Gforth

for version 1.0.0, October 25, 2026

Neal Crook
Anton Ertl
David Kuehling
Bernd Paysan
Jens Wilke
Gerald Wodni

This manual is for Gforth (version 1.0.0, October 25, 2026), a fast and portable implementation of the Standard Forth language. It serves as reference manual, but it also contains an introduction to Forth and a Forth tutorial.

Authors: Bernd Paysan, Anton Ertl, Gerald Wodni, Neal Crook, David Kuehling, Jens Wilke Copyright © 1995, 1996, 1997, 1998, 2000, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020, 2021, 2022, 2023, 2024,2025 Free Software Foundation, Inc.

> Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with no Invariant Sections, with the Front-Cover texts being "A GNU Manual," and with the Back-Cover Texts as in (a) below. A copy of the license is included in the section entitled "GNU Free Documentation License."
>
> (a) The FSF's Back-Cover Text is: "You have freedom to copy and modify this GNU Manual, like GNU software. Copies published by the Free Software Foundation raise funds for GNU development."

Table of Contents

Preface 1

1 Goals of Gforth 2
1.1 Stability Goals 2

2 Gforth Environment 4
2.1 Invoking Gforth 4
2.1.1 Code generation options 6
2.1.2 Image-specific options of `gforth.fi` 9
2.2 Leaving Gforth 9
2.3 Help on Gforth 9
2.4 Command-line editing 10
2.5 Environment variables 11
2.6 Gforth files 11
2.7 Gforth in pipes 12
2.8 Startup speed 12

3 Forth Tutorial 14
3.1 Starting Gforth 14
3.2 Syntax 14
3.3 Crash Course 15
3.4 Stack 15
3.5 Arithmetics 15
3.6 Stack Manipulation 16
3.7 Using files for Forth code 17
3.8 Comments 17
3.9 Colon Definitions 18
3.10 Decompilation 18
3.11 Stack-Effect Comments 18
3.12 Types 20
3.13 Factoring 20
3.14 Designing the stack effect 21
3.15 Local Variables 21
3.16 Conditional execution 22
3.17 Flags and Comparisons 23
3.18 General Loops 24
3.19 Counted loops 25
3.20 Recursion 26
3.21 Leaving definitions or loops 27
3.22 Return Stack 27
3.23 Memory 28
3.24 Characters and Strings 29

3.25 Alignment 30
3.26 Floating Point 31
3.27 Files 32
3.27.1 Open file for input 32
3.27.2 Create file for output 32
3.27.3 Scan file for a particular line 32
3.27.4 Copy input to output 33
3.27.5 Close files 33
3.28 Interpretation and Compilation Semantics and Immediacy 33
3.29 Execution Tokens 35
3.30 Exceptions 36
3.31 Defining Words 37
3.32 Arrays and Records 38
3.33 `POSTPONE` 39
3.34 `Literal` 40
3.35 Advanced macros 40
3.36 Compilation Tokens 41
3.37 Wordlists and Search Order 41

4 An Introduction to Standard Forth 43

4.1 Introducing the Text Interpreter 43
4.2 Stacks, postfix notation and parameter passing 45
4.3 Your first Forth definition 48
4.4 How does that work? 49
4.5 Forth is written in Forth 51
4.6 Review - elements of a Forth system 52
4.7 Where To Go Next 52
4.8 Exercises 53

5 Literals in source code 54

5.1 Integer and character literals 54
5.2 Floating-point number and complex literals 54
5.3 String and environment variable literals 56
5.4 Literals for tokens and addresses 56
5.5 Disambiguating recognizers 56

6 Forth Words 57

6.1 Notation 57
6.2 Case insensitivity 59
6.3 Comments 59
6.4 Boolean Flags 60
6.5 Arithmetic 60
6.5.1 Single precision 60
6.5.2 Double precision 61
6.5.3 Mixed precision 61

6.5.4 Integer division ... 62
6.5.5 Two-stage integer division ... 64
6.5.6 Bitwise operations ... 66
6.5.7 Numeric comparison ... 68
6.5.8 Pseudo-random number generator ... 69
6.5.9 Floating Point ... 69
6.6 Floating-point comparisons ... 72
6.7 Stack Manipulation ... 72
6.7.1 Data stack ... 72
6.7.2 Floating point stack ... 73
6.7.3 Return stack ... 73
6.7.4 Locals stack ... 75
6.7.5 Stack pointer manipulation ... 75
6.8 Memory ... 75
6.8.1 Memory model ... 75
6.8.2 Dictionary allocation ... 76
6.8.3 Sections ... 78
6.8.4 Heap allocation ... 80
6.8.4.1 Memory blocks and heap allocation ... 80
6.8.4.2 Growable memory buffers ... 81
6.8.5 Memory Access ... 81
6.8.6 Special Memory Accesses ... 82
6.8.7 Address arithmetic ... 84
6.8.8 Memory Blocks ... 87
6.9 Strings and Characters ... 88
6.9.1 String representations ... 88
6.9.2 Xchars and Unicode ... 89
6.9.3 String and character literals ... 91
6.9.4 String words ... 93
6.9.5 $tring words ... 96
6.9.6 Internationalization and localization ... 98
6.9.7 Substitute ... 101
6.9.8 Counted string words ... 102
6.10 Control Structures ... 103
6.10.1 Selection ... 103
6.10.2 General Loops ... 104
6.10.3 Counted Loops ... 104
6.10.4 General loops with multiple exits ... 109
6.10.5 General control structures with `case` ... 110
6.10.6 Arbitrary control structures ... 112
6.10.7 Calls and returns ... 113
6.10.8 Exception Handling ... 114
6.11 Defining Words ... 119
6.11.1 `CREATE` ... 119
6.11.2 Variables ... 120
6.11.3 Constants ... 120
6.11.4 Values ... 121

6.11.5 Colon Definitions ... 123
6.11.6 Inline Definitions ... 123
6.11.7 Anonymous Definitions ... 124
6.11.8 Quotations ... 125
6.11.9 Supplying the name of a defined word ... 125
6.11.10 User-defined Defining Words ... 125
6.11.10.1 User-defined defining words with colon definitions ... 126
6.11.10.2 User-defined defining words using `create` ... 126
6.11.10.3 Applications of `CREATE..DOES>` ... 128
6.11.10.4 The gory details of `CREATE..DOES>` ... 129
6.11.10.5 Advanced does> usage example ... 130
6.11.10.6 Words with user-defined `to` etc. ... 132
6.11.10.7 User-defined `compile,` ... 134
6.11.10.8 Creating from a prototype ... 136
6.11.10.9 Making a word current ... 137
6.11.10.10 `Const-does>` ... 137
6.11.11 Deferred Words ... 138
6.11.12 Synonyms ... 141
6.12 Structures ... 141
6.12.1 Standard Structures ... 142
6.12.2 Value-Flavoured and Defer-Flavoured Fields ... 144
6.12.3 Structure Extension ... 146
6.12.4 Gforth structs ... 147
6.13 User-defined Stacks ... 149
6.14 Interpretation and Compilation Semantics ... 150
6.14.1 Where are interpretation semantics used? ... 150
6.14.2 Where are compilation semantics used? ... 151
6.14.3 Which semantics do existing words have? ... 151
6.14.4 What semantics do normal definitions have? ... 152
6.14.5 How to define immediate words ... 152
6.14.6 How to define combined words ... 154
6.15 Tokens for Words ... 156
6.15.1 Execution token ... 156
6.15.2 Name token ... 158
6.15.3 Compilation token ... 160
6.16 Compiling words ... 160
6.16.1 Literals ... 160
6.16.2 Macros ... 162
6.17 The Text Interpreter ... 166
6.17.1 Input Sources ... 168
6.17.2 Interpret/Compile states ... 169
6.17.3 Interpreter Directives ... 169
6.17.4 Recognizers ... 171
6.17.4.1 Default recognizers ... 171
6.17.4.2 Recognizer order ... 175
6.17.4.3 Defining recognizers ... 176
6.17.4.4 Defining translation tokens ... 178

6.17.4.5 Performing translation actions 180
6.17.5 Text Interpreter Hooks 181
6.18 The Input Stream 181
6.19 Word Lists 182
6.19.1 Why use word lists? 185
6.19.2 Wordlist and vocabulary usage 186
6.20 Number conversion 187
6.20.1 Base and integer decimal point 187
6.20.2 String to number conversion 188
6.20.3 Integer to string conversion 189
6.20.4 Floating-point to string conversion 192
6.21 Environmental Queries 192
6.22 Files 196
6.22.1 Forth source files 196
6.22.2 General files 197
6.22.3 Redirection 199
6.22.4 Directories 199
6.22.5 Search Paths 200
6.22.5.1 Source Search Paths 201
6.22.5.2 General Search Paths 201
6.23 Blocks 202
6.24 Other I/O 205
6.24.1 Simple numeric output 205
6.24.2 Floating-point output 206
6.24.3 Miscellaneous output 208
6.24.4 Displaying characters and strings 209
6.24.5 Terminal output 209
6.24.5.1 Color output 210
6.24.5.2 Color themes 211
6.24.6 Single-key input 211
6.24.7 String input from the terminal 214
6.24.8 Pipes 215
6.24.9 CSV Reader 215
6.25 OS command line arguments 216
6.26 Locals 217
6.26.1 Gforth locals 217
6.26.1.1 Locals definitions words 218
6.26.1.2 Where are locals visible by name? 220
6.26.1.3 How long do locals live? 223
6.26.1.4 POSTPONEing locals 223
6.26.1.5 Locals programming style 224
6.26.1.6 Locals implementation 225
6.26.2 Standard Forth locals 227
6.27 Object-oriented Forth 228
6.27.1 Why object-oriented programming? 228
6.27.2 Object-Oriented Terminology 228
6.27.3 The `objects.fs` model 229

6.27.3.1 Properties of the `objects.fs` model 229
6.27.3.2 Basic `objects.fs` Usage 230
6.27.3.3 The `object.fs` base class 231
6.27.3.4 Creating objects 231
6.27.3.5 Object-Oriented Programming Style 231
6.27.3.6 Class Binding 231
6.27.3.7 Method conveniences 232
6.27.3.8 Classes and Scoping 233
6.27.3.9 Dividing classes 233
6.27.3.10 Object Interfaces 234
6.27.3.11 `objects.fs` Implementation 235
6.27.3.12 `objects.fs` Glossary 236
6.27.4 The `oof.fs` model 239
6.27.4.1 Properties of the `oof.fs` model 239
6.27.4.2 Basic `oof.fs` Usage 240
6.27.4.3 The `oof.fs` base class 241
6.27.4.4 Class Declaration 242
6.27.5 The `mini-oof.fs` model 242
6.27.5.1 Basic `mini-oof.fs` Usage 243
6.27.5.2 Mini-OOF Example 243
6.27.5.3 `mini-oof.fs` Implementation 244
6.27.6 Mini-OOF2 246
6.27.7 Comparison of object models 246
6.28 Closures 248
6.28.1 How do I read outer locals? 251
6.28.2 How do I write outer locals? 251
6.29 Regular Expressions 254
6.30 Programming Tools 257
6.30.1 Text interpreter status 257
6.30.2 Locating source code definitions 258
6.30.3 Locating uses of a word 259
6.30.4 Locating backtrace entry source 260
6.30.5 Examining compiled code 261
6.30.6 Examining data 263
6.30.7 Forgetting words 264
6.30.8 Debugging 265
6.30.9 Assertions 266
6.30.10 Singlestep Debugger 267
6.30.11 Code Coverage and Execution Frequency 268
6.30.12 `ans-report.fs`: Report the words used, sorted by wordset 270
6.30.12.1 Caveats 270
6.30.13 Stack depth changes during interpretation 271
6.31 Multitasker 271
6.31.1 Pthreads 271
6.31.1.1 Basic multi-tasking 272
6.31.1.2 Task-local data 273

6.31.1.3 Mutexes ... 274
6.31.1.4 Atomic memory accesses and memory ordering ... 274
6.31.1.5 Message queues ... 275
6.31.2 Cilk ... 276
6.32 C Interface ... 276
6.32.1 Existing C interface libraries ... 277
6.32.2 Calling C functions ... 277
6.32.3 Declaring C Functions ... 278
6.32.4 Calling C function pointers from Forth ... 280
6.32.5 Defining library interfaces ... 280
6.32.6 C compiler and linker options ... 281
6.32.7 Callbacks ... 282
6.32.8 How the C interface works ... 283
6.32.9 Low-Level C Interface Words ... 283
6.32.10 Automated interface generation using SWIG ... 284
6.32.10.1 Basic operation ... 284
6.32.10.2 Detailed operation: ... 284
6.32.10.3 Examples ... 284
6.32.11 Migrating from Gforth 0.7 ... 285
6.33 Assembler and Code Words ... 285
6.33.1 Definitions in assembly language ... 285
6.33.2 Common Assembler ... 288
6.33.3 Common Disassembler ... 288
6.33.4 386 Assembler ... 290
6.33.5 Alpha Assembler ... 291
6.33.6 AMD64 (x86_64) Assembler ... 291
6.33.7 ARM A32 assembler ... 295
6.33.8 LoongArch assembler ... 296
6.33.9 MIPS assembler ... 297
6.33.10 PowerPC assembler ... 298
6.33.11 RISC-V assembler ... 298
6.33.12 Other assemblers ... 298
6.34 Carnal words ... 299
6.34.1 Header fields ... 299
6.34.2 Header methods ... 300
6.34.3 Threading Words ... 303
6.35 Passing Commands to the Operating System ... 305
6.36 Keeping track of Time ... 305
6.37 Miscellaneous Words ... 306

7 Error messages ... 307

8 Standard conformance ... 309

8.1 The Core Words ... 310
8.1.1 Implementation Defined Options ... 310
8.1.2 Ambiguous conditions ... 312

8.1.3 Other system documentation ... 316
8.2 The optional Block word set ... 316
8.2.1 Implementation Defined Options ... 316
8.2.2 Ambiguous conditions ... 316
8.2.3 Other system documentation ... 317
8.3 The optional Double Number word set ... 317
8.3.1 Ambiguous conditions ... 317
8.4 The optional Exception word set ... 317
8.4.1 Implementation Defined Options ... 317
8.5 The optional Facility word set ... 317
8.5.1 Implementation Defined Options ... 317
8.5.2 Ambiguous conditions ... 318
8.6 The optional File-Access word set ... 318
8.6.1 Implementation Defined Options ... 318
8.6.2 Ambiguous conditions ... 319
8.7 The optional Floating-Point word set ... 319
8.7.1 Implementation Defined Options ... 320
8.7.2 Ambiguous conditions ... 320
8.8 The optional Locals word set ... 321
8.8.1 Implementation Defined Options ... 321
8.8.2 Ambiguous conditions ... 321
8.9 The optional Memory-Allocation word set ... 322
8.9.1 Implementation Defined Options ... 322
8.10 The optional Programming-Tools word set ... 322
8.10.1 Implementation Defined Options ... 322
8.10.2 Ambiguous conditions ... 322
8.11 The optional Search-Order word set ... 323
8.11.1 Implementation Defined Options ... 323
8.11.2 Ambiguous conditions ... 323

9 Should I use Gforth extensions? ... 324

10 Integrating Gforth into C programs ... 325

10.1 Types ... 326
10.2 Variables ... 326
10.3 Functions ... 326
10.4 Signals ... 326

11 Emacs and Gforth ... 327

11.1 Installing gforth.el ... 327
11.2 Emacs Tags ... 327
11.3 Hilighting ... 328
11.4 Auto-Indentation ... 328
11.5 Blocks Files ... 329

12 Image Files ... 330
12.1 Image Licensing Issues ... 330
12.2 Image File Background ... 330
12.3 Fully Relocatable Image Files ... 331
12.3.1 `gforthmi` ... 331
12.3.2 `cross.fs` ... 333
12.4 Non-Relocatable Image Files ... 333
12.5 Stack and Dictionary Sizes ... 334
12.6 Running Image Files ... 334
12.7 Modifying the Startup Sequence ... 335

13 Optimizations ... 337
13.1 Threaded code ... 337
13.2 Dynamic Superinstructions ... 339
13.3 Static Superinstructions ... 342
13.4 Stack caching ... 343
13.5 IP-update optimization ... 344
13.6 Literal-based optimizations ... 346
13.7 Differences between the engines ... 346
13.8 Performance ... 347

14 Engine Construction ... 349
14.1 Portability **and** Performance ... 349
14.2 Primitives ... 350
14.3 Native code of primitive variants ... 350
14.4 Source code of primitive variants ... 353

15 Cross Compiler ... 355
15.1 Using the Cross Compiler ... 355
15.2 How the Cross Compiler Works ... 357

16 MINOS2, a GUI library ... 358
16.1 MINOS2 object framework ... 358
16.1.1 `actor` methods: ... 358
16.1.2 `widget` methods: ... 358
16.2 MINOS2 tutorial ... 358

Appendix A Bugs ... 360

Appendix B Authors and Ancestors of Gforth ... 361
B.1 Authors and Contributors ... 361
B.2 Pedigree ... 361

Appendix C Other Forth-related information . 362

Appendix D Licenses 363
D.1 GNU Free Documentation License 363
D.1.1 ADDENDUM: How to use this License for your documents . 369
D.2 GNU GENERAL PUBLIC LICENSE 369

Word Index 381

Concept and Word Index 400

Preface

This manual documents Gforth. Some introductory material is provided for readers who are unfamiliar with Forth or who are migrating to Gforth from other Forth compilers. However, this manual is primarily a reference manual.

1 Goals of Gforth

The goal of the Gforth Project is to develop a standard model for Standard Forth. This can be split into several subgoals:

- Gforth should conform to the Forth Standard.
- It should be a model, i.e. it should define all the implementation-dependent things.
- It should become standard, i.e. widely accepted and used. This goal is the most difficult one.

To achieve these goals Gforth should be

- Similar to previous models (fig-Forth, F83)
- Powerful. It should provide for all the things that are considered necessary today and even some that are not yet considered necessary.
- Efficient. It should not get the reputation of being exceptionally slow.
- Free.
- Available on many machines/easy to port.

Have we achieved these goals?

Gforth conforms to the Forth-94 (ANS Forth) and Forth-2012 standards.

We have changed some of the internal data structures (in particular, the headers) over time, so Gforth does not provide the stability of implementation details that we originally aimed for; they were too constraining for a long-term project like Gforth. However, we still aim for a high level of stability.

Gforth is quite popular and is treated by some like a de-facto standard.

It has some similarities to and some differences from previous models.

It has powerful features, and the version 1.0 indicates that it can do everything (and more) that we originally envisioned. That does not mean that we will stop development.

We certainly have achieved and exceeded our execution speed goals (see Section 13.8 [Performance], page 347)[1].

Gforth is free and available on many platforms.

1.1 Stability Goals

Programs that work on earlier versions of Gforth should also work on newer versions. However, there are some caveats:

Internal data structures (including the representation of code) of Gforth may change between versions, unless they are documented.

Moreover, we only feel obliged to keep standard words (i.e., with standard wordset names) and words documented as stable Gforth extensions (with wordset name `gforth` or `gforth-<version>`, see Section 6.1 [Notation], page 57). Other words may be removed in newer releases.

[1] However, in 1998 the bar was raised when the major commercial Forth vendors switched to native code compilers.

In particular, you may find a word by using `locate` or otherwise inspecting Gforth's source code. You can see the wordset in a comment right after the stack-effect comment. E.g., in

```
: execute-parsing ( ... addr u xt -- ... ) \ gforth
```

the wordset is `gforth`.

If there is no wordset for a word, it is an internal factor and may be removed in a future version. If the wordset is `gforth-experimental`, `gforth-internal`, or `gforth-obsolete`, the word may also be removed in a future version. In particular, `gforth-experimental` indicates that this is a supported word that we do not consider stable yet; `gforth-obsolete` indicates an intent to remove the word in the next version; and `gforth-internal` (or no wordset) indicates that we may remove the word as soon as we no longer use it in Gforth.

If you want to use a particular word that is not marked as stable, please let us know, and we will consider to add the word as stable word (or we may suggest an alternative to using this word).

2 Gforth Environment

Note: ultimately, the Gforth man page will be auto-generated from the material in this chapter.

For related information about the creation of images see Chapter 12 [Image Files], page 330.

2.1 Invoking Gforth

Gforth is made up of two parts; an executable "engine" (named `gforth` or `gforth-fast`) and an image file. To start it, you will usually just say `gforth` – this automatically loads the default image file `gforth.fi`. In many other cases the default Gforth image will be invoked like this:

```
gforth [file | -e forth-code] ...
```

This interprets the contents of the files and the Forth code in the order they are given.

In addition to the `gforth` engine, there is also an engine called `gforth-fast`, which is faster, but gives less informative error messages (see Chapter 7 [Error messages], page 307) and may catch some errors (in particular, stack underflows and integer division errors) later or not at all. You should use it for debugged, performance-critical programs.

Moreover, there is an engine called `gforth-itc`, which uses indirect-threaded code and is useful in some backwards-compatibility situations.

In general, the command line looks like this:

```
gforth[-fast] [engine options] [image options]
```

The engine options must come before the rest of the command line. They are:

`--image-file` *file*
`-i` *file*
: Loads the Forth image *file* instead of the default `gforth.fi` (see Chapter 12 [Image Files], page 330).

`--appl-image` *file*
: Loads the image *file* and leaves all further command-line arguments to the image (instead of processing them as engine options). This is useful for building executable application images on Unix, built with `gforthmi --application ...`.

`--no-0rc`
: Do not load `~/.config/gforthrc0` nor the file specified by `GFORTH_ENV`.

`--path` *path*
`-p` *path*
: Uses *path* for searching the image file and Forth source code files instead of the default in the environment variable `GFORTHPATH` or the path specified at installation time and the working directory `.` (e.g., `/usr/local/share/gforth/0.2.0:.`). A path is given as a list of directories, separated by '`:`' (previous versions had '`;`' for other OSes, but since Cygwin now only accepts `/cygdrive/<letter>`, and we dropped support for OS/2 and MS-DOS, it is '`:`' everywhere).

`--dictionary-size` *size*
`-m` *size* Allocate *size* space for the Forth dictionary space instead of using the default specified in the image (default: 8M). The *size* specification for this and subsequent options consists of an integer and a unit (e.g., `1G`). The unit can be one of `b` (bytes), `e` (element size, in this case Cells), `k` (kilobytes), `M` (Megabytes), `G` (Gigabytes), and `T` (Terabytes). If no unit is specified, `e` is used.

`--data-stack-size` *size*
`-d` *size* Allocate *size* space for the data stack instead of using the default specified in the image (default: 16K).

`--return-stack-size` *size*
`-r` *size* Allocate *size* space for the return stack instead of using the default specified in the image (default: 15K).

`--fp-stack-size` *size*
`-f` *size* Allocate *size* space for the floating point stack instead of using the default specified in the image (default: 15.5K). In this case the unit specifier `e` refers to floating point numbers.

`--locals-stack-size` *size*
`-l` *size* Allocate *size* space for the locals stack instead of using the default specified in the image (default: 14.5K).

`--map_32bit`
Allocate the dictionary and some other areas in the lower 2GB of the address space, if possible. The purpose of this option is debugging convenience.

`--vm-commit`
Normally, Gforth tries to start up even if there is not enough virtual memory for the dictionary and the stacks (using `MAP_NORESERVE` on OSs that support it); so you can ask for a really big dictionary and/or stacks, and as long as you don't use more virtual memory than is available, everything will be fine (but if you use more, processes get killed). With this option you just use the default allocation policy of the OS; for OSs that don't overcommit (e.g., Solaris), this means that you cannot and should not ask for as big dictionary and stacks, but once Gforth successfully starts up, out-of-memory won't kill it.

`--help`
`-h` Print a message about the command-line options

`--version`
`-v` Print version and exit

`--diag`

`-D` Checks for and reports some performance problems.

`--debug` Print some information useful for debugging on startup.

`--debug-mcheck`
Try to find and report erroneous usage of `allocate`, `free`, and the C functions `malloc()`, `free()`, etc.

`--offset-image`
: Start the dictionary at a slightly different position than would be used otherwise (useful for creating relocatable images, see Section 12.3 [Fully Relocatable Image Files], page 331).

`--no-offset-im`
: Start the dictionary at the normal position.

`--clear-dictionary`
: Initialize all bytes in the dictionary to 0 before loading the image (see Section 12.3 [Fully Relocatable Image Files], page 331).

`--die-on-signal [`*`number`*`]`
: Normally Gforth handles most signals (e.g., the user interrupt SIGINT, or the segmentation violation SIGSEGV) by translating it into a Forth `THROW`. With this option, Gforth exits if it receives such a signal. This option is useful when the engine and/or the image might be severely broken (such that it causes another signal before recovering from the first); this option avoids endless loops in such cases. The optional number set the number of signals to be handled; only the last one will cause Gforth to exit.

`--ignore-async-signals`
: Ignore asynchronous signals (e.g., `SIGINT` generated with *`Ctrl-c`*).

2.1.1 Code generation options

`--no-dynamic`
`--dynamic`
: Disable or enable dynamic superinstructions with replication (see Section 13.2 [Dynamic Superinstructions], page 339). Default enabled.

`--no-dynamic-image`
: Disable dynamic native-code generation when loading the Gforth image, but generate dynamic native code afterwards. This option is useful when debugging Gforth's code generator.

`--no-super`
: Disable dynamic superinstructions, use just dynamic replication; this is useful if you want to patch threaded code (see Section 13.2 [Dynamic Superinstructions], page 339).

`--ss-number=`*`N`*
: Use only the first *N* static superinstructions compiled into the engine (default: use them all; note that only `gforth-fast` has any). This option is useful for measuring the performance impact of static superinstructions.

`--ss-min-codesize`
`--ss-min-ls`
`--ss-min-lsu`
`--ss-min-nexts`
: Use specified metric for determining the cost of a primitive or static superinstruction for static superinstruction selection. `Codesize` is the native code size

of the primitive or static superinstruction, `ls` is the number of loads and stores, `lsu` is the number of loads, stores, and updates, and `nexts` is the number of dispatches (not taking dynamic superinstructions into account), i.e. every primitive or static superinstruction has cost 1. Default: `codesize` if you use dynamic code generation, otherwise `nexts`.

`--ss-greedy`
: This option is useful for measuring the performance impact of static superinstructions. By default, an optimal shortest-path algorithm is used for selecting static superinstructions. With `--ss-greedy` this algorithm is modified to assume that anything after the static superinstruction currently under consideration is not combined into static superinstructions. With `--ss-min-nexts` this produces the same result as a greedy algorithm that always selects the longest superinstruction available at the moment. E.g., if there are superinstructions AB and BCD, then for the sequence A B C D the optimal algorithm will select A BCD and the greedy algorithm will select AB C D.

`--opt-ip-updates=`*n*
: Set the level of IP-update optimization (default: 31 (7+3*8)). *n* is computed as *n1*+8**n2*.

  n1 indicates the use of IP-update optimization in straight-line code: 0 means no IP-update optimization, 1 combines IP-update optimizations of primitives without inline arguments, 2 also eliminates the dead IP updates of `;s`, `execute-;s` and *fast-throw*, `>2` eliminates the IP updates in front of several frequently-used primitives with inline arguments.

  n2 is the number of ip-updates that can replace a load in a backwards or unconditional branch; for conditional forward branches only *n2*/2 ip-updates replace a load (to avoid too many additional updates in the fall-through path).

`--code-block-size=`*size*
: Size of native-code blocks (default: 2M). Gforth allocates as many blocks of this size as necessary.

`--print-metrics`
: On exit from Gforth: Print some metrics used during static superinstruction selection: `code size` is the actual size of the dynamically generated code. `Metric codesize` is the sum of the codesize metrics as seen by static superinstruction selection; there is a difference from `code size`, because not all primitives and static superinstructions are compiled into dynamically generated code, and because of markers. The other metrics correspond to the `ss-min-...` options. This option is useful for evaluating the effects of the `--ss-...` options.

`--print-prims`
: When exiting GforthL: Print the primitives with static usage counts. E.g., one line might look like:

```
?branch           1-1  0   21 1575   73 0x5573e4048c33 len= 4+ 25+ 3 send=0
```

  The columns are, from left to right: name of the primitive, stack-caching state transition (from a state with 1 stack item in a register to the same state in the example), IP offset for this version of the primitive (0 for most primitives,

but, e.g., for `?branch` there are also versions with 0-zero offset), index of the primitive, index of the corresponding branch-to-IP variant (in case of a branch), static number of occurrences of the primitive in the loaded/compiled code, address of the code of the primitive (or `(nil)` if the primitive is not relocatable), length of the parts of this code: ip-update+main+dispatch, and whether the primitive ends a superblock (i.e., an unconditional branch or the like).

`--print-nonreloc`
When starting Gforth: Print the non-relocatable primitives.

`--print-sequences`
When loading the image: For each superblock in the image, print the sequence of primitives.

`--tpa-noautomaton`
`--tpa-noequiv`
These options are about using an automaton for speeding up startup and compilation, in particular the shortest-path algorithm used for selecting static superinstructions and stack caching variants; tpa stands for for "tree-parsing automaton" (although we only have sequences, not trees). In the `gforth` engine the default is to use an automaton with state equivalence (state equivalence reduces the number of states compared to having one state for every sequence prefix), which is the fastest option and requires the least memory.

With static superinstructions the automaton does not work correctly, so Gforth falls back to `--tpa-noautomaton` in that case unless you ask for `--tpa-noequiv` (`gforth-fast` uses static superinstructions and therefore `--tpa-noautomaton` by default).

`--tpa-noequiv` turns off state equivalence, which costs memory and compiles a little slower than using an automaton.

`--tpa-noautomaton` turns off using the automaton. This consumes quite a bit more compile time, and should in theory use less memory than using an automaton, but apparently there is a bug in Gforth, and it consumes more memory.

The following shows the startup speed and memory consumption of Gforth 0.7.9_20240821 run with `gforth-fast -e bye` (plus the options given in the table) on a Core-i5 6600K (Skylake):

```
 cycles    instructions KB(RSS) other options
23_309_239  43_534_167   9228   --ss-number=0
26_399_456  51_895_687  11316   --ss-number=0 --tpa-noequiv
40_427_672  93_709_354  10988   --ss-number=0 --tpa-noautomaton
27_599_969  53_126_621  11320
27_732_944  53_128_381  11320   --tpa-noequiv
42_960_520  95_466_840  11044   --tpa-noautomaton
```

`--tpa-trace`
This option produces data about the number of states generated during startup and compilation.

2.1.2 Image-specific options of `gforth.fi`

As explained above, the image-specific command-line arguments for the default image `gforth.fi` consist of a sequence of filenames and `-e` *`forth-code`* options that are interpreted in the sequence in which they are given. The `-e` *`forth-code`* or `--evaluate` *`forth-code`* option evaluates the Forth code. This option takes only one argument; if you want to evaluate more Forth words, you have to quote them or use `-e` several times. To exit after processing the command line (instead of entering interactive mode) append `-e bye` to the command line. You can also process the command-line arguments with a Forth program (see Section 6.25 [OS command line arguments], page 216). If there is an uncaught exception while processing image options, Gforth will exit with a non-zero exit code. This is how you run scripts in Gforth.

If you have several versions of Gforth installed, `gforth` will invoke the version that was installed last. `gforth-<`*`version`*`>` invokes a specific version. If your environment contains the variable `GFORTHPATH`, you may want to override it by using the `--path` option.

On startup, before processing any of the image options, the user initialization file is included, if it exists. The user initialization file is `~/.config/gforthrc0`, or, if the environment variable `GFORTH_ENV` is set, it contains the name of the user initialization file. You can suppress loading this file with by setting `GFORTH_ENV` to `off`, or with the option `--no-0rc`.

After processing all the image options and just before printing the boot message, the user initialization file `~/.config/gforthrc` from your home directory is included, unless the option `--no-rc` is given.

Warning levels can be set with

`-W` Turn off warnings

`-Won` Turn on warnings (level 1)

`-Wall` Turn on beginner warnings (level 2)

`-Wpedantic`
Turn on pedantic warnings (level 3)

`-Werror` Turn warnings into errors (level 4)

2.2 Leaving Gforth

You can leave Gforth by typing `bye` or *`Ctrl-d`* (at the start of a line) or (if you invoked Gforth with the `--die-on-signal` option) *`Ctrl-c`*. When you leave Gforth, all of your definitions and data are discarded. For ways of saving the state of the system before leaving Gforth see Chapter 12 [Image Files], page 330.

`bye` (–) tools-ext

Exit Gforth (with exit status 0).

2.3 Help on Gforth

Gforth has a simple, text-based online help system.

`help` ("*rest-of-line*" –) gforth-1.0

If no name is given, show basic help. If a documentation node name is given followed by "::", show the start of the node. If the name of a word is given, show the documentation of the word if it exists, or its source code if not. If something else is given that is recognized, shows help on the recognizer. You can then use the same keys and commands as after using `locate` (see Section 6.30.2 [Locating source code definitions], page 258).

`authors` (–) gforth-1.0

show the list of authors

`license` (–) gforth-0.2

print the license statement

2.4 Command-line editing

Gforth maintains a history file that records every line that you type to the text interpreter. This file is preserved between sessions, and is used to provide a command-line recall facility; if you type *`Ctrl-P`* repeatedly you can recall successively older commands from this (or previous) session(s). The full list of command-line editing facilities is:

- *`Ctrl-p`* ("previous") (or up-arrow) to recall successively older lines from the history buffer.
- *`Ctrl-n`* ("next") (or down-arrow) to recall successively newer lines from the history buffer. If you moved to an older line earlier and gave it to Gforth for text-interpretation, asking for the next line as the first editing command gives you the next line after the one you selected last time.
- *`Ctrl-f`* (or right-arrow) to move the cursor right, non-destructively.
- *`Ctrl-b`* (or left-arrow) to move the cursor left, non-destructively.
- *`Ctrl-h`* (backspace) to delete the character to the left of the cursor, closing up the line.
- *`Ctrl-k`* to delete ("kill") from the cursor to the end of the line.
- *`Ctrl-a`* to move the cursor to the start of the line.
- *`Ctrl-e`* to move the cursor to the end of the line.
- `RET` (*`Ctrl-m`*) or `LFD` (*`Ctrl-j`*) to submit the current line.
- `TAB` to step through all possible full-word completions of the word currently being typed.
- *`Ctrl-d`* on an empty line line to terminate Gforth (gracefully, using `bye`).
- *`Ctrl-x`* (or `Ctrl-d` on a non-empty line) to delete the character under the cursor.

When editing, displayable characters are inserted to the left of the cursor position; the line is always in "insert" (as opposed to "overstrike") mode.

On Unix systems, the history file is `$HOME/.local/share/gforth/history` by default[1] . You can find out the name and location of your history file using:

```
history-file type \ Unix-class systems

history-file type \ Other systems
history-dir  type
```

[1] i.e. it is stored in the user's home directory.

If you enter long definitions by hand, you can use a text editor to paste them out of the history file into a Forth source file for reuse at a later time.

Gforth never trims the size of the history file, so you should do this periodically, if necessary.

2.5 Environment variables

Gforth uses these environment variables:

- `GFORTHHIST` – (Unix systems only) specifies the path for the history file `.gforth-history`. Default: `$HOME/.local/share/gforth/history`.
- `GFORTHPATH` – specifies the path used when searching for the gforth image file and for Forth source-code files (usually '`.`', the current working directory). Path separator is '`:`', a typical path would be `/usr/local/share/gforth/1.0:.`.
- `LANG` – see `LC_CTYPE`
- `LC_ALL` – see `LC_CTYPE`
- `LC_CTYPE` – If this variable contains "UTF-8" on Gforth startup, Gforth uses the UTF-8 encoding for strings internally and expects its input and produces its output in UTF-8 encoding, otherwise the encoding is 8bit (see Section 6.9.2 [Xchars and Unicode], page 89). If this environment variable is unset, Gforth looks in `LC_ALL`, and if that is unset, in `LANG`.
- `GFORTHSYSTEMPREFIX` – specifies what to prepend to the argument of `system` before passing it to C's `system()`. Default: `"./$COMSPEC /c "` on Windows, `""` on other OSs. The prefix and the command are directly concatenated, so if a space between them is necessary, append it to the prefix.
- `GFORTH` – used by `gforthmi`, See Section 12.3.1 [gforthmi], page 331.
- `GFORTHD` – used by `gforthmi`, See Section 12.3.1 [gforthmi], page 331.
- `TMP`, `TEMP` - (non-Unix systems only) used as a potential location for the history file.

All the Gforth environment variables default to sensible values if they are not set.

2.6 Gforth files

When you install Gforth on a Unix system, it installs files in these locations by default:

- `/usr/local/bin/gforth`
- `/usr/local/bin/gforthmi`
- `/usr/local/man/man1/gforth.1` - man page.
- `/usr/local/info` - the Info version of this manual.
- `/usr/local/lib/gforth/<version>/...` - Gforth `.fi` files.
- `/usr/local/share/gforth/<version>/TAGS` - Emacs TAGS file.
- `/usr/local/share/gforth/<version>/...` - Gforth source files.
- `.../emacs/site-lisp/gforth.el` - Emacs gforth mode.

You can select different places for installation by using `configure` options (listed with `configure --help`).

2.7 Gforth in pipes

Gforth can be used in pipes created elsewhere (described in the following). It can also create pipes on its own (see Section 6.24.8 [Pipes], page 215).

If you pipe into Gforth, your program should read with `read-file` or `read-line` from `stdin` (see Section 6.22.2 [General files], page 197). `Key` does not recognize the end of input. Words like `accept` echo the input and are therefore usually not useful for reading from a pipe. You have to invoke the Forth program with an OS command-line option, as you have no chance to use the Forth command line (the text interpreter would try to interpret the pipe input).

You can output to a pipe with `type`, `emit`, `cr` etc.

When you write to a pipe that has been closed at the other end, Gforth receives a SIGPIPE signal ("pipe broken"). Gforth translates this into the exception `broken-pipe-error`. If your application does not catch that exception, the system catches it and exits, usually silently (unless you were working on the Forth command line; then it prints an error message and exits). This is usually the desired behaviour.

If you do not like this behaviour, you have to catch the exception yourself, and react to it.

Here's an example of an invocation of Gforth that is usable in a pipe:

```
gforth -e ": foo begin pad dup 10 stdin read-file throw dup while \
 type repeat ; foo bye"
```

This example just copies the input verbatim to the output. A very simple pipe containing this example looks like this:

```
cat startup.fs |
gforth -e ": foo begin pad dup 80 stdin read-file throw dup while \
 type repeat ; foo bye"|
head
```

Pipes involving Gforth's `stderr` output do not work.

2.8 Startup speed

If Gforth is used for CGI scripts or in shell scripts, its startup speed may become a problem. On a 3GHz Core 2 Duo E8400 under 64-bit Linux 2.6.27.8 with libc-2.7, `gforth-fast -e bye` takes 13.1ms user and 1.2ms system time (`gforth -e bye` is faster on startup with about 3.4ms user time and 1.2ms system time, because it subsumes some of the options discussed below).

If startup speed is a problem, you may consider the following ways to improve it; or you may consider ways to reduce the number of startups (for example, by using Fast-CGI). Note that the first steps below improve the startup time at the cost of run-time (including compile-time), so whether they are profitable depends on the balance of these times in your application.

An easy step that influences Gforth startup speed is the use of a number of options that increase run-time, but decrease image-loading time.

The first of these that you should try is `--ss-number=0 --ss-states=1` because this option buys relatively little run-time speedup and costs quite a bit of time at startup.

`gforth-fast --ss-number=0 --ss-states=1 -e bye` takes about 2.8ms user and 1.5ms system time.

The next option is `--no-dynamic` which has a substantial impact on run-time (about a factor of 2-4 on several platforms), but still makes startup speed a little faster: `gforth-fast --ss-number=0 --ss-states=1 --no-dynamic -e bye` consumes about 2.6ms user and 1.2ms system time.

If the script you want to execute contains a significant amount of code, it may be profitable to compile it into the image to avoid the cost of compiling it at startup time.

3 Forth Tutorial

The difference of this chapter from the Introduction (see Chapter 4 [Introduction], page 43) is that this tutorial is more fast-paced, should be used while sitting in front of a computer, and covers much more material, but does not explain how the Forth system works.

This tutorial can be used with any Standard-compliant Forth; any Gforth-specific features are marked as such and you can skip them if you work with another Forth. This tutorial does not explain all features of Forth, just enough to get you started and give you some ideas about the facilities available in Forth. Read the rest of the manual when you are through this.

The intended way to use this tutorial is that you work through it while sitting in front of the console, take a look at the examples and predict what they will do, then try them out; if the outcome is not as expected, find out why (e.g., by trying out variations of the example), so you understand what's going on. There are also some assignments that you should solve.

This tutorial assumes that you have programmed before and know what, e.g., a loop is.

3.1 Starting Gforth

You can start Gforth by typing its name:

```
gforth
```

That puts you into interactive mode; you can leave Gforth by typing `bye`. While in Gforth, you can edit the command line and access the command line history with cursor keys, similar to bash.

3.2 Syntax

A *word* is a sequence of arbitrary characters (except white space). Words are separated by white space. E.g., each of the following lines contains exactly one word:

```
word
!@#$%^&*()
1234567890
5!a
```

A frequent beginner's error is to leave out necessary white space, resulting in an error like '`Undefined word`'; so if you see such an error, check if you have put spaces wherever necessary.

```
." hello, world" \ correct
."hello, world"  \ gives an "Undefined word" error
```

Gforth and most other Forth systems ignore differences in case (they are case-insensitive), i.e., '`word`' is the same as '`Word`'. If your system is case-sensitive, you may have to type all the examples given here in upper case.

3.3 Crash Course

Forth does not prevent you from shooting yourself in the foot. Let's try a few ways to crash Gforth:

```
0 0 !
here execute
' catch >body 20 erase abort
' (quit1) >body 20 erase
```

The last two examples are guaranteed to destroy important parts of Gforth (and most other systems), so you better leave Gforth afterwards (if it has not finished by itself). On some systems you may have to kill gforth from outside (e.g., in Unix with `kill`).

You will find out later what these lines do and then you will get an idea why they produce crashes.

Now that you know how to produce crashes (and that there's not much to them), let's learn how to produce meaningful programs.

3.4 Stack

The most obvious feature of Forth is the stack. When you type in a number, it is pushed on the stack. You can display the contents of the stack with `.s`.

```
1 2 .s
3 .s
```

`.s` displays the top-of-stack to the right, i.e., the numbers appear in `.s` output as they appeared in the input.

You can print the top element of the stack with `.`.

```
1 2 3 . . .
```

In general, words consume their stack arguments (`.s` is an exception).

Assignment: What does the stack contain after `5 6 7 .`?

3.5 Arithmetics

The words `+`, `-`, `*`, `/`, and `mod` always operate on the top two stack items:

```
2 2 .s
+ .s
.
2 1 - .
7 3 mod .
```

The operands of `-`, `/`, and `mod` are in the same order as in the corresponding infix expression (this is generally the case in Forth).

Parentheses are superfluous (and not available), because the order of the words unambiguously determines the order of evaluation and the operands:

```
3 4 + 5 * .
3 4 5 * + .
```

Assignment: What are the infix expressions corresponding to the Forth code above? Write `6-7*8+9` in Forth notation[1].

To change the sign, use `negate`:

```
2 negate .
```

Assignment: Convert -(-3)*4-5 to Forth.

`/mod` performs both `/` and `mod`.

```
7 3 /mod . .
```

Reference: Section 6.5 [Arithmetic], page 60.

3.6 Stack Manipulation

Stack manipulation words rearrange the data on the stack.

```
1 .s drop .s
1 .s dup .s drop drop .s
1 2 .s over .s drop drop drop
1 2 .s swap .s drop drop
1 2 3 .s rot .s drop drop drop
```

These are the most important stack manipulation words. There are also variants that manipulate twice as many stack items:

```
1 2 3 4 .s 2swap .s 2drop 2drop
```

Two more stack manipulation words are:

```
1 2 .s nip .s drop
1 2 .s tuck .s 2drop drop
```

Assignment: Replace `nip` and `tuck` with combinations of other stack manipulation words.

Given:	How do you get:
1 2 3	3 2 1
1 2 3	1 2 3 2
1 2 3	1 2 3 3
1 2 3	1 3 3
1 2 3	2 1 3
1 2 3 4	4 3 2 1
1 2 3	1 2 3 1 2 3
1 2 3 4	1 2 3 4 1 2
1 2 3	
1 2 3	1 2 3 4
1 2 3	1 3

```
5 dup * .
```

Assignment: Write 17^3 and 17^4 in Forth, without writing `17` more than once. Write a piece of Forth code that expects two numbers on the stack (*a* and *b*, with *b* on top) and computes `(a-b)(a+1)`.

Reference: Section 6.7 [Stack Manipulation], page 72.

[1] This notation is also known as Postfix or RPN (Reverse Polish Notation).

3.7 Using files for Forth code

While working at the Forth command line is convenient for one-line examples and short one-off code, you probably want to store your source code in files for convenient editing and persistence. You can use your favourite editor (Gforth includes Emacs support, see Chapter 11 [Emacs and Gforth], page 327) to create *file.fs* and use

```
s" file.fs" included
```

to load it into your Forth system. The file name extension I use for Forth files is '`.fs`'.

You can easily start Gforth with some files loaded like this:

```
gforth file1.fs file2.fs
```

If an error occurs during loading these files, Gforth terminates, whereas an error during `INCLUDED` within Gforth usually gives you a Gforth command line. Starting the Forth system every time gives you a clean start every time, without interference from the results of earlier tries.

I often put all the tests in a file, then load the code and run the tests with

```
gforth code.fs tests.fs -e bye
```

(often by performing this command with *C-x C-e* in Emacs). The `-e bye` ensures that Gforth terminates afterwards so that I can restart this command without ado.

The advantage of this approach is that the tests can be repeated easily every time the program is changed, making it easy to catch bugs introduced by the change.

Reference: Section 6.22.1 [Forth source files], page 196.

3.8 Comments

```
\ That's a comment; it ends at the end of the line
( Another comment; it ends here: )  .s
```

`\` and `(` are ordinary Forth words and therefore have to be separated with white space from the following text.

```
\This gives an "Undefined word" error
```

The first `)` ends a comment started with `(`, so you cannot nest `(`-comments; and you cannot comment out text containing a `)` with `( ... )`[2].

I use `\`-comments for descriptive text and for commenting out code of one or more line; I use `(`-comments for describing the stack effect, the stack contents, or for commenting out sub-line pieces of code.

The Emacs mode `gforth.el` (see Chapter 11 [Emacs and Gforth], page 327) supports these uses by commenting out a region with *C-x *, uncommenting a region with *C-u C-x *, and filling a `\`-commented region with *M-q*.

Reference: Section 6.3 [Comments], page 59.

[2] therefore it's a good idea to avoid `)` in word names.

3.9 Colon Definitions

are similar to procedures and functions in other programming languages.

```
: squared ( n -- n^2 )
   dup * ;
5 squared .
7 squared .
```

`:` starts the colon definition; its name is `squared`. The following comment describes its stack effect. The words `dup *` are not executed, but compiled into the definition. `;` ends the colon definition.

The newly-defined word can be used like any other word, including using it in other definitions:

```
: cubed ( n -- n^3 )
   dup squared * ;
-5 cubed .
: fourth-power ( n -- n^4 )
   squared squared ;
3 fourth-power .
```

Assignment: Write colon definitions for `nip`, `tuck`, `negate`, and `/mod` in terms of other Forth words, and check if they work (hint: test your tests on the originals first). Don't let the '`redefined`'-Messages spook you, they are just warnings.

Reference: Section 6.11.5 [Colon Definitions], page 123.

3.10 Decompilation

You can decompile colon definitions with `see`:

```
see squared
see cubed
```

In Gforth `see` shows you a reconstruction of the source code from the executable code. Informations that were present in the source, but not in the executable code, are lost (e.g., comments).

You can also decompile the predefined words:

```
see .
see +
```

3.11 Stack-Effect Comments

By convention the comment after the name of a definition describes the stack effect: The part in front of the '`--`' describes the state of the stack before the execution of the definition, i.e., the parameters that are passed into the colon definition; the part behind the '`--`' is the state of the stack after the execution of the definition, i.e., the results of the definition. The stack comment only shows the top stack items that the definition accesses and/or changes.

You should put a correct stack effect on every definition, even if it is just `( -- )`. You should also add some descriptive comment to more complicated words (I usually do this in the lines following `:`). If you don't do this, your code becomes unreadable (because you have to work through every definition before you can understand any).

Assignment: The stack effect of `swap` can be written like this: `x1 x2 -- x2 x1`. Describe the stack effect of `-`, `drop`, `dup`, `over`, `rot`, `nip`, and `tuck`. Hint: When you are done, you can compare your stack effects to those in this manual (see [Word Index], page 381).

Sometimes programmers put comments at various places in colon definitions that describe the contents of the stack at that place (stack comments); i.e., they are like the first part of a stack-effect comment. E.g.,

```
: cubed ( n -- n^3 )
   dup squared  ( n n^2 ) * ;
```

In this case the stack comment is pretty superfluous, because the word is simple enough. If you think it would be a good idea to add such a comment to increase readability, you should also consider factoring the word into several simpler words (see Section 3.13 [Factoring], page 20), which typically eliminates the need for the stack comment; however, if you decide not to refactor it, then having such a comment is better than not having it.

The names of the stack items in stack-effect and stack comments in the standard, in this manual, and in many programs specify the type through a type prefix, similar to Fortran and Hungarian notation. The most frequent prefixes are:

`n` signed integer

`u` unsigned integer

`c` character

`f` Boolean flags, i.e. `false` or `true`.

`a-addr,a-` Cell-aligned address

`c-addr,c-` Char-aligned address (note that a Char may have two bytes in Windows NT)

`xt` Execution token, same size as Cell

`w,x` Cell, can contain an integer or an address. It usually takes 32, 64 or 16 bits (depending on your platform and Forth system). A cell is more commonly known as machine word, but the term *word* already means something different in Forth.

`d` signed double-cell integer

`ud` unsigned double-cell integer

`r` Float (on the FP stack)

You can find a more complete list in Section 6.1 [Notation], page 57.

Assignment: Write stack-effect comments for all definitions you have written up to now.

3.12 Types

In Forth the names of the operations are not overloaded; so similar operations on different types need different names; e.g., `+` adds integers, and you have to use `f+` to add floating-point numbers. The following prefixes are often used for related operations on different types:

`(none)`	signed integer
`u`	unsigned integer
`c`	character
`d`	signed double-cell integer
`ud, du`	unsigned double-cell integer
`2`	two cells (not-necessarily double-cell numbers)
`m, um`	mixed single-cell and double-cell operations
`f`	floating-point (note that in stack comments '`f`' represents flags, and '`r`' represents FP numbers; also, you need to include the exponent part in literal FP numbers, see Section 3.26 [Floating Point Tutorial], page 31).

If there are no differences between the signed and the unsigned variant (e.g., for `+`), there is only the prefix-less variant.

Forth does not perform type checking, neither at compile time, nor at run time. If you use the wrong operation, the data are interpreted incorrectly:

```
-1 u.
```

If you have only experience with type-checked languages until now, and have heard how important type-checking is, don't panic! In my experience (and that of other Forthers), type errors in Forth code are usually easy to find (once you get used to it), the increased vigilance of the programmer tends to catch some harder errors in addition to most type errors, and you never have to work around the type system, so in most situations the lack of type-checking seems to be a win (projects to add type checking to Forth have not caught on).

3.13 Factoring

If you try to write longer definitions, you will soon find it hard to keep track of the stack contents. Therefore, good Forth programmers tend to write only short definitions (e.g., three lines). The art of finding meaningful short definitions is known as factoring (as in factoring polynomials).

Well-factored programs offer additional advantages: smaller, more general words, are easier to test and debug and can be reused more and better than larger, specialized words.

So, if you run into difficulties with stack management, when writing code, try to define meaningful factors for the word, and define the word in terms of those. Even if a factor contains only two words, it is often helpful.

Good factoring is not easy, and it takes some practice to get the knack for it; but even experienced Forth programmers often don't find the right solution right away, but only when rewriting the program. So, if you don't come up with a good solution immediately, keep trying, don't despair.

3.14 Designing the stack effect

In other languages you can use an arbitrary order of parameters for a function; and since there is only one result, you don't have to deal with the order of results, either.

In Forth (and other stack-based languages, e.g., PostScript) the parameter and result order of a definition is important and should be designed well. The general guideline is to design the stack effect such that the word is simple to use in most cases, even if that complicates the implementation of the word. Some concrete rules are:

- Words consume all of their parameters (e.g., `.`).
- If there is a convention on the order of parameters (e.g., from mathematics or another programming language), stick with it (e.g., `-`).
- If one parameter usually requires only a short computation (e.g., it is a constant), pass it on the top of the stack. Conversely, parameters that usually require a long sequence of code to compute should be passed as the bottom (i.e., first) parameter. This makes the code easier to read, because the reader does not need to keep track of the bottom item through a long sequence of code (or, alternatively, through stack manipulations). E.g., `!` (store, see Section 6.8 [Memory], page 75) expects the address on top of the stack because it is usually simpler to compute than the stored value (often the address is just a variable).
- Similarly, results that are usually consumed quickly should be returned on the top of stack, whereas a result that is often used in long computations should be passed as bottom result. E.g., the file words like `open-file` return the error code on the top of stack, because it is usually consumed quickly by `throw`; moreover, the error code has to be checked before doing anything with the other results.

These rules are just general guidelines, don't lose sight of the overall goal to make the words easy to use. E.g., if the convention rule conflicts with the computation-length rule, you might decide in favour of the convention if the word will be used rarely, and in favour of the computation-length rule if the word will be used frequently (because with frequent use the cost of breaking the computation-length rule would be quite high, and frequent use makes it easier to remember an unconventional order).

3.15 Local Variables

You can define local variables (*locals*) in a colon definition:

```
: swap { a b -- b a }
  b a ;
1 2 swap .s 2drop
```

(If your Forth system does not support this syntax, include `compat/anslocal.fs` first).

In this example `{ a b -- b a }` is the locals definition; it takes two cells from the stack, puts the top of stack in `b` and the next stack element in `a`. `--` starts a comment ending with `}`. After the locals definition, using the name of the local will push its value on the stack. You can omit the comment part (`-- b a`):

```
: swap ( x1 x2 -- x2 x1 )
  { a b } b a ;
```

In Gforth you can have several locals definitions, anywhere in a colon definition; in contrast, in a standard program you can have only one locals definition per colon definition, and that locals definition must be outside any control structure.

With locals you can write slightly longer definitions without running into stack trouble. However, I recommend trying to write colon definitions without locals for exercise purposes to help you gain the essential factoring skills.

Assignment: Rewrite your definitions until now with locals

Reference: Section 6.26 [Locals], page 217.

3.16 Conditional execution

In Forth you can use control structures only inside colon definitions. An `if`-structure looks like this:

```
: abs ( n1 -- +n2 )
    dup 0 < if
        negate
    endif ;
5 abs .
-5 abs .
```

`if` takes a flag from the stack. If the flag is non-zero (true), the following code is performed, otherwise execution continues after the `endif` (or `else`). `<` compares the top two stack elements and produces a flag:

```
1 2 < .
2 1 < .
1 1 < .
```

Actually the standard name for `endif` is `then`. This tutorial presents the examples using `endif`, because this is often less confusing for people familiar with other programming languages where `then` has a different meaning. If your system does not have `endif`, define it with

```
: endif postpone then ; immediate
```

You can optionally use an `else`-part:

```
: min ( n1 n2 -- n )
  2dup < if
    drop
  else
    nip
  endif ;
2 3 min .
3 2 min .
```

Assignment: Write `min` without `else`-part (hint: what's the definition of `nip`?).

Reference: Section 6.10.1 [Selection], page 103.

3.17 Flags and Comparisons

In a false-flag all bits are clear (0 when interpreted as integer). In a canonical true-flag all bits are set (-1 as a twos-complement signed integer); in many contexts (e.g., `if`) any non-zero value is treated as true flag.

```
false .
true .
true hex u. decimal
```

Comparison words produce canonical flags:

```
1 1 = .
1 0= .
0 1 < .
0 0 < .
-1 1 u< . \ type error, u< interprets -1 as large unsigned number
-1 1 < .
```

Gforth supports all combinations of the prefixes `0 u d d0 du f f0` (or none) and the comparisons `= <> < > <= >=`. Only a part of these combinations are standard (for details see the standard, Section 6.5.7 [Numeric comparison], page 68, Section 6.5.9 [Floating Point], page 69, or [Word Index], page 381).

You can use `and or xor invert` as operations on canonical flags. Actually they are bitwise operations:

```
1 2 and .
1 2 or .
1 3 xor .
1 invert .
```

You can convert a zero/non-zero flag into a canonical flag with `0<>` (and complement it on the way with `0=`; indeed, it is more common to use `0=` instead of `invert` for canonical flags).

```
1 0= .
1 0<> .
```

While you can use `if` without `0<>` to test for zero/non-zero, you sometimes need to use `0<>` when combining zero/non-zero values with `and or xor` because of their bitwise nature. The simplest, least error-prone, and probably clearest way is to use `0<>` in all these cases, but in some cases you can use fewer `0<>`s. Here are some stack effects, where *fc* represents a canonical flag, and *fz* represents zero/non-zero (every *fc* also works as *fz*):

```
or  ( fz1 fz2 -- fz3 )
and ( fz1 fc  -- fz2 )
and ( fc  fz1 -- fz2 )
```

So, if you see code like this:

```
( n1 n2 ) 0<> and if
```

This tests whether n1 and n2 are non-zero and if yes, performs the code after `if`; it treats n1 as zero/non-zero and uses `0<>` to convert n2 into a canonical flag; the `and` then produces an fz, which is consumed by the `if`.

You can use the all-bits-set feature of canonical flags and the bitwise operation of the Boolean operations to avoid `if`s:

```
: foo ( n1 -- n2 )
  0= if
    14
  else
    0
  endif ;
0 foo .
1 foo .

: foo ( n1 -- n2 )
  0= 14 and ;
0 foo .
1 foo .
```

Assignment: Write `min` without `if`.

For reference, see Section 6.4 [Boolean Flags], page 60, Section 6.5.7 [Numeric comparison], page 68, and Section 6.5.6 [Bitwise operations], page 66.

3.18 General Loops

The endless loop is the most simple one:

```
: endless ( -- )
  0 begin
    dup . 1+
  again ;
endless
```

Terminate this loop by pressing *Ctrl-C* (in Gforth). `begin` does nothing at run-time, `again` jumps back to `begin`.

A loop with one exit at any place looks like this:

```
: log2 ( +n1 -- n2 )
\ logarithmus dualis of n1>0, rounded down to the next integer
  assert( dup 0> )
  2/ 0 begin
    over 0> while
      1+ swap 2/ swap
  repeat
  nip ;
7 log2 .
8 log2 .
```

At run-time `while` consumes a flag; if it is 0, execution continues behind the `repeat`; if the flag is non-zero, execution continues behind the `while`. `Repeat` jumps back to `begin`, just like `again`.

In Forth there are a number of combinations/abbreviations, like `1+`. However, `2/` is not one of them; it shifts its argument right by one bit (arithmetic shift right), and viewed

as division that always rounds towards negative infinity (floored division), like Gforth's `/` (since Gforth 0.7), but unlike `/` in many other Forth systems.

```
-5 2 / . \ -2 or -3
-5 2/ .  \ -3
```

`assert(` is no standard word, but you can get it on systems other than Gforth by including `compat/assert.fs`. You can see what it does by trying

```
0 log2 .
```

Here's a loop with an exit at the end:

```
: log2 ( +n1 -- n2 )
\ logarithmus dualis of n1>0, rounded down to the next integer
  assert( dup 0 > )
  -1 begin
    1+ swap 2/ swap
    over 0 <=
  until
  nip ;
```

`Until` consumes a flag; if it is zero, execution continues at the `begin`, otherwise after the `until`.

Assignment: Write a definition for computing the greatest common divisor.

Reference: Section 6.10.2 [General Loops], page 104.

3.19 Counted loops

```
: ^ ( n1 u -- n )
\ n = the uth power of n1
  1 swap 0 u+do
    over *
  loop
  nip ;
3 2 ^ .
4 3 ^ .
```

`U+do` (from `compat/loops.fs`, if your Forth system doesn't have it) takes two numbers of the stack `( u3 u4 -- )`, and then performs the code between `u+do` and `loop` for `u3-u4` times (or not at all, if `u3-u4<0`).

You can see the stack effect design rules at work in the stack effect of the loop start words: Since the start value of the loop is more frequently constant than the end value, the start value is passed on the top-of-stack.

You can access the counter of a counted loop with `i`:

```
: fac ( u -- u! )
  1 swap 1+ 1 u+do
    i *
  loop ;
5 fac .
7 fac .
```

There is also `+do`, which expects signed numbers (important for deciding whether to enter the loop).

Assignment: Write a definition for computing the nth Fibonacci number.

You can also use increments other than 1:

```
: up2 ( n1 n2 -- )
  +do
    i .
  2 +loop ;
10 0 up2

: down2 ( n1 n2 -- )
  -do
    i .
  2 -loop ;
0 10 down2
```

Reference: Section 6.10.3 [Counted Loops], page 104.

3.20 Recursion

Usually the name of a definition is not visible in the definition; but earlier definitions are usually visible:

```
1 0 / . \ "Floating-point unidentified fault" in Gforth on some platforms
: / ( n1 n2 -- n )
  dup 0= if
    -10 throw \ report division by zero
  endif
  /           \ old version
;
1 0 /
```

For recursive definitions you can use `recursive` (non-standard) or `recurse`:

```
: fac1 ( n -- n! ) recursive
 dup 0> if
   dup 1- fac1 *
 else
   drop 1
 endif ;
7 fac1 .

: fac2 ( n -- n! )
 dup 0> if
   dup 1- recurse *
 else
   drop 1
 endif ;
8 fac2 .
```

Assignment: Write a recursive definition for computing the nth Fibonacci number.

Reference (including indirect recursion): See Section 6.10.7 [Calls and returns], page 113.

3.21 Leaving definitions or loops

`EXIT` exits the current definition right away. For every counted loop that is left in this way, an `UNLOOP` has to be performed before the `EXIT`:

```
: ...
 ... u+do
   ... if
     ... unloop exit
   endif
   ...
 loop
 ... ;
```

`LEAVE` leaves the innermost counted loop right away:

```
: ...
 ... u+do
   ... if
     ... leave
   endif
   ...
 loop
 ... ;
```

Reference: Section 6.10.7 [Calls and returns], page 113, Section 6.10.3 [Counted Loops], page 104.

3.22 Return Stack

In addition to the data stack Forth also has a second stack, the return stack; most Forth systems store the return addresses of procedure calls there (thus its name). Programmers can also use this stack:

```
: foo ( n1 n2 -- )
 .s
 >r .s
 r@ .
 >r .s
 r@ .
 r> .
 r@ .
 r> . ;
1 2 foo
```

`>r` takes an element from the data stack and pushes it onto the return stack; conversely, `r>` moves an element from the return to the data stack; `r@` pushes a copy of the top of the return stack on the data stack.

Forth programmers usually use the return stack for storing data temporarily, if using the data stack alone would be too complex, and factoring and locals are not an option:

```
: 2swap ( x1 x2 x3 x4 -- x3 x4 x1 x2 )
 rot >r rot r> ;
```

The return address of the definition and the loop control parameters of counted loops usually reside on the return stack, so you have to take all items, that you have pushed on the return stack in a colon definition or counted loop, from the return stack before the definition or loop ends. You cannot access items that you pushed on the return stack outside some definition or loop within the definition of loop.

If you miscount the return stack items, this usually ends in a crash:

```
: crash ( n -- )
  >r ;
5 crash
```

You cannot mix using locals and using the return stack (according to the standard; Gforth has no problem). However, they solve the same problems, so this shouldn't be an issue.

Assignment: Can you rewrite any of the definitions you wrote until now in a better way using the return stack?

Reference: Section 6.7.3 [Return stack], page 73.

3.23 Memory

You can create a global variable `v` with

```
variable v ( -- addr )
```

`v` pushes the address of a cell in memory on the stack. This cell was reserved by `variable`. You can use `!` (store) to store values from the stack into this cell and `@` (fetch) to load the value from memory onto the stack:

```
v .
5 v ! .s
v @ .
```

You can see a raw dump of memory with `dump`:

```
v 1 cells .s dump
```

`Cells ( n1 -- n2 )` gives you the number of bytes (or, more generally, address units (aus)) that `n1 cells` occupy. You can also reserve more memory:

```
create v2 20 cells allot
v2 20 cells dump
```

creates a variable-like word `v2` and reserves 20 uninitialized cells; the address pushed by `v2` points to the start of these 20 cells (see Section 6.11.1 [CREATE], page 119). You can use address arithmetic to access these cells:

```
3 v2 5 cells + !
v2 20 cells dump
```

You can reserve and initialize memory with `,`:

```
create v3
```

```
  5 , 4 , 3 , 2 , 1 ,
v3 @ .
v3 cell+ @ .
v3 2 cells + @ .
v3 5 cells dump
```

Assignment: Write a definition `vsum ( addr u -- n )` that computes the sum of `u` cells, with the first of these cells at `addr`, the next one at `addr cell+` etc.

The difference between `variable` and `create` is that `variable` allots a cell, and that you cannot allot additional memory to a variable in Standard Forth.

You can also reserve memory without creating a new word:

```
here 10 cells allot .
here .
```

The first `here` pushes the start address of the memory area, the second `here` the address after the dictionary area. You should store the start address somewhere, or you will have a hard time finding the memory area again.

`Allot` manages dictionary memory. The dictionary memory contains the system's data structures for words etc. on Gforth and most other Forth systems. It is managed like a stack: You can free the memory that you have just `allot`ed with

```
-10 cells allot
here .
```

Note that you cannot do this if you have created a new word in the meantime (because then your `allot`ed memory is no longer on the top of the dictionary "stack").

Alternatively, you can use `allocate` and `free` which allow freeing memory in any order:

```
10 cells allocate throw .s
20 cells allocate throw .s
swap
free throw
free throw
```

The `throw`s deal with errors (e.g., out of memory).

And there is also a garbage collector (`https://www.complang.tuwien.ac.at/forth/garbage-collection.zip`), which eliminates the need to `free` memory explicitly.

Reference: Section 6.8 [Memory], page 75.

3.24 Characters and Strings

On the stack characters take up a cell, like numbers. In memory they have their own size (one 8-bit byte on most systems), and therefore require their own words for memory access:

```
create v4
  104 c, 97 c, 108 c, 108 c, 111 c,
v4 4 chars + c@ .
v4 5 chars dump
```

The preferred representation of strings on the stack is `addr u-count`, where `addr` is the address of the first character and `u-count` is the number of characters in the string.

```
v4 5 type
```

You get a string constant with

```
s" hello, world" .s
type
```

Make sure you have a space between `s"` and the string; `s"` is a normal Forth word and must be delimited with white space (try what happens when you remove the space).

However, this interpretive use of `s"` is quite restricted: the string exists only until the next call of `s"` (some Forth systems keep more than one of these strings, but usually they still have a limited lifetime).

```
s" hello," s" world" .s
type
type
```

You can also use `s"` in a definition, and the resulting strings then live forever (well, for as long as the definition):

```
: foo s" hello," s" world" ;
foo .s
type
type
```

Assignment: `Emit ( c -- )` types `c` as character (not a number). Implement `type ( addr u -- )`.

Reference: Section 6.8.8 [Memory Blocks], page 87.

3.25 Alignment

On many processors cells have to be aligned in memory, if you want to access them with `@` and `!` (and even if the processor does not require alignment, access to aligned cells is faster).

`Create` aligns `here` (i.e., the place where the next allocation will occur, and that the `create`d word points to). Likewise, the memory produced by `allocate` starts at an aligned address. Adding a number of `cells` to an aligned address produces another aligned address.

However, address arithmetic involving `char+` and `chars` can create an address that is not cell-aligned. `Aligned ( addr -- a-addr )` produces the next aligned address:

```
v3 char+ aligned .s @ .
v3 char+ .s @ .
```

Similarly, `align` advances `here` to the next aligned address:

```
create v5 97 c,
here .
align here .
1000 ,
```

Note that you should use aligned addresses even if your processor does not require them, if you want your program to be portable.

Reference: Section 6.8.7 [Address arithmetic], page 84.

3.26 Floating Point

Floating-point (FP) numbers and arithmetic in Forth works mostly as one might expect, but there are a few things worth noting:

The first point is not specific to Forth, but so important and yet not universally known that I mention it here: FP numbers are not reals. Many properties (e.g., arithmetic laws) that reals have and that one expects of all kinds of numbers do not hold for FP numbers. If you want to use FP computations, you should learn about their problems and how to avoid them; a good starting point is *David Goldberg, What Every Computer Scientist Should Know About Floating-Point Arithmetic (*`https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html`*), ACM Computing Surveys 23(1):5–48, March 1991.*

In Forth source code literal FP numbers need an exponent, e.g., `1e0`; this can also be written shorter as `1e`, longer as `+1.0e+0`, and many variations in between. The reason for this is that, for historical reasons, Forth interprets a decimal point alone (e.g., `1.`) as indicating a double-cell integer. Examples:

```
2e 2e f+ f.
```

Another requirement for literal FP numbers is that the current base is decimal; with a hex base `1e` is interpreted as an integer.

Forth has a separate stack for FP numbers in conformance with Forth-2012. One advantage of this model is that cells are not in the way when accessing FP values, and vice versa. Forth has a set of words for manipulating the FP stack: `fdup fswap fdrop fover frot` and (non-standard) `fnip ftuck fpick`.

FP arithmetic words are prefixed with `F`. There is the usual set `f+ f- f* f/ f** fnegate` as well as a number of words for other functions, e.g., `fsqrt fsin fln fmin`. One word that you might expect is `f=`; but `f=` is non-standard, because FP computation results are usually inaccurate, so exact comparison is usually a mistake, and one should use approximate comparison. Unfortunately, `f~`, the standard word for that purpose, is not well designed, so Gforth provides `f~abs` and `f~rel` as well.

And of course there are words for accessing FP numbers in memory (`f@ f!`), and for address arithmetic (`floats float+ faligned`). There are also variants of these words with an `sf` and `df` prefix for accessing IEEE format single-precision and double-precision numbers in memory; their main purpose is for accessing external FP data (e.g., that has been read from or will be written to a file).

Here is an example of a dot-product word and its use:

```
: v* ( f_addr1 nstride1 f_addr2 nstride2 ucount -- r )
  >r swap 2swap swap 0e r> 0 ?DO
    dup f@ over + 2swap dup f@ f* f+ over + 2swap
  LOOP
  2drop 2drop ;

create v 1.23e f, 4.56e f, 7.89e f,

v 1 floats  v 1 floats  3  v* f.
```

Assignment: Write a program to solve a quadratic equation. Then read *Henry G. Baker, You Could Learn a Lot from a Quadratic*

(https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.111.4448&rep=rep1&type=pdf), *ACM SIGPLAN Notices, 33(1):30–39, January 1998*, and see if you can improve your program. Finally, find a test case where the original and the improved version produce different results.

Reference: Section 6.5.9 [Floating Point], page 69; Section 6.7.2 [Floating point stack], page 73; Section 6.20 [Number conversion], page 187; Section 6.8.5 [Memory Access], page 81; Section 6.8.7 [Address arithmetic], page 84.

3.27 Files

This section gives a short introduction into how to use files inside Forth. It's broken up into five easy steps:

1. Open an ASCII text file for input
2. Open a file for output
3. Read input file until string matches (or some other condition is met)
4. Write some lines from input (modified or not) to output
5. Close the files.

Reference: Section 6.22.2 [General files], page 197.

3.27.1 Open file for input

```
s" foo.in"  r/o open-file throw Value fd-in
```

3.27.2 Create file for output

```
s" foo.out" w/o create-file throw Value fd-out
```

The available file modes are `r/o` for read-only access, `r/w` for read-write access, and `w/o` for write-only access. You could open both files with `r/w`, too, if you like. All file words return error codes; for most applications, it's best to pass the error codes with `throw` to the outer error handler.

If you want words for opening and assigning, define them as follows:

```
0 Value fd-in
0 Value fd-out
: open-input ( addr u -- )  r/o open-file throw to fd-in ;
: open-output ( addr u -- )  w/o create-file throw to fd-out ;
```

Usage example:

```
s" foo.in" open-input
s" foo.out" open-output
```

3.27.3 Scan file for a particular line

```
256 Constant max-line
Create line-buffer  max-line 2 + allot

: scan-file ( addr u -- )
  begin
      line-buffer max-line fd-in read-line throw
```

```
      while
            >r 2dup line-buffer r> compare 0=
         until
      else
         drop
      then
      2drop ;
```

`read-line ( addr u1 fd -- u2 flag ior )` reads up to u1 bytes into the buffer at addr, and returns the number of bytes read, a flag that is false when the end of file is reached, and an error code.

`compare ( addr1 u1 addr2 u2 -- n )` compares two strings and returns zero if both strings are equal. It returns a positive number if the first string is lexically greater, a negative if the second string is lexically greater.

We haven't seen this loop here; it has two exits. Since the `while` exits with the number of bytes read on the stack, we have to clean up that separately; that's after the `else`.

Usage example:

```
s" The text I search is here" scan-file
```

3.27.4 Copy input to output

```
: copy-file ( -- )
  begin
      line-buffer max-line fd-in read-line throw
  while
      line-buffer swap fd-out write-line throw
  repeat
  drop ;
```

3.27.5 Close files

```
fd-in close-file throw
fd-out close-file throw
```

Likewise, you can put that into definitions, too:

```
: close-input ( -- )  fd-in close-file throw ;
: close-output ( -- )  fd-out close-file throw ;
```

Assignment: How could you modify `copy-file` so that it copies until a second line is matched? Can you write a program that extracts a section of a text file, given the line that starts and the line that terminates that section?

3.28 Interpretation and Compilation Semantics and Immediacy

When a word is compiled, it behaves differently from being interpreted. E.g., consider `+`:

```
1 2 + .
: foo + ;
```

These two behaviours are known as compilation and interpretation semantics. For normal words (e.g., `+`), the compilation semantics is to append the interpretation semantics to

the currently defined word (`foo` in the example above). I.e., when `foo` is executed later, the interpretation semantics of `+` (i.e., adding two numbers) will be performed.

However, there are words with non-default compilation semantics, e.g., the control-flow words like `if`. You can use `immediate` to change the compilation semantics of the last defined word to be equal to the interpretation semantics:

```
: [FOO] ( -- )
 5 . ; immediate

[FOO]
: bar ( -- )
  [FOO] ;
bar
see bar
```

Two conventions to mark words with non-default compilation semantics are names with brackets (more frequently used) and to write them all in upper case (less frequently used).

For some words, such as `if`, using their interpretation semantics is usually a mistake, so we mark them as `compile-only`, and you get a warning when you interpret them.

```
: flip ( -- )
 6 . ; compile-only \ but not immediate
flip

: flop ( -- )
 flip ;
flop
```

In this example, first the interpretation semantics of `flip` is used (and you get a warning); the second use of `flip` uses the compilation semantics (and you get no warning). You can also see in this example that compile-only is a property that is evaluated at text interpretation time, not at run-time.

The text interpreter has two states: in interpret state, it performs the interpretation semantics of words it encounters; in compile state, it performs the compilation semantics of these words.

Among other things, `:` switches into compile state, and `;` switches back to interpret state. They contain the factors `]` (switch to compile state) and `[` (switch to interpret state), that do nothing but switch the state.

```
: xxx ( -- )
  [ 5 . ]
;

xxx
see xxx
```

These brackets are also the source of the naming convention mentioned above.

Reference: Section 6.14 [Interpretation and Compilation Semantics], page 150.

3.29 Execution Tokens

`' word` gives you the execution token (XT) of a word. The XT is a cell representing the interpretation semantics of a word. You can execute these semantics with `execute`:

```
' + .s
1 2 rot execute .
```

The XT is similar to a function pointer in C. However, parameter passing through the stack makes it a little more flexible:

```
: map-array ( ... addr u xt -- ... )
\ executes xt ( ... x -- ... ) for every element of the array starting
\ at addr and containing u elements
  { xt }
  cells over + swap ?do
    i @ xt execute
  1 cells +loop ;

create a 3 , 4 , 2 , -1 , 4 ,
a 5 ' . map-array .s
0 a 5 ' + map-array .
s" max-n" environment? drop .s
a 5 ' min map-array .
```

You can use map-array with the XTs of words that consume one element more than they produce. In theory you can also use it with other XTs, but the stack effect then depends on the size of the array, which is hard to understand.

Since XTs are cell-sized, you can store them in memory and manipulate them on the stack like other cells. You can also compile the XT into a word with `compile,`:

```
: foo1 ( n1 n2 -- n )
   [ ' + compile, ] ;
see foo1
```

This is non-standard, because `compile,` has no interpretation semantics in the standard, but it works in good Forth systems. For the bad ones, use

```
: [compile,] compile, ; immediate

: foo1 ( n1 n2 -- n )
   [ ' + ] [compile,] ;
see foo1
```

`'` is a word with default compilation semantics; it parses the next word when its interpretation semantics are executed, not during compilation:

```
: foo ( -- xt )
  ' ;
see foo
: bar ( ... "word" -- ... )
  ' execute ;
see bar
1 2 bar + .
```

You often want to parse a word during compilation and compile its XT so it will be pushed on the stack at run-time. `[']` does this:

```
: xt-+ ( -- xt )
  ['] + ;
see xt-+
1 2 xt-+ execute .
```

Many programmers tend to see `'` and the word it parses as one unit, and expect it to behave like `[']` when compiled, and are confused by the actual behaviour. If you are, just remember that the Forth system just takes `'` as one unit and has no idea that it is a parsing word (attempts to convenience programmers in this issue have usually resulted in even worse pitfalls, see '`State`'-smartness—Why it is evil and How to Exorcise it (`https://www.complang.tuwien.ac.at/papers/ertl98.ps.gz`)).

Note that the state of the interpreter does not come into play when creating and executing XTs. I.e., even when you execute `'` in compile state, it still gives you the interpretation semantics. And whatever that state is, `execute` performs the semantics represented by the XT (i.e., for XTs produced with `'` the interpretation semantics).

Reference: Section 6.15 [Tokens for Words], page 156.

3.30 Exceptions

`throw ( n -- )` causes an exception unless n is zero.

```
100 throw .s
0 throw .s
```

`catch ( ... xt -- ... n )` behaves similar to `execute`, but it catches exceptions and pushes the number of the exception on the stack (or 0, if the xt executed without exception). If there was an exception, the stacks have the same depth as when entering `catch`:

```
.s
3 0 ' / catch .s
3 2 ' / catch .s
```

Assignment: Try the same with `execute` instead of `catch`.

`Throw` always jumps to the dynamically next enclosing `catch`, even if it has to leave several call levels to achieve this:

```
: foo 100 throw ;
: foo1 foo ." after foo" ;
: bar ['] foo1 catch ;
bar .
```

It is often important to restore a value upon leaving a definition, even if the definition is left through an exception. You can ensure this like this:

```
: ...
   save-x
   ['] word-changing-x catch ( ... n )
   restore-x
   ( ... n ) throw ;
```

However, this is still not safe against, e.g., the user pressing *`Ctrl-C`* when execution is between the `catch` and `restore-x`.

Gforth provides an alternative exception handling syntax that is safe against such cases: `try ... restore ... endtry`. If the code between `try` and `endtry` has an exception, the stack depths are restored, the exception number is pushed on the stack, and the execution continues right after `restore`.

The safer equivalent to the restoration code above is

```
: ...
  save-x
  try
    word-changing-x 0
  restore
    restore-x
  endtry
  throw ;
```

Reference: Section 6.10.8 [Exception Handling], page 114.

3.31 Defining Words

`:`, `create`, and `variable` are definition words: They define other words. `Constant` is another definition word:

```
5 constant foo
foo .
```

You can also use the prefixes `2` (double-cell) and `f` (floating point) with `variable` and `constant`.

You can also define your own defining words. E.g.:

```
: variable ( "name" -- )
  create 0 , ;
```

You can also define defining words that create words that do something other than just producing their address:

```
: constant ( n "name" -- )
  create ,
does> ( -- n )
  ( addr ) @ ;

5 constant foo
foo .
```

The definition of `constant` above ends at the `does>`; i.e., `does>` replaces `;`, but it also does something else: It changes the last defined word such that it pushes the address of the body of the word and then performs the code after the `does>` whenever it is called.

In the example above, `constant` uses `,` to store 5 into the body of `foo`. When `foo` executes, it pushes the address of the body onto the stack, then (in the code after the `does>`) fetches the 5 from there.

The stack comment near the `does>` reflects the stack effect of the defined word, not the stack effect of the code after the `does>` (the difference is that the code expects the address of the body that the stack comment does not show).

You can use these definition words to do factoring in cases that involve (other) definition words. E.g., a field offset is always added to an address. Instead of defining

```
2 cells constant offset-field1
```

and using this like

```
( addr ) offset-field1 +
```

you can define a definition word

```
: simple-field ( n "name" -- )
  create ,
does> ( n1 -- n1+n )
  ( addr ) @ + ;
```

Definition and use of field offsets now look like this:

```
2 cells simple-field field1
create mystruct 4 cells allot
mystruct .s field1 .s drop
```

If you want to do something with the word without performing the code after the `does>`, you can access the body of a `create`d word with `>body ( xt -- addr )`:

```
: value ( n "name" -- )
  create ,
does> ( -- n1 )
  @ ;
: to ( n "name" -- )
  ' >body ! ;

5 value foo
foo .
7 to foo
foo .
```

Assignment: Define `defer ( "name" -- )`, which creates a word that stores an XT (at the start the XT of `abort`), and upon execution `execute`s the XT. Define `is ( xt "name" -- )` that stores `xt` into `name`, a word defined with `defer`. Indirect recursion is one application of `defer`.

Reference: Section 6.11.10 [User-defined Defining Words], page 125.

3.32 Arrays and Records

Forth has no standard words for defining arrays, but you can build them yourself based on address arithmetic. You can also define words for defining arrays and records (see Section 3.31 [Defining Words], page 37).

One of the first projects a Forth newcomer sets out upon when learning about defining words is an array defining word (possibly for n-dimensional arrays). Go ahead and do it, I did it, too; you will learn something from it. However, don't be disappointed when you later learn that you have little use for these words (inappropriate use would be even worse). I have not found a set of useful array words yet; the needs are just too diverse, and named, global arrays (the result of naive use of defining words) are often not flexible enough (e.g.,

consider how to pass them as parameters). Another such project is a set of words to help dealing with strings.

On the other hand, there is a useful set of record words, and it has been defined in `compat/struct.fs`; these words are predefined in Gforth. They are explained in depth elsewhere in this manual (see see Section 6.12 [Structures], page 141). The `simple-field` example above is simplified variant of fields in this package.

3.33 POSTPONE

You can compile the compilation semantics (instead of compiling the interpretation semantics) of a word with `POSTPONE`:

```
: MY-+ ( Compilation: -- ; Run-time of compiled code: n1 n2 -- n )
 POSTPONE + ; immediate
: foo ( n1 n2 -- n )
 MY-+ ;
1 2 foo .
see foo
```

During the definition of `foo` the text interpreter performs the compilation semantics of `MY-+`, which performs the compilation semantics of `+`, i.e., it compiles `+` into `foo`.

This example also displays separate stack comments for the compilation semantics and for the stack effect of the compiled code. For words with default compilation semantics these stack effects are usually not displayed; the stack effect of the compilation semantics is always `( -- )` for these words, the stack effect for the compiled code is the stack effect of the interpretation semantics.

Note that the state of the interpreter does not come into play when performing the compilation semantics in this way. You can also perform it interpretively, e.g.:

```
: foo2 ( n1 n2 -- n )
 [ MY-+ ] ;
1 2 foo .
see foo
```

However, there are some broken Forth systems where this does not always work, and therefore this practice was been declared non-standard in 1999.

Here is another example for using `POSTPONE`:

```
: MY-- ( Compilation: -- ; Run-time of compiled code: n1 n2 -- n )
 POSTPONE negate POSTPONE + ; immediate compile-only
: bar ( n1 n2 -- n )
  MY-- ;
2 1 bar .
see bar
```

You can define `ENDIF` (which you can use instead of `THEN`) in this way:

```
: ENDIF ( Compilation: orig -- )
  POSTPONE then ; immediate
```

Assignment: Write `MY-2DUP` that has compilation semantics equivalent to `2dup`, but compiles `over over`.

3.34 Literal

You cannot POSTPONE numbers:

```
: [FOO] POSTPONE 500 ; immediate
```

Instead, you can use LITERAL (compilation: n --; run-time: -- n):

```
: [FOO] ( compilation: --; run-time: -- n )
  500 POSTPONE literal ; immediate

: flip [FOO] ;
flip .
see flip
```

LITERAL consumes a number at compile-time (when it's compilation semantics are executed) and pushes it at run-time (when the code it compiled is executed). A frequent use of LITERAL is to compile a number computed at compile time into the current word:

```
: bar ( -- n )
  [ 2 2 + ] literal ;
see bar
```

Assignment: Write]L which allows writing the example above as : bar (-- n) [2 2 +]L ;

3.35 Advanced macros

Reconsider map-array from Section 3.29 [Execution Tokens], page 35. It frequently performs execute, a relatively expensive operation in some Forth implementations. You can use compile, and POSTPONE to eliminate these executes and produce a word that contains the word to be performed directly:

```
: compile-map-array ( compilation: xt -- ; run-time: ... addr u -- ... )
\ at run-time, execute xt ( ... x -- ... ) for each element of the
\ array beginning at addr and containing u elements
  { xt }
  POSTPONE cells POSTPONE over POSTPONE + POSTPONE swap POSTPONE ?do
    POSTPONE i POSTPONE @ xt compile,
  1 cells POSTPONE literal POSTPONE +loop ;

: sum-array ( addr u -- n )
 0 rot rot [ ' + compile-map-array ] ;
see sum-array
a 5 sum-array .
```

You can use the full power of Forth for generating the code; here's an example where the code is generated in a loop:

```
: compile-vmul-step ( compilation: n --; run-time: n1 addr1 -- n2 addr2 )
\ n2=n1+(addr1)*n, addr2=addr1+cell
  POSTPONE tuck POSTPONE @
  POSTPONE literal POSTPONE * POSTPONE +
  POSTPONE swap POSTPONE cell+ ;
```

```
: compile-vmul ( compilation: addr1 u -- ; run-time: addr2 -- n )
\ n=v1*v2 (inner product), where the v_i are represented as addr_i u
  0 postpone literal postpone swap
  [ ' compile-vmul-step compile-map-array ]
  postpone drop ;
see compile-vmul

: a-vmul ( addr -- n )
\ n=a*v, where v is a vector that's as long as a and starts at addr
 [ a 5 compile-vmul ] ;
see a-vmul
a a-vmul .
```

This example uses `compile-map-array` to show off, but you could also use `map-array` instead (try it now!).

You can use this technique for efficient multiplication of large matrices. In matrix multiplication, you multiply every row of one matrix with every column of the other matrix. You can generate the code for one row once, and use it for every column. The only downside of this technique is that it is cumbersome to recover the memory consumed by the generated code when you are done (and in more complicated cases it is not possible portably).

3.36 Compilation Tokens

This section is Gforth-specific. You can skip it.

`' word compile,` compiles the interpretation semantics. For words with default compilation semantics this is the same as performing the compilation semantics. To represent the compilation semantics of other words (e.g., words like `if` that have no interpretation semantics), Gforth has the concept of a compilation token (CT, consisting of two cells), and words `comp'` and `[comp']`. You can perform the compilation semantics represented by a CT with `execute`:

```
: foo2 ( n1 n2 -- n )
   [ comp' + execute ] ;
see foo
```

You can compile the compilation semantics represented by a CT with `postpone,`:

```
: foo3 ( -- )
  [ comp' + postpone, ] ;
see foo3
```

`[ comp' word postpone, ]` is equivalent to `POSTPONE word`. `comp'` is particularly useful for words that have no interpretation semantics:

```
' if
comp' if .s 2drop
```

Reference: Section 6.15 [Tokens for Words], page 156.

3.37 Wordlists and Search Order

The dictionary is not just a memory area that allows you to allocate memory with `allot`, it also contains the Forth words, arranged in several wordlists. When searching for a word

in a wordlist, conceptually you start searching at the youngest and proceed towards older words (in reality most systems nowadays use hash-tables); i.e., if you define a word with the same name as an older word, the new word shadows the older word.

Which wordlists are searched in which order is determined by the search order. You can display the search order with `order`. It displays first the search order, starting with the wordlist searched first, then it displays the wordlist that will contain newly defined words.

You can create a new, empty wordlist with `wordlist ( -- wid )`:

```
wordlist constant mywords
```

`Set-current ( wid -- )` sets the wordlist that will contain newly defined words (the *current* wordlist):

```
mywords set-current
order
```

Gforth does not display a name for the wordlist in `mywords` because this wordlist was created anonymously with `wordlist`.

You can get the current wordlist with `get-current ( -- wid)`. If you want to put something into a specific wordlist without overall effect on the current wordlist, this typically looks like this:

```
get-current mywords set-current ( wid )
create someword
( wid ) set-current
```

You can write the search order with `set-order ( wid1 .. widn n -- )` and read it with `get-order ( -- wid1 .. widn n )`. The first searched wordlist is topmost.

```
get-order mywords swap 1+ set-order
order
```

Yes, the order of wordlists in the output of `order` is reversed from stack comments and the output of `.s` and thus unintuitive.

> **Assignment:** Define `>order ( wid -- )` which adds `wid` as first searched wordlist to the search order. Define `previous ( -- )`, which removes the first searched wordlist from the search order. Experiment with boundary conditions (you will see some crashes or situations that are hard or impossible to leave).

The search order is a powerful foundation for providing features similar to Modula-2 modules and C++ namespaces. However, trying to modularize programs in this way has disadvantages for debugging and reuse/factoring that overcome the advantages in my experience (I don't do huge projects, though). These disadvantages are not so clear in other languages/programming environments, because these languages are not so strong in debugging and reuse.

Reference: Section 6.19 [Word Lists], page 182.

4 An Introduction to Standard Forth

The difference of this chapter from the Tutorial (see Chapter 3 [Tutorial], page 14) is that it is slower-paced in its examples, but uses them to dive deep into explaining Forth internals (not covered by the Tutorial). Apart from that, this chapter covers far less material. It is suitable for reading without using a computer.

The primary purpose of this manual is to document Gforth. However, since Forth is not a widely-known language and there is a lack of up-to-date teaching material, it seems worthwhile to provide some introductory material. For other sources of Forth-related information, see Appendix C [Forth-related information], page 362.

The examples in this section should work on any Standard Forth; the output shown was produced using Gforth. Each example attempts to reproduce the exact output that Gforth produces. If you try out the examples (and you should), what you should type is shown *`like this`* and Gforth's response is shown `like this`. The single exception is that, where the example shows `RET` it means that you should press the "carriage return" key. Unfortunately, some output formats for this manual cannot show the difference between *`this`* and `this` which will make trying out the examples harder (but not impossible).

Forth is an unusual language. It provides an interactive development environment which includes both an interpreter and compiler. Forth programming style encourages you to break a problem down into many small fragments (*factoring*), and then to develop and test each fragment interactively. Forth advocates assert that breaking the edit-compile-test cycle used by conventional programming languages can lead to great productivity improvements.

4.1 Introducing the Text Interpreter

When you invoke the Forth image, you will see a startup banner printed and nothing else (if you have Gforth installed on your system, try invoking it now, by typing *`gforth`*`RET`). Forth is now running its command line interpreter, which is called the *Text Interpreter* (also known as the *Outer Interpreter*). (You will learn a lot about the text interpreter as you read through this chapter, for more detail see Section 6.17 [The Text Interpreter], page 166).

Although it's not obvious, Forth is actually waiting for your input. Type a number and press the `RET` key:

```
45RET  ok
```

Rather than give you a prompt to invite you to input something, the text interpreter prints a status message *after* it has processed a line of input. The status message in this case (" `ok`" followed by carriage-return) indicates that the text interpreter was able to process all of your input successfully. Now type something illegal:

```
qwer341RET
*the terminal*:2: Undefined word
>>>qwer341<<<
Backtrace:
$2A95B42A20 throw
$2A95B57FB8 no.extensions
```

The exact text, other than the "Undefined word" may differ slightly on your system, but the effect is the same; when the text interpreter detects an error, it discards any remaining

text on a line, resets certain internal state and prints an error message. For a detailed description of error messages see Chapter 7 [Error messages], page 307.

The text interpreter waits for you to press carriage-return, and then processes your input line. Starting at the beginning of the line, it breaks the line into groups of characters separated by spaces. For each group of characters in turn, it makes two attempts to do something:

- It tries to treat it as a command. It does this by searching a *name dictionary*. If the group of characters matches an entry in the name dictionary, the name dictionary provides the text interpreter with information that allows the text interpreter to perform some actions. In Forth jargon, we say that the group of characters names a *word*, that the dictionary search returns an *execution token (xt)* corresponding to the *definition* of the word, and that the text interpreter executes the xt. Often, the terms *word* and *definition* are used interchangeably.
- If the text interpreter fails to find a match in the name dictionary, it tries to treat the group of characters as a number in the current number base (when you start up Forth, the current number base is base 10). If the group of characters legitimately represents a number, the text interpreter pushes the number onto a stack (we'll learn more about that in the next section).

If the text interpreter is unable to do either of these things with any group of characters, it discards the group of characters and the rest of the line, then prints an error message. If the text interpreter reaches the end of the line without error, it prints the status message " `ok`" followed by carriage-return.

This is the simplest command we can give to the text interpreter:

```
RET  ok
```

The text interpreter did everything we asked it to do (nothing) without an error, so it said that everything is " `ok`". Try a slightly longer command:

```
12 dup fred dupRET
*the terminal*:3: Undefined word
12 dup >>>fred<<< dup
Backtrace:
$2A95B42A20 throw
$2A95B57FB8 no.extensions
```

When you press the carriage-return key, the text interpreter starts to work its way along the line:

- When it gets to the space after the `2`, it takes the group of characters `12` and looks them up in the name dictionary[1]. There is no match for this group of characters in the name dictionary, so it tries to treat them as a number. It is able to do this successfully, so it puts the number, 12, "on the stack" (whatever that means).
- The text interpreter resumes scanning the line and gets the next group of characters, `dup`. It looks it up in the name dictionary and (you'll have to take my word for this) finds it, and executes the word `dup` (whatever that means).

[1] We can't tell if it found them or not, but assume for now that it did not

- Once again, the text interpreter resumes scanning the line and gets the group of characters `fred`. It looks them up in the name dictionary, but can't find them. It tries to treat them as a number, but they don't represent any legal number.

At this point, the text interpreter gives up and prints an error message. The error message shows exactly how far the text interpreter got in processing the line. In particular, it shows that the text interpreter made no attempt to do anything with the final character group, `dup`, even though we have good reason to believe that the text interpreter would have no problem looking that word up and executing it a second time.

4.2 Stacks, postfix notation and parameter passing

In procedural programming languages (like C and Pascal), the building-block of programs is the *function* or *procedure*. These functions or procedures are called with *explicit parameters*. For example, in C we might write:

```
total = total + new_volume(length,height,depth);
```

where new_volume is a function-call to another piece of code, and total, length, height and depth are all variables. length, height and depth are parameters to the function-call.

In Forth, the equivalent of the function or procedure is the *definition* and parameters are implicitly passed between definitions using a shared stack that is visible to the programmer. Although Forth does support variables, the existence of the stack means that they are used far less often than in most other programming languages. When the text interpreter encounters a number, it will place (*push*) it on the stack. There are several stacks (the actual number is implementation-dependent ...) and the particular stack used for any operation is implied unambiguously by the operation being performed. The stack used for all integer operations is called the *data stack* and, since this is the stack used most commonly, references to "the data stack" are often abbreviated to "the stack".

The stacks have a last-in, first-out (LIFO) organisation. If you type:

```
1 2 3RET  ok
```

Then this instructs the text interpreter to placed three numbers on the (data) stack. An analogy for the behaviour of the stack is to take a pack of playing cards and deal out the ace (1), 2 and 3 into a pile on the table. The 3 was the last card onto the pile ("last-in") and if you take a card off the pile then, unless you're prepared to fiddle a bit, the card that you take off will be the 3 ("first-out"). The number that will be first-out of the stack is called the *top of stack*, which is often abbreviated to *TOS*.

To understand how parameters are passed in Forth, consider the behaviour of the definition `+` (pronounced "plus"). You will not be surprised to learn that this definition performs addition. More precisely, it adds two numbers together and produces a result. Where does it get the two numbers from? It takes the top two numbers off the stack. Where does it place the result? On the stack. You can act out the behaviour of `+` with your playing cards like this:

- Pick up two cards from the stack on the table
- Stare at them intently and ask yourself "what *is* the sum of these two numbers"
- Decide that the answer is 5
- Shuffle the two cards back into the pack and find a 5

- Put a 5 on the remaining ace that's on the table.

If you don't have a pack of cards handy but you do have Forth running, you can use the definition `.s` to show the current state of the stack, without affecting the stack. Type:

```
clearstacks 1 2 3RET ok
.sRET <3> 1 2 3  ok
```

The text interpreter looks up the word `clearstacks` and executes it; it tidies up the stacks (data and floating point stack) and removes any entries that may have been left on them by earlier examples. The text interpreter pushes each of the three numbers in turn onto the stack. Finally, the text interpreter looks up the word `.s` and executes it. The effect of executing `.s` is to print the "<3>" (the total number of items on the stack) followed by a list of all the items on the stack; the item on the far right-hand side is the TOS.

You can now type:

```
+ .sRET <2> 1 5  ok
```

which is correct; there are now 2 items on the stack and the result of the addition is 5.

If you're playing with cards, try doing a second addition: pick up the two cards, work out that their sum is 6, shuffle them into the pack, look for a 6 and place that on the table. You now have just one item on the stack. What happens if you try to do a third addition? Pick up the first card, pick up the second card – ah! There is no second card. This is called a *stack underflow* and constitutes an error. If you try to do the same thing with Forth it often reports an error (probably a Stack Underflow or an Invalid Memory Address error).

The opposite situation to a stack underflow is a *stack overflow*, which simply accepts that there is a finite amount of storage space reserved for the stack. To stretch the playing card analogy, if you had enough packs of cards and you piled the cards up on the table, you would eventually be unable to add another card; you'd hit the ceiling. Gforth allows you to set the maximum size of the stacks. In general, the only time that you will get a stack overflow is because a definition has a bug in it and is generating data on the stack uncontrollably.

There's one final use for the playing card analogy. If you model your stack using a pack of playing cards, the maximum number of items on your stack will be 52 (I assume you didn't use the Joker). The maximum *value* of any item on the stack is 13 (the King). In fact, the only possible numbers are positive integer numbers 1 through 13; you can't have (for example) 0 or 27 or 3.52 or -2. If you change the way you think about some of the cards, you can accommodate different numbers. For example, you could think of the Jack as representing 0, the Queen as representing -1 and the King as representing -2. Your *range* remains unchanged (you can still only represent a total of 13 numbers) but the numbers that you can represent are -2 through 10.

In that analogy, the limit was the amount of information that a single stack entry could hold, and Forth has a similar limit. In Forth, the size of a stack entry is called a *cell*. The actual size of a cell is implementation dependent and affects the maximum value that a stack entry can hold. A Standard Forth provides a cell size of at least 16-bits, and most desktop systems use a cell size of 32-bits.

Forth does not do any type checking for you, so you are free to manipulate and combine stack items in any way you wish. A convenient way of treating stack items is as 2's

complement signed integers, and that is what Standard words like + do. Therefore you can type:

```
-5 12 + .sRET <1> 7  ok
```

If you use numbers and definitions like + in order to turn Forth into a great big pocket calculator, you will realise that it's rather different from a normal calculator. Rather than typing 2 + 3 = you had to type 2 3 + (ignore the fact that you had to use .s to see the result). The terminology used to describe this difference is to say that your calculator uses *Infix Notation* (parameters and operators are mixed) whilst Forth uses *Postfix Notation* (parameters and operators are separate), also called *Reverse Polish Notation*.

Whilst postfix notation might look confusing to begin with, it has several important advantages:

- it is unambiguous
- it is more concise
- it fits naturally with a stack-based system

To examine these claims in more detail, consider these sums:

```
6 + 5 * 4 =
4 * 5 + 6 =
```

If you're just learning maths or your maths is very rusty, you will probably come up with the answer 44 for the first and 26 for the second. If you are a bit of a whizz at maths you will remember the *convention* that multiplication takes precedence over addition, and you'd come up with the answer 26 both times. To explain the answer 26 to someone who got the answer 44, you'd probably rewrite the first sum like this:

```
6 + (5 * 4) =
```

If what you really wanted was to perform the addition before the multiplication, you would have to use parentheses to force it.

If you did the first two sums on a pocket calculator you would probably get the right answers, unless you were very cautious and entered them using these keystroke sequences:

6 + 5 = * 4 = 4 * 5 = + 6 =

Postfix notation is unambiguous because the order that the operators are applied is always explicit; that also means that parentheses are never required. The operators are *active* (the act of quoting the operator makes the operation occur) which removes the need for "=".

The sum 6 + 5 * 4 can be written (in postfix notation) in two equivalent ways:

```
6 5 4 * +      or:
5 4 * 6 +
```

An important thing that you should notice about this notation is that the *order* of the numbers does not change; if you want to subtract 2 from 10 you type `10 2 -`.

The reason that Forth uses postfix notation is very simple to explain: it makes the implementation extremely simple, and it follows naturally from using the stack as a mechanism for passing parameters. Another way of thinking about this is to realise that all Forth definitions are *active*; they execute as they are encountered by the text interpreter. The result of this is that the syntax of Forth is trivially simple.

4.3 Your first Forth definition

Until now, the examples we've seen have been trivial; we've just been using Forth as a bigger-than-pocket calculator. Also, each calculation we've shown has been a "one-off" – to repeat it we'd need to type it in again[2] In this section we'll see how to add new words to Forth's vocabulary.

The easiest way to create a new word is to use a *colon definition*. We'll define a few and try them out before worrying too much about how they work. Try typing in these examples; be careful to copy the spaces accurately:

```
: add-two 2 + . ;
: greet ." Hello and welcome" ;
: demo 5 add-two ;
```

Now try them out:

```
greetRET Hello and welcome  ok
greet greetRET Hello and welcomeHello and welcome  ok
4 add-twoRET 6  ok
demoRET 7  ok
9 greet demo add-twoRET Hello and welcome7 11  ok
```

The first new thing that we've introduced here is the pair of words `:` and `;`. These are used to start and terminate a new definition, respectively. The first word after the `:` is the name for the new definition.

As you can see from the examples, a definition is built up of words that have already been defined; Forth makes no distinction between definitions that existed when you started the system up, and those that you define yourself.

The examples also introduce the words `.` (dot), `."` (dot-quote) and `dup` (dewp). Dot takes the value from the top of the stack and displays it. It's like `.s` except that it only displays the top item of the stack and it is destructive; after it has executed, the number is no longer on the stack. There is always one space printed after the number, and no spaces before it. Dot-quote defines a string (a sequence of characters) that will be printed when the word is executed. The string can contain any printable characters except `"`. A `"` has a special function; it is not a Forth word but it acts as a delimiter (the way that delimiters work is described in the next section). Finally, `dup` duplicates the value at the top of the stack. Try typing `5 dup .s` to see what it does.

We already know that the text interpreter searches through the dictionary to locate names. If you've followed the examples earlier, you will already have a definition called `add-two`. Lets try modifying it by typing in a new definition:

```
: add-two dup . ." + 2 = " 2 + . ;RET redefined add-two  ok
```

Forth recognised that we were defining a word that already exists, and printed a message to warn us of that fact. Let's try out the new definition:

```
9 add-twoRET 9 + 2 = 11  ok
```

All that we've actually done here, though, is to create a new definition, with a particular name. The fact that there was already a definition with the same name did not make

[2] That's not quite true. If you press the up-arrow key on your keyboard you should be able to scroll back to any earlier command, edit it and re-enter it.

any difference to the way that the new definition was created (except that Forth printed a warning message). The old definition of add-two still exists (try `demo` again to see that this is true). Any new definition will use the new definition of `add-two`, but old definitions continue to use the version that already existed at the time that they were `compiled`.

Before you go on to the next section, try defining and redefining some words of your own.

4.4 How does that work?

Now we're going to take another look at the definition of `add-two` from the previous section. From our knowledge of the way that the text interpreter works, we would have expected this result when we tried to define `add-two`:

```
: add-two 2 + . ;RET
*the terminal*:4: Undefined word
: >>>add-two<<< 2 + . ;
```

The reason that this didn't happen is bound up in the way that `:` works. The word `:` does two special things. The first special thing that it does is to prevent the text interpreter from ever seeing the characters `add-two`. The text interpreter uses a variable called `>IN` (pronounced "to-in") to keep track of where it is in the input line. When it encounters the word `:` it behaves in exactly the same way as it does for any other word; it looks it up in the name dictionary, finds its xt and executes it. When `:` executes, it looks at the input buffer, finds the word `add-two` and advances the value of `>IN` to point past it. It then does some other stuff associated with creating the new definition (including creating an entry for `add-two` in the name dictionary). When the execution of `:` completes, control returns to the text interpreter, which is oblivious to the fact that it has been tricked into ignoring part of the input line.

Words like `:` – words that advance the value of `>IN` and so prevent the text interpreter from acting on the whole of the input line – are called *parsing words*.

The second special thing that `:` does is change the value of a variable called `state`, which affects the way that the text interpreter behaves. When Gforth starts up, `state` has the value 0, and the text interpreter is said to be *interpreting*. During a colon definition (started with `:`), `state` is set to -1 and the text interpreter is said to be *compiling*.

In this example, the text interpreter is compiling when it processes the string "`2 + . ;`". It still breaks the string down into character sequences in the same way. However, instead of pushing the number `2` onto the stack, it lays down (*compiles*) some magic into the definition of `add-two` that will make the number `2` get pushed onto the stack when `add-two` is *executed*. Similarly, the behaviours of `+` and `.` are also compiled into the definition.

Certain kinds of words do not get compiled. These so-called *immediate words* get executed (performed *now*) regardless of whether the text interpreter is interpreting or compiling. The word `;` is an immediate word. Rather than being compiled into the definition, it executes. Its effect is to terminate the current definition, which includes changing the value of `state` back to 0.

When you execute `add-two`, it has a *run-time effect* that is exactly the same as if you had typed `2 + . RET` outside of a definition.

In Forth, every word or number can be described in terms of two properties:

- Its *interpretation semantics* describe how it will behave when the text interpreter encounters it in *interpret* state. The interpretation semantics of a word are represented by its *execution token* (see Section 6.15.1 [Execution token], page 156).
- Its *compilation semantics* describe how it will behave when the text interpreter encounters it in *compile* state. The compilation semantics of a word are represented by its *compilation token* (see Section 6.15.3 [Compilation token], page 160).

Numbers are always treated in a fixed way:

- When the number is *interpreted*, its behaviour is to push the number onto the stack.
- When the number is *compiled*, a piece of code is appended to the current definition that pushes the number when it runs. (In other words, the compilation semantics of a number are to postpone its interpretation semantics until the run-time of the definition that it is being compiled into.)

Words don't always behave in such a regular way, but most have *default semantics* which means that they behave like this:

- The *interpretation semantics* of the word are to do something useful.
- The *compilation semantics* of the word are to append its *interpretation semantics* to the current definition (so that its run-time behaviour is to do something useful).

The actual behaviour of any particular word can be controlled by using the words `immediate` and `compile-only` when the word is defined. These words set flags in the name dictionary entry of the most recently defined word, and these flags are retrieved by the text interpreter when it finds the word in the name dictionary.

A word that is marked as *immediate* has compilation semantics that are identical to its interpretation semantics. In other words, it behaves like this:

- The *interpretation semantics* of the word are to do something useful.
- The *compilation semantics* of the word are to do something useful (and actually the same thing); i.e., it is executed during compilation.

Marking a word as *compile-only* means that the text interpreter produces a warning when encountering this word in interpretation state; ticking the word (with `'` or `[']` also produces a warning.

It is never necessary to use `compile-only` (and it is not even part of Standard Forth, though it is provided by many implementations) but it is good etiquette to apply it to a word that will not behave correctly (and might have unexpected side-effects) in interpret state. For example, it is only legal to use the conditional word `IF` within a definition. If you forget this and try to use it elsewhere, the fact that (in Gforth) it is marked as `compile-only` allows the text interpreter to generate a helpful warning.

This example shows the difference between an immediate and a non-immediate word:

```
: show-state state @ . ;
: show-state-now show-state ; immediate
: word1 show-state ;
: word2 show-state-now ;
```

The word `immediate` after the definition of `show-state-now` makes that word an immediate word. These definitions introduce a new word: `@` (pronounced "fetch"). This word

fetches the value of a variable, and leaves it on the stack. Therefore, the behaviour of `show-state` is to print a number that represents the current value of `state`.

When you execute `word1`, it prints the number 0, indicating that the system is interpreting. When the text interpreter compiled the definition of `word1`, it encountered `show-state` whose compilation semantics are to append its interpretation semantics to the current definition. When you execute `word1`, it performs the interpretation semantics of `show-state`. At the time that `word1` (and therefore `show-state`) is executed, the system is interpreting.

When you pressed `RET` after entering the definition of `word2`, you should have seen the number -1 printed, followed by " `ok`". When the text interpreter compiled the definition of `word2`, it encountered `show-state-now`, an immediate word, whose compilation semantics are therefore to perform its interpretation semantics. It is executed straight away (even before the text interpreter has moved on to process another group of characters; the `;` in this example). The effect of executing it is to display the value of `state` *at the time that the definition of* `word2` *is being defined*. Printing -1 demonstrates that the system is compiling at this time. If you execute `word2` it does nothing at all.

Before leaving the subject of immediate words, consider the behaviour of `."` in the definition of `greet`, in the previous section. This word is both a parsing word and an immediate word. Notice that there is a space between `."` and the start of the text `Hello and welcome`, but that there is no space between the last letter of `welcome` and the `"` character. The reason for this is that `."` is a Forth word; it must have a space after it so that the text interpreter can identify it. The `"` is not a Forth word; it is a *delimiter*. The examples earlier show that, when the string is displayed, there is neither a space before the `H` nor after the `e`. Since `."` is an immediate word, it executes at the time that `greet` is defined. When it executes, its behaviour is to search forward in the input line looking for the delimiter. When it finds the delimiter, it updates `>IN` to point past the delimiter. It also compiles some magic code into the definition of `greet`; the xt of a run-time routine that prints a text string. It compiles the string `Hello and welcome` into memory so that it is available to be printed later. When the text interpreter gains control, the next word it finds in the input stream is `;` and so it terminates the definition of `greet`.

4.5 Forth is written in Forth

When you start up a Forth compiler, a large number of definitions already exist. In Forth, you develop a new application using bottom-up programming techniques to create new definitions that are defined in terms of existing definitions. As you create each definition you can test and debug it interactively.

If you have tried out the examples in this section, you will probably have typed them in by hand; when you leave Gforth, your definitions will be lost. You can avoid this by using a text editor to enter Forth source code into a file, and then loading code from the file using `include` (see Section 6.22.1 [Forth source files], page 196). A Forth source file is processed by the text interpreter, just as though you had typed it in by hand[3].

Gforth also supports the traditional Forth alternative to using text files for program entry (see Section 6.23 [Blocks], page 202).

[3] Actually, there are some subtle differences – see Section 6.17 [The Text Interpreter], page 166.

In common with many, if not most, Forth compilers, most of Gforth is actually written in Forth. All of the `.fs` files in the installation directory[4] are Forth source files, which you can study to see examples of Forth programming.

Gforth maintains a history file that records every line that you type to the text interpreter. This file is preserved between sessions, and is used to provide a command-line recall facility. If you enter long definitions by hand, you can use a text editor to paste them out of the history file into a Forth source file for reuse at a later time (for more information see Section 2.4 [Command-line editing], page 10).

4.6 Review - elements of a Forth system

To summarise this chapter:

- Forth programs use *factoring* to break a problem down into small fragments called *words* or *definitions*.
- Forth program development is an interactive process.
- The main command loop that accepts input, and controls both interpretation and compilation, is called the *text interpreter* (also known as the *outer interpreter*).
- Forth has a very simple syntax, consisting of words and numbers separated by spaces or carriage-return characters. Any additional syntax is imposed by *parsing words*.
- Forth uses a stack to pass parameters between words. As a result, it uses postfix notation.
- To use a word that has previously been defined, the text interpreter searches for the word in the *name dictionary*.
- Words have *interpretation semantics* and *compilation semantics*.
- The text interpreter uses the value of `state` to select between the use of the *interpretation semantics* and the *compilation semantics* of a word that it encounters.
- The relationship between the *interpretation semantics* and *compilation semantics* for a word depends upon the way in which the word was defined (for example, whether it is an *immediate* word).
- Forth definitions can be implemented in Forth (called *high-level definitions*) or in some other way (usually a lower-level language and as a result often called *low-level definitions*, *code definitions* or *primitives*).
- Many Forth systems are implemented mainly in Forth.

4.7 Where To Go Next

Amazing as it may seem, if you have read (and understood) this far, you know almost all the fundamentals about the inner workings of a Forth system. You certainly know enough to be able to read and understand the rest of this manual and the Standard Forth document, to learn more about the facilities that Forth in general and Gforth in particular provide. Even scarier, you know almost enough to implement your own Forth system. However, that's not a good idea just yet... better to try writing some programs in Gforth.

[4] For example, `/usr/local/share/gforth...`

Forth has such a rich vocabulary that it can be hard to know where to start in learning it. This section suggests a few sets of words that are enough to write small but useful programs. Use the word index in this document to learn more about each word, then try it out and try to write small definitions using it. Start by experimenting with these words:

- Arithmetic: `+ - * / /MOD */ ABS INVERT`
- Comparison: `MIN MAX =`
- Logic: `AND OR XOR NOT`
- Stack manipulation: `DUP DROP SWAP OVER`
- Loops and decisions: `IF ELSE THEN ?DO I LOOP`
- Input/Output: `. ." EMIT CR KEY`
- Defining words: `: ; CREATE`
- Memory allocation words: `ALLOT ,`
- Tools: `SEE WORDS .S MARKER`

When you have mastered those, go on to:

- More defining words: `VARIABLE CONSTANT VALUE TO CREATE DOES>`
- Memory access: `@ !`

When you have mastered these, there's nothing for it but to read through the whole of this manual and find out what you've missed.

4.8 Exercises

TODO: provide a set of programming exercises linked into the stuff done already and into other sections of the manual. Provide solutions to all the exercises in a .fs file in the distribution.

5 Literals in source code

5.1 Integer and character literals

To push an integer number on the data stack, you write the number in source code, e.g., `123`. You can prefix the digits with `-` to indicate a negative number, e.g. `-123`. This works both inside colon definitions and outside. The number is interpreted according to the value in `base` (see Section 6.20.1 [Base and integer decimal point], page 187). The digits are `0` to `9` and `a` (decimal 10) to `z` (decimal 35), but only digits smaller than `base @` are recognized. The conversion is case-insensitive, so `A` and `a` are the same digit.

You can make the base explicit for the number by using a prefix:

- `#` – decimal
- `%` – binary
- `$` – hexadecimal
- `&` – decimal (non-standard)
- `0x` – hexadecimal, if base<33 (non-standard).

For combinations including base-prefix and sign, the standard order is to have the base-prefix first (e.g., `#-123`); Gforth supports both orders.

You can put a decimal point `.` at the end of a number (or, non-standardly, anywhere else except before a prefix) to get a double-cell integer (e.g., `#-123.` or `#-.123` (the same number)).

By default (`.-is-dcell?` pushes true), prefixless numbers with a decimal point (e.g., `-123.`) are also recognized as double-cell integers. This is confusing to users experienced in other programming language. To clear up the confusion early, Gforth warns of such usage; to avoid the warnings, the best approach is to always write double numbers with a base prefix (e.g., `#-123.`). This also works in the setting `0 to .-is-dcell?`.

Here are some examples, with the equivalent decimal number shown after in braces:

`$-41` (-65), `%1001101` (205), `%1001.0001` (145, a double-precision number), `#905` (905), `$abc` (2478), `$ABC` (2478).

You can get the numeric value of a (character) code point by surrounding the character with `'` (e.g., `'a'`). The trailing `'` is required by the standard, but you can leave it away in Gforth. Note that this also works for non-ASCII characters. For many uses, it is more useful to have the character as a string rather than as a cell; see below for the string syntax.

5.2 Floating-point number and complex literals

For floating-point numbers in Forth, you recognize them due to their exponent. I.e., by default `1.` is a double-cell integer, and `1e0` is a floating-point number; the latter can be (and usually is) shortened to `1e`. Both the significand (the part before the `e` or `E`) and the exponent may have signs (including `+`); the significand must contain at least one digit and may contain a decimal point, the exponent can be empty. Floating-point numbers always use decimal base for both significand and exponent, and are only recognized when the base is decimal. Examples are: `1e 1e0 1.e 1.e0 +1e+0` (which all represent the same number) `+12.E-4`.

With the setting `0 to .-is-dcell?`, Gforth (since 1.0) does not recognize prefixless numbers with a decimal point as double-cell integers, but recognizes them as FP numbers instead. Note that standard Forth systems (such as the default setting of Gforth) behave differently, so we recommend always using `e` to signify FP numbers.

A Gforth extension (since 1.0) is to write a floating-point number in scaled notation: It can optionally have a sign, then one or more digits, then use one of the mostly SI-defined scaling symbols (aka metric prefixes) or `%`, and then optionally more digits. Here's the full list of scaling symbols that Gforth accepts:

- `Q` `e30` quetta
- `R` `e27` ronna
- `Y` `e24` yotta
- `Z` `e21` zetta
- `X` `e18` exa (not `E`)
- `P` `e15` peta
- `T` `e12` tera
- `G` `e9` giga
- `M` `e6` mega
- `k` `e3` kilo
- `h` `e2` hecto
- `d` `e-1` deci
- `%` `e-2` percent (not `c`)
- `m` `e-3` milli
- `u` `e-6` micro (not μ)
- `n` `e-9` nano
- `p` `e-12` pico
- `f` `e-15` femto
- `a` `e-18` atto
- `z` `e-21` zepto
- `y` `e-24` yocto
- `r` `e-27` ronto
- `q` `e-30` quecto

Unlike most of the rest of Gforth, scaling symbols are treated case-sensitively. Using the scaled notation is equivalent to using a decimal point instead of the scaling symbol and appending the exponential notation at the end. Examples of scaled notation: `6k5` (6500e) `23%` (0.23e).

In Gforth (since 1.0) you can input a complex number with *`real`*`+`*`imaginary`*`i`, where both *`real`* and *`imaginary`* are strings that are recognized as floating-point numbers. E.g., `1e+2ei`. This pushes the values `1e` and `2e` on the floating-point stack, so one might just as well have written `1e 2e`, but `1e+2ei` makes the intent obvious.

5.3 String and environment variable literals

In Gforth (since 1.0) you can input a string by surrounding it with `"` (e.g. `"abc"`, `"a b"`). The result is the starting address and byte (=char) count of the string on the data stack.

You have to escape any `"` inside the string with `\` (e.g., `"double-quote->\"<-"`). In addition, this string syntax supports all the ways to write control characters that are supported by `s\"` (see Section 6.9.3 [String and character literals], page 91). A disadvantage of this string syntax is that it is non-standard; for standard programs, use `s\"` instead.

In Gforth (since 1.0) you can input an environment variable by surrounding its name with `${...}`, e.g., `${HOME}`; the result is a string descriptor on the data stack in the format described above. This is equivalent to `"HOME" getenv`, i.e., the environment variable is resolved at run-time.

5.4 Literals for tokens and addresses

Gforth (since 1.0) also recognizes the following literals:

You can input an execution token (xt) of a word by prefixing the name of the word with the backquote `` ` `` (e.g., `` `dup ``). An advantage over using `'` or `[']` is you do not need to switch between them when copying and pasting code from inside to outside a colon definition or vice versa. A disadvantage is that this syntax is non-standard.

You can input a name token (nt) of a word by prefixing the name of the word with ``` `` ``` (e.g., ``` ``dup ```). This syntax is also non-standard.

You can input a body address of a word by surrounding it with `<` and `>` (e.g., `<spaces>`). You can also input an address that is at a positive offset from the body address (typically an address in that body), by putting `+` and a number (see syntax above) between the word name and the closing `>` (e.g., `<spaces+$15>`, `<spaces+-3>`). You will get the body address plus the number. This feature exists to allow copying and pasting the output of `...` (see Section 6.30.6 [Examining data], page 263).

In addition, by default Gforth recognizes words with `rec-name` and `rec-scope`, and stores in or adds to value-flavoured words with `rec-to`, but these do not recognize literals, so they are discussed elsewhere (see Section 6.17.4.1 [Default recognizers], page 171).

5.5 Disambiguating recognizers

In some cases where two recognizers match the same string, you can specify in Gforth (since 1.0) which recognizer you want to use, with *`recognizer?string`*, where *`recognizer`* is the name of the recognizer without the `rec-` prefix, and *`string`* is the string you want to recognize. E.g., `float?1.` uses `rec-float` to recognize a string that would otherwise be recognized as a double-cell integer number (because `rec-number` is earlier in the recognizer sequence than `rec-float`).

6 Forth Words

6.1 Notation

The Forth words are described in this section in the glossary notation that has become a de-facto standard for Forth texts:

word *Stack effect* *wordset* *pronunciation*

Description

word The name of the word.

Stack effect

The stack effect is written in the notation `before -- after`, where *before* and *after* describe the top of stack entries before and after the execution of the word. The rest of the stack is not touched by the word. The top of stack is rightmost, i.e., a stack sequence is written as it is typed in.

Gforth has several stacks, in particular, the data stack, return stack and floating-point stack. However, it uses a unified stack effect notation, where one stack effect description describes all three stack effects, and the name of the item indicates which stack the item is on: floating-point stack items start with *r*. Return stack items are prefixed with *R:*, but are otherwise the same as data stack items. E.g., in the stack effect `( w1 w2 -- R:w1 R:w2 )` *w1* is a cell on the data stack, and *R:w1* is a cell on the return stack with the same value. So a unified stack effect

```
( r1 n1 R:n2 -- R:n3 n4 r2 )
```

is equivalent to the separated stack effect notation

```
( n1 -- n4 ) ( R: n2 -- n3 ) ( F: r1 -- r2 )
```

The name of a stack item describes the type and/or the function of the item. See below for a discussion of the types.

Words generally have different stack effects in different contexts. If only one stack effect is shown, it's the stack effect for the execution/interpretation semantics.[1] The stack effect of default compilation semantics is `( -- )` and is not shown.

The stack-effects of non-default compilation semantics are shown if they are other than `( -- )`. Such words usually also have a run-time semantics, and their stack effects are then shown as in this example

```
; ( compilation colon-sys -- ; run-time nest-sys -- )
```

Further stack effects, such as those of defined words, of passed xts, are shown in the description part of the glossary entry.

Also note that in code templates or examples there can be comments in parentheses that display the stack picture at this point; there is no `--` in these places, because there is no before-after situation.

[1] Gforth 1.0 does not make a difference between interpretation and execution semantics.

pronunciation
: How the word is pronounced.

wordset
: The wordset specifies if a word has been standardized (indicated by a capitalized wordset name), it is an environmental query string (indicated by "environment"), or if it is a Gforth-specific word (lower case).

  The Forth standard is divided into several word sets. In theory, a standard system need not support all of them, but in practice, serious systems on non-tiny machines support almost all standardized words (some systems require explicit loading of some word sets, however), so it does not increase portability in practice to be parsimonious in using word sets.

  For the Gforth-specific words, we have the following categories:

  `gforth`

  `gforth-<`*`version`*`>`
  : We intend to permanently support this word in Gforth and it has been available since Gforth <*version*> (possibly not as stable word at that time).

    You see `gforth` in the source code (e.g., when using `locate`), and `gforth-<`*`version`*`>` in the documentation (e.g., when using `help`). So if you want to know since which Gforth version a word is available, use `help` *`word`*.

  `library`
  : The word belongs to a library that is independent of Gforth, but is delivered with Gforth and documented in this manual. Gforth 1.0 includes libraries with the following wordset names: mini-oof mini-oof2 minos2 minos2-bidi objects oof regexp-cg regexp-pattern regexp-replace cilk

  `gforth-experimental`
  : This word is available in the present version and may turn into a stable word or may be removed in a future release of Gforth. Feedback welcome.

  `gforth-internal`
  : This word is an internal factor, not a supported word, and it may be removed in a future release of Gforth. If you see a word in the source code (e.g., with `locate`) without a wordset, that word is also an internal factor.

  `gforth-obsolete`
  : This word will be removed in a future release of Gforth.

Description
: A description of the behaviour of the word.

The type of a stack item is specified by the prefix of the name:

`f`
: Boolean flags, i.e. `false` or `true`.

`c`
: Char

`w`	
`x`	Cell, can contain an integer or an address
`n`	signed integer
`u`	unsigned integer
`d`	signed double-cell integer
`ud`	unsigned double-cell integer
`r`	Float (on the FP stack)
`addr`	Address without further information
`a-`	Cell-aligned address
`c-`	Char-aligned address, address used to point to a character or start of a string.
`f-`	Float-aligned address
`df-`	Address aligned for IEEE double precision float
`sf-`	Address aligned for IEEE single precision float
`xt`	Execution token, same size as Cell
`nt`	Name token, same size as Cell
`wid`	Word list ID, same size as Cell
`ior, wior`	I/O result code, cell-sized. In Gforth, you can `throw` iors.
`"`	String in the input stream (not on the stack), typically space-delimited.
`'`	String in the input stream, delimited by the last character before the closing `'`. E.g., `'ccc"'` indicates a string in the input stream that is terminated by `"`.

6.2 Case insensitivity

Gforth is case-insensitive for ASCII characters and case-sensitive for non-ASCII characters. I.e., you can invoke Standard words using upper, lower or mixed case.

For now, Standard Forth only *requires* implementations to recognise Standard words when they are typed entirely in upper case. You can use whatever case you like for words that you define, but in a Standard program you have to use the words in the same case that you defined them.

Gforth supports case sensitivity through `cs-wordlists` (case-sensitive wordlists, see Section 6.19 [Word Lists], page 182).

6.3 Comments

Forth supports two styles of comment; the *in-line* comment starting with `(` and ending with `)`, and the comment to the end of the line line starting with `\`. Don't forget the space after the starting word.

`(` (*compilation 'ccc<close-paren>' – ; run-time –*) core,file “paren”

Comment, usually till the next `):` parse and discard all subsequent characters in the parse area until `")"` is encountered. During interactive input, an end-of-line also acts as a comment terminator. For file input, it does not; if the end-of-file is encountered whilst parsing for the `")"` delimiter, Gforth will generate a warning.

`\` (*compilation 'ccc<newline>' – ; run-time –*) core-ext,block-ext "backslash"

Comment until the end of line: parse and discard all remaining characters in the parse area, except while `load`ing from a block: while `load`ing from a block, parse and discard all remaining characters in the 64-byte line.

`\G` (*compilation 'ccc<newline>' – ; run-time –*) gforth-0.2 "backslash-gee"

Equivalent to `\`. Used right below the start of a definition to describe the behaviour of a word. In Gforth's source code these comments are those that are then inserted in the documentation.

6.4 Boolean Flags

A Boolean flag is cell-sized. A cell with all bits clear represents the flag `false` and a flag with all bits set represents the flag `true`. Words that check a flag (for example, `IF`) will treat a cell that has *any* bit set as `true`.

`true` (*– f*) core-ext

`Constant` – *f* is a cell with all bits set.

`false` (*– f*) core-ext

`Constant` – *f* is a cell with all bits clear.

`on` (*a-addr –*) gforth-0.2

Set the (value of the) variable at *a-addr* to `true`.

`off` (*a-addr –*) gforth-0.2

Set the (value of the) variable at *a-addr* to `false`.

`select` (*u1 u2 f – u*) gforth-1.0 "select"

If *f* is false, *u* is *u2*, otherwise *u1*.

6.5 Arithmetic

Forth arithmetic is not checked, i.e., you will not hear about integer overflow on addition or multiplication, you may hear about division by zero if you are lucky. The operator is written after the operands, but the operands are still in the original order. I.e., the infix `2-1` corresponds to `2 1 -`.

6.5.1 Single precision

By default, numbers in Forth are single-precision integers that are one cell (a machine word, e.g., 64 bits on a 64-bit system) in size. They can be signed or unsigned, depending upon how you treat them. For the rules used by the text interpreter for recognising single-precision integers see Section 6.16.1 [Literals], page 160.

`+`, `1+`, `under+`, `-`, `1-`, `*` are defined for signed operands, but they also work for unsigned numbers. For division words see Section 6.5.4 [Integer division], page 62.

`+` (*n1 n2 – n*) core "plus"

`1+` (*n1 – n2*) core “one-plus”

`under+` (*n1 n2 n3 – n n2*) gforth-0.3 “under-plus”

add *n3* to *n1* (giving *n*)

`-` (*n1 n2 – n*) core “minus”

`1-` (*n1 – n2*) core “one-minus”

`*` (*n1 n2 – n*) core “star”

`negate` (*n1 – n2*) core “negate”

`abs` (*n – u*) core “abs”

`min` (*n1 n2 – n*) core “min”

`max` (*n1 n2 – n*) core “max”

`umin` (*u1 u2 – u*) gforth-0.5 “umin”

`umax` (*u1 u2 – u*) gforth-1.0 “umax”

6.5.2 Double precision

For the rules used by the text interpreter for recognising double-precision integers, see Section 6.16.1 [Literals], page 160.

A double precision number is represented by a cell pair, with the most significant cell at the top-of-stack (TOS). It is trivial to convert an unsigned single to a double: simply push a `0` onto the TOS. Numbers are represented by Gforth using 2's complement arithmetic, so converting a signed single to a (signed) double requires sign-extension across the most significant cell. This can be achieved using `s>d`. You cannot convert a number from single-cell to double-cell without knowing whether it represents an unsigned or a signed number. By contrast, in 2's complement arithmetic the conversion from double to single just `drop`s the most significant cell, and `d>s` just documents the intent.

`D+` and `d-` are defined for signed operands, but also work for unsigned numbers.

`s>d` (*n – d*) core “s-to-d”

`d>s` (*d – n*) double “d-to-s”

`d+` (*ud1 ud2 – ud*) double “d-plus”

`d-` (*d1 d2 – d*) double “d-minus”

`dnegate` (*d1 – d2*) double “d-negate”

`dabs` (*d – ud*) double “d-abs”

`dmin` (*d1 d2 – d*) double “d-min”

`dmax` (*d1 d2 – d*) double “d-max”

6.5.3 Mixed precision

`m+` (*d1 n – d2*) double “m-plus”

`m*` (*n1 n2 – d*) core “m-star”

`um*` (*u1 u2 – ud*) core “u-m-star”

6.5.4 Integer division

Below you find a considerable number of words for dealing with divisions. A major difference between them is in dealing with signed division: Do the words support signed division? Those with the `u` prefix do not.

Do signed division words round towards negative infinity (floored division, suffix `F`), or towards 0 (symmetric division, suffix `S`). The standard leaves the issue implementation-defined for most standard words (`/ mod /mod */ */mod m*/`). Gforth implements these words as floored (since Gforth 0.7), but there are systems that implement them as symmetric. There is only a difference between floored and symmetric division if the dividend and the divisor have different signs, and the dividend is not a multiple of the divisor. The following table illustrates the results:

```
                        floored          symmetric
dividend divisor remainder quotient remainder quotient
    10      7           3   1              3   1
   -10      7           4  -2             -3  -1
    10     -7          -4  -2              3  -1
   -10     -7          -3   1             -3   1
```

The common case where floored vs. symmetric makes a difference is when dividends n1 with varying sign are divided by the same positive divisor n2; in that case you usually want floored division, because then the remainder is always positive and does not change sign depending on the dividend; also, with floored division, the quotient always increases by 1 when n1 increases by n2, while with symmetric division there is no increase in the quotient for -n2<n1<n2 (the quotient is 0 in this range).

In any case, if you divide numbers where floored vs. symmetric makes a difference, you should think about which variant is the right one for you, and then use either the appropriately suffixed Gforth words, or the standard words `fm/mod` or `sm/rem`.

In the following, "remainder" (symmetric) has the same sign as the dividend or is 0, while "modulus" (floored) has the same sign as the divisor or is 0.

The following words perform single-by-single-cell division:

`/` (*n1 n2 – n*) core "slash"

n=n1/n2

`/s` (*n1 n2 – n*) gforth-1.0 "slash-s"

`/f` (*n1 n2 – n*) gforth-1.0 "slash-f"

`u/` (*u1 u2 – u*) gforth-1.0 "u-slash"

`mod` (*n1 n2 – n*) core

n is the modulus of n1/n2

`mods` (*n1 n2 – n*) gforth-1.0 "mod-s"

`modf` (*n1 n2 – n*) gforth-1.0 "modf"

`umod` (*u1 u2 – u*) gforth-1.0 "umod"

`/mod` (*n1 n2 – n3 n4*) core "slash-mod"

n1=n2*n4+n3; n3 is the modulus, n4 the quotient.

`/mods` (*n1 n2 – n3 n4*) gforth-1.0 "slash-mod-s"

n1=n2*n4+n3; n3 is the remainder, n4 the quotient

`/modf` (*n1 n2 – n3 n4*) gforth-1.0 “slash-mod-f”

n1=n2*n4+n3; n3 is the modulus, n4 the quotient

`u/mod` (*u1 u2 – u3 u4*) gforth-1.0 “u-slash-mod”

u1=u2*u4+u3; u3 is the modulus, u4 the quotient

The following words perform double-by-single-cell division with single-cell results; these words are roughly as fast as the words above on some architectures (e.g., AMD64), but much slower on others (e.g., an order of magnitude on various ARM A64 CPUs).

`fm/mod` (*d1 n1 – n2 n3*) core “f-m-slash-mod”

Floored division: *d1* = *n3***n1*+*n2*, *n1*>*n2*>=0 or 0>=*n2*>*n1*.

`sm/rem` (*d1 n1 – n2 n3*) core “s-m-slash-rem”

Symmetric division: *d1* = *n3***n1*+*n2*, sign(*n2*)=sign(*d1*) or 0.

`um/mod` (*ud u1 – u2 u3*) core “u-m-slash-mod”

ud=u3*u1+u2, 0<=u2<u1

`du/mod` (*d u – n u1*) gforth-1.0 “du-slash-mod”

d=n*u+u1, 0<=u1<u; PolyForth style mixed division

`*/` (*(n1 n2 n3 – n4*) core “star-slash”

n4=(n1*n2)/n3, with the intermediate result being double

`*/s` (*n1 n2 n3 – n4*) gforth-1.0 “star-slash-s”

n4=(n1*n2)/n3, with the intermediate result being double

`*/f` (*n1 n2 n3 – n4*) gforth-1.0 “star-slash-f”

n4=(n1*n2)/n3, with the intermediate result being double

`u*/` (*u1 u2 u3 – u4*) gforth-1.0 “u-star-slash”

u4=(u1*u2)/u3, with the intermediate result being double.

`*/mod` (*n1 n2 n3 – n4 n5*) core “star-slash-mod”

n1*n2=n3*n5+n4, with the intermediate result (n1*n2) being double; n4 is the modulus, n5 the quotient.

`*/mods` (*n1 n2 n3 – n4 n5*) gforth-1.0 “star-slash-mod-s”

n1*n2=n3*n5+n4, with the intermediate result (n1*n2) being double; n4 is the remainder, n5 the quotient

`*/modf` (*n1 n2 n3 – n4 n5*) gforth-1.0 “star-slash-mod-f”

n1*n2=n3*n5+n4, with the intermediate result (n1*n2) being double; n4 is the modulus, n5 the quotient

`u*/mod` (*u1 u2 u3 – u4 u5*) gforth-1.0 “u-star-slash-mod”

u1*u2=u3*u5+u4, with the intermediate result (u1*u2) being double.

The following words perform division with double-cell results; these words are much slower than the words above.

`ud/mod` (*ud1 u2 – urem udquot*) gforth-0.2 “ud-slash-mod”

divide unsigned double *ud1* by *u2*, resulting in a unsigned double quotient *udquot* and a single remainder *urem*.

`m*/` (*d1 n2 u3 – dquot*) double "m-star-slash"

dquot=(d1*n2)/u3, with the intermediate result being triple-precision. In Forth-2012 u3 is only allowed to be a positive signed number.

You can use the environmental query `floored` (see Section 6.21 [Environmental Queries], page 192) to learn whether `/ mod /mod */ */mod m*/` use floored or symmetric division on the system your program is being loaded on; alternatively, `-1 3 /` also produces -1 on floored and 0 on symmetric systems.

One other aspect of the integer division words is that most of them can overflow, and division by zero is mathematically undefined. What happens if you hit one of these conditions depends on the engine, the hardware, and the operating system: The engine `gforth` tries hard to throw the appropriate error -10 (Division by zero) or -11 (Result out of range), but on some platforms throws -55 (Floating-point unidentified fault). The engine `gforth-fast` may produce an inappropriate throw code (and error message), or may produce no error, just produce a bogus value. I.e., you should not bet on such conditions being thrown, but for quicker debugging `gforth` catches more and produces more accurate errors than `gforth-fast`.

6.5.5 Two-stage integer division

On most hardware, multiplication is significantly faster than division. So if you have to divide many numbers by the same divisor, it is usually faster to determine the reciprocal of the divisor once and multiply the numbers with the reciprocal. If you divide by a constant, Gforth performs this optimization automatically.

However, for cases where the divisor is not known during compilation, Gforth provides words that allow you to implement this optimization without to much fuss.

Let's start with an example: You want to divide all elements of an array of cells by the same number n. A straightforward way to implement this is:

```
: array/ ( addr u n -- )
  -rot cells bounds u+do
    i @ over / i !
  1 cells +loop
  drop ;
```

A possibly more efficient version looks like this:

```
: array/ ( addr u n -- )
  {: | reci[ staged/-size ] :}
  reci[ /f-stage1m
  cells bounds u+do
    i @ reci[ /f-stage2m i !
  1 cells +loop ;
```

This example first creates a local buffer `reci[` with size `staged/-size` for storing the reciprocal data. Then `/f-stage1m` computes the reciprocal of n and stores it in `reci[`. Finally, inside the loop `/f-stage2m` uses the data in `reci[` to compute the quotient.

There are some limitations: Only positive divisors are supported for `/f-stage1m`; for `u/-stage1m` you can use a divisor of 2 or higher. You get an error if you try to use an

unsupported divisor. You must initialize the reciprocal buffer for the floored second-stage words with `/f-stage1m` and for the unsigned second-stage words with `u/-stage1m`. You must not modify the reciprocal buffer between the first stage and the second stage; basically, don't treat it as a memory buffer, but as something that is only mutable by the first stage; the point of this rule is that future versions of Gforth will not consider aliasing of this buffer.

Measurements show that staged division is not always beneficial:

```
break  100 elem
even   speedup  core
  7      2.09   Skylake (Core i5-6600K)
  -      0.94   Rocket Lake (Xeon E-2388G)
 40      1.09   Golden Cove (Core i3-1315U P-core)
  -      0.85   Gracemont (Core i3-1315U E-core)
  6      1.68   Zen2 (Ryzen 9 3900X)
  -      0.56   Zen3 (Ryzen 7 5800X)
```

The words are:

`staged/-size` (– *u*) gforth-1.0 "staged-slash-size"

Size of buffer for `u/-stage1m` or `/f-stage1m`.

`/f-stage1m` (*n a-reci* –) gforth-1.0 "slash-f-stage1m"

Compute the reciprocal of *n* and store it in the buffer *a-reci* of size `staged/-size`. Throws an error if *n*<1.

`/f-stage2m` (*n1 a-reci* – *nquotient*) gforth-1.0 "slash-f-stage2m"

Nquotient is the result of dividing *n1* by the divisor represented by *a-reci*, which was computed by `/f-stage1m`.

`modf-stage2m` (*n1 a-reci* – *umodulus*) gforth-1.0 "mod-f-stage2m"

Umodulus is the remainder of dividing *n1* by the divisor represented by *a-reci*, which was computed by `/f-stage1m`.

`/modf-stage2m` (*n1 a-reci* – *umodulus nquotient*) gforth-1.0 "slash-mod-f-stage2m"

Nquotient is the quotient and *umodulus* is the remainder of dividing *n1* by the divisor represented by *a-reci*, which was computed by `/f-stage1m`.

`u/-stage1m` (*u a-reci* –) gforth-1.0 "u-slash-stage1m"

Compute the reciprocal of *u* and store it in the buffer *a-reci* of size `staged/-size`. Throws an error if *u*<2.

`u/-stage2m` (*u1 a-reci* – *uquotient*) gforth-1.0 "u-slash-stage2m"

Uquotient is the result of dividing *u1* by the divisor represented by *a-reci*, which was computed by `u/-stage1m`.

`umod-stage2m` (*u1 a-reci* – *umodulus*) gforth-1.0 "u-mod-stage2m"

Umodulus is the remainder of dividing *u1* by the divisor represented by *a-reci*, which was computed by `u/-stage1m`.

`u/mod-stage2m` (*u1 a-reci* – *umodulus uquotient*) gforth-1.0 "u-slash-mod-stage2m"

Uquotient is the quotient and *umodulus* is the remainder of dividing *u1* by the divisor represented by *a-reci*, which was computed by `u/-stage1m`.

Gforth currently does not support staged symmetrical division.

You can recover the divisor from (the address of) a reciprocal with `staged/-divisor @`:

`staged/-divisor` (*addr1 – addr2*) gforth-1.0 “staged-slash-divisor”

Addr1 is the address of a reciprocal, *addr2* is the address containing the divisor from which the reciprocal was computed.

This can be useful when looking at the decompiler output of Gforth: a division by a constant is often compiled to a literal containing the address of a reciprocal followed by a second-stage word.

The performance impact of using these words strongly depends on the architecture (does it have hardware division?) and the specific implementation (how fast is hardware division?), but just to give you an idea about the relative performance of these words, here are the cycles per iteration of a microbenchmark (which performs the mentioned word once per iteration) on two AMD64 implementations; the *norm* column shows the normal division word (e.g., `u/`), while the *stg2* column shows the corresponding stage2 word (e.g., `u/-stage2m`):

```
 Skylake              Zen2
norm stg2           norm stg2
41.3 15.8 u/        35.2 21.4 u/
39.8 19.7 umod      36.9 25.8 umod
44.0 25.3 u/mod     43.0 33.9 u/mod
48.7 16.9 /f        36.2 22.5 /f
47.9 20.5 modf      37.9 27.1 modf
53.0 24.6 /modf     45.8 35.4 /modf
    227.2 u/stage1      101.9 u/stage1
    159.8 /fstage1       97.7 /fstage1
```

6.5.6 Bitwise operations

`and` (*w1 w2 – w*) core “and”

`or` (*w1 w2 – w*) core “or”

`xor` (*w1 w2 – w*) core “x-or”

`invert` (*w1 – w2*) core “invert”

`mux` (*u1 u2 u3 – u*) gforth-1.0 “mux”

Multiplex: For every bit in *u3*: for a 1 bit, select the corresponding bit from *u1*, otherwise the corresponding bit from *u2*. E.g., `%0011 %1100 %1010 mux` gives `%0110`

`lshift` (*u1 u – u2*) core “l-shift”

Shift *u1* left by *u* bits.

`rshift` (*u1 u – u2*) core “r-shift”

Shift *u1* (cell) right by *u* bits, filling the shifted-in bits with zero (logical/unsigned shift).

`arshift` (*n1 u – n2*) gforth-1.0 “ar-shift”

Shift *n1* (cell) right by *u* bits, filling the shifted-in bits from the sign bit of *n1* (arithmetic shift).

`dlshift` (*ud1 u – ud2*) gforth-1.0 “dlshift”

Shift *ud1* (double-cell) left by *u* bits.

`drshift` (*ud1 u – ud2*) gforth-1.0 “drshift”

Shift *ud1* (double-cell) right by *u* bits, filling the shifted-in bits with zero (logical/unsigned shift).

`darshift` (*d1 u – d2*) gforth-1.0 “darshift”

Shift *d1* (double-cell) right by *u* bits, filling the shifted-in bits from the sign bit of *d1* (arithmetic shift).

`2*` (*n1 – n2*) core “two-star”

Shift left by 1; also works on unsigned numbers

`2/` (*n1 – n2*) core “two-slash”

Arithmetic shift right by 1. For signed numbers this is a floored division by 2 (note that `/` is symmetric on some systems, but `2/` always floors).

`d2*` (*d1 – d2*) double “d-two-star”

Shift double-cell left by 1; also works on unsigned numbers

`d2/` (*d1 – d2*) double “d-two-slash”

Arithmetic shift right by 1. For signed numbers this is a floored division by 2.

`>pow2` (*u1 – u2*) gforth-1.0 “to-pow2”

u2 is the lowest power-of-2 number with *u2*>=*u1*.

`log2` (*u – n*) gforth-1.0 “log2”

N is the rounded-down binary logarithm of *u*, i.e., the index of the first set bit; if *u*=0, *n*=-1.

`pow2?` (*u – f*) gforth-1.0 “pow-two-query”

f is true if and only if *u* is a power of two, i.e., there is exactly one bit set in *u*.

`ctz` (*x – u*) gforth-1.0 “c-t-z”

count trailing zeros in binary representation of x

Unlike most other operations, rotation of narrower units cannot easily be synthesized from rotation of wider units, so using cell-wide and double-wide rotation operations means that the results depend on the cell width. For published algorithms or cell-width-independent results, you usually need to use a fixed-width rotation operation.

`wrol` (*u1 u – u2*) gforth-1.0 “wrol”

Rotate the least significant 16 bits of *u1* left by *u* bits, set the other bits to 0.

`wror` (*u1 u – u2*) gforth-1.0 “wror”

Rotate the least significant 16 bits of *u1* right by *u* bits, set the other bits to 0.

`lrol` (*u1 u – u2*) gforth-1.0 “lrol”

Rotate the least significant 32 bits of *u1* left by *u* bits, set the other bits to 0.

`lror` (*u1 u – u2*) gforth-1.0 “lror”

Rotate the least significant 32 bits of *u1* right by *u* bits, set the other bits to 0.

`rol` (*u1 u – u2*) gforth-1.0 “rol”

Rotate all bits of *u1* left by *u* bits.

`ror` (*u1 u – u2*) gforth-1.0 “ror”

Rotate all bits of *u1* right by *u* bits.

`drol` (*ud1 u – ud2*) gforth-1.0 “drol”

Rotate all bits of *ud1* (double-cell) left by *u* bits.

`dror` (*ud1 u – ud2*) gforth-1.0 “dror”

Rotate all bits of *ud1* (double-cell) right by *u* bits.

6.5.7 Numeric comparison

All these comparison words produce -1 (all bits set) if the condition is true, otherwise 0. Note that the words that compare for equality (`= <> 0= 0<> d= d<> d0= d0<>`) work for for both signed and unsigned numbers.

`<` (*n1 n2 – f*) core “less-than”

`<=` (*n1 n2 – f*) gforth-0.2 “less-or-equal”

`<>` (*n1 n2 – f*) core-ext “not-equals”

`=` (*n1 n2 – f*) core “equals”

`>` (*n1 n2 – f*) core “greater-than”

`>=` (*n1 n2 – f*) gforth-0.2 “greater-or-equal”

`0<` (*n – f*) core “zero-less-than”

`0<=` (*n – f*) gforth-0.2 “zero-less-or-equal”

`0<>` (*n – f*) core-ext “zero-not-equals”

`0=` (*n – f*) core “zero-equals”

`0>` (*n – f*) core-ext “zero-greater-than”

`0>=` (*n – f*) gforth-0.2 “zero-greater-or-equal”

`u<` (*u1 u2 – f*) core “u-less-than”

`u<=` (*u1 u2 – f*) gforth-0.2 “u-less-or-equal”

`u>` (*u1 u2 – f*) core-ext “u-greater-than”

`u>=` (*u1 u2 – f*) gforth-0.2 “u-greater-or-equal”

`within` (*u1 u2 u3 – f*) core-ext “within”

u2<u3 and u1 in [u2,u3) or: u2>=u3 and u1 not in [u3,u2). This works for unsigned and signed numbers (but not a mixture). Another way to think about this word is to consider the numbers as a circle (wrapping around from `max-u` to 0 for unsigned, and from `max-n` to min-n for signed numbers); now consider the range from u2 towards increasing numbers up to and excluding u3 (giving an empty range if u2=u3); if u1 is in this range, `within` returns true.

`d<` (*d1 d2 – f*) double “d-less-than”

`d<=` (*d1 d2 – f*) gforth-0.2 “d-less-or-equal”

`d<>` (*d1 d2 – f*) gforth-0.2 “d-not-equals”

`d=` (*d1 d2 – f*) double “d-equals”

`d>` (*d1 d2 – f*) gforth-0.2 “d-greater-than”

`d>=` (*d1 d2 – f*) gforth-0.2 “d-greater-or-equal”

`d0<` (*d – f*) double “d-zero-less-than”

`d0<=` (*d – f*) gforth-0.2 “d-zero-less-or-equal”

`d0<>` (*d – f*) gforth-0.2 “d-zero-not-equals”

`d0=` (*d – f*) double "d-zero-equals"
`d0>` (*d – f*) gforth-0.2 "d-zero-greater-than"
`d0>=` (*d – f*) gforth-0.2 "d-zero-greater-or-equal"
`du<` (*ud1 ud2 – f*) double-ext "d-u-less-than"
`du<=` (*ud1 ud2 – f*) gforth-0.2 "d-u-less-or-equal"
`du>` (*ud1 ud2 – f*) gforth-0.2 "d-u-greater-than"
`du>=` (*ud1 ud2 – f*) gforth-0.2 "d-u-greater-or-equal"

6.5.8 Pseudo-random number generator

The pseudo-random number generator is initialised based on the time, so in the usual case it will start at a different value in different sessions.

`rnd` (– *x*) gforth-1.0

x is a random number, with all bits randomly set.

`random` (*u1 – u2*) gforth-1.0

U2 is a random number 0<=*u2*<*u1*.

`seed!` (*x* –) gforth-1.0 "seed-store"

Set seed for `rnd` and `random` to a specific value, resulting in a deterministic sequence.

6.5.9 Floating Point

For the rules used by the text interpreter for recognising floating-point numbers see `rec-float` (see Section 6.17.4.1 [Default recognizers], page 171).

Gforth has a separate floating point stack, but the documentation uses the unified notation.[2]

Floating point numbers have a number of unpleasant surprises for the unwary (e.g., floating point addition is not associative) and even a few for the wary. You should not use them unless you know what you are doing or you don't care that the results you get may be totally bogus. If you want to learn about the problems of floating point numbers (and how to avoid them), you might start with *David Goldberg, What Every Computer Scientist Should Know About Floating-Point Arithmetic (*`https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html`*), ACM Computing Surveys 23(1):5–48, March 1991.*

Conversion between integers and floating-point:

`s>f` (*n – r*) floating-ext "s-to-f"
`d>f` (*d – r*) floating "d-to-f"
`f>s` (*r – n*) floating-ext "f-to-s"
`f>d` (*r – d*) floating "f-to-d"

Arithmetics:

`f+` (*r1 r2 – r3*) floating "f-plus"
`f-` (*r1 r2 – r3*) floating "f-minus"
`f*` (*r1 r2 – r3*) floating "f-star"
`f*+` (*r1 r2 r3 – r*) gforth-1.0 "f-star-plus"

[2] It's easy to generate the separate notation from that by just separating the floating-point numbers out: e.g. `( n r1 u r2 -- r3 )` becomes `( n u -- ) ( F: r1 r2 -- r3 )`.

*r=r1+r2*r3*, on some platforms with fewer rounding errors than `f* f+`).

`f/` (*r1 r2 – r3*) floating "f-slash"

`fnegate` (*r1 – r2*) floating "f-negate"

`fabs` (*r1 – r2*) floating-ext "f-abs"

`fcopysign` (*r1 r2 – r3*) gforth-1.0

r3 takes its absolute value from r1 and its sign from r2

`fmax` (*r1 r2 – r3*) floating "f-max"

`fmin` (*r1 r2 – r3*) floating "f-min"

`floor` (*r1 – r2*) floating "floor"

Round towards the next smaller integral value, i.e., round toward negative infinity.

`fround` (*r1 – r2*) floating "f-round"

Round to the nearest integral value.

`ftrunc` (*r1 – r2*) floating-ext "f-trunc"

round towards 0

`f**` (*r1 r2 – r3*) floating-ext "f-star-star"

$r3 = r1^{r2}$

`fsqrt` (*r1 – r2*) floating-ext "f-square-root"

`fexp` (*r1 – r2*) floating-ext "f-e-x-p"

$r2 = e^{r1}$

`fexpm1` (*r1 – r2*) floating-ext "f-e-x-p-m-one"

$r2 = e^{r1} - 1$

`fln` (*r1 – r2*) floating-ext "f-l-n"

Natural logarithm: $r1 = e^{r2}$

`flnp1` (*r1 – r2*) floating-ext "f-l-n-p-one"

Inverse of `fexpm1`: $r1 + 1 = e^{r2}$

`flog` (*r1 – r2*) floating-ext "f-log"

The decimal logarithm: $r1 = 10^{r2}$

`falog` (*r1 – r2*) floating-ext "f-a-log"

$r2 = 10^{r1}$

`f2*` (*r1 – r2*) gforth-0.2 "f-two-star"

Multiply *r1* by 2.0e0

`f2/` (*r1 – r2*) gforth-0.2 "f-two-slash"

Multiply *r1* by 0.5e0

`1/f` (*r1 – r2*) gforth-0.2 "one-slash-f"

Divide 1.0e0 by *r1*.

Vector arithmetics:

`v*` (*f-addr1 nstride1 f-addr2 nstride2 ucount – r*) gforth-0.5 "v-star"

dot-product: r=v1*v2. The first element of v1 is at f_addr1, the next at f_addr1+nstride1 and so on (similar for v2). Both vectors have ucount elements.

`faxpy` (*ra f-x nstridex f-y nstridey ucount* –) gforth-0.5 "faxpy"

vy=ra*vx+vy, where *vy* is the vector starting at *f_y* with stride *nstridey* bytes, and *vx* is the vector starting at *f_x* with stride *nstridex*, and both vectors contain *ucount elements*.

Angles in floating point operations are given in radians (a full circle has 2 pi radians).

`fsin` (*r1* – *r2*) floating-ext "f-sine"

`fcos` (*r1* – *r2*) floating-ext "f-cos"

`fsincos` (*r1* – *r2 r3*) floating-ext "f-sine-cos"

r2=sin(*r1*), *r3*=cos(*r1*)

`ftan` (*r1* – *r2*) floating-ext "f-tan"

`fasin` (*r1* – *r2*) floating-ext "f-a-sine"

`facos` (*r1* – *r2*) floating-ext "f-a-cos"

`fatan` (*r1* – *r2*) floating-ext "f-a-tan"

`fatan2` (*r1 r2* – *r3*) floating-ext "f-a-tan-two"

r1/r2=tan(*r3*). Forth-2012 does not require, but probably intends this to be the inverse of `fsincos`. In Gforth it is.

`fsinh` (*r1* – *r2*) floating-ext "f-cinch"

`fcosh` (*r1* – *r2*) floating-ext "f-cosh"

`ftanh` (*r1* – *r2*) floating-ext "f-tan-h"

`fasinh` (*r1* – *r2*) floating-ext "f-a-cinch"

`facosh` (*r1* – *r2*) floating-ext "f-a-cosh"

`fatanh` (*r1* – *r2*) floating-ext "f-a-tan-h"

`pi` (– *r*) gforth-0.2

`Fconstant` – *r* is the value pi; the ratio of a circle's area to its diameter.

Special values in IEEE754 can be derived by for example dividing by zero. The most common ones are defined as floating point constants for easy usage.

`infinity` (– *r*) gforth-1.0

floating point infinity

`inf` (– *r*) gforth-1.0

Synonym of `infinity` to allow copying and pasting from the output of `...`, See Section 6.30.6 [Examining data], page 263.

`-infinity` (– *r*) gforth-1.0

floating point -infinity

`-inf` (– *r*) gforth-1.0

Synonym of `-infinity` to allow copying and pasting from the output of `...`, See Section 6.30.6 [Examining data], page 263.

`NaN` (– *r*) gforth-1.0

floating point Not a Number

6.6 Floating-point comparisons

One particular problem with floating-point arithmetic is that comparison for equality often fails when you would expect it to succeed. For this reason approximate equality is often preferred (but you still have to know what you are doing). Also note that IEEE NaNs may compare differently from what you might expect. The comparison words are:

`f~rel` (*r1 r2 r3 – flag*) gforth-0.5 "f-tilde-rel"

Approximate equality with relative error: `|r1-r2|<r3*|r1+r2|`.

`f~abs` (*r1 r2 r3 – flag*) gforth-0.5 "f-tilde-abs"

Approximate equality with absolute error: `|r1-r2|<r3`.

`f~` (*r1 r2 r3 – flag*) floating-ext "f-proximate"

Forth-2012 medley for comparing r1 and r2 for equality: r3>0: `f~abs`; r3=0: bitwise comparison; r3<0: `fnegate f~rel`.

`f=` (*r1 r2 – f*) gforth-0.2 "f-equals"

`f<>` (*r1 r2 – f*) gforth-0.2 "f-not-equals"

`f<` (*r1 r2 – f*) floating "f-less-than"

`f<=` (*r1 r2 – f*) gforth-0.2 "f-less-or-equal"

`f>` (*r1 r2 – f*) gforth-0.2 "f-greater-than"

`f>=` (*r1 r2 – f*) gforth-0.2 "f-greater-or-equal"

`f0<` (*r – f*) floating "f-zero-less-than"

`f0<=` (*r – f*) gforth-0.2 "f-zero-less-or-equal"

`f0<>` (*r – f*) gforth-0.2 "f-zero-not-equals"

`f0=` (*r – f*) floating "f-zero-equals"

`f0>` (*r – f*) gforth-0.2 "f-zero-greater-than"

`f0>=` (*r – f*) gforth-0.2 "f-zero-greater-or-equal"

6.7 Stack Manipulation

Gforth maintains a number of separate stacks:

- A data stack (also known as the *parameter stack*) – for characters, cells, addresses, and double cells.
- A floating point stack – for holding floating point (FP) numbers.
- A return stack – for holding the return addresses of colon definitions and other (non-FP) data.
- A locals stack – for holding local variables.

6.7.1 Data stack

`drop` (*w –*) core "drop"

`nip` (*w1 w2 – w2*) core-ext "nip"

`dup` (*w – w w*) core "dupe"

`over` (*w1 w2 – w1 w2 w1*) core "over"

`third` (*w1 w2 w3 – w1 w2 w3 w1*) gforth-1.0 "third"

`fourth` (*w1 w2 w3 w4 – w1 w2 w3 w4 w1*) gforth-1.0 “fourth”

`swap` (*w1 w2 – w2 w1*) core “swap”

`rot` (*w1 w2 w3 – w2 w3 w1*) core “rote”

`-rot` (*w1 w2 w3 – w3 w1 w2*) gforth-0.2 “not-rote”

`tuck` (*w1 w2 – w2 w1 w2*) core-ext “tuck”

`pick` (*S:... u – S:... w*) core-ext “pick”

Actually the stack effect is `x0 ... xu u -- x0 ... xu x0` .

`roll` (*x0 x1 .. xn n – x1 .. xn x0*) core-ext

`?dup` (*w – S:... w*) core “question-dupe”

Actually the stack effect is: `( 0 -- 0 | x\0 -- x x )`. It performs a `dup` if x is nonzero.

`2drop` (*w1 w2 –*) core “two-drop”

`2nip` (*w1 w2 w3 w4 – w3 w4*) gforth-0.2 “two-nip”

`2dup` (*w1 w2 – w1 w2 w1 w2*) core “two-dupe”

`2over` (*w1 w2 w3 w4 – w1 w2 w3 w4 w1 w2*) core “two-over”

`2swap` (*w1 w2 w3 w4 – w3 w4 w1 w2*) core “two-swap”

`2rot` (*w1 w2 w3 w4 w5 w6 – w3 w4 w5 w6 w1 w2*) double-ext “two-rote”

`2tuck` (*w1 w2 w3 w4 – w3 w4 w1 w2 w3 w4*) gforth-0.2 “two-tuck”

6.7.2 Floating point stack

`fdrop` (*r –*) floating “f-drop”

`fnip` (*r1 r2 – r2*) gforth-0.2 “f-nip”

`fdup` (*r – r r*) floating “f-dupe”

`fover` (*r1 r2 – r1 r2 r1*) floating “f-over”

`fthird` (*r1 r2 r3 – r1 r2 r3 r1*) gforth-1.0 “fthird”

`ffourth` (*r1 r2 r3 r4 – r1 r2 r3 r4 r1*) gforth-1.0 “ffourth”

`fswap` (*r1 r2 – r2 r1*) floating “f-swap”

`frot` (*r1 r2 r3 – r2 r3 r1*) floating “f-rote”

`f-rot` (*r1 r2 r3 – r3 r1 r2*) gforth-1.0 “f-not-rote”

`ftuck` (*r1 r2 – r2 r1 r2*) gforth-0.2 “f-tuck”

`fpick` (*f:... u – f:... r*) gforth-0.4 “fpick”

Actually the stack effect is `r0 ... ru u -- r0 ... ru r0` .

6.7.3 Return stack

The return stack primarily exists for storing system data, such as return addresses and loop control parameters, but Forth also allows programmers to make use of it, albeit with restrictions stemming from the other uses. The primary use is for temporary storage of data; locals also provide this capability, and usually in a more convenient way; some purists (or puritans) prefer to avoid locals, though.

In Gforth 1.0 you can use the return stack during text interpretation (and you cannot use locals for that). The only limitation here is that you cannot pass data on the return stack into or out of an included file, block, or `evaluate`d string. Example:

```
1 >r
: foo [ r> ] literal ;
foo . \ prints 1
```

This interpretive usage of return-stack words is non-standard, and many other Forth systems do not have support this usage, or limit it to within one line.

In Gforth you can use the return stack for storing data while you also keep and access data in locals. However, the standard puts restrictions on mixing return stack and locals usage, for easy locals implementations, and there are systems that actually rely on these restrictions. So, if you want to produce a standard compliant program and you are using local variables in a definition, forget about return stack manipulations in that word (refer to the standard document for the exact rules).

`>r` (*w* – *R:w*) core “to-r”

`r>` (*R:w* – *w*) core “r-from”

`r@` (*R:w* – *R:w w*) core “r-fetch”

`r'@` (*r:w r:w2* – *r:w r:w2 w*) gforth-1.0 “r-tick-fetch”

The second item on the return stack

`rpick` (*R:wu ... R:w0 u* – *R:wu ... R:w0 wu*) gforth-1.0

wu is the *u*th element on the return stack; `0 rpick` is equivalent to `r@`.

`rdrop` (*R:w* –) gforth-0.2 “rdrop”

`2>r` (*w1 w2* – *R:w1 R:w2*) core-ext “two-to-r”

`2r>` (*R:w1 R:w2* – *w1 w2*) core-ext “two-r-from”

`2r@` (*R:w1 R:w2* – *R:w1 R:w2 w1 w2*) core-ext “two-r-fetch”

`2rdrop` (*R:w1 R:w2* –) gforth-0.2 “two-r-drop”

`n>r` (*x1 .. xn n* – *R:xn..R:x1 R:n*) tools-ext “n-to-r”

In Standard Forth, the order of items on the return stack is not specified, and the only thing you can do with the items on the return stack is to use `nr>`

`nr>` (*R:xn..R:x1 R:n* – *x1 .. xn n*) tools-ext “n-r-from”

In Standard Forth, the order of items on the return stack is not specified, and the only thing you can do with the items on the return stack is to use `nr>`

On some platforms (particularly, 32-bit platforms) floating-point numbers are not naturally aligned on the return stack and this can lead to (usually, but not always) small performance disadvantages.

`f>r` (*r* –) gforth-experimental “f-to-r”

Actual stack effect: `( r -- R:r )`

`fr>` (– *r*) gforth-experimental “f-r-from”

Actual stack effect: `( R:r -- r )`

`fr@` (– *r*) gforth-experimental “f-r-fetch”

Actual stack effect: `( R:r -- R:r r )`

6.7.4 Locals stack

Gforth uses a separate locals stack. It is described, along with the reasons for its existence, in Section 6.26.1.6 [Locals implementation], page 225.

6.7.5 Stack pointer manipulation

In the stack effects of the following words, ignore the occurrences of "..." in the stack-pointer fetching words.

`sp0` (– *a-addr*) gforth-0.4 "sp-zero"

User variable – initial value of the data stack pointer.

`sp@` (*S:...* – *a-addr*) gforth-0.2 "sp-fetch"

`sp!` (*a-addr* – *S:...*) gforth-0.2 "sp-store"

`fp0` (– *a-addr*) gforth-0.4 "fp-zero"

User variable – initial value of the floating-point stack pointer.

`fp@` (*f:...* – *f-addr*) gforth-0.2 "fp-fetch"

`fp!` (*f-addr* – *f:...*) gforth-0.2 "fp-store"

`rp0` (– *a-addr*) gforth-0.4 "rp-zero"

User variable – initial value of the return stack pointer.

`rp@` (– *a-addr*) gforth-0.2 "rp-fetch"

`rp!` (*a-addr* –) gforth-0.2 "rp-store"

`lp0` (– *a-addr*) gforth-0.4 "lp-zero"

User variable – initial value of the locals stack pointer.

`lp@` (– *c-addr*) gforth-internal "lp-fetch"

C_addr is the current value of the locals stack pointer.

`lp!` (*c-addr* –) gforth-internal "lp-store"

Set the locals pointer to *c-addr*.

6.8 Memory

In addition to the Standard Forth memory allocation words, there is also a garbage collector (`https://www.complang.tuwien.ac.at/forth/garbage-collection.zip`).

6.8.1 Memory model

Standard Forth considers a Forth system as consisting of several address spaces, of which only *data space* is managed and accessible with the memory words in standard programs. Memory not necessarily in data space includes the stacks, the code (called code space) and the headers (called name space). Gforth allows at least read access to all these logical spaces, but does not guarantee that code accessing the stacks, the threaded or native code, or the headers is portable or will work in the next Gforth version; Gforth provides some accessor words for these purposes, however.

Another division of memory is between dictionary and heap memory.[3] In heap memory you can free allocations in arbitrary order, but you cannot grow allocations in-place (see

[3] The term *dictionary* is also used to refer to the search data structure embodied in word lists and headers. The search data (word headers) reside in dictionary memory.

Section 6.8.4 [Heap Allocation], page 80). In dictionary memory deallocation is impractical for the most part, but you can grow allocations in place (see Section 6.8.2 [Dictionary allocation], page 76). Gforth (since 1.0) allows having several sections of dictionary memory in order to allow more flexibility in this growing (see Section 6.8.3 [Sections], page 78).

One relevant concept in this context is the *contiguous region*: It means a piece of memory that is contiguous, without any system data interleaved with it. In heap memory each allocation forms one contiguous region, and separate allocations are not contiguous with any other allocations. In dictionary memory all allocations in a section are contiguous, unless something happens that ends the contiguous region; a typical reason for ending a contiguous region is defining a word in that section.

Gforth provides one big address space, and address arithmetic can be performed between any addresses. However, in the dictionary headers or code are interleaved with data, so almost the only contiguous regions are those described by Standard Forth as contiguous; but you can be sure that, within a section the dictionary is allocated towards increasing addresses even between contiguous regions. The memory order of allocations in the heap is platform-dependent (and possibly different from one run to the next).

6.8.2 Dictionary allocation

Dictionary allocation is a stack-oriented allocation scheme, i.e., if you want to deallocate X, you also deallocate everything allocated after X.

The allocations using the words below are contiguous and grow the region towards increasing addresses. Other words that allocate dictionary memory of any kind (i.e., defining words including `:noname`) in the same section end the contiguous region and start a new one, but allocating memory in a different section does not end a contiguous region.

In Standard Forth only `create`d words are guaranteed to produce an address that is the start of the following contiguous region. In particular, the cell allocated by `variable` is not guaranteed to be contiguous with following `allot`ed memory.

You can deallocate memory by using `allot` with a negative argument (with some restrictions, see `allot`). For larger deallocations use `marker`.

`here` (– *addr*) core

Return the address of the next free location in data space.

`unused` (– *u*) core-ext

Return the amount of free space remaining (in address units) in the region addressed by `here`.

`allot` (*n* –) core

Reserve *n* address units of data space without initialization. *n* is a signed number, passing a negative *n* releases memory. In Forth-2012 you can only deallocate memory from the current contiguous region in this way. In Gforth you can deallocate anything in this way but named words. The system does not check this restriction.

`->here` (*addr* –) gforth-1.0 “to-here”

Change the value of `here` to *addr*.

`c,` (*c* –) core “c-comma”

Reserve data space for one char and store *c* in the space.

`f,` (*f* –) gforth-0.2 “f-comma”

Reserve data space for one floating-point number and store *f* in the space.

`,` (*w* –) core "comma"

Reserve data space for one cell and store *w* in the space.

`2,` (*w1 w2* –) gforth-0.2 "two-comma"

Reserve data space for two cells and store the double *w1 w2* there, *w2* first (lower address).

`w,` (*x* –) gforth-1.0 "w-comma"

Reserve 2 bytes of data space and store the least significant 16 bits of *x* there.

`l,` (*l* –) gforth-1.0 "l-comma"

Reserve 4 bytes of data space and store the least significant 32 bits of *x* there.

`x,` (*x* –) gforth-1.0 "x-comma"

Reserve 8 bytes of data space and store (the least significant 64 bits) of *x* there. Reserve 8 bytes of data space and store *w* there.

`xd,` (*xd* –) gforth-1.0 "x-d-comma"

Reserve 8 bytes of data space and store the least significant 64 bits of *x* there.

`A,` (*addr* –) gforth-0.2 "a-comma"

Reserve data space for one cell, and store *addr* there. For our cross-compiler this provides the type information necessary for a relocatable image; normally, though, this is equivalent to `,`.

`mem,` (*addr u* –) gforth-0.6 "mem-comma"

Reserve *u* bytes of dictionary space and copy *u* bytes starting at *addr* there. If you want the memory to be aligned, precede `mem,` with an alignment word.

`save-mem-dict` (*addr1 u – addr2 u*) gforth-0.7

Copy the memory block *addr1 u* to a newly `allot`ed memory block of size *u*; the target memory block starts at *addr2*.

Memory accesses have to be aligned (see Section 6.8.7 [Address arithmetic], page 84). So of course you should allocate memory in an aligned way, too. I.e., before allocating a cell, `here` must be cell-aligned, etc. The words below align `here` if it is not already. Basically it is only already aligned for a type, if the last allocation was a multiple of the size of this type and if `here` was aligned for this type before.

After freshly `create`ing a word, `here` is `align`ed in Standard Forth (`maxalign`ed in Gforth).

`align` (–) core

If the data-space pointer is not aligned, reserve enough space to align it.

`falign` (–) floating "f-align"

If the data-space pointer is not float-aligned, reserve enough space to align it.

`sfalign` (–) floating-ext "s-f-align"

If the data-space pointer is not single-float-aligned, reserve enough space to align it.

`dfalign` (–) floating-ext "d-f-align"

If the data-space pointer is not double-float-aligned, reserve enough space to align it.

`maxalign` (–) gforth-0.2

Align data-space pointer for all Forth alignment requirements.

6.8.3 Sections

If you want to do something that allocates memory from the dictionary or compiles code in the middle of a contiguous region of another dictionary allocation, or in the middle of a colon definition, that's not possible with a single dictionary pointer, leading to complicated workarounds.

Gforth provides dictionary sections to address this problem. Each section has its own dictionary pointer, and allocating or compiling something in one section does not interrupt the contiguity of allocations in other sections. In this respect Gforth's sections are similar to sections and segments in assembly languages.

One difference is that the most common usage of sections is as a stack of sections, which is useful for building nested definitions or dictionary-allocated data structures: Use `next-section` for the inner definition or data structure, switch back with `previous-section`.

Words like `latest` (see Section 6.15.2 [Name token], page 158) and `latestxt` (see Section 6.11.7 [Anonymous Definitions], page 124) refer to the most recent definition in the current section. Quotations (see Section 6.11.8 [Quotations], page 125) and the implicit quotation of `does>` (see Section 6.11.10.2 [User-defined defining words using CREATE], page 126) are in a different section than the containing definition, so after the quotation ends (and the section is switched back), words like `latest` report the outer definition rather than the quotation.

An example of such a usage of the section stack is:

```
create my2x2matrix
  next-section here 1 , 2 , previous-section ,
  next-section here 3 , 4 , previous-section ,

\ now print my2x2matrix[0,1], i.e., "2":
my2x2matrix 0 cells + @ 1 cells + @ .
```

This works also for allocating section memory while compiling a definition, or defining a definition during a contiguous region, e.g.:

```
create mydispatchtable
  next-section :noname ." foo" ; previous-section ,
  next-section :noname ." bar" ; previous-section ,

\ now dispatch mydispatchtable[1]
mydispatchtable 1 cells + @ execute
```

Note that the interpretation semantics of `[:` (see Section 6.11.8 [Quotations], page 125) switches to the next section internally, so you can write `mydispatchtable` also as follows:

```
create mydispatchtable
  [: ." foo" ;] ,
  [: ." bar" ;] ,
```

The interpretation semantics of `does>` uses a separate section, so the `does>` does not end the contiguous region, and you can define a word `mydispatch` that includes the dispatch code, as follows:

```
create mydispatch
does> ( u -- )
```

```
    ( u addr ) swap cells + @ execute ;
  [: ." foo" ;] ,
  [: ." bar" ;] ,

1 mydispatch \ prints "bar"
```

`next-section` (–) gforth-1.0

Switch to the next section in the section stack. If there is no such section yet, create it (with the size being a quarter of the size of the current section).

`previous-section` (–) gforth-1.0

Switch to the previous section in the section stack; the now-next section continues to exist with everything that was put there. Throw an exception if there is no previous section.

The bottom section in the section stack has the size given with the `--dictionary-size` command-line parameter (see Section 2.1 [Invoking Gforth], page 4).

In addition to the stack of anonymous sections you can also have named sections that you define with:

`extra-section` (*usize* "*name*" –) gforth-1.0

Define a new word *name* and create a section *s* with at least *usize* unused bytes. *Name* execution `( ... xt -- ... )`: When calling *name*, the current section is *c*. Switch the current section to be *s*, *execute* xt, then switch the current section back to *c*.

As an example, here's a variant of the `my2x2matrix` definition:

```
4 cells extra-section myvec

create my2x2matrix
  ' here myvec 1 ' , myvec 2 ' , myvec ,
  ' here myvec 3 ' , myvec 4 ' , myvec ,
```

Currently a named section does not start a dictionary stack, and using `next-section` inside a named section throws an error.

You can show the existing sections with:

`.sections` (–) gforth-1.0 "dot-sections"

Show all the sections and their status.

At the time of this writing this outputs:

```
                start      size      used name
      $7F9A5A516000     32768        96 noname
      $7F9A5A1A1000    131072       208 noname
      $7F9A5A1C2000    524288      2128 noname
      $7F9A4BDFD000   2097152     32680 noname
>     $7F9A4BFFE040   8388608    659272 Forth
      $7F9A5A51F000     20480      1448 locals-headers
```

The lines describe the different sections: First the section stack, with sections called `noname` and (the bottom) `Forth`, then the extra-sections. The columns are the start address of the section, the gross size (including section management overhead), how much of the section is already used, and the name. The size and used columns are in decimal base.

In the section `Forth`, not all of the remaining size can be used for `allot`ting memory, because room must be left for `pad` (see Section 6.8.8 [Memory Blocks], page 87). The current section is marked with `>`. Also, if you use `word` (see Section 6.18 [The Input Stream], page 181), you must leave room in the current section for the parsed string and its length byte.

6.8.4 Heap allocation

Heap allocation supports deallocation of allocated memory in any order. It does not affect dictionary allocation (i.e., heap allocation does not end a contiguous region). In Gforth, these words are implemented using the standard C library calls malloc(), free() and realloc().

The memory region produced by one invocation of `allocate` or `resize` is internally contiguous. There is no contiguity between such a region and any other region (including others allocated from the heap).

`allocate` (*u – a_addr wior*) memory

Allocate *u* address units of contiguous data space. This data space is not initialized. If the allocation is successful, *a-addr* is the start address of the allocated region and *wior* is 0. If the allocation fails, *a-addr* is arbitrary and *wior* is a non-zero I/O result code.

`free` (*a_addr – wior*) memory

Return the region of data space starting at *a-addr* to the system. The region must originally have been obtained using `allocate` or `resize`, otherwise the result of `free` is unpredictable. If the operation is successful, *wior* is 0. If the operation fails, *wior* is a non-zero I/O result code.

`resize` (*a_addr1 u – a_addr2 wior*) memory

Change the size of the allocated area at *a-addr1* to *u* address units, possibly moving the contents to a different area. *a-addr2* is the address of the resulting area. If the operation is successful, *wior* is 0. If the operation fails, *wior* is a non-zero I/O result code. If *a-addr1* is 0, Gforth's (but not the Standard) `resize` `allocate`s *u* address units.

6.8.4.1 Memory blocks and heap allocation

Additional words for dealing with memory blocks are described in Section 6.8.8 [Memory Blocks], page 87. An alternative to the following words are among the $tring words (see Section 6.9.5 [$tring words], page 96).

`save-mem` (*addr1 u – addr2 u*) gforth-0.2

Copy the memory block *addr u* to *addr2*, which is the start of a newly heap allocated *u*-byte region.

`extend-mem` (*addr1 u1 u – addr addr2 u2*) gforth-experimental

Addr1 u1 is a memory block in heap memory. Increase the size of this memory block by *u* aus, possibly reallocating it. *C-addr2 u2* is the resulting memory block (*u2=u1+u*), *addr* is the start of the *u* additional aus (*addr=addr2+u1*).

`free-mem-var` (*addr –*) gforth-experimental

Addr is the address of a 2variable containing a memory block descriptor *c-addr u* in heap memory; `free-mem-var` frees the memory block and stores 0 0 in the 2variable.

Usage example:

```
2variable myblock
```

```
"foo" save-mem myblock 2!
myblock 2@ "bar" tuck >r >r extend-mem myblock 2! r> swap r> move
myblock 2@ type \ prints "foobar"
myblock free-mem-var
```

6.8.4.2 Growable memory buffers

The following words are useful for growable memory buffers. One can alternatively use $trings (see Section 6.9.5 [$tring words], page 96), and the differences are: When the used memory in the buffer shrinks, $trings may resize the buffer, while `adjust-buffer` does not, which may be preferable for a buffer that is reused all the time. However, $strings have one cell less memory overhead, and for longer-term storage the shrinking may be worthwhile.

`buffer%` (– *u1 u2*) gforth-experimental "buffer-percent"

u1 is the alignment and *u2* is the size of a buffer descriptor.

`init-buffer` (*addr* –) gforth-experimental

`adjust-buffer` (*u addr* –) gforth-experimental

Adjust buffer% at addr to length u. This may grow the allocated area, but never shrinks it.

You can get the current address and length of such a buffer with `2@`.

Typical usage:

```
create mybuf  buffer% %allot  mybuf init-buffer
s" frobnicate" mybuf adjust-buffer  mybuf 2@ move
mybuf 2@ type
s" foo"        mybuf adjust-buffer  mybuf 2@ move
mybuf 2@ type
```

6.8.5 Memory Access

`@` (*a-addr* – *w*) core "fetch"

w is the cell stored at *a_addr*.

`!` (*w a-addr* –) core "store"

Store *w* into the cell at *a-addr*.

`+!` (*n a-addr* –) core "plus-store"

Add *n* to the cell at *a-addr*.

`!@` (*w1 a-addr* – *w2*) gforth-experimental "store-fetch"

Fetch *w2* from *a_addr*, then store *w1* there. There is also `atomic!@` (see [Hardware operations for multi-tasking], page 274).

`+!@` (*u1 a-addr* – *u2*) gforth-experimental "plus-store-fetch"

Fetch *u2* from *a_addr*, then increment this location by *u1*. There is also `atomic+!@` (see [Hardware operations for multi-tasking], page 274).

`c@` (*c-addr* – *c*) core "c-fetch"

c is the char stored at *c_addr*.

`c!` (*c c-addr* –) core "c-store"

Store *c* into the char at *c-addr*.

`2@` (*a-addr* – *w1 w2*) core "two-fetch"

w2 is the content of the cell stored at *a-addr*, *w1* is the content of the next cell.

`2!` (*w1 w2 a-addr* –) core "two-store"

Store *w2* into the cell at *c-addr* and *w1* into the next cell.

`f@` (*f-addr* – *r*) floating "f-fetch"

r is the float at address *f-addr*.

`f!` (*r f-addr* –) floating "f-store"

Store *r* into the float at address *f-addr*.

`sf@` (*sf-addr* – *r*) floating-ext "s-f-fetch"

Fetch the single-precision IEEE floating-point value *r* from the address *sf-addr*.

`sf!` (*r sf-addr* –) floating-ext "s-f-store"

Store *r* as single-precision IEEE floating-point value to the address *sf-addr*.

`df@` (*df-addr* – *r*) floating-ext "d-f-fetch"

Fetch the double-precision IEEE floating-point value *r* from the address *df-addr*.

`df!` (*r df-addr* –) floating-ext "d-f-store"

Store *r* as double-precision IEEE floating-point value to the address *df-addr*.

6.8.6 Special Memory Accesses

This section is about memory accesses useful for communicating with other software or other computers. This means that the accesses are of a certain bit width (independent of Gforth's cell width), are possibly not naturally aligned and typically have a certain byte order that may be different from the native byte order of the system that Gforth runs on.

We use the following prefixes:

`c`	8 bits (character)
`w`	16 bits
`l`	32 bits
`x`	64 bits represented as one cell
`xd`	64 bits represented as two cells

The `x`-prefix words do not work properly on 32-bit systems, so for code that is intended to be portable to 32-bit systems you should use `xd`-prefix words. Note that `xd`-prefix words work on 64-bit systems: there the upper cell is just 0 (for unsigned values) or a sign extension of the lower cell.

The memory-access words below all work with arbitrarily (un)aligned addresses (unlike `@`, `!`, `f@`, `f!`, which require alignment on some hardware), and use native byte order (like these words),

`w@` (*c-addr* – *u*) gforth-0.5 "w-fetch"

u is the zero-extended 16-bit value stored at *c_addr*.

`w!` (*w c-addr* –) gforth-0.7 "w-store"

Store the bottom 16 bits of *w* at *c_addr*.

`l@` (*c-addr – u*) gforth-0.7 “l-fetch”

u is the zero-extended 32-bit value stored at *c_addr*.

`l!` (*w c-addr –*) gforth-0.7 “l-store”

Store the bottom 32 bits of *w* at *c_addr*.

`x@` (*c-addr – u*) gforth-1.0 “x-fetch”

u is the zero-extended 64-bit value stored at *c_addr*.

`x!` (*w c-addr –*) gforth-1.0 “x-store”

Store the bottom 64 bits of *w* at *c_addr*.

`xd@` (*c-addr – ud*) gforth-1.0 “x-d-fetch”

ud is the zero-extended 64-bit value stored at *c_addr*.

`xd!` (*ud c-addr –*) gforth-1.0 “x-d-store”

Store the bottom 64 bits of *ud* at *c_addr*.

For accesses with a specific byte order, you have to perform byte-order adjustment immediately after a fetch (before the sign-extension), or immediately before the store. The results of these byte-order adjustment words are always zero-extended.

`wbe` (*u1 – u2*) gforth-1.0

Convert 16-bit value in *u1* from native byte order to big-endian or from big-endian to native byte order (the same operation)

`wle` (*u1 – u2*) gforth-1.0

Convert 16-bit value in *u1* from native byte order to little-endian or from little-endian to native byte order (the same operation)

`lbe` (*u1 – u2*) gforth-1.0

Convert 32-bit value in *u1* from native byte order to big-endian or from big-endian to native byte order (the same operation)

`lle` (*u1 – u2*) gforth-1.0

Convert 32-bit value in *u1* from native byte order to little-endian or from little-endian to native byte order (the same operation)

`xbe` (*u1 – u2*) gforth-1.0

Convert 64-bit value in *u1* from native byte order to big-endian or from big-endian to native byte order (the same operation)

`xle` (*u1 – u2*) gforth-1.0

Convert 64-bit value in *u1* from native byte order to little-endian or from little-endian to native byte order (the same operation)

`xdbe` (*ud1 – ud2*) gforth-1.0

Convert 64-bit value in *ud1* from native byte order to big-endian or from big-endian to native byte order (the same operation)

`xdle` (*ud1 – ud2*) gforth-1.0

Convert 64-bit value in *ud1* from native byte order to little-endian or from little-endian to native byte order (the same operation)

For signed fetches with a specific byte order, you have first have to perform an unsigned fetch and a byte-order correction, and finally use a sign-extension word:

`c>s` (*x* – *n*) gforth-1.0 "c-to-s"

Sign-extend the 8-bit value in *x* to cell *n*.

`w>s` (*x* – *n*) gforth-1.0 "w-to-s"

Sign-extend the 16-bit value in *x* to cell *n*.

`l>s` (*x* – *n*) gforth-1.0 "l-to-s"

Sign-extend the 32-bit value in *x* to cell *n*.

`x>s` (*x* – *n*) gforth-1.0 "x-to-s"

Sign-extend the 64-bit value in *x* to cell *n*.

`xd>s` (*xd* – *d*) gforth-1.0 "xd-to-s"

Sign-extend the 64-bit value in *xd* to double-cell *d*.

Overall, this leads to sequences like

```
w@ wbe w>s   \ 16-bit unaligned signed big-endian fetch
>r lle r> l! \ 32-bit unaligned little-endian store
```

6.8.7 Address arithmetic

Address arithmetic is the foundation on which you can build data structures like arrays, records (see Section 6.12 [Structures], page 141) and objects (see Section 6.27 [Object-oriented Forth], page 228).

Standard Forth does not specify the sizes of the data types. Instead, it offers a number of words (e.g., `cells`) for computing sizes and doing address arithmetic.

Address arithmetic is performed in terms of address units (aus); on most systems the address unit is one byte. There is also word-addressed[4] hardware in some embedded systems, and on these systems the au is one cell. Finally, Forth-2012 also supports systems where a char needs more than one au. However, the common practice is that `1 chars` produces 1, and this will be standardized in the next release of the standard.

The basic address arithmetic words are `+` and `-`. E.g., if you have the address of a cell, perform `1 cells +`, and you will have the address of the next cell.

Standard Forth also defines words for aligning addresses for specific types. Some hardware requires that accesses to specific data types must only occur at specific addresses; e.g., that (4-byte) cells may only be accessed at addresses divisible by 4. Even if a machine allows unaligned accesses, it can usually perform aligned accesses faster.

For the performance-conscious: alignment operations are usually only necessary during the definition of a data structure, not during the (more frequent) accesses to it.

Standard Forth defines no words for character-aligning addresses, but given that `1 chars`=1 is common practice, that's not a big loss.

Standard Forth guarantees that addresses returned by `CREATE`d words are cell-aligned; in addition, Gforth guarantees that these addresses are aligned for all Forth purposes.[5]

[4] In Forth terminology: cell-addressed.

[5] Some SIMD extensions of some instruction sets impose more severe alignment constraints that `create` currently does not satisfy.

Note that the Standard Forth word `char` has nothing to do with address arithmetic.

`chars` (*n1 – n2*) core

n2 is the number of address units of *n1* chars.

`char+` (*c-addr1 – c-addr2*) core "char-plus"

`1 chars +.`

`char-` (*c-addr1 – c-addr2*) gforth-0.7 "char-minus"

`1 chars -`

`cells` (*n1 – n2*) core "cells"

n2 is the number of address units of *n1* cells.

`cell+` (*a-addr1 – a-addr2*) core "cell-plus"

`1 cells +`

`cell-` (*a-addr1 – a-addr2*) core "cell-minus"

`1 cells -`

`cell/` (*n1 – n2*) gforth-1.0 "cell-divide"

N2 is the number of cells that fit into *n1* aus, rounded towards negative infinity.

`cell` (*– u*) gforth-0.2

`Constant` – `1 cells`

`aligned` (*c-addr – a-addr*) core "aligned"

a-addr is the smallest aligned address greater than or equal to *c-addr*.

`floats` (*n1 – n2*) floating "floats"

n2 is the number of address units of *n1* floats.

`float+` (*f-addr1 – f-addr2*) floating "float-plus"

`1 floats +.`

`float` (*– u*) gforth-0.3

`Constant` – the number of address units corresponding to a floating-point number.

`float/` (*n1 – n2*) gforth-1.0 "float-divide"

N2 is the number of floats that fit into *n1* aus, rounded towards negative infinity.

`faligned` (*c-addr – f-addr*) floating "f-aligned"

f-addr is the first float-aligned address greater than or equal to *c-addr*.

`sfloats` (*n1 – n2*) floating-ext "s-floats"

n2 is the number of address units of *n1* single-precision IEEE floating-point numbers.

`sfloat+` (*sf-addr1 – sf-addr2*) floating-ext "s-float-plus"

`1 sfloats +.`

`sfloat/` (*n1 – n2*) gforth-1.0 "s-float-divide"

N2 is the number of sfloats that fit into *n1* aus, rounded towards negative infinity.

`sfaligned` (*c-addr – sf-addr*) floating-ext "s-f-aligned"

sf-addr is the first single-float-aligned address greater than or equal to *c-addr*.

`dfloats` (*n1 – n2*) floating-ext "d-floats"

n2 is the number of address units of *n1* double-precision IEEE floating-point numbers.

`dfloat+` (*df-addr1 – df-addr2*) floating-ext "d-float-plus"

`1 dfloats +`.

`dfloat/` (*n1 – n2*) gforth-1.0 "d-float-divide"

N2 is the number of dfloats that fit into *n1* aus, rounded towards negative infinity.

`dfaligned` (*c-addr – df-addr*) floating-ext "d-f-aligned"

df-addr is the first double-float-aligned address greater than or equal to *c-addr*.

`maxaligned` (*addr1 – addr2*) gforth-0.2

addr2 is the first address after *addr1* that satisfies all alignment restrictions.

`*aligned` (*addr1 n – addr2*) gforth-1.0 "star-aligned"

addr2 is the aligned version of *addr1* with respect to the alignment *n*; *n* must be a power of 2.

`*align` (*n* –) gforth-1.0 "star-align"

Align `here` with respect to the alignment *n*.

`waligned` (*addr – addr'*) gforth-1.0

Addr' is the next even address `>=` *addr*.

`walign` (–) gforth-1.0

Align `here` to even.

`laligned` (*addr – addr'*) gforth-1.0

Addr' is the next address `>=` *addr* divisible by 4.

`lalign` (–) gforth-1.0

Align `here` to be divisible by 4.

`xaligned` (*addr – addr'*) gforth-1.0

Addr' is the next address `>=` *addr* divisible by 8.

`xalign` (–) gforth-1.0

Align `here` to be divisible by 8.

For cell-based addess calculations, there are three shortcuts, based on

```
: th ( a-addr0 n -- a-addr1 ) cells + ;
```

`th` (*a-addr1 n – a-addr2*) gforth-0.7 "th"

Array address calculator, `cells +`

`th@` (*a-addr n – u*) gforth-1.0 "th-fetch"

Array fetch, `cells + @`

`th!` (*u a-addr n* –) gforth-1.0 "th-store"

Array store, `cells + !`

The environmental query `address-unit-bits` (see Section 6.21 [Environmental Queries], page 192) and the following words may be useful to those who want to write software portable to non-byte-addressed machines.

`/w` (– *u*) gforth-0.7 "slash-w"

address units for a 16-bit value

`/l` (– *u*) gforth-0.7 “slash-l”

address units for a 32-bit value

`/x` (– *u*) gforth-1.0 “slash-x”

address units for a 64-bit value

6.8.8 Memory Blocks

Memory blocks often represent character strings; For ways of storing character strings in memory see Section 6.9.1 [String representations], page 88. For other string-processing words see Section 6.24.4 [Displaying characters and strings], page 209.

In case you want to write a program that is portable to systems with `1 chars >` 1 (not recommended), you have to note the difference between words that take a number of aus (e.g., `erase`) and words that take a number of chars (e.g., `blank`), and insert `chars` as appropriate.

When copying characters between overlapping memory regions, use `move`. `Cmove` and `cmove>` tend to be slower than a well-implemented `move`.

`move` (*c-from c-to ucount* –) core “move”

Copy the contents of *ucount* aus at *c-from* to *c-to*. `move` works correctly even if the two areas overlap.

`cmove` (*c-from c-to u* –) string “c-move”

Copy the contents of *ucount* characters from data space at *c-from* to *c-to*. The copy proceeds `char`-by-`char` from low address to high address; i.e., for overlapping areas it is safe if *c-to*`<=`*c-from*.

`cmove>` (*c-from c-to u* –) string “c-move-up”

Copy the contents of *ucount* characters from data space at *c-from* to *c-to*. The copy proceeds `char`-by-`char` from high address to low address; i.e., for overlapping areas it is safe if *c-to*`>=`*c-from*.

`fill` (*c-addr u c* –) core “fill”

Store *c* in *u* chars starting at *c-addr*.

`erase` (*addr u* –) core-ext

Clear all bits in *u* aus starting at *addr*.

`blank` (*c-addr u* –) string

Store the space character into *u* chars starting at *c-addr*.

`pad` (– *c-addr*) core-ext

c-addr is the address of a transient region that can be used as temporary data storage. At least 84 characters of space is available.

6.9 Strings and Characters

A Forth char is a byte. Forth programs use chars to represent ASCII characters or other data that fits into a byte.

But the world has moved on since ASCII, and now the dominating character set is Unicode, and it is supported by Forth in its UTF-8 encoding. Forth has extended characters (xchars, see Section 6.9.2 [Xchars and Unicode], page 89) which map to Unicode code points. An xchar for an Unicode code point is represented by 1-4 chars in memory, or one cell on the data stack.

So is an xchar a character (a *user-perceived character* in Unicode terms)? Unfortunately, the writing systems unified by Unicode are too complex for that idea to work in general; e.g., characters can be composed of base characters (one code point) modified by diacritical marks (0 or more additional code points). So in general a user-perceived character cannot be represented by a single cell. So the way to go is to represent text (including a single user-perceived characer) as string.

Once you embrace the idea of working with strings instead of with characters, you find out that you rarely need to deal with individual code points, and therefore rarely need to use words from the xchar words (see Section 6.9.2 [Xchars and Unicode], page 89). Also, given that a string consists of bytes aka chars, you can use the words that deal with chars to work on strings containing Unicode characters. In Unicode terms: the strings are processed on the level of code units (bytes for UTF-8), not code points.

You can use the usual integer words on chars and Xchars on the stack.

In UTF-8 each ASCII character is encoded as single byte with the same value as in ASCII. The same holds true for all other character encodings supported in Gforth. So you can use the character words (e.g., `c@`) to deal with ASCII characters. Only ASCII characters are represented as single bytes in UTF-8, so this benefit stops there. In particular, the Unicode code points 128–255 are represented by two-byte sequences in UTF-8.

6.9.1 String representations

Forth commonly represents strings as cell pair *c-addr u* on the stack; *u* is the length of the string in bytes (aka chars), and *c-addr* is the address of the first byte of the string. Note that a code point may be represented by a sequence of several chars in the string (and a user-perceived character may consist of several code points). See Section 6.9.4 [String words], page 93.

Another string representation is used with the string library of words containing `$` (see Section 6.9.5 [$tring words], page 96). It uses the address of a cell-sized string handle to represent the string when its allocation plays a role, e.g., when appending to the string; this corresponds to owned strings in Rust. When only the content of the string is of interest, the *c-addr u* representation for the string is used with these words, too; the validity of a *c-addr u* pair ends when the underlying string is modified or freed; this corresponds to string slices in Rust.

A legacy string representation are *counted strings*, represented on the stack by *c-addr*. The char addressed by *c-addr* contains a character-count, *n*, of the string and the string occupies the subsequent *n* char addresses in memory. Counted strings are limited to 255 bytes in length. While counted strings may look attractive due to needing only one stack

item, due to their limitations we recommend avoiding them, especially as input parameters of words. See Section 6.9.8 [Counted string words], page 102.

6.9.2 Xchars and Unicode

An xchar is represented as a single cell on the stack and as a sequence of one or more chars (as string) in memory.[6]

The actual supported xchars depend on the encoding, which is determined automatically from the environment variable `LC_CTYPE`, `LC_ALL`, or `LANG` (see Section 2.5 [Environment variables], page 11). If any of them contains "UTF-8" on Gforth startup, Gforth uses UTF-8 encoding (for Unicode), otherwise it uses fixed-width 8-bit encoding. The encoding cannot be changed after Gforth startup, and if any non-ASCII characters are stored in an image, the image must be invoked in a way that produces the same encoding and charset setting.

Any text I/O is expected to happen in the encoding and character set specified in the environment variables, so if any encoding or character set conversions are needed, perform them outside Gforth, with tools such as `recode` or `iconv`.

The fixed-width 8-bit encoding is used for 8-bit encodings of legacy character sets, such as ISO Latin-1, ISO Latin-2, or KOI8-R.

Whatever the environment, an xchar corresponds to a code point of the character set. In some character sets, each code point represents a character, but in Unicode a user-perceived character may consist of a sequence of code points. Gforth currently provides no facilities for dealing with user-perceived characters, and dealing with code points rarely provides any benefit, so the usual way to deal with text is as strings of chars.

The only thing that's in common between Unicode and all the charsets for which Gforth supports a fixed-width encoding is ASCII: These character sets all have the code points 0–127 which have the same meaning, have the same on-stack representation (a number in the range 0–127), and have the same in-memory representation (a single byte (aka char) that has a value in the range 0–127).

When using UTF-8 encoding, all other codepoints take more than one byte/char. In most cases, you can just treat such characters as strings in memory and don't need to use the following words, but if you want to deal with individual codepoints, the following words are useful.

When using the fixed-width encoding, all other code points take only one byte, but you can still use the xchar words to access the code points, and your code will also continue to work when you switch to UTF-8 encoding.

The xchar words add a few data types:

- *xc* is an extended char (xchar) on the stack. It occupies one cell, and is a subset of unsigned cell.
- *xc-addr* is the address of an xchar in memory. Alignment requirements are the same as *c-addr*. The memory representation of an xchar differs from the stack representation, and depends on the encoding used. An xchar may use a variable number of chars in memory.

[6] Of course, you can also store the xchar cell in memory, but Gforth has no words for dealing with sequences of such cells.

- *xc-addr u* is a string (or buffer) of xchars in memory, starting at *xc-addr*, *u* chars (i.e., bytes, not xchars) long.

`xc-size` (*xc – u*) xchar "x-c-size"

The xchar *xc* occupies *u* chars in memory.

`x-size` (*xc-addr u1 – u2*) xchar

The first xchar at *xc-addr* occupies *u2* chars; if *xc-addr u1* does not contain a complete xchar, *u2* is *u1*.

`xc@` (*xc-addr – xc*) xchar-ext "x-c-fetch"

xc is the xchar starting at *xc-addr1*.

`xc@+` (*xc-addr1 – xc-addr2 xc*) xchar "x-c-fetch-plus"

xc is the xchar starting at *xc-addr1*. *xc-addr2* points to the first memory location after *xc*.

`xc@+?` (*xc-addr1 u1 – xc-addr2 u2 xc*) gforth-experimental "x-c-fetch-plus-query"

xc is the xchar starting at *xc-addr1*. *xc-addr2 u2* is the remaining string behind *xc*. If the start of *xc-addr1 u1* contains no valid xchar, *xc* is `invalid-char`, and *xc-addr2 u2* is the remaining string after skipping at least one byte. If *u1*=0, the current behaviour does not make much sense and may change in the future: *xc-addr2*=*xc-addr1*+1, *u2*=MAX-U, and *xc* is either 0 or `invalid-char`.

`xc!+?` (*xc xc-addr1 u1 – xc-addr2 u2 f*) xchar "x-c-store-plus-query"

Stores the xchar *xc* into the buffer starting at address *xc-addr1*, *u1* chars large. *xc-addr2* points to the first memory location after *xc*, *u2* is the remaining size of the buffer. If the xchar *xc* did fit into the buffer, *f* is true, otherwise *f* is false, and *xc-addr2 u2* equal *xc-addr1 u1*. XC!+? is safe against buffer overflows, and therefore preferred over XC!+.

`xc!+` (*xc xc-addr1 – xc-addr2*) xchar "x-c-store"

Stores the xchar *xc* at *xc-addr1*. *xc-addr2* is the next unused address in the buffer. Note that this writes up to 4 bytes, so you need at least 3 bytes of padding after the end of the buffer to avoid overwriting useful data if you only check the address against the end of the buffer.

`xchar+` (*xc-addr1 – xc-addr2*) xchar "x-char-plus"

xc-addr2 is the address of the next xchar behind the one pointed to by *xc-addr*.

`xchar-` (*xc-addr1 – xc-addr2*) xchar-ext "x-char-minus"

xc-addr2 is the address of the previous xchar in front of the one pointed to by *xc-addr*.

`+x/string` (*xc-addr1 u1 – xc-addr2 u2*) xchar-ext "plus-x-slash-string"

xc-addr1 u1 is a string of *u1* chars. *xc-addr2* is the address of the next xchar behind the one pointed to by *xc-addr*. *u2* is the size (in chars) of the rest of the string.

`x\string-` (*xc-addr u1 – xc-addr u2*) xchar-ext "x-backslash-string-minus"

xc-addr1 u1 is a string of *u1* chars. *u2* is the size of the string without its last xchar.

`-trailing-garbage` (*xc-addr u1 – xc-addr u2*) xchar-ext "minus-trailing-garbage"

xc-addr1 u1 is a string of *u1* chars. *u2* is the size of the string after removing the chars from the end that do not constitute a complete, valid xchar.
The idea here is that if you read a fixed number of chars, e.g., with `read-file`, there may

be an incomplete xchar at the end; you eliminate that with `-trainling-garbage`, leaving a valid xchar string for processing (if the string starts with a complete xchar and only contains valid xchars). You prepend the eliminated chars to the next read block of chars so you do not miss any parts.

`x-width` (*xc-addr u – n*) xchar-ext

n is the number of monospace ASCII chars that take the same space to display as *xc-addr u* needs on a monospaced display.

`xc-width` (*xc – n*) xchar-ext "x-c-width"

xc has a width of *n* times the width of a normal fixed-width glyph.

`xhold` (*xc –*) xchar-ext "x-hold"

Used between `<<#` and `#>`. Prepend *xc* to the pictured numeric output string. We recommend that you use `holds` instead.

`xc,` (*xc –*) xchar "x-c-comma"

Reserve data space for *xc*, and store *xc* in that space.

`invalid-char` (*– xc*) gforth-experimental

Unicode code point returned for cases where the string does not contain a valid Unicode encoding. Current value: the Unicode replacement character U+FFFD.

`toupper` (*xc1 – xc2*) gforth-0.2 "toupper"

If *xc1* is a lower-case ASCII character, *xc2* is the equivalent upper-case character, otherwise *xc2* is *xc1*.

See also `xemit` (see Section 6.24.4 [Displaying characters and strings], page 209) and `xkey` (see Section 6.24.6 [Single-key input], page 211).

6.9.3 String and character literals

The nicest way to write a string literal is to write it as `"`*`STRING`*`"`. For these kinds of string literals as well as for `s\"` some sequences are not put in the resulting string as is, but are replaced as shown below. The sequences are mostly the same as in C (exceptions noted):

`\a`	7 `#bell` (alert)
`\b`	8 `#bs` (backspace)
`\e`	27 `#esc` (escape, not in C99)
`\f`	12 `#ff` (form feed)
`\l`	10 `#lf` (line feed, not in C)
`\m`	13 10 CR LF (not in C)
`\n`	sequence produced by `newline` (in C this produces a LF)
`\q`	34 `"` (double quote, not in C)
`\r`	13 `#cr` (carriage return)
`\t`	9 `#tab` (horizontal tab)
`\u`*`XXXX`*	Unicode code point *XXXX* (in hex); auto-merges surrogate pairs (not in Forth-2012 nor C)

`\UXXXXXXXX`
: Unicode code point *XXXXXXXX* (in hex, not in Forth-2012 nor C)

`\v`
: 11 VT (vertical tab)

`\xXX`
: raw byte (not code point) *XX* (in hex)

`\z`
: 0 NUL (not in C)

`\\`
: \

`\"`
: " (the \" does not terminate the string; not in Forth-2012)

`\XXX`
: raw byte; *XXX* is 1-3 octal digits (not in Forth-2012).

A \ before any other character is reserved.

Note that \xXX produces raw bytes, while \uXXXX and \UXXXXXXXX produce code points for the current encoding. E.g., if we use UTF-8 encoding and want to encode ä (code point U+00E4), you can write the letter ä itself, or write `\xc3\xa4` (the UTF-8 bytes for this code point), `\u00e4`, or `\U000000e4`.

The `"STRING"` syntax is non-standard, so for portability you may want to use one of the following words:

`s\"` (*Interpretation 'ccc"' – c-addr u*) core-ext,file-ext "s-backslash-quote"

Interpretation: Parse the string *ccc* delimited by a " (but not \"), and convert escaped characters as described above. Store the resulting string in newly allocated heap memory, and push its descriptor *c-addr u*.
Compilation `( 'ccc"' -- )`: Parse the string *ccc* delimited by a " (but not \"), and convert escaped characters as described above. Append the run-time semantics below to the current definition.
Run-time `( -- c-addr u )`: Push a descriptor for the resulting string.

`S"` (*Interpretation 'ccc"' – c-addr u*) core,file "s-quote"

Interpretation: Parse the string *ccc* delimited by a " (double quote). Store the resulting string in newly allocated heap memory, and push its descriptor *c-addr u*.
Compilation `( 'ccc"' -- )`: Parse the string *ccc* delimited by a " (double quote). Append the run-time semantics below to the current definition.
Run-time `( -- c-addr u )`: Push a descriptor for the parsed string.

All these ways of interpreting strings consume heap memory; normally you can just live with the string consuming memory until the end of the Gforth session, but if that is a problem for some reason, you can `free` the string when you no longer need it. Forth-2012 only guarantees two buffers of 80 characters each, so in standard programs you should assume that the string lives only until the next-but-one `s"`.

On the other hand, the compilation semantics of string literals of any form allocates the string in the dictionary, and you cannot `free` it, and it lives as long as the word it is compiled into (also in Forth-2012).

Likewise, You can get the code *xc* of a character *C* with `'C'`. This way has been standardized since Forth-2012. An older way to get it is to use one of the following words:

`char` (*'<spaces>ccc' – c*) core,xchar-ext

Skip leading spaces. Parse the string *ccc* and return *c*, the display code representing the first character of *ccc*.

`[char]` (*compilation '<spaces>ccc' – ; run-time – c*) core,xchar-ext "bracket-char"

Compilation: skip leading spaces. Parse the string *ccc*. Run-time: return *c*, the display code representing the first character of *ccc*. Interpretation semantics for this word are undefined.

You usually use `char` outside and `[char]` inside colon definitions, or you just use `'C'`.

Note that, e.g.,

```
"C" type
```

is (slightly) more efficient than

```
'C' xemit
```

because the latter converts the code point into a sequence of bytes and individually `emit`s them. Similarly, dealing with general characters is usually more efficient when representing them as strings rather than code points.

There are the following words for producing commonly-used characters and strings that cannot be produced with `S"` or `'C'`:

`newline` (– *c-addr u*) gforth-0.5 "newline"

String containing the newline sequence of the host OS

`bl` (– *c-char*) core "b-l"

c-char is the character value for a space.

`#tab` (– *c*) gforth-0.2 "number-tab"

`#lf` (– *c*) gforth-0.2 "number-l-f"

`#cr` (– *c*) gforth-0.2 "number-c-r"

`#ff` (– *c*) gforth-0.2 "number-f-f"

`#bs` (– *c*) gforth-0.2 "number-b-s"

`#del` (– *c*) gforth-0.2 "number-del"

`#bell` (– *c*) gforth-0.2 "number-bell"

`#esc` (– *c*) gforth-0.5 "number-esc"

`#eof` (– *c*) gforth-0.7 "number-e-o-f"

actually EOT (ASCII code 4 aka `^D`)

See also `invalid-char` (see Section 6.9.2 [Xchars and Unicode], page 89).

6.9.4 String words

Words that are used for memory blocks are also useful for strings, so for words that move, copy, and fill strings, see Section 6.8.8 [Memory Blocks], page 87. For words that display characters and strings, see Section 6.24.4 [Displaying characters and strings], page 209.

The following words work on previously existing strings:

`compare` (*c-addr1 u1 c-addr2 u2 – n*) string "compare"

Compare two strings lexicographically, based on the values of the bytes in the strings (i.e., case-sensitive and without locale-specific collation order). If they are equal, *n* is 0; if the string in *c_addr1 u1* is smaller, *n* is -1; if it is larger, *n* is 1.

`str=` (*c-addr1 u1 c-addr2 u2 – f*) gforth-0.6 "str-equals"

Bytewise equality

`str<` (*c-addr1 u1 c-addr2 u2 – f*) gforth-0.6 "str-less-than"

Bytewise lexicographic comparison.

`string-prefix?` (*c-addr1 u1 c-addr2 u2 – f*) gforth-0.6 "string-prefix-question"

Is *c-addr2 u2* a prefix of *c-addr1 u1*?

`string-suffix?` (*c-addr1 u1 c-addr2 u2 – f*) gforth-1.0 "string-suffix-question"

Is *c-addr2 u2* a suffix of *c-addr1 u1*?

`search` (*c-addr1 u1 c-addr2 u2 – c-addr3 u3 flag*) string

Search the string specified by *c-addr1, u1* for the string specified by *c-addr2, u2*. If *flag* is true: match was found at *c-addr3* with *u3* characters remaining. If *flag* is false: no match was found; *c-addr3, u3* are equal to *c-addr1, u1*.

`scan` (*c-addr1 u1 c – c-addr2 u2*) gforth-0.2 "scan"

Skip all characters not equal to c. The result starts with c or is empty. `Scan` is limited to single-byte (ASCII) characters. Use `search` to search for multi-byte characters.

`scan-back` (*c-addr u1 c – c-addr u2*) gforth-0.7

The last occurrence of *c* in *c-addr u1* is at *c-addr*+*u2*−1; if it does not occur, *u2*=0.

`skip` (*c-addr1 u1 c – c-addr2 u2*) gforth-0.2 "skip"

Skip all characters equal to c. The result starts with the first non-c character, or it is empty. `Scan` is limited to single-byte (ASCII) characters.

`$split` (*c-addr u char – c-addr u1 c-addr2 u2*) gforth-0.7 "string-split"

Divides a string *c-addr u* into two, with *char* as separator. *U1* is the length of the string up to, but excluding the first occurrence of the separator, *c-addr2 u2* is the part of the input string behind the separator. If the separator does not occur in the string, *u1*=*u*, *u2*=0 and *c-addr2*=*c-addr*+*u*.

`nosplit?` (*addr1 u1 addr2 u2 – addr1 u1 addr2 u2 flag*) gforth-experimental "nosplit-question"

Used on the result of `$split`, flag is true if and only if the separator does not occur in the input string of `$split`.

`-trailing` (*c_addr u1 – c_addr u2*) string "dash-trailing"

Adjust the string specified by *c-addr, u1* to remove all trailing spaces. *u2* is the length of the modified string.

`/string` (*c-addr1 u1 n – c-addr2 u2*) string "slash-string"

Adjust the string specified by *c-addr1, u1* to remove *n* characters from the start of the string.

`safe/string` (*c-addr1 u1 n – c-addr2 u2*) gforth-1.0 "safe-slash-string"

Adjust the string specified by *c-addr1, u1* to remove *n* characters from the start of the string. Unlike `/string`, `safe/string` removes at least 0 and at most *u1* characters.

`insert` (*c-addr1 u1 c-addr2 u2 –*) gforth-0.7

Move the contents of the buffer *c-addr2 u2* towards higher addresses by *u1* chars, and copy the string *c-addr1 u1* into the first *u1* chars of the buffer.

`delete` (*c-addr u u1 –*) gforth-0.7

In the memory block *c-addr u*, delete the first *u1* chars by copying the contents of the block starting at *c-addr+u1* there; fill the *u1* characters at the end of the block with blanks.

`cstring>sstring` (*c-addr – c-addr u*) gforth-0.2 "cstring-to-sstring"

C-addr is the start address of a zero-terminated string, *u* is its length.

The following words compare case-insensitively for ASCII characters, but case-sensitively for non-ASCII characters (like in lookup in wordlists).

`capscompare` (*c-addr1 u1 c-addr2 u2 – n*) gforth-0.7 "capscompare"

Compare two strings lexicographically, based on the values of the bytes in the strings, but comparing ASCII characters case-insensitively, and non-ASCII characters case-sensitively and without locale-specific collation order. If they are equal, *n* is 0; if the first string is smaller, *n* is -1; if the first string is larger, *n* is 1.

`capsstring-prefix?` (*c-addr1 u1 c-addr2 u2 – f*) gforth-1.0 "capsstring-prefix-question"

Like `string-prefix?`, but case-insensitive for ASCII characters: Is *c-addr2 u2* a prefix of *c-addr1 u1*?

`capssearch` (*c-addr1 u1 c-addr2 u2 – c-addr3 u3 flag*) gforth-1.0

Like `search`, but case-insensitive for ASCII characters: Search for *c-addr2 u2* in *c-addr1 u1*; *flag* is true if found.

The following words create or extend strings on the heap:

`s+` (*c-addr1 u1 c-addr2 u2 – c-addr u*) gforth-0.7 "s-plus"

c-addr u is a newly `allocate`d string that contains the concatenation of *c-addr1 u1* (first) and *c-addr2 u2* (second).

`append` (*c-addr1 u1 c-addr2 u2 – c-addr u*) gforth-0.7

C-addr u is the concatenation of *c-addr1 u1* (first) and *c-addr2 u2* (second). *c-addr1 u1* is an `allocate`d string, and `append` `resize`s it (possibly moving it to a new address) to accomodate *u* characters.

`>string-execute` (*... xt – ... c-addr u*) gforth-1.0 "to-string-execute"

Execute *xt* while the standard output (`type`, `emit`, and everything that uses them) is redirected to a string. The resulting string is *c-addr u*, which is in heap memory; it is the responsibility of the caller of `>string-execute` to `free` this string.

`$tmp` (*xt – addr u*) gforth-1.0 "string-t-m-p"

Like `>string-execute`, but the result is deallocated when `$tmp` is invoked the next time, and you must not `free` it yourself.

One could define `s+` using `>string-execute`, as follows:

```
: s+ ( c-addr1 u1 c-addr2 u2 -- c-addr u )
  [: 2swap type type ;] >string-execute ;
```

For concatenating just two strings `>string-execute` is inefficient, but for concatenating many strings `>string-execute` can be more efficient.

6.9.5 $string words

The following string library stores strings in ordinary cell-size variables (string handles). These handles contain a pointer to a cell-counted string allocated from the heap. The string library originates from bigFORTH.

Because there is only one permanent reference to the contents (the one in the handle), the string can be relocated or deleted without worrying about dangling references; this requires that the programmer uses references produced by, e.g., `$@` only for temporary purposes, i.e., these references are not passed out, e.g., as return values or stored in global memory, and words that may change the handle are not called while these references exist.

This library is complemented by the cell-pair representation: You use the $string words for variable strings which are cumbersome with the c-addr u representation. You use the cell-pair representation for processing (e.g., inspecting) strings while they do not change.

`$!` (*addr1 u $addr* –) gforth-0.7 "string-store"

stores a newly allocated string buffer at an address, frees the previous buffer if necessary.

`$@` (*$addr* – *addr2 u*) gforth-0.7 "string-fetch"

returns the stored string.

`$@len` (*$addr* – *u*) gforth-0.7 "string-fetch-len"

returns the length of the stored string.

`$!len` (*u $addr* –) gforth-0.7 "string-store-len"

changes the length of the stored string. Therefore we must change the memory area and adjust address and count cell as well.

`$+!len` (*u $addr* – *addr*) gforth-1.0 "string-plus-store-len"

make room for u bytes at the end of the memory area referenced by $addr; addr is the address of the first of these bytes.

`$del` (*$addr off u* –) gforth-0.7 "string-del"

Deletes *u* bytes at offset *off* bytes in the string *$addr*.

`$ins` (*addr1 u $addr off* –) gforth-0.7 "string-ins"

Inserts string *addr1 u* at offset *off* bytes in the string *$addr*.

`$+!` (*addr1 u $addr* –) gforth-0.7 "string-plus-store"

appends a string to another.

`c$+!` (*char $addr* –) gforth-1.0 "c-string-plus-store"

append a character to a string.

`$free` (*$addr* –) gforth-1.0 "string-free"

free the string pointed to by addr, and set addr to 0

`$init` (*$addr* –) gforth-1.0 "string-init"

store an empty string there, regardless of what was in before

`$iter` (*.. $addr char xt* – *..*) gforth-0.7 "string-iter"

Splits the string in *$addr* using *char* as separator. For each part, its descriptor *c-addr u* is pushed and *xt* `( ... c-addr u -- ... )` is executed.

`$over` (*addr u $addr off* –) gforth-1.0 "string-over"

Overwrite *u* bytes at offset *off* bytes in the string *$addr* with the string at *addr u*.

`$exec` (*xt $addr* –) gforth-1.0 "string-exec"

execute xt while the standard output (TYPE, EMIT, and everything that uses them) is appended to the string in *$addr*.

`$.` (*$addr* –) gforth-1.0 "string-dot"

print a string, shortcut

`$slurp` (*fid $addr* –) gforth-1.0 "string-slurp"

Read the file *fid* until the end (without closing it) and put the read data into the string at *$addr*.

`$slurp-file` (*c-addr u $addr* –) gforth-1.0 "string-slurp-file"

Put all the data in the file named *c-addr u* into the string at *$addr*.

`$+slurp` (*fid $addr* –) gforth-1.0 "string-plus-slurp"

Read the file *fid* until the end (without closing it) and append the read data to the string at *$addr*.

`$+slurp-file` (*c-addr u $addr* –) gforth-1.0 "string-plus+slurp-file"

Append all the data in the file named *c-addr u* to the string at *$addr*.

`$[]` (*u $[]addr – addr'*) gforth-1.0 "string-array"

Addr' is the address of the *u*th element of the string array *$[]addr*. The array is resized if needed.

`$[]!` (*c-addr u n $[]addr* –) gforth-1.0 "string-array-store"

Store string *c-addr y* into the string array *$[]addr* at index *n*. The array is resized if needed.

`$[]+!` (*c-addr u n $[]addr* –) gforth-1.0 "string-array-plus-store"

Append the string *c-addr u* to the string at index *n*. The array is resized if needed. Don't confuse this with `$+[]!`.

`$+[]!` (*c-addr u $[]addr* –) gforth-1.0 "string-append-array"

Store the string *c-addr u* as the new last element of string array *$[]addr*. The array is resized if needed.

`$[]@` (*n $[]addr – addr u*) gforth-1.0 "string-array-fetch"

fetch a string from array index *n* — return the zero string if empty, and don't accidentally grow the array.

`$[]#` (*$[]addr – len*) gforth-1.0 "string-array-num"

return the number of elements in an array

`$[]map` (*$[]addr xt* –) gforth-1.0 "string-array-map"

execute *xt* for all elements of the string array *$[]addr*. xt is *(addr u –)*, getting one string at a time

`$[]slurp` (*fid $[]addr* –) gforth-1.0 "string-array-slurp"

slurp a file *fid* line by line into a string array *$[]addr*

`$[]slurp-file` (*addr u $[]addr* –) gforth-1.0 "string-array-slurp-file"

slurp a named file *addr u* line by line into a string array *$[]addr*

`$[].` (*$[]addr* –) gforth-1.0 “string-array-dot”

print all array entries

`$[]free` (*$[]addr* –) gforth-1.0 “string-array-free”

$[]addr contains the address of a cell-counted string that contains the addresses of a number of cell-counted strings; $[]free frees these strings, frees the array, and sets addr to 0

`$Variable` ("*name*" –) gforth-1.0 “string-variable”

Defines a string variable whose content is preserved across savesystem

`$[]Variable` ("*name*" –) gforth-1.0 “string-array-variable”

Defines a string array variable whose content is preserved across savesystem

6.9.6 Internationalization and localization

A program may need to communicate with its user in the user's language, and it may have users with different languages. We do not want to produce one version of the program for each language, so we write one internationalized program that can use localization features to communicate in the user's language.

Apart from the words mentioned here, you will probably want to use Unicode to write the localized strings; you probably do not need to use the xchar words (see Section 6.9.2 [Xchars and Unicode], page 89) in that context, but they are there if you need them.

Moreover, you may need to put placeholders (e.g., for amounts of currency) in localized strings that you substitute for the real values later. The word `substitute` and friends (see Section 6.9.7 [Substitute], page 101) have been designed for that purpose.

The basic idea in an internationalized program is that instead of, e.g.,

```
." Please enter your name:"
```

you write

```
L" Please enter your name:" locale@ type
```

In the following examples, we use the locales defined with

```
locale: de    \ German (generic)
locale: de_AT \ German (as used in Austria)
locale: de_CH \ German (as used in Switzerland)
locale: de_DE \ German (as used in Germany)
```

In addition, there are the locales `program` and `default`.

You can activate a locale, i.e., make it the current locale, with, e.g.,

```
locales:de_AT
```

Note that, unlike most Forth words, locales are case-sensitive, so `locales:de_at` would not work.

In the following examples, we use the following code to output localized strings, after first setting the strings (shown later):

```
L" Please enter your name:" locale@ cr type
L" cauliflower" locale@ cr type
L" street" locale@ cr type
L" something else" locale@ cr type
```

```
L" bank [geography]" locale@ cr type
L" bank [finance]" locale@ cr type
```

When the current locale is `de_AT`, the output is:

```
Bitte geben Sie Ihren Namen ein:
Karfiol
Straße
something else
Ufer
Bank
```

When the current locale is `de_CH`, the output is:

```
Bitte geben Sie Ihren Namen ein:
Blumenkohl
Strasse
something else
Ufer
Bank
```

In the localization data used for this example (see below), most of these localizations are inherited from the locale `de`, with the only `de_AT`-specific localization being "Karfiol" and the only `de_CH`-specific localization being "Strasse". There is no `de` and no `default` localization for "something else", so the text in the string `L" something else"` is used (we always get that if we use the `program` locale).

So there is a sequence of fallbacks for looking up localizations: For the general locale *`X_Y`*, the first fallback is to *`X`*, next to `default`, and finally to `program`. If the current locale's name contains no underscore, the fallback sequence starts at `default`.

The locale `default` is a fallback if there is no more specific localization (typically, if the localization is missing), as for "something else". In many cases (as for "something else") there is no `default` localization, and the fallback continues to the `L"` string. But in some cases (e.g., in `L" bank [geography]"` and `L" bank [finance]"`) these strings are too developer-oriented, and we put a user-oriented string ("bank" for both) in the `default` locale.

So how do we provide the localized string for a given `program` string? A simple way is to use `locale!`, e.g.:

```
locales:de_AT
"Karfiol" L" cauliflower" locale!
```

However, defining localizations for all `L"` strings with `locale!` is cumbersome and error-prone (you have to use the exact same spelling for `L"` strings with `locale!` as before `locale@` in the code).

A better alternative is to use `locale-csv-out` after loading the program to save all its `L"` strings and all the existing localizations to a CSV (comma-separated values) file. The first column of this file contains the `L"` strings, the others the various localizations for that string. E.g., the CSV file for the example in this section contains:

```
"program","default","de","de_AT","de_CH","de_DE"
"bank [finance]","bank","Bank","","",""
"bank [geography]","bank","Ufer","","",""
```

```
"Please enter your name:","","Bitte geben Sie Ihren Namen ein:","","",""
"cauliflower","","Blumenkohl","Karfiol","",""
"street","","Straße","","Strasse",""
"something else","","","","",""
```

The first line contains the names of the locales. Many entries contain empty strings; in that case, there is no localization for the `L"` string for the locale of that column (and `locale@` will use the next fallback that is not empty.

The way to add localizations is to edit the CSV file. This is not easy with a simple text editor if there are many columns. One way to work around this is to use a spreadsheet program or an editor that has good CSV support. Another way is to distribute the localizations across several files (which is also better for letting several people work on the localizations at the same time). E.g., one file could contain the localizations for `de` and its variants, while another could contain the localizations for `fr` (French) and its variants.

Once a localization has been edited into a CSV file, one can load the CSV file with `locale-csv`. All the locales mentioned in the CSV file will be defined automatically; if you use `locale-csv`, do not use `locale:` afterwards for the locales in the CSV file.

The lines in the CSV file are ordered by the order in which the `L"` strings are found by Gforth. If the CSV file is generated just from the `L"` strings in the program, this order may be helpful for producing the localizations, because related `L"` strings are grouped together. If, during maintenance, new `L"` strings are added, and you first load the CSV file, then the program, and then write out a CSV file, you will find the new `L"` strings (for localizing) at the end of the new CSV file.

A *locale-string identifier* (*lsid*) is an opaque token that occupies a cell, and it identifies the `L"` string.

`L"` (*Interpretation* `"`*string*`<">"` – *lsid; Compilation* `"`*string*`<">"` –) gforth-experimental "l-quote"

At text interpretation time, parse *string*. At run-time, push the *lsid* associated with *string*. Each string has a unique lsid. If no lsid for the string exists yet, a new one is created. If an lsid for the string exists already, that lsid is returned. This means that one can refer to and use the same lsid with `L"` in different locations in the source code, by uing the same *string*. If you want two different lsids (e.g., because you refer to two different concepts), but would use the same user-centric text in `L"`, append `" [specifier]"` to the text, e.g. `L" bank [finance]"` or `L" bank [geography]"`. You may then want to add a user-centric non-unique `default` localization (e.g., "bank").

`locales` (–) gforth-experimental

This case-sensitive vocabulary contains the locales. Typical use: `locales:`*`locale`*.

`native@` (*lsid* – *c-addr u*) gforth-experimental "native-fetch"

c-addr u is the `L"` string for *lsid* (i.e., the text-interpretation argument of `L"`).

`locale@` (*lsid* – *c-addr u*) gforth-experimental "locale-fetch"

c-addr u is the localized string for *lsid* in the current locale. If no localized string is found in the current locale with a name of the form `X_Y`, *lsid* is looked up in locale `X`. If no localized string is found in the locale `X`, *lsid* is looked up in the locale `default`. If no localized string is found in the locale `default`, *lsid* is looked up in the locale `program` (i.e., *c-addr u* is the text-interpretation argument of `L"`).

`program` (–) gforth-experimental

`locales:program` becomes the current locale. When this locale is current, `locale@` produces the string used for identifying the lsid (i.e., the string parsed by `L"`). This locale is useful for development: One can see which lsid is used in which context.

`default` (–) gforth-experimental

`locales:default` is the default locale if the user has not set one. Most lsids don't have a specific default string, so fallback to the `program` locale happens. But if you have a developer-centric program string that is inappropriate for end users (in particular, if the program string contains an extra specifier), you will prefer to define a user-centric string in the default locale.

`locale-csv-out` (`"`*name*`"` –) gforth-experimental "locale-csv-out"

Create file *name* and write the locale database to this in CSV format.

`locale-csv` (`"`*name*`"` –) gforth-experimental "locale-csv"

Import comma-separated value (CSV) table into locales. The first line contains the locale names (column headers). The `program` locale must be leftmost. Fallback locales like `de` must precede more specific locales like `de_AT`. The other lines contain the `L"` string (first column) and the corresponding localizations. Each column contains the localizations for a specific locale. Empty entries mean that this locale does not define a localization for this `L"` string, resulting in using the localization from a fallback locale instead.

`.locale-csv` (–) gforth-experimental "dot-locale-csv"

Write the locale database in CSV format to the user output device.

`locale!` (*addr u lsid* –) gforth-experimental "locale-store"

After executing `locale!`, the localized string for *lsid* in the current locale is *c-addr u*.

`Locale:` (`"`*name*`"` –) gforth-experimental "Locale-colon"

Defines a new locale *l* with name *name* in `locales`.
name execution: (–) *l* becomes the current locale.
For locales with names of the form `X_Y`, define `X` first in order to establish `X` as a fallback for `X_Y`.

6.9.7 Substitute

This is a simple text macro replacement facility; it is particularly useful for working with localized strings (see Section 6.9.6 [Internationalization and localization], page 98). In other contexts, using `>string-execute` (see Section 6.9.4 [String words], page 93) is often easier.

When using `substitute`, strings in the form `"text%macro%text"` (with an arbitrary number of macros) are processed, and the macro variables enclosed in `'%'` are replaced by their associated strings. Two consecutive `%` are replaced by one `%`. Macros are defined in a specific wordlist, and return a string upon execution; the standard defines only one way to declare macros, `replaces`, which creates a macro that just returns a string.

`macros-wordlist` (– *wid*) gforth-experimental

wordlist for string replacement macros

`replaces` (*addr1 len1 addr2 len2* –) string-ext

create a macro with name *addr2 len2* and content *addr1 len1*. If the macro already exists, just change the content.

`replacer:` ("*name*" –) gforth-experimental "replacer-colon"

Start a colon definition *name* in `macros-wordlist`, i.e. this colon definition is a macro. It must have the stack effect *(– addr u)*.

`.substitute` (*addr1 len1 – n / ior*) gforth-experimental "dot-substitute"

substitute all macros in text *addr1 len1* and print the result. *n* is the number of substitutions or, if negative, a throwable *ior*.

`$substitute` (*addr1 len1 – addr2 len2 n/ior*) gforth-experimental "string-substitute"

substitute all macros in text *addr1 len1*. *n* is the number of substitutions, if negative, it's a throwable *ior*, *addr2 len2* the result.

`substitute` (*addr1 len1 addr2 len2 – addr2 len3 n/ior*) string-ext

substitute all macros in text *addr1 len1*, and copy the result to *addr2 len2*. *n* is the number of substitutions or, if negative, a throwable *ior*, *addr2 len3* the result.

`unescape` (*addr1 u1 dest – dest u2*) string-ext

double all delimiters in *addr1 u1*, so that substitute will result in the original text. Note that the buffer *dest* does not have a size, as in worst case, it will need just twice as many characters as *u1*. *dest u2* is the resulting string.

`$unescape` (*addr1 u1 – addr2 u2*) gforth-experimental "string-unescape"

same as `unescape`, but creates a temporary destination string with `$tmp`.

6.9.8 Counted string words

Counted strings store the length as byte at the address pointed to, followed by the bytes of the string. Their possible length is severely limited, and you cannot create a substring in-place without destroying the input string. Therefore we recommend against using counted strings. Nevertheless, if you have to deal with counted strings, here are some words for that:

`count` (*c-addr1 – c-addr2 u*) core "count"

c-addr2 is the first character and *u* the length of the counted string at *c-addr1*.

The following word has no useful interpretation semantics (unlike `s"`) and no interpretive counterpart (unlike `[char]`), so you should use it only inside colon definitions (if at all):

`C"` (*compilation "ccc<quote>" – ; run-time – c-addr*) core-ext "c-quote"

Compilation: parse a string *ccc* delimited by a `"` (double quote). At run-time, return *c-addr* which specifies the counted string *ccc*. Interpretation semantics are undefined.

`place` (*c-addr1 u c-addr2 –*) gforth-experimental "place"

Create a counted string of length *u* at *c-addr2* and copy the string *c-addr1 u* into that location. Up to 256 bytes starting at *c-addr2* will be written, so make sure that the buffer at *c-addr2* has that much space (or check that *u*+1 does not exceed the buffer size before calling `place`)

`string,` (*c-addr u –*) gforth-0.2 "string-comma"

Reserve *u*+1 bytes of dictionary space and store the string *c-addr u* there as counted string.

6.10 Control Structures

Control structures in Forth cannot be used interpretively, only in a colon definition[7]. We do not like this limitation, but have not seen a satisfying way around it yet, although many schemes have been proposed.

6.10.1 Selection

```
flag IF
  code
THEN
```

If *flag* is non-zero (as far as `IF` etc. are concerned, a non-zero cell represents truth), *code* is executed.

You may wonder why `then` ends an `if` construct, which is at odds with the usage of `then` in some other programming languages, and with the idiom "if ... then ..." in English. According to *Webster's New Encyclopedic Dictionary*, *then (adv.)* has the following meanings:

> ... 2b: following next after in order ... 3d: as a necessary consequence (if you were there, then you saw them).

Forth's `then` has the meaning 2b, whereas `THEN` in Pascal and many other programming languages has the meaning 3d. If you do not like this usage of `then`, Gforth (but not Standard Forth) also has `endif`, which can be used in its place. Adding `ENDIF` to a system that only supplies `THEN` is simple:

```
: ENDIF   POSTPONE then ; immediate
```

```
flag IF
  code1
ELSE
  code2
THEN
```

If *flag* is true, *code1* is executed, otherwise *code2* is executed.

Gforth also provides the words `?DUP-IF` and `?DUP-0=-IF`, so you can avoid using `?dup`. Using these alternatives is also more efficient than using `?dup`. Definitions in Standard Forth for `ENDIF`, `?DUP-IF` and `?DUP-0=-IF` are provided in `compat/control.fs`.

```
x
CASE
  x1 OF code1 ENDOF
  x2 OF code2 ENDOF
  ...
  ( x ) default-code ( x )
ENDCASE ( )
```

Executes the first *codei*, where the *xi* is equal to *x*. If no *xi* matches, the optional *default-code* is executed. The optional default case can be added by simply writing the code after

[7] To be precise, in Standard Forth the control-flow words have no interpretation semantics, and in Gforth the interpretation semantics of the control-flow words are not useful for interpretive control flow (see Section 6.14 [Interpretation and Compilation Semantics], page 150).

the last `ENDOF`. It may use *x*, which is on top of the stack, but must not consume it. The value *x* is consumed by this construction (either by an `OF` that matches, or by the `ENDCASE`, if no OF matches). Example:

```
: num-name ( n -- c-addr u )
 case
   0 of s" zero " endof
   1 of s" one "  endof
   2 of s" two "  endof
   \ default case:
   s" other number"
   rot \ get n on top so ENDCASE can drop it
 endcase ;
```

Programming style note:

To keep the code understandable, you should ensure that you change the stack in the same way (wrt. number and types of stack items consumed and pushed) on all paths through a selection structure.

6.10.2 General Loops

```
BEGIN
  code1
  flag WHILE
    code2
REPEAT
```

code1 is executed and *flag* is computed. If it is true, *code2* is executed and the loop is restarted; If *flag* is false, execution continues after the `REPEAT`.

```
BEGIN
  code
  flag
UNTIL
```

code is executed. The loop is restarted if `flag` is false.

Programming style note:

To keep the code understandable, a complete iteration of the loop should not change the number and types of the items on the stacks.

```
BEGIN
  code
AGAIN
```

This is an endless loop. You can leave it by leaving the enclosing colon definition with `exit` or `throw`, or with `while` (see Section 6.10.4 [General loops with multiple exits], page 109).

6.10.3 Counted Loops

The basic counted loop is:

```
limit start ?DO
  body
```

```
LOOP
```

This performs one iteration for every integer, starting from *start* and up to, but excluding *limit*. The counter, or *index*, can be accessed with `i`. For example, the loop:

```
10 0 ?DO
  i .
LOOP
```

prints `0 1 2 3 4 5 6 7 8 9`

The index of the innermost loop can be accessed with `i`, the index of the next loop with `j`, and the index of the third loop with `k`.

You can access the limit of the innermost loop with `i'` and `i'-i` with `delta-i`. E.g., running

```
: foo 7 5 ?do cr i . i' . delta-i . loop ;
```

prints

```
5 7 2
6 7 1
```

The loop control data are kept on the return stack, so there are some restrictions on mixing return stack accesses and counted loop words. In particuler, if you put values on the return stack outside the loop, you cannot read them inside the loop[8]. If you put values on the return stack within a loop, you have to remove them before the end of the loop and before accessing the index of the loop.

There are several variations on the counted loop:

- `LEAVE` leaves the innermost counted loop immediately; execution continues after the associated `LOOP` or `NEXT`. For example:

  ```
  10 0 ?DO  i DUP . 3 = IF LEAVE THEN LOOP
  ```

  prints `0 1 2 3`

- `UNLOOP` prepares for an abnormal loop exit, e.g., via `EXIT`. `UNLOOP` removes the loop control parameters from the return stack so `EXIT` can get to its return address. For example:

  ```
  : demo 10 0 ?DO i DUP . 3 = IF UNLOOP EXIT THEN LOOP ." Done" ;
  ```

  prints `0 1 2 3`

- If *start* is greater than *limit*, a `?DO` loop is entered (and `LOOP` iterates until they become equal by wrap-around arithmetic). This behaviour is usually not what you want. Therefore, Gforth offers `+DO` and `U+DO` (as replacements for `?DO`), which do not enter the loop if *start* is greater than *limit*; `+DO` is for signed loop parameters, `U+DO` for unsigned loop parameters.

- `?DO` can be replaced by `DO`. `DO` always enters the loop, independent of the loop parameters. Do not use `DO`, even if you know that the loop is entered in any case. Such knowledge tends to become invalid during maintenance of a program, and then the `DO` will make trouble.

[8] Not in a way that is portable.

- `LOOP` can be replaced with *n* `+LOOP`; this updates the index by *n* instead of by 1. The loop is terminated when the border between *limit-1* and *limit* is crossed. E.g.:

  ```
  4 0 +DO  i .  2 +LOOP
  ```

  prints `0 2`

  ```
  4 1 +DO  i .  2 +LOOP
  ```

  prints `1 3`

- The behaviour of *n* `+LOOP` is peculiar when *n* is negative:

  ```
  -1 0 ?DO  i .  -1 +LOOP
  ```

  prints `0 -1`

  ```
  0 0 ?DO  i .  -1 +LOOP
  ```

  prints nothing.

  We recommend not combining `?DO` with `+LOOP`. Gforth offers several alternatives:

  If you want `-1 +LOOP`'s behaviour of including an iteration where `I`=limit, start the loop with `-[DO` or `U-[DO` (where the `[` is inspired by the mathematical notation for inclusive ranges, e.g., [1,n]):

  ```
  -1 0 -[DO  i .  -1 +LOOP
  ```

  prints `0 -1`.

  ```
  0 0 -[DO  i .  -1 +LOOP
  ```

  prints `0`.

  ```
  0 -1 -[DO  i .  -1 +LOOP
  ```

  prints nothing.

  If you want to exclude the limit, you instead use `1 -LOOP` (or generally *u* `-LOOP`) and start the loop with `?DO`, `-DO` or `U-DO`. `-LOOP` terminates the loop when the border between *limit+1* and *limit* is crossed. E.g.:

  ```
  -2 0 -DO  i .  1 -LOOP
  ```

  prints `0 -1`

  ```
  -1 0 -DO  i .  1 -LOOP
  ```

  prints `0`

  ```
  0 0 -DO  i .  1 -LOOP
  ```

  prints nothing.

  Unfortunately, `+DO`, `U+DO`, `-DO`, `U-DO` and `-LOOP` are not defined in Standard Forth. However, an implementation for these words that uses only standard words is provided in `compat/loops.fs`.

- A common task is to iterate over the elements of an array, forwards or backwards. Iterating over the addresses of the elements has two benefits: It avoids the need to keep the start address of the array around, reducing the data stack load; and it avoids the need to perform address computations in every iteration. The disadvantage is that, starting with the usual array representations *addr uelems* or *addr ubytes*, some processing is required to produce a start and limit address. Gforth has `bounds` for getting there from the *addr ubytes* representation, so you can write a forward loop through a cell array `v` as:

  ```
  create v 1 , 3 , 7 ,
  ```

```
: foo v 3 cells bounds U+DO i @ . cell +LOOP ;
foo
```

which prints `1 3 7`. Preprocessing the inputs for walking backwards is more involved, so Gforth provide a loop construct of the form `MEM-DO...LOOP` that does it for you: It takes an array in *addr ubytes* representation and the element size, and iterates over the addresses of the elements in backwards order:

```
create v 1 , 3 , 7 ,
: foo1 v 3 cell array>mem MEM-DO i @ . LOOP ;
foo1
```

This prints `7 3 1`. `ARRAY>MEM` converts the *addr uelems uelemsize* representation into the *addr ubytes uelemsize* representation expected by `MEM-DO`. This loop is finished with `LOOP` which decrements by *uelemsize* when it finishes a `MEM-DO`.

Gforth also adds `MEM+DO` for completeness. It takes the same parameters as `MEM-DO`, but walks forwards through the array:

```
create v 1 , 3 , 7 ,
: foo2 v 3 cell array>mem MEM+DO i @ . LOOP ;
foo2
```

prints `1 3 7`.

- Another counted loop is:

```
n
FOR
  body
NEXT
```

This is the preferred loop of native code compiler writers who are too lazy to optimize `?DO` loops properly. This loop structure is not defined in Standard Forth. In Gforth, this loop iterates *n+1* times; `i` produces values starting with *n* and ending with 0. Other Forth systems may behave differently, even if they support `FOR` loops. To avoid problems, don't use `FOR` loops.

The counted-loop words are:

`?DO` (*compilation – do-sys ; run-time w1 w2 – | loop-sys*) core-ext "question-do"

See Section 6.10.3 [Counted Loops], page 104.

`+DO` (*compilation – do-sys ; run-time n1 n2 – | loop-sys*) gforth-0.2 "plus-do"

See Section 6.10.3 [Counted Loops], page 104.

`U+DO` (*compilation – do-sys ; run-time u1 u2 – | loop-sys*) gforth-0.2 "u-plus-do"

See Section 6.10.3 [Counted Loops], page 104.

`bounds` (*u1 u2 – u3 u1*) gforth-0.2 "bounds"

Given a memory block represented by starting address *addr* and length *u* in aus, produce the end address *addr+u* and the start address in the right order for `u+do` or `?do`.

`-[do` (*compilation – do-sys ; run-time n1 n2 – | loop-sys*) gforth-experimental "minus-bracket-do"

Start of a counted loop with negative stride; Skips the loop if *n2<n1*; such a counted loop ends with `+loop` where the increment is negative; it runs as long as `I>=`*n1*.

`u-[do` (*compilation – do-sys ; run-time u1 u2 –* `|` *loop-sys*) gforth-experimental "u-minus-bracket-do"

Start of a counted loop with negative stride; Skips the loop if *u2*<*u1*; such a counted loop ends with `+loop` where the increment is negative; it runs as long as `I>=`*u1*.

`-DO` (*compilation – do-sys ; run-time n1 n2 –* `|` *loop-sys*) gforth-0.2 "minus-do"

See Section 6.10.3 [Counted Loops], page 104.

`U-DO` (*compilation – do-sys ; run-time u1 u2 –* `|` *loop-sys*) gforth-0.2 "u-minus-do"

See Section 6.10.3 [Counted Loops], page 104.

`array>mem` (*uelements uelemsize – ubytes uelemsize*) gforth-experimental "array-to-mem"

*ubytes=uelements*uelemsize*

`mem+do` (*compilation – w xt do-sys; run-time addr ubytes* `+`*nstride –*) gforth-experimental "mem-plus-do"

Starts a counted loop that starts with `I` as *addr* and then steps upwards through memory with *nstride* wide steps as long as `I<`*addr*`+`*ubytes*. Must be finished with `loop`.

`mem-do` (*compilation – w xt do-sys; run-time addr ubytes* `+`*nstride –*) gforth-experimental "mem-minus-do"

Starts a counted loop that starts with `I` as *addr*`+`*ubytes-nstride* and then steps backwards through memory with *-nstride* wide steps as long as `I>=`*addr*. Must be finished with `loop`.

`DO` (*compilation – do-sys ; run-time w1 w2 – loop-sys*) core

See Section 6.10.3 [Counted Loops], page 104.

`FOR` (*compilation – do-sys ; run-time u – loop-sys*) gforth-0.2

See Section 6.10.3 [Counted Loops], page 104.

`LOOP` (*compilation do-sys – ; run-time loop-sys1 –* `|` *loop-sys2*) core

Finish a counted loop. If started with `mem+do` or `mem-do`, the stride (increment) and terminating condition is given by these words, otherwise the stride is 1 and the loop ends when the limit is reached (the last iteration has `i`=limit-1).

`+LOOP` (*compilation do-sys – ; run-time loop-sys1 n –* `|` *loop-sys2*) core "plus-loop"

See Section 6.10.3 [Counted Loops], page 104.

`-LOOP` (*compilation do-sys – ; run-time loop-sys1 u –* `|` *loop-sys2*) gforth-0.2 "minus-loop"

See Section 6.10.3 [Counted Loops], page 104.

`NEXT` (*compilation do-sys – ; run-time loop-sys1 –* `|` *loop-sys2*) gforth-0.2

See Section 6.10.3 [Counted Loops], page 104.

`i` (*R:n – R:n n*) core "i"

n is the index of the innermost counted loop.

`j` (*R:n R:w1 R:w2 – n R:n R:w1 R:w2*) core "j"

n is the index of the next-to-innermost counted loop.

`k` (*R:n R:w1 R:w2 R:w3 R:w4 – n R:n R:w1 R:w2 R:w3 R:w4*) gforth-0.3 "k"

n is the index of the third-innermost counted loop.

`i'` (*R:w R:w2 – R:w R:w2 w*) gforth-0.2 "i-tick"

The limit of the innermost counted loop

`delta-i` (*r:ulimit r:u – r:ulimit r:u u2*) gforth-1.0 "delta-i"

u2=`I'-I` (difference between limit and index).

`LEAVE` (*compilation – ; run-time loop-sys –*) core

See Section 6.10.3 [Counted Loops], page 104.

`?LEAVE` (*compilation – ; run-time f | f loop-sys –*) gforth-0.2 "question-leave"

See Section 6.10.3 [Counted Loops], page 104.

`unloop` (*R:w1 R:w2 –*) core "unloop"

`DONE` (*compilation do-sys – ; run-time –*) gforth-0.2

resolves all LEAVEs up to the do-sys

The standard does not allow using `CS-PICK` and `CS-ROLL` on *do-sys*. Gforth allows it, except for the do-sys produced by `MEM+DO` and `MEM-DO`, but it's your job to ensure that for every `?DO` etc. there is exactly one `UNLOOP` on any path through the definition (`LOOP` etc. compile an `UNLOOP` on the fall-through path). Also, you have to ensure that all `LEAVE`s are resolved (by using one of the loop-ending words or `DONE`).

6.10.4 General loops with multiple exits

For counted loops, you can use `leave` in several places. For `begin` loops, you have the following options:

Use `exit` (possibly several times) in the loop to leave not just the loop, but the whole colon definition. E.g.,:

```
: foo
  begin
    condition1 while
      condition2 if
        exit-code2 exit then
      condition3 if
        exit-code3 exit then
    ...
  repeat
  exit-code1 ;
```

The disadvantage of this approach is that, if you want to have some common code afterwards, you either have to wrap `foo` in another definition that contains the common code, or you have to call the common code several times, from each exit-code.

Another approach is to use several `while`s in a `begin` loop. You have to append a `then` behind the loop for every additional `while`. E.g.,;

```
begin
  condition1 while
    condition2 while
      condition3 while
again then then then
```

Here I used `again` at the end of the loop so that I would have a `then` for each `while`; `repeat` would result in one less `then`, but otherwise the same behaviour. For an explanation of why this works, See Section 6.10.6 [Arbitrary control structures], page 112.

We can have common code afterwards, but, as presented above, we cannot have different exit-codes for the different exits. You can have these different exit-codes, as follows:

```
begin
  condition1 while
    condition2 while
      condition3 while
again then exit-code3
else exit-code2 then
else exit-code1 then
```

This is relatively hard to comprehend, because the exit-codes are relatively far from the exit conditions (it does not help that we are not used to such control structures, either). The following extended `case` does not have this problem.

6.10.5 General control structures with `case`

Gforth provides an extended `case` that solves the problems of the multi-exit loops discussed above, and offers additional options. You can find a portable implementation of this extended `case` in `compat/caseext.fs`.

There are three additional words in the extension. The first is `?of` which allows general tests (rather than just testing for equality) in a `case`; e.g.,

```
: sgn ( n -- -1|0|1 )
  ( n ) case
    dup 0 < ?of drop -1 endof
    dup 0 > ?of drop 1  endof
    \ otherwise leave the 0 on the stack
  0 endcase ;
```

Note that `endcase` drops a value, which works fine much of the time with `of`, but usually not with `?of`, so we leave a 0 on the stack for `endcase` to drop. The n that is passed into `sgn` is also 0 if neither `?of` triggers, and that is then passed out.

The second additional word is `next-case`, which allows turning `case` into a loop. Our triple-exit loop becomes:

```
case
  condition1 ?of exit-code1 endof
  condition2 ?of exit-code2 endof
  condition3 ?of exit-code3 endof
  ...
next-case
common code afterwards
```

As you can see, this solves both problems of the variants discussed above (see Section 6.10.4 [General loops with multiple exits], page 109). Note that `next-case` does not drop a value, unlike `endcase`.[9]

[9] The name `next-case` has a `-`, unlike the other `case` words, because VFX Forth has a `next-case` that works like Gforth's `next-case`, but also contains a `nextcase` that drops a value; in VFX you need to pair `next-case` with `begincase`, however.

The last additional word is `contof`, which is used instead of `endof` and starts the next iteration instead of leaving the loop. This can be used in ways similar to Dijkstra's guarded command *do*, e.g.:

```
: gcd ( n1 n2 -- n )
    case
        2dup > ?of tuck - contof
        2dup < ?of over - contof
    endcase ;
```

Here the two `?of`s have different ways of continuing the loop; when neither `?of` triggers, the two numbers are equal and are the gcd. `Endcase` drops one of them, leaving the other as n.

You can also combine these words. Here's an example that uses each of the `case` words once, except `endcase`:

```
: collatz ( u -- )
    \ print the 3n+1 sequence starting at u until we reach 1
    case
        dup .
        1 of endof
        dup 1 and ?of 3 * 1+ contof
        2/
    next-case ;
```

This example keeps the current value of the sequence on the stack. If it is 1, the `of` triggers, drops the value, and leaves the `case` structure. For odd numbers, the `?of` triggers, computes 3n+1, and starts the next iteration with `contof`. Otherwise, if the number is even, it is divided by 2, and the loop is restarted with `next-case`.

The `case` words are:

`case` (*compilation – case-sys ; run-time –*) core-ext

Start a `case` structure.

`endcase` (*compilation case-sys – ; run-time x –*) core-ext "end-case"

Finish the `case` structure; drop x, and continue behind the `endcase`. Dropping x is useful in the original `case` construct (with only `of`s), but you may have to supply an x in other cases (especially when using `?of`).

`next-case` (*compilation case-sys – ; run-time –*) gforth-1.0

Restart the `case` loop by jumping to the matching `case`. Note that `next-case` does not drop a cell, unlike `endcase`.

`of` (*compilation – of-sys ; run-time x1 x2 – \| x1*) core-ext

If x1=x2, continue (dropping both); otherwise, leave x1 on the stack and jump behind `endof` or `contof`.

`?of` (*compilation – of-sys ; run-time f –*) gforth-1.0 "question-of"

If f is true, continue; otherwise, jump behind `endof` or `contof`.

`endof` (*compilation case-sys1 of-sys – case-sys2 ; run-time –*) core-ext "end-of"

Exit the enclosing `case` structure by jumping behind `endcase`/`next-case`.

`contof` (*compilation case-sys1 of-sys – case-sys2 ; run-time –*) gforth-1.0 "cont-of"

Restart the `case` loop by jumping to the enclosing `case`.

Internally, *of-sys* is an `orig`; and *case-sys* is a cell and some stack-depth information, 0 or more `orig`s, and a `dest`.

6.10.6 Arbitrary control structures

Standard Forth permits and supports using control structures in a non-nested way. Information about incomplete control structures is stored on the control-flow stack. This stack may be implemented on the Forth data stack, and this is what we have done in Gforth.

An *orig* entry represents an unresolved forward branch, a *dest* entry represents a backward branch target. A few words are the basis for building any control structure possible (except control structures that need storage, like calls, coroutines, and backtracking).

`IF` (*compilation – orig ; run-time f –*) core

At run-time, if *f*=0, execution continues after the `THEN` (or `ELSE`) that consumes the *orig*, otherwise right after the `IF` (see Section 6.10.1 [Selection], page 103).

`AHEAD` (*compilation – orig ; run-time –*) tools-ext

At run-time, execution continues after the `THEN` that consumes the *orig*.

`THEN` (*compilation orig – ; run-time –*) core

The `IF`, `AHEAD`, `ELSE` or `WHILE` that pushed *orig* jumps right after the `THEN` (see Section 6.10.1 [Selection], page 103).

`BEGIN` (*compilation – dest ; run-time –*) core

The `UNTIL`, `AGAIN` or `REPEAT` that consumes the *dest* jumps right behind the `BEGIN` (see Section 6.10.2 [General Loops], page 104).

`UNTIL` (*compilation dest – ; run-time f –*) core

At run-time, if *f*=0, execution continues after the `BEGIN` that produced *dest*, otherwise right after the `UNTIL` (see Section 6.10.2 [General Loops], page 104).

`AGAIN` (*compilation dest – ; run-time –*) core-ext

At run-time, execution continues after the `BEGIN` that produced the *dest* (see Section 6.10.2 [General Loops], page 104).

`CS-PICK` (*dest0/orig0 dest1/orig1 ... destu/origu u – ... dest0/orig0*) tools-ext “c-s-pick”

`CS-ROLL` (*destu/origu .. dest0/orig0 u – .. dest0/orig0 destu/origu*) tools-ext “c-s-roll”

`CS-DROP` (*dest/orig –*) gforth-1.0

The Standard words `cs-pick` and `cs-roll` allow you to manipulate the control-flow stack in a portable way. Without them, you would need to know how many stack items are occupied by a control-flow entry (Many systems use one cell. In Gforth they currently take four cells, but this may change in the future).

When using `cs-pick` and `cs-drop` on an orig, you need to use one `cs-drop` for every `cs-pick` (and vice versa) of a given orig, because the orig must be resolved by `then` exactly once.

Some standard control structure words are built from these words:

`ELSE` (*compilation orig1 – orig2 ; run-time –*) core

At run-time, execution continues after the `THEN` that consumes the *orig*; the `IF`, `AHEAD`, `ELSE` or `WHILE` that pushed *orig1* jumps right after the `ELSE`. (see Section 6.10.1 [Selection], page 103).

`WHILE` (*compilation dest – orig dest ; run-time f –*) core

At run-time, if *f*=0, execution continues after the `REPEAT` (or `THEN` or `ELSE`) that consumes the *orig*, otherwise right after the `WHILE` (see Section 6.10.2 [General Loops], page 104).

`REPEAT` (*compilation orig dest – ; run-time –*) core

At run-time, execution continues after the `BEGIN` that produced the *dest*; the `WHILE`, `IF`, `AHEAD` or `ELSE` that pushed *orig* jumps right after the `REPEAT`. (see Section 6.10.2 [General Loops], page 104).

Gforth adds some more control-structure words:

`ENDIF` (*compilation orig – ; run-time –*) gforth-0.2

Same as `THEN`.

`?dup-IF` (*compilation – orig ; run-time n – n|*) gforth-0.2 "question-dupe-if"

This is the preferred alternative to the idiom `"?DUP IF"`, since it can be better handled by tools like stack checkers. Besides, it's faster.

`?DUP-0=-IF` (*compilation – orig ; run-time n – n|*) gforth-0.2 "question-dupe-zero-equals-if"

6.10.7 Calls and returns

A definition can be called simply be writing the name of the definition to be called. Normally a definition is invisible during its own definition. If you want to write a directly recursive definition, you can use `recursive` to make the current definition visible, or `recurse` to call the current definition directly.

`recursive` (*compilation – ; run-time –*) gforth-0.2

Make the current definition visible, enabling it to call itself recursively.

`recurse` (*... – ...*) core

Alias to the current definition.

For examples of using these words, See Section 3.20 [Recursion Tutorial], page 26.

Programming style note:

I prefer using `recursive` to `recurse`, because calling the definition by name is more descriptive (if the name is well-chosen) than the somewhat cryptic `recurse`. E.g., in a quicksort implementation, it is much better to read (and think) "now sort the partitions" than to read "now do a recursive call".

For mutual recursion, Gforth offers the defining word `forward`. You can use it to create a forward reference which is resolved automatically, and does not incur additional costs like the indirection of `Defer`. However, these forward definitions only work for colon definitions. Here's a usage example:

```
forward foo

: bar ( ... -- ... )
```

```
... foo ... ;

: foo ( ... -- ... ) \ resolves the forward definition
 ... bar ... ;
```

The words used for forward definitions are:

`forward` ("*name*" –) gforth-1.0

Defines a forward reference to a colon definition. Defining a colon definition with the same name in the same wordlist resolves the forward references. Use `.unresolved` to check whether any forwards are unresolved.

`.unresolved` (–) gforth-1.0 "dot-unresolved"

print all unresolved forward references

In Standard Forth, you use `Defer`red words for mutual recursion, like this:

```
Defer foo

: bar ( ... -- ... )
 ... foo ... ;

:noname ( ... -- ... )
 ... bar ... ;
IS foo
```

Deferred words are discussed in more detail in Section 6.11.11 [Deferred Words], page 138.

The current definition returns control to the calling definition when the end of the definition is reached or `EXIT` is encountered.

`EXIT` (*compilation – ; run-time nest-sys –*) core

Return to the calling definition; usually used as a way of forcing an early return from a definition. Before `EXIT`ing you must clean up the return stack and `UNLOOP` any outstanding `?DO`...`LOOP`s.

`?EXIT` (–) gforth-0.2 "question-exit"

Return to the calling definition if *f* is true.

6.10.8 Exception Handling

If a word detects an error condition that it cannot handle, it can `throw` an exception. In the simplest case, this will terminate your program, and report an appropriate error.

`throw` (*y1 .. ym nerror – y1 .. ym / z1 .. zn nerror*) exception

If *nerror* is 0, drop it and continue. Otherwise, transfer control to the next dynamically enclosing exception handler, reset the stacks accordingly, and push *nerror*.

`fast-throw` (*... nerror – ... nerror*) gforth-experimental "fast-throw"

Lightweight `throw` variant: only for non-zero *nerror*s, and does not store a backtrace or deal with missing `catch`.

`Throw` consumes a cell-sized error number on the stack. There are some predefined error numbers in Standard Forth (see `errors.fs`). In Gforth (and most other systems) you can use the iors produced by various words as error numbers (e.g., a typical use of `allocate` is `allocate throw`). Gforth also provides the word `exception` to define your own error

numbers (with decent error reporting); a Standard Forth version of this word (but without the error messages) is available in `compat/except.fs`. And finally, you can use your own error numbers (anything outside the range -4095..0), but won't get nice error messages, only numbers. For example, try:

```
-10 throw                     \ Standard defined
-267 throw                    \ system defined
s" my error" exception throw \ user defined
7 throw                       \ arbitrary number
```

`exception` (*addr u – n*) gforth-0.2

n is a previously unused `throw` value in the range (-4095...-256). Consecutive calls to `exception` return consecutive decreasing numbers. Gforth uses the string *addr u* as an error message.

There are also cases where you have a word (typically modeled after POSIX' `strerror`) for converting an error number into a string. You can use the following word to get these strings into Gforth's error handling:

`exceptions` (*xt n1 – n2*) gforth-1.0

Xt `( +n -- c-addr u )` converts an error number in the range 0<=*n*<*n1* into an error message. `Exceptions` reserves *n1* error codes in the range n2-n1<n3<=n2. When (at some later point in time) the Gforth error code *n3* in that range is thrown, it pushes *n2-n3* and then executes *xt* to produce the error message.

As an example, if the `errno` errors (and the conversion using `strerror`) was not already directly supported by Gforth, you could tie `strerror` in as follows:

```
' strerror 1536 exceptions constant errno-base
: errno-ior ( -- n )
\ n is the Gforth ior corresponding to the value in errno, so
\ we have to convert between the ranges here.
\ ERRNO is not a Gforth word, so you  would have to use the
\ C interface to access it.
  errno errno-base over - swap 0<> and ;
```

When you call a C function that can set `errno` (with the C interface, see Section 6.32 [C Interface], page 276), you can use one of the following words for converting that error into a `throw`:

`?errno-throw` (*f* –) gforth-1.0 "question-errno-throw"

If *f*<>0, throws an error code based on the value of `errno`.

`?ior` (*x* –) gforth-1.0 "question-i-o-r"

If *f*=-1, throws an error code based on the value of `errno`.

Which of these you should use depends on how the C function indicates that an error has happened. When the system then catches a throw performed by one of these words, it produces the proper error message (such as "Permission denied").

Note that the errno numbers are not directly used as throw codes (because the Forth standard specifies that positive throw codes must not be system-defined), but maps them into a different number range.

A common idiom to `THROW` a specific *err#* if a flag is true is this:

```
( flag ) 0<> err# and throw
```

Your program can provide exception handlers to catch exceptions. An exception handler can be used to correct the problem, or to clean up some data structures and just throw the exception to the next exception handler. Note that `throw` jumps to the dynamically innermost exception handler. The system's exception handler is outermost, and just prints an error and restarts command-line interpretation (or, in batch mode (i.e., while processing the shell command line), leaves Gforth).

The Standard Forth way to catch exceptions is `catch`:

`catch` (*x1 .. xn xt – y1 .. ym 0 / z1 .. zn error*) exception

`Executes` *xt*. If execution returns normally, `catch` pushes 0 on the stack. If execution returns through `throw`, all the stacks are reset to the depth on entry to `catch`, and the TOS (the *xt* position) is replaced with the throw code.

`catch-nobt` (*x1 .. xn xt – y1 .. ym 0 / z1 .. zn error*) gforth-experimental

perform a catch that does not record backtraces on errors

`nothrow` (–) gforth-0.7

Use this (or the standard sequence `[']` `false` `catch` `2drop`) after a `catch` or `endtry` that does not rethrow; this ensures that the next `throw` will record a backtrace.

The most common use of exception handlers is to clean up the state when an error happens. E.g.,

```
base @ >r hex \ actually the HEX should be inside foo to protect
              \ against exceptions between HEX and CATCH
['] foo catch ( nerror|0 )
r> base !
( nerror|0 ) throw \ pass it on
```

A use of `catch` for handling the error `myerror` might look like this:

```
['] foo catch
CASE
  myerror OF ... ( do something about it ) nothrow ENDOF
  dup throw \ default: pass other errors on, do nothing on non-errors
ENDCASE
```

Having to wrap the code into a separate word is often cumbersome, therefore Gforth provides an alternative syntax:

```
TRY
  code1
  IFERROR
    code2
  THEN
  code3
ENDTRY
```

This performs *code1*. If *code1* completes normally, execution continues with *code3*. If there is an exception in *code1* or before `endtry`, the stacks are reset to the depth during `try`, the throw value is pushed on the data stack, and execution continues at *code2*, and finally falls through to *code3*.

`try` (*compilation – orig ; run-time – R:sys1*) gforth-0.5

Start an exception-catching region.

`endtry` (*compilation* – *; run-time R:sys1* –) gforth-0.5

End an exception-catching region.

`iferror` (*compilation orig1* – *orig2 ; run-time* –) gforth-0.7

Starts the exception handling code (executed if there is an exception between `try` and `endtry`). This part has to be finished with `then`.

If you don't need *code2*, you can write `restore` instead of `iferror then`:

```
TRY
  code1
RESTORE
  code3
ENDTRY
```

The cleanup example from above in this syntax:

```
base @ { oldbase }
TRY
  hex foo \ now the hex is placed correctly
  0       \ value for throw
RESTORE
  oldbase base !
ENDTRY
throw
```

An additional advantage of this variant is that an exception between `restore` and `endtry` (e.g., from the user pressing *Ctrl-C*) restarts the execution of the code after `restore`, so the base will be restored under all circumstances.

However, you have to ensure that this code does not cause an exception itself, otherwise the `iferror`/`restore` code will loop. Moreover, you should also make sure that the stack contents needed by the `iferror`/`restore` code exist everywhere between `try` and `endtry`; in our example this is achieved by putting the data in a local before the `try` (you cannot use the return stack because the exception frame (*sys1*) is in the way there).

This kind of usage corresponds to Lisp's `unwind-protect`.

If you do not want this exception-restarting behaviour, you achieve this as follows:

```
TRY
  code1
ENDTRY-IFERROR
  code2
THEN
```

If there is an exception in *code1*, then *code2* is executed, otherwise execution continues behind the `then` (or in a possible `else` branch). This corresponds to the construct

```
TRY
  code1
RECOVER
  code2
ENDTRY
```

in Gforth before version 0.7. So you can directly replace `recover`-using code; however, we recommend that you check if it would not be better to use one of the other `try` variants while you are at it.

To ease the transition, Gforth provides two compatibility files: `endtry-iferror.fs` provides the `try ... endtry-iferror ... then` syntax (but not `iferror` or `restore`) for old systems; `recover-endtry.fs` provides the `try ... recover ... endtry` syntax on new systems, so you can use that file as a stopgap to run old programs. Both files work on any Gforth (they just do nothing if the system already has the syntax it implements), so you can unconditionally `require` one of these files, even if you use a mix old and new Gforths.

`restore` (*compilation orig1 – ; run-time –*) gforth-0.7

Starts restoring code, that is executed if there is an exception, and if there is no exception.

`endtry-iferror` (*compilation orig1 – orig2 ; run-time R:sys1 –*) gforth-0.7

End an exception-catching region while starting exception-handling code outside that region (executed if there is an exception between `try` and `endtry-iferror`). This part has to be finished with `then` (or `else`...`then`).

Here's the error handling example:

```
TRY
  foo
ENDTRY-IFERROR
  CASE
    myerror OF ... ( do something about it ) nothrow ENDOF
    throw \ pass other errors on
  ENDCASE
THEN
```

Programming style note:

As usual, you should ensure that the stack depth is statically known at the end: either after the `throw` for passing on errors, or after the `ENDTRY` (or, if you use `catch`, after the end of the selection construct for handling the error).

There are two alternatives to `throw`: `Abort"` is conditional and you can provide an error message. `Abort` just produces an "Aborted" error.

The problem with these words is that exception handlers cannot differentiate between different `abort"`s; they just look like `-2 throw` to them (the error message cannot be accessed by standard programs). Similarly, `abort` looks like `-1 throw` to exception handlers.

`ABORT"` (*compilation 'ccc" ' – ; run-time ... f –*) core,exception-ext "abort-quote"

If any bit of *f* is non-zero, perform the function of `-2 throw`, displaying the string *ccc* if there is no exception frame on the exception stack.

`abort` (*?? – ??*) core,exception-ext

`-1 throw`.

For problems that are not that awful that you need to abort execution, you can just display a warning. The variable `warnings` allows to tune how many warnings you see.

`WARNING"` (*compilation 'ccc" ' – ; run-time f –*) gforth-1.0 "warning-quote"

if *f* is non-zero, display the string *ccc* as warning message.

`warnings` (*– addr*) gforth-0.2

Set warnings level to

`0` turns warnings off

`-1` turns normal warnings on

`-2` turns beginner warnings on

`-3` turns pedantic warnings on

`-4` turns warnings into errors (including beginner warnings)

6.11 Defining Words

Defining words are used to extend Forth by creating new entries in the dictionary.

6.11.1 CREATE

The simplest defining word is `CREATE`, used like this:

```
CREATE new-word1
```

`CREATE` is a parsing word, i.e., it takes an argument from the input stream (`new-word1` in our example). It generates a dictionary entry for `new-word1`. When `new-word1` is executed, all that it does is leave an address on the stack. The address represents the value of the dictionary pointer (`HERE`) at the time that `new-word1` was defined. Therefore, `CREATE` is a way of associating a name with the address of a region of memory.

`Create` (`"`*name*`"` –) core

Note that Standard Forth guarantees only for `create` that its body is contiguous with the following dictionary allocations (e.g., `allot`, see Section 6.8.2 [Dictionary allocation], page 76). Also, in Standard Forth only `create`d words can be modified with `does>` (see Section 6.11.10 [User-defined Defining Words], page 125). And in Standard Forth `>body` can only be applied to `create`d words.

By extending this example to reserve some memory in data space, we end up with something like a *variable*. Here are two different ways to do it:

```
CREATE new-word2 1 cells allot  \ reserve 1 cell without initializing it
CREATE new-word3 4 ,            \ reserve 1 cell and initialise it (to 4)
```

The variable can be examined and modified using `@` ("fetch") and `!` ("store") like this:

```
new-word2 @ .      \ get address, fetch from it and display
1234 new-word2 !   \ new value, get address, store to it
```

A similar mechanism can be used to create arrays. For example, an 80-character text buffer:

```
CREATE text-buf 80 allot \ uninitialized

text-buf 0 + c@ \ the 1st character (offset 0)
text-buf 3 + c@ \ the 4th character (offset 3)
```

You can build arbitrarily complex data structures by allocating appropriate areas of memory. For further discussions of this, and to learn about some Gforth tools that make it easier, See Section 6.12 [Structures], page 141.

6.11.2 Variables

The previous section showed how a sequence of commands could be used to generate a variable. As a final refinement, the whole code sequence can be wrapped up in a defining word, making it easier to create new variables:

```
: myvariableX ( "name" -- a-addr ) CREATE 1 cells allot ;
: myvariable0 ( "name" -- a-addr ) CREATE 0 , ;

myvariableX foo \ variable foo starts off with an unknown value
myvariable0 joe \ whilst joe is initialised to 0

45 3 * foo !   \ set foo to 135
1234 joe !     \ set joe to 1234
3 joe +!       \ increment joe by 3.. to 1237
```

Not surprisingly, there is no need to define `myvariableX`, since Forth already has a definition `Variable`. Standard Forth does not guarantee that a `Variable` is initialised when it is created (i.e., it may behave like `myvariableX`). In contrast, Gforth's `Variable` initialises the variable to 0 (i.e., it behaves exactly like `myvariable0`). Forth also provides `2Variable` and `fvariable` for double and floating-point variables, respectively – they are initialised to #0. and 0e in Gforth. If you use a `Variable` to store a boolean, you can use `on` and `off` to toggle its state (see Section 6.4 [Boolean Flags], page 60).

`Variable` ("*name*" –) core

Define *name* and reserve a cell at *addr*.
name execution: `( -- addr )`.

`AVariable` ("*name*" –) gforth-0.2

Works like `variable`, but (when used in cross-compiled code) tells the cross-compiler that the cell stored in the variable is an address.

`2Variable` ("*name*" –) double "two-variable"

Define *name* and reserve two cells starting at *addr*.
name execution: `( -- addr )`.

`fvariable` ("*name*" –) floating "f-variable"

Define *name* and reserve a float at *f-addr*.
name execution: `( -- f-addr )`.

Finally, for buffers of arbitrary length there is

`buffer:` (*u* "*name*" –) core-ext "buffer-colon"

Define *name* and reserve *u* bytes starting at *addr*. Gforth initializes the reserved bytes to 0, but the standard does not guarantee this.
name execution: `( -- addr )`.

6.11.3 Constants

`Constant` allows you to declare a fixed value and refer to it by name. For example:

```
12 Constant INCHES-PER-FOOT \ is integer appropriate
2.54e fconstant CM-PER-INCH
```

A `Variable` can be both read and written, so its run-time behaviour is to supply an address through which its current value can be manipulated. In contrast, the value of a `Constant` cannot be changed once it has been declared[10] so it's not necessary to supply the address – it is more efficient to return the value of the constant directly. That's exactly what happens; the run-time effect of a constant is to put its value on the top of the stack (You can find one way of implementing `Constant` in Section 6.11.10 [User-defined Defining Words], page 125).

Forth also provides `2Constant` and `fconstant` for defining double and floating-point constants, respectively.

`Constant` (*w* "*name*" –) core

Define *name*.
name execution: *(– w)*

`AConstant` (*addr* "*name*" –) gforth-0.2

Like `constant`, but defines a constant for an address (this only makes a difference in the cross-compiler).

`2Constant` (*w1 w2* "*name*" –) double "two-constant"

Define *name*.
name execution: *(– w1 w2)*

`fconstant` (*r* "*name*" –) floating "f-constant"

Define *name*.
name execution: *(– r)*

6.11.4 Values

A `Value` behaves like a `Constant`, but it can be changed. `TO` and `+TO` are parsing words that change a value. Alternatively, you can change a value `v` by writing `->v` (equivalent to `TO v`) or `+>v` (equivalent to `+TO v`).

Here are some examples:

```
12 value apples \ Define APPLES with an initial value of 12
34 to apples    \ Change the value of APPLES. TO is a parsing word
34 ->apples     \ Change the value of APPLES. Non-standard usage
1  +to apples   \ Increment APPLES.  Non-standard usage.
1  +>apples     \ Increment APPLES.  Non-standard usage.
apples          \ puts 36 on the top of the stack.
```

`Value` (*w* "*name*" –) core-ext

Define *name* with the initial value *w*
name execution: *(– w2)* push the current value of *name*.
`to` ***name*** run-time: *(w3 –)* change the value of *name* to *w3*.
`+to` ***name*** run-time: *(n|u –)* increment the value of *name* by *n|u*

`AValue` (*w* "*name*" –) gforth-0.6

Like `value`, but defines a value for an address (this only makes a difference in the cross-compiler).

`2Value` (*w1 w2* "*name*" –) double-ext "two-value"

[10] Well, often it can be – but not in a Standard, portable way. It's safer to use a `Value` (read on).

Define *name* with the initial value *w*
name execution: *(– w3 w4)* push the current value of *name*.
`to` ***name*** run-time: *(w5 w6 –)* change the value of *name* to *w5 w6*.
`+to` ***name*** run-time: *(d|ud –)* increment the value of *name* by *d|ud*

`fvalue` (*r* "*name*" –) floating-ext "f-value"

Define *name* with the initial value *r*
name execution: *(– r2)* push the current value of *name*.
`to` ***name*** run-time: *(r3 –)* change the value of *name* to *r3*.
`+to` ***name*** run-time: *(r4 –)* increment the value of *name* by *r4*

`TO` (*value ...* "*name*" –) core-ext

Name is a value-flavoured word, *...* is optional additional addressing information, e.g., for a value-flavoured field. At run-time, perform the *to name* semantics: change *name* (with the same additional addressing information) to push *value*. The type of *value* depends on the type of *name* (see the defining word for *name* for the actual type). An alternative syntax is to write `->`***name***.

`+TO` (*value ...* "*name*" –) gforth-1.0 "plus-TO"

Name is a value-flavoured word, *...* is optional additional addressing information, e.g., for a value-flavoured field. At run-time, perform the *+to name* semantics: if *name* (with the same additional addressing information) pushed *value1* before, change it to push *value2*, the sum of the *value1* and *value*. The type of *value* depends on the type of *name* (see the defining word for *name* for the actual type). An alternative syntax is to write `+>`***name***.

Words that produce their value on execution and that can be changed with `to` or `+to` are called value-flavoured (in contrast to the variable-flavoured words that produce their address on execution). They are defined be some of the words listed above, but also by some locals definitions words (see Section 6.26.1.1 [Locals definition words], page 218) and some field definition words (see Section 6.12.2 [Value-Flavoured and Defer-Flavoured Fields], page 144).

Sometimes you want to take the address of a value-flavoured word. Because this has some potential performance disadvantages, Gforth asks you to be explicit about it, and define the word as addressable. Once you have done that, you can get the address with `addr`. The following example is equivalent to the one above:

```
12 addressable: value apples
34 addr apples ! \ Change the value of APPLES.  ADDR is a parsing word
1 +to apples     \ Increment APPLES
addr apples @    \ puts 35 on the top of the stack.
```

`addressable:` (–) gforth-experimental "addressable-colon"

`Addressable:` should be used in front of a defining word for a value-flavoured word (e.g., `value`). It allows to use `addr` on the word defined by that defining word.

`addr` (*interpretation* "*name*" *... – addr; compilation* "*name*" *– ; run-time ... – addr*) gforth-1.0

Name is an `addressable:` value-flavoured word, *...* is optional additional addressing information, e.g., for a value-flavoured field. *Addr* is the address where the value of *name* (taking the additional address information into account) is stored.

For now using `addr` on a non-`addressable:` value results in a warning. In the future, when we change the code generation in a way that results in potentially faster code for non-`addressable:` values, but where the use of `addr` on such values could produce unexpected results, such usage will result in an error.

6.11.5 Colon Definitions

```
: name ( ... -- ... )
    word1 word2 word3 ;
```

Creates a word called `name` that, upon execution, executes `word1` `word2` `word3`. `name` is a *(colon) definition*.

The explanation above is somewhat superficial. For simple examples of colon definitions see Section 4.3 [Your first definition], page 48. For an in-depth discussion of some of the issues involved, See Section 6.14 [Interpretation and Compilation Semantics], page 150.

`:` ("*name*" – *colon-sys*) core "colon"

`;` (*compilation colon-sys* – *; run-time nest-sys* –) core "semicolon"

6.11.6 Inline Definitions

We plan to to perform automatic inlining eventually, but for now you can perform inlining with

`inline:` ("*name*" – *inline:-sys*) gforth-experimental "inline-colon"

Start inline colon definition. The code between `inline:` and `;inline` has to compile (not perform) the code to be inlined, but the resulting definition *name* is a colon definition that performs the inlined code. Note that the compiling code must have the stack effect `( -- )`, otherwise you will get an error when Gforth tries to create the colon definition for *name*.

`;inline` (*inline:-sys* –) gforth-experimental "semi-inline"

end inline definition started with `inline:`

As an example, you can define an inlined word and use it with

```
inline: my2dup ( a b -- a b a b )
    ]] over over [[ ;inline

#1. my2dup d. d.
: foo my2dup ;
#1. foo d. d.
see foo
```

Inline words are related to macros (see Section 6.16.2 [Macros], page 162); the difference is that a macro has immediate compilation semantics while an `inline:`-defined word has default compilation semantics; this means that you normally use a macro only inside a colon definition, while you can use an `inline:` word also interpretively. But that also means that you can do some things with macros that you cannot do as an `inline:` word. E.g.,

```
\ Doesn't work:
\   inline: endif ]] then [[ ;inline
\ Instead, write a macro:
: endif ]] then [[ ; immediate
```

Conversely, for words that would be fine as non-immediate colon definitions, define them as non-immediate colon definitions or (if utmost performance is required) as `inline:` words; don't define them as macros, because then you cannot properly use them interpretively:

```
: another2dup ]] over over [[ ; immediate
\ Doesn't work:
\   #1. another2dup d. d.
```

You may wonder why you have to write compiling code between `inline:` and `;inline`. That's because the implementation of an inline word like `my2dup` above works similar to:

```
: compile-my2dup ( xt -- )
    drop ]] over over [[ ;

: my2dup [ 0 compile-my2dup ] ;
' compile-my2dup set-optimizer
```

The `DROP` and `0` are there because `compile-my2dup` is the implementation of `compile,` for `my2dup`, and `compile,` expects an xt (see Section 6.11.10.7 [User-defined compile-comma], page 134).

6.11.7 Anonymous Definitions

Sometimes you want to define an *anonymous word*; a word without a name. You can do this with:

`:noname` (– *xt colon-sys*) core-ext "colon-no-name"

This leaves the execution token for the word on the stack after the closing `;`. Here's an example in which a deferred word is initialised with an `xt` from an anonymous colon definition:

```
Defer deferred
:noname ( ... -- ... )
  ... ;
IS deferred
```

Gforth provides an alternative way of doing this, using two separate words:

`noname` (–) gforth-0.2

The next defined word will be anonymous. The defining word will leave the input stream alone. The xt of the defined word will be given by `latestxt`, its nt by `latestnt` (see Section 6.15.2 [Name token], page 158).

`latestxt` (– *xt*) gforth-0.6

xt is the execution token of the most recent word defined in the current section.

The previous example can be rewritten using `noname` and `latestxt`:

```
Defer deferred
noname : ( ... -- ... )
  ... ;
latestxt IS deferred
```

`noname` works with any defining word, not just `:`.

`latestxt` also works when the last word was not defined as `noname`. It also has the useful property that it is valid as soon as the header for a definition has been built. Thus:

```
latestxt . : foo [ latestxt . ] ; ' foo .
```

prints 3 numbers; the last two are the same.

6.11.8 Quotations

A quotation is an anonymous colon definition inside another colon definition. Quotations are useful when dealing with words that consume an execution token, like `catch` or `outfile-execute`. E.g. consider the following example of using `outfile-execute` (see Section 6.22.3 [Redirection], page 199):

```
: some-warning ( n -- )
    cr ." warning# " . ;

: print-some-warning ( n -- )
    ['] some-warning stderr outfile-execute ;
```

Here we defined `some-warning` as a helper word whose xt we could pass to outfile-execute. Instead, we can use a quotation to define such a word anonymously inside `print-some-warning`:

```
: print-some-warning ( n -- )
  [: cr ." warning# " . ;] stderr outfile-execute ;
```

The quotation is bounded by `[:` and `;]`. It produces an execution token at run-time.

`[:` (*compile-time: – quotation-sys flag colon-sys*) gforth-1.0 "bracket-colon"

Starts a quotation in the next section.

`;]` (*compile-time: quotation-sys – ; run-time: – xt*) gforth-1.0 "semi-bracket"

Ends a quotation (represented by *xt*) and switch to the previous section. `Latestxt` and `latestnt` refer to the last word in the current section, i.e., not to the quotation.

6.11.9 Supplying the name of a defined word

By default, a defining word takes the name for the defined word from the input stream. Sometimes you want to supply the name from a string. You can do this with:

`nextname` (*c-addr u* –) gforth-0.2

The next defined word will have the name *c-addr u*; the defining word will leave the input stream alone.

For example:

```
s" foo" nextname create
```

is equivalent to:

```
create foo
```

`nextname` works with any defining word.

6.11.10 User-defined Defining Words

You can define new defining words in terms of any existing defining word, but `:` and `create`...`does>`/`set-does>` are particularly flexible, whereas the children of, e.g., `constant` are all just constants.

6.11.10.1 User-defined defining words with colon definitions

Colon definitions are very flexible, so you can write a defining word that defines a new colon definition at its run-time. Here is an example:

```
: myconstant {: w -- :}
  : w postpone literal postpone ; ;
```

When defining `5 myconstant five`, `myconstant` first stashes *w* in a local (for reasons explained later), then calls `:`, which starts the definition of `five`. Then it uses `postpone literal` (see Section 6.16 [Compiling words], page 160) to compile *w* (i.e., 5) into this colon definition, and then `postpone ;` to end it. You can look at the result with `see five`:

```
: five  #5 ;
```

Can't we just leave *w* on the data stack for consumption by `postpone literal`? No: `:` pushes a colon-sys on the data stack, so we have to first move *w* elsewhere so we can later access it. In this example, we used a local variable, but moving *w* on the return stack and back would also have been an option.

A more convenient, but Gforth-specific way to write `myconstant` is:

```
: myconstant {: w -- :}
  : ]] w ; [[ ;
```

The features used in this code are explained elsewhere (see Section 6.16.2 [Macros], page 162).

A disadvantage of this approach is that it consumes more memory than the approach of the next section: E.g, here are the memory costs of defining `five` with the various implementations:

```
builtin  : does> set-does> opt
   48   64   48     48      48  bytes header+threaded code
    0   23    0      0       0  bytes native code
   16   16   32     16      16  compiled threaded code
    4   23   34      7       4  compiled native code
```

Builtin refers to using `constant`, *:* to using `myconstant` (defined above), *does>* to using `myconstant2`, *set-does>* to using `myconstant3` (both from see Section 6.11.10.2 [User-defined defining words using CREATE], page 126), and opt to using `myconstant4` (see Section 6.11.10.7 [User-defined compile-comma], page 134).

The lines where the label starts with "bytes" report the space consumption of defining the word `five` itself; the native code is for gforth-fast on AMD64 (native code for the gforth engine is larger).

The lines where the label starts with "compiled" report the space consumption (also in bytes) for the invocation of `five` in the word `: foo five * ;`. The native code can be bigger or smaller in other contexts.

6.11.10.2 User-defined defining words using `create`

If you want the words defined with your defining words to behave differently from words defined with standard defining words, you can write your defining word like this:

```
: def-word ( "name" -- )
    CREATE code1
```

```
DOES> ( ... -- ... )
    code2 ;

def-word name
```

This fragment defines a *defining word* `def-word` and then executes it. When `def-word` executes, it `CREATE`s a new word *name*, and executes the code *code1*. The code *code2* is not executed at this time. The word *name* is sometimes called a *child* of `def-word`.

When you execute *name*, the address of the body of *name* is pushed on the data stack and *code2* is executed. The address of the body of *name* is the address `HERE` returns immediately after the `CREATE`, i.e., the address a `create`d word returns by default).

You can understand the behaviour of `def-word` and `name` by considering the following definitions:

```
: def-word1 ( "name" -- )
    CREATE code1 ;

: action1 ( ... -- ... )
    code2 ;

def-word1 name1
```

Using `name1 action1` is equivalent to using `name`.

You can use `def-word` to define a set of child words that behave similarly; they all have a common run-time behaviour determined by *code2*. Typically, the *code1* sequence builds a data area in the body of the child word. The structure of the data is common to all children of `def-word`, but the data values are specific – and private – to each child word.

As an example, here's how you can define `myconstant2` with `does>`:

```
: myconstant2 ( w "name" -- )
    create ,
does> ( -- w )
    @ ;
```

Here `create` defines a word *name*, then `,` stores *w* in *name*'s data area, then the `does>` changes *name*'s behaviour and returns to the caller of `myconstant2`: When *name* is invoked, the new behaviour first pushes the address of the data area (as before), but then also performs the code after the `does>`. In the present case, this code fetches the value of the constant from the data area.

The stack effect besides the `does` reflects the stack effect of *name* execution, not the stack effect of the code after the `does>` (this is not common practice yet but we still recommend it).

`Does>` splits the definition into two subdefinitions and has a number of disadvantages. Alternatively, Gforth allows you to provide the second part as an execution token by using `set-does>`. So the general scheme is:

```
: def-word ( "name" -- ; name execution: ... -- ... )
    create code1
    xt-code2 set-does>
    code3 ;
```

The difference from the definition using `does>` is that on *name* execution, after pushing the data address, *xt-code2* is `execute`d, rather than calling the code after the `does>`. This also allows putting *code3* in *def-word*; this is particularly relevant when you want to also use `set-optimizer` (see Section 6.11.10.7 [User-defined compile-comma], page 134) on the defined word, because `does>`/`set-does>` calls `set-optimizer` itself, so using `set-optimizer` before `does>`/`set-does>` has no effect.

Here *xt-code2* could be the xt of an existing word, or it could be provided through a quotation (see Section 6.11.8 [Quotations], page 125).

Another advantage of `set-does>` is that the result is a little more efficient if the execution token passed to it is that of a primitive. This advantages comes to fruition in:

```
: myconstant3 ( w "name" -- ; name execution: -- w )
    create ,
    ['] @ set-does> ;
```

During *name* execution, after pushing the body address of *name*, `@` is executed.

The efficiency advantage shows up in the comparisons of compiled code size (see Section 6.11.10.1 [User-defined defining words with colon definitions], page 126); the execution time also benefits.

6.11.10.3 Applications of `CREATE..DOES>`

You may wonder how to use this feature. Here are some usage patterns:

When you see a sequence of code occurring several times, and you can identify a meaning, you will factor it out as a colon definition. When you see similar colon definitions, you can factor them using `CREATE..DOES>`. E.g., an assembler usually defines several words that look very similar:

```
: ori, ( reg-target reg-source n -- )
    0 asm-reg-reg-imm ;
: andi, ( reg-target reg-source n -- )
    1 asm-reg-reg-imm ;
```

This could be factored with:

```
: reg-reg-imm ( op-code -- )
    CREATE ,
DOES> ( reg-target reg-source n -- )
    @ asm-reg-reg-imm ;

0 reg-reg-imm ori,
1 reg-reg-imm andi,
```

Another view of `CREATE..DOES>` is to consider it as a crude way to supply a part of the parameters for a word (known as *currying* in the functional language community). E.g., `+` needs two parameters. Creating versions of `+` with one parameter fixed can be done like this:

```
: curry+ ( n1 "name" -- )
    CREATE ,
DOES> ( n2 -- n1+n2 )
    @ + ;
```

```
 3 curry+ 3+
-2 curry+ 2-
```

6.11.10.4 The gory details of CREATE..DOES>

DOES> (*compilation colon-sys1 – colon-sys2*) core "does"

Changes the current word such that it pushes its body address and then calls the code behind the does>. Also changes the compile, implementation accordingly. Call set-optimizer afterwards if you want a more efficient implementation.

You can put the does>-part in a different definition than the create part. This allows us to, e.g., select among different DOES>-parts:

```
: does1
DOES> ( ... -- ... )
    code1 ;

: does2
DOES> ( ... -- ... )
    code2 ;

: def-word ( ... -- ... )
    create ...
    IF
       does1
    ELSE
       does2
    THEN ;
```

In this example, the selection of whether to use does1 or does2 is made at definition-time, i.e., at the time when the child word is CREATEd.

Note that the property of does> to end the definition makes it necessary to introduce extra definitions does1 and does2. You can avoid that with set-does>:

```
: def-word ( ... -- ... )
    create ...
    IF
       [: code1 ;] set-does>
    ELSE
       [: code2 ;] set-does>
    THEN ;
```

set-does> (*xt* –) gforth-1.0 "set-does-to"

Changes the current word such that it pushes its body address and then executes *xt*. Also changes the compile, implementation accordingly. Call set-optimizer afterwards if you want a more efficient implementation.

In a standard program you can apply a DOES>-part only if the last word was defined with CREATE. In Gforth, the DOES>-part will override the behaviour of the last word defined in any case. In a standard program, you can use DOES> only in a colon definition. In Gforth, you can also use it in interpretation state, in a kind of one-shot mode; for example:

```
CREATE name ( ... -- ... )
  initialization
DOES>
  code ;
```

is equivalent to the standard:

```
:noname
DOES>
    code ;
CREATE name EXECUTE ( ... -- ... )
    initialization
```

Gforth also supports quotations in interpreted code, and quotations save and restore the current definition, so you can also write the example above also as:

```
CREATE name ( ... -- ... )
  initialization
[: code ;] set-does>
```

`>body` (*xt* – *a-addr*) core "to-body"

a-addr is the address of the body (aka parameter field or data field) of the word represented by *xt*

You can access the data area of a `create`d word with `>body`, including words where the behaviour has been changed with `does>`/`set-does>`. So if you know that `five` has been defined with, e.g., `myconstant3` (see Section 6.11.10.2 [User-defined defining words using CREATE], page 126), you can change its value with

```
7 ' five >body !
```

and performing `five` will then push 7. By contrast, for words defined with `myconstant` (defined using `:`, see Section 6.11.10.1 [User-defined defining words with colon definitions], page 126) you cannot change the value in this way.

However, if a word uses `set-optimizer` (see Section 6.11.10.7 [User-defined compile-comma], page 134) for a more efficient implementation of the compiled code for a word, in many cases the compiled code does not read data from the body of this word, and in that case changing the data by using `>body` will not have the desired effect. So looking at the source code of a defining word and seeing `create` is not enough to conclude that you can change the data and it will affect all existing uses. An example of that is `myconstant4` (see Section 6.11.10.7 [User-defined compile-comma], page 134).

So it's a good idea to document whether the intention behind a defining word using `create` is that it's data should be changeable through `>body`.

6.11.10.5 Advanced does> usage example

The MIPS disassembler (`arch/mips/disasm.fs`) contains many words for disassembling instructions, that follow a very repetitive scheme:

```
:noname disasm-operands s" inst-name" type ;
entry-num cells table + !
```

Of course, this inspires the idea to factor out the commonalities to allow a definition like

```
disasm-operands entry-num table define-inst inst-name
```

The parameters *disasm-operands* and *table* are usually correlated. Moreover, before I wrote the disassembler, there already existed code that defines instructions like this:

```
entry-num inst-format inst-name
```

This code comes from the assembler and resides in `arch/mips/insts.fs`.

So I had to define the *inst-format* words that performed the scheme above when executed. At first I chose to use run-time code-generation:

```
: inst-format ( entry-num "name" -- ; compiled code: addr w -- )
  :noname Postpone disasm-operands
  name Postpone sliteral Postpone type Postpone ;
  swap cells table + ! ;
```

Note that this supplies the other two parameters of the scheme above.

An alternative would have been to write this using `create`/`does>`:

```
: inst-format ( entry-num "name" -- )
  here name string, ( entry-num c-addr ) \ parse and save "name"
  noname create , ( entry-num )
  latestxt swap cells table + !
does> ( addr w -- )
  \ disassemble instruction w at addr
  @ >r
  disasm-operands
  r> count type ;
```

Somehow the first solution is simpler, mainly because it's simpler to shift a string from definition-time to use-time with `sliteral` than with `string,` and friends.

I wrote a lot of words following this scheme and soon thought about factoring out the commonalities among them. Note that this uses a two-level defining word, i.e., a word that defines ordinary defining words.

This time a solution involving `postpone` and friends seemed more difficult (try it as an exercise), so I decided to use a `create`/`does>` word; since I was already at it, I also used `create`/`does>` for the lower level (try using `postpone` etc. as an exercise), resulting in the following definition:

```
: define-format ( disasm-xt table-xt -- )
    \ define an instruction format that uses disasm-xt for
    \ disassembling and enters the defined instructions into table
    \ table-xt
    create 2,
does> ( u "inst" -- )
    \ defines an anonymous word for disassembling instruction inst,
    \ and enters it as u-th entry into table-xt
    2@ swap here name string, ( u table-xt disasm-xt c-addr ) \ remember string
    noname create 2,      \ define anonymous word
    execute latestxt swap ! \ enter xt of defined word into table-xt
does> ( addr w -- )
    \ disassemble instruction w at addr
    2@ >r ( addr w disasm-xt R: c-addr )
```

```
    execute ( R: c-addr ) \ disassemble operands
    r> count type ; \ print name
```

Note that the tables here (in contrast to above) do the `cells +` by themselves (that's why you have to pass an xt). This word is used in the following way:

```
' disasm-operands ' table define-format inst-format
```

As shown above, the defined instruction format is then used like this:

```
entry-num inst-format inst-name
```

In terms of currying, this kind of two-level defining word provides the parameters in three stages: first *disasm-operands* and *table*, then *entry-num* and *inst-name*, finally `addr w`, i.e., the instruction to be disassembled.

Of course this did not quite fit all the instruction format names used in `insts.fs`, so I had to define a few wrappers that conditioned the parameters into the right form.

If you have trouble following this section, don't worry. First, this is involved and takes time (and probably some playing around) to understand; second, this is the first two-level `create/does>` word I have written in seventeen years of Forth; and if I did not have `insts.fs` to start with, I may well have elected to use just a one-level defining word (with some repeating of parameters when using the defining word). So it is not necessary to understand this, but it may improve your understanding of Forth.

6.11.10.6 Words with user-defined `to` etc.

When you define a word *x*, you can set its execution semantics with `set-does>` (see Section 6.11.10.2 [User-defined defining words using CREATE], page 126) or `set-execute` (see Section 6.34.2 [Header methods], page 300). But you can also change the semantics of

```
to x        \ aka ->x
+to x       \ aka +>x
action-of x \ aka `x defer@
is x        \ aka `x defer!
addr x
```

This is all achieved through a common mechanism described in this section. As an example, let's define `dvalue` (it behaves in Gforth exactly like `2value`, see Section 6.11.4 [Values], page 121). The code is as follows, explained below:

```
: d+! ( d addr -- )
  dup >r 2@ d+ r> 2! ;

\                  to +to action-of is  addr
to-table: d!-table 2! d+!    n/a    n/a [noop]

' >body d!-table to-class: dvalue-to

: dvalue ( d "name" -- )
  create 2,
  ['] 2@ set-does>
  ['] dvalue-to set-to ;
```

```
#5. dvalue x
#2. +to x
x d. \ prints 7
```

First, we define the support word `d+!`.

Next, we define `d!-table`, a table of the various `to`-like actions.

For actions that are not supported, we put `n/a` in the table, and when you try to use, e.g., `is x`, exception -21 (unsupported operation) is thrown.

For the `to` and `+to` actions, we have to provide words with the stack effect `( d addr -- )`, where *addr* is the address where the data of the value-flavoured word is stored. For `addr` (only supported for `addressable:` words), we have to provide a word with the stack effect `( addr1 -- addr2 )`; in the usual case, both addresses are the same, and we can just provide `[noop]`. For the general case, see the description of `to-table:` below.

Next, the defining word `to-class:` combines the `d!-table` with the address-computation word `>body`, resulting in the definition of `dvalue-to`.[11] The address-computation word has the stack effect `( ... xt -- addr )`. When invoking `+to x`, the xt of `x` is pushed and then the address-computation word (`>body`) is called; the result is the address that is then passed on to `d+!` (from `d!-table`).

For `dvalue` that's all, but in other cases, e.g., value-flavoured fields (see Section 6.12.2 [Value-Flavoured and Defer-Flavoured Fields], page 144), additional stack items can be consumed by the address-computation word, and those have to be provided by the user as stack-top values when invoking, e.g., `to` *`field`*.

Next, we have the definition of `dvalue`, which is a straightforward `create...set-does>` word that also tells *name* with `set-to` how it should behave for `to` etc.

Finally, we use `dvalue` to define `x` and use it. The line using `+to` exercises the `set-to` mechanism:

```
#2. +to x
```

performs

```
#2. ' x >body d+!
```

The `>body` is the address-computation word given in the definition of `dvalue-to`, and the `d+!` is the `+to` entry in `d!-table`.

These are the words mentioned above:

`to-table:` ("*name*" "*to-word*" "*+to-word*" "*addr-word*" "*action-of-word*" "*is-word*" –) gforth-experimental "to-table-colon"

Create a table *name* with entries for `TO`, `+TO`, `ACTION-OF`, `IS`, and `ADDR`. The words for these entries are called with *xt* on the stack, where xt belongs to the word behind `to` (or `+to` etc.). Use `n/a` to mark unsupported operations. Default entries operations can be left away at the end of the line; the default is for the `addr` entry is `[noop]` while the default for the other entries is `n/a`.

The stack effects of the actions are:

- For the `to` and `+to` action: `( value addr -- )`, where *value* has the appropriate type (e.g., a double-cell in our `dvalue` example).

[11] The same to-table is often combined with different address computation words (e.g., for global values, user values, value-flavoured fields and locals), that's why the definition of the to-table is separated from the definition of the to-class.

- For the `action-of` action: `( addr -- xt )`
- For the `is` action: `( xt addr -- )`
- For the `addr` action: `( addr1 -- addr2 )`

The *addr* input parameter in all these cases is the address of the memory where the value of the xt is stored.

The default mechanism means that `d!-table` could instead have been defined as follows:

```
\                   to +to action-of is  addr
to-table: d!-table 2! d+!
```

`n/a` (–) gforth-experimental "not-available"

This word can be ticked, but throws an "Operation not supported" exception on interpretation and compilation. Use this for methods etc. that aren't supported.

`to-class:` (*xt table* "*name*" –) gforth-experimental "to-class-colon"

Create a to-class implementation *name*, where *xt* `( ... xt -- addr )` computes the address to access the data, and *table* (created with `to-table:`) contains the words for accessing it.

`>uvalue` (*xt – addr*) gforth-internal "to-uvalue"

Xt is the xt of a word *x* defined with `uvalue`; *addr* is the address of the data of *x* in the current task. This word is useful for building, e.g., `uvalue`. Do not use it to circumvent that you cannot get the address of a uvalue with `addr`; in the future Gforth may perform optimizations that assume that uvalues can only be accessed through their name.

`set-to` (*to-xt* –) gforth-1.0

Changes the implementations of the to-class methods of the most recently defined word to come from the to-class that has the xt *to-xt*.

6.11.10.7 User-defined `compile,`

You can also change the implementation of `compile,` for a word, with

`set-optimizer` (*xt* –) gforth-1.0

Changes the current word such that `compile,`ing it executes *xt* (with the same stack contents as passed to `compile,`). Note that `compile,` must be consistent with `execute`, so you must use `set-optimizer` only to install a more efficient implementation of the same behaviour.

`opt:` (*compilation – colon-sys2 ; run-time – nest-sys*) gforth-1.0 "opt-colon"

Starts a nameless colon definition; when it is complete, this colon definition will become the `compile,` implementation of the latest word (before the `opt:`).

Note that the resulting `compile,` must still be equivalent to `postpone literal postpone execute`, so `set-optimizer` is useful for efficiency, not for changing the behaviour. There is nothing that prevents you from shooting yourself in the foot, however. You can check whether your uses of `set-optimizer` are correct by comparing the results when you use it with the results you get when you disable your uses by first defining

```
: set-optimizer drop ;
```

As an example of the use of `set-optimizer`, we can enhance `myconstant3` as follows.

```
: myconstant4 ( n "name" -- ; name: -- n )
```

```
  create ,
  ['] @ set-does>
  [: >body @ postpone literal ;] set-optimizer
;
```

The only change is the addition of the `set-optimizer` line. When you define a constant and compile it:

```
5 myconstant4 five
: foo five ;
```

the compiled `five` in `foo` is now compiled to the literal 5 instead of a generic invocation of `five`. The quotation has the same stack effect as `compile,`: `( xt -- )`. The passed xt belongs to the `compile,`d word, i.e., `five` in the example. In the example the xt is first converted to the body address, then the value 5 at that place is fetched, and that value is compiled with the `postpone literal` (see Section 6.16.1 [Literals], page 160).

This use of `set-optimizer` assumes that the user does not change the value of a constant with, e.g., `6 ' five >body !`. While `five` has been defined with `create`, that is an implementation detail of `CONSTANT`, and if you don't document it, the user must not rely on it. And if you use `set-optimizer` in a way that assumes that the body does not change (like is done here), you must not document that `create` is used; and conversely, if you document it, you have to write the `compile,` implementation such that it can deal with changing bodies.

Note that the call to `set-optimizer` has to be performed after the call to `set-does>` (or `does>`, because `set-does>` overwrites the `compile,` implementation itself.

We can also apply `set-optimizer` to individual words rather than inside a defining word like `constant`. In this case, the xt of the word passed to optimizer is usually unnecessary and is `drop`ped. As an simple example, let's define a word that is inlined when being compiled:

```
: compile-my2dup ( xt -- )
  drop ]] over over [[ ;

: my2dup over over ;
' compile-my2dup set-optimizer

: foo my2dup ;
see my2dup
```

An alternative way to define `my2dup` is:

```
: my2dup over over ;
opt: drop ]] over over [[ ;
```

`Opt:` starts an anonymous definition that is then (internally) attached to `my2dup` with `set-optimize`.

One disadvantage of these approaches is that `over over` occurs twice. This can be avoided by using `compile-my2dup` to define `my2dup`:

```
: compile-my2dup ( xt -- )
    drop ]] over over [[ ;
```

```
: my2dup [ 0 compile-my2dup ] ;
' compile-my2dup set-optimizer
```

Finally, a convenient way to write words like `my2dup` is to use `inline:` (see Section 6.11.6 [Inline Definitions], page 123), but it is limited to inlining.

The engine `gforth-itc` uses `,` for `compile,` in nearly all cases and `set-optimizer` usually has no effect there.

6.11.10.8 Creating from a prototype

In the above we show how to define a word by first using `create`, and then modifying it with `immediate`, `set-does>`, `set-to`, `set-optimizer` etc.

An alternative way is to create a prototype using these words, and then create a new word from that prototype. This kind of copying does not cover the body, so that has to be allocated and initialized explicitly. Taking `dvalue` above, we could instead define it as:

```
create dvalue-prototype ( -- d )
`2@ set-does>
`dvalue-to set-to

: dvalue ( d "name" -- ; name: -- d )
  ``dvalue-prototype create-from 2, reveal ;
```

An advantage of this approach is that creating words with `dvalue` is now faster.[12] But this advantage is only relevant if the number of words created with this defining word is huge.

`create-from` (*nt* "*name*" –) gforth-1.0

Create a word *name* that behaves like *nt*, but with an empty body. *nt* must be the nt of a named word. The resulting header is not yet `reveal`ed; use `reveal` to reveal it or `latest` to get its xt. Creating a word with `create-from` without using any `set-` words is faster than if you create a word using `set-` words, `immediate`, or `does>`. You can use `noname` with `create-from`.

`reveal` (–) gforth-0.2

Put the current word in the wordlist current at the time of the header definition.

`reveal!` (*xt wid* –) core-ext “reveal-store”

Add *xt* to a wordlist. Mapped to `DEFER!`.

The performance advantage does not extend to using `noname` with the defining word. Therefore we also have

`noname-from` (*xt* –) gforth-1.0

Create a nameless word that behaves like *xt*, but with an empty body. *xt* must be the nt of a nameless word.

Here’s a usage example:

```
``dvalue-prototype noname create-from
latestnt constant noname-dvalue-prototype
```

[12] The non-prototype method first duplicates the header methods of `create`, modify them, and eventually deduplicate them. The `create-from` approach eliminates this overhead.

```
: noname-dvalue ( d -- xt ; xt execution: -- d )
  noname-dvalue-prototype noname-from 2,
  latestxt ;
```

6.11.10.9 Making a word current

Many words mentioned above, such as `immediate` or `set-optimizer` change the "current" or "most recently defined" word. Sometimes you want to change an earlier word. You can do this with

`make-latest` (*nt* –) gforth-1.0

Make *nt* the latest definition, which can be manipulated by `immediate` and `set-*` operations. If you have used (especially compiled) the word referred to by nt already, do not change the behaviour of the word (only its implementation), otherwise you may get a surprising mix of behaviours that is not consistent between Gforth engines and versions.

6.11.10.10 `Const-does>`

A frequent use of `create...does>` is for transferring some values from definition-time to run-time. Other ways of achieving this are closures (see Section 6.28 [Closures], page 248), and with colon definitions (see Section 6.11.10.1 [User-defined defining words with colon definitions], page 126), but another way of achieving this is to use

`const-does>` (*run-time: w*uw r*ur uw ur* "*name*" –) gforth-obsolete "const-does"

Defines *name* and returns.

name execution: pushes *w*uw r*ur*, then performs the code following the `const-does>`.

A typical use of this word is:

```
: curry+ ( n1 "name" -- )
1 0 CONST-DOES> ( n2 -- n1+n2 )
    + ;

3 curry+ 3+
```

Here the `1 0` means that 1 cell and 0 floats are transferred from definition to run-time.

The advantages of using `const-does>` compared to `create...does>` are:

- You don't have to deal with storing and retrieving the values, i.e., your program becomes more writable and readable.
- When using `does>`, you have to introduce a `@` that cannot be optimized away automatically (because the system does not know whether you allow to access the data with `>body...!`). You can address this problem with `set-optimizer` (see Section 6.11.10.7 [User-defined compile-comma], page 134), but `const-does>` avoids it; however, the current implementation is still not particularly efficient.

A Standard Forth implementation of `const-does>` is available in `compat/const-does.fs`.

6.11.11 Deferred Words

The defining word `Defer` allows you to define a word by name without defining its behaviour; the definition of its behaviour is deferred. Here are two situation where this can be useful:

- Where you want to allow the behaviour of a word to be altered later, and for all precompiled references to the word to change when its behaviour is changed.
- For mutual recursion: See Section 6.10.7 [Calls and returns], page 113.

In the following example, `foo` always invokes the version of `greet` that prints "`Good morning`" whilst `bar` always invokes the version that prints "`Hello`". There is no way of getting `foo` to use the later version without re-ordering the source code and recompiling it.

```
: greet ." Good morning" ;
: foo ... greet ... ;
: greet ." Hello" ;
: bar ... greet ... ;
```

This problem can be solved by defining `greet` as a `Defer`red word. The behaviour of a `Defer`red word can be defined and redefined at any time by using `IS` to associate the xt of a previously-defined word with it. The previous example becomes:

```
Defer greet ( -- )
: foo ... greet ... ;
: bar ... greet ... ;
: greet1 ( -- ) ." Good morning" ;
: greet2 ( -- ) ." Hello" ;
' greet2 IS greet  \ make greet behave like greet2
```

Programming style note:

You should write a stack comment for every deferred word, and put only XTs into deferred words that conform to this stack effect. Otherwise it's too difficult to use the deferred word.

One thing to note is that `IS` has special compilation semantics, such that it parses the name at compile time (like `TO`):

```
: set-greet ( xt -- )
  IS greet ;

' greet1 set-greet
```

`Defer` ("*name*" –) core-ext

Define a deferred word *name*; you have to set it to an xt before executing it.
name execution: execute the xt that *name* has most recently been set to execute.
`Is` ***name*** run-time: *(xt1 –)* Set *name* to execute *xt1*.
`Action-of` ***name*** run-time: *(– xt2) xt2* is the xt that *name* has most recently been set to execute.

`IS` (*xt ...* "*name*" –) core-ext

Name is a defer-flavoured word, *...* is optional additional addressing information, e.g., for a defer-flavoured field. At run-time, perform the *is name* semantics: change *name* (with the same additional addressing information) to execute *xt*.

You can extract the xt of a `defer`red word with `action-of`:

```
action-of greet ( xt ) >name id.
```

`action-of` (*interpretation* "*name*" *... – xt; compilation* "*name*" *– ; run-time ... – xt*) core-ext

Name is a defer-flavoured word, *...* is optional additional addressing information, e.g., for a defer-flavoured field. At run-time, perform the *action-of name* semantics: Push the *xt*, that *name* (possibly with additional addressing data on the stack) executes.

One usage for deferred words is the definition of a an action (e.g., initialization) in several pieces, each piece in a different source file dealing with the matters of that source file. This can be done with

```
defer myspeech ( -- )
:noname cr ." <central message>" ; is myspeech

\ and in every source file where you want to add a piece, something like
:noname ( -- )
  cr ." <introduction>"
  [ action-of myspeech compile, ]
  cr ." <conclusion>"
; is myspeech
```

The `[ action-of myspeech compile, ]` calls the previous content of `myspeech`. Gforth offers the words `:is` and `defers` to express the last definition more conveniently:

```
:is myspeech ( -- )
  cr ." <introduction>"
  defers myspeech
  cr ." <conclusion>" ;
```

`:is` ("*name*" –) gforth-experimental "colon-is"

define a noname that is assigned to the deferred word *name* at `;`.

`defers` (*compilation* "*name*" *– ; run-time ... – ...*) gforth-0.2

Compiles the present contents of the deferred word *name* into the current definition. I.e., this produces static binding as if *name* was not deferred.

Another usage is to change a deferred word temporarily, and later change it back. Gforth provides words for supporting this usage. The use of `preserve` is shown in this example:

```
: smalltalk ( -- )
  greet ."  Isn't the weather nice?" ;

\ here GREET performs GREET2
: when-in-rome ( xt -- )
  [: ." Buon Giorno!" ;] is greet
  execute
  preserve greet \ Equivalent to: ['] greet2 is greet
;

' greet1 is greet
greet \ "Good Morning"
```

```
' smalltalk when-in-rome \ "Buon Giorno! Isn't the weather nice?"
greet \ "Hello"
```

`preserve` (*compilation* "*name*" – *; run-time* –) gforth-1.0

Name has to be a defer-flavoured word that does not consume additional stack items for addressing (i.e., not a defer-flavoured field). `Preserve` *`name`* changes *name* at run-time to execute the same XT that it had at compile time. I.e., `Preserve` *`name`* is equivalent to `[ action-of` *`name`* `] literal is` *name*.

`Preserve` is only appropriate when you want to restore the deferred word to a fixed xt. If you want to change a deferred temporarily and then restore its old run-time value, use `wrap-xt`:

```
: when-in-rome2 ( xt -- )
  [: ." Buon Giorno!" ;] ['] greet rot wrap-xt ;

' greet1 is greet
greet \ "Good Morning"
' smalltalk when-in-rome2 \ "Buon Giorno! Isn't the weather nice?"
greet \ "Good Morning"
```

`wrap-xt` (*... xt1 xt2 xt3* – *...*) gforth-1.0

Set deferred word xt2 to xt1 and execute xt3. Restore afterwards.

For implementing words like `wrap-xt` to which you pass the xt of a deferred word, you cannot use `is` and `action-of`, which consume a name from the input stream. Instead, you use the words `defer!` and `defer@`.

`defer!` (*xt xt-deferred* –) core-ext "defer-store"

xt-deferred belongs to a word defined with `defer`, it is changed to execute *xt* on execution.
If *xt-deferred* belongs to another defer-flavoured word (e.g., a defer-flavoured field), the location associated with *... xt-deferred* is changed to execut *xt*.
If *xt-deferred* is the xt of a word that is not defer-flavoured, throw -21 (Unsupported operation).

`defer@` (*... xt-deferred* – *xt*) core-ext "defer-fetch"

If *xt-deferred* belongs to a word defined with `defer`, *xt* represents the word currently associated with the deferred word *xt-deferred*.
If *xt-deferred* belongs to another defer-flavoured word (e.g., a defer-flavoured field), *xt* is the word associated with the location indicated by *... xt-deferred* (e.g., for a defer-flavoured field *...* is the structure address).
If *xt-deferred* is the xt of a word that is not defer-flavoured, throw -21 (Unsupported operation).

A deferred word can only inherit execution semantics from the xt (because that is all that an xt can represent – for more discussion of this see Section 6.15 [Tokens for Words], page 156); by default it will have default interpretation and compilation semantics deriving from this execution semantics. However, you can change the interpretation and compilation semantics of the deferred word in the usual ways:

```
: bar .... ; immediate
Defer fred immediate
```

```
Defer jim

' bar IS jim  \ jim has default semantics
' bar IS fred \ fred is immediate
```

6.11.12 Synonyms

The defining word `synonym` allows you to define a word by name that has the same behaviour as some other word. Here are two situation where this can be useful:

- When you want access to a word's definition from a different word list (for an example of this, see the definitions in the `Root` word list in the Gforth source).
- When you want to create a synonym; a definition that can be known by either of two names (for example, `THEN` and `ENDIF` can be defined as synonyms).

`Synonym` ("*name*" "*oldname*" –) tools-ext

Define *name* to behave the same way as *oldname*: Same interpretation semantics, same compilation semantics, same `to`, `+to`, `is`, `action-of` and `addr` semantics.

Gforth also offers the Gforth-specific `alias`, that allows to define another word with the same execution token, but otherwise default semantics (no copying of compilation or other semantics). You can then change, e.g., the compilation semantics with, e.g., `immediate`.

`Alias` (*xt* "*name*" –) gforth-0.2

Define *name* as a word that performs *xt*. Unlike for deferred words, aliases don't have an indirection overhead when compiled.

Example:

```
: foo ." foo" ; immediate

' foo Alias bar1           \ bar1 is not an immediate word
' foo Alias bar2 immediate \ bar2 is an immediate word
synonym bar3 foo           \ bar3 is an immediate word
: test-bar1 bar1 ; \ no output
test-bar1          \ "foo"
: test-bar2 bar2 ; \ "foo"
test-bar2          \ no output
: test-bar3 bar3 ; \ "foo"
test-bar3          \ no output
```

Both synonyms and aliases have a different nt than the original, but ticking it (or using `name>interpret`) produces the same xt as the original (see Section 6.15 [Tokens for Words], page 156).

6.12 Structures

A structure (aka record) is a collection of fields that are stored together. The fields can have different types and are accessed by name. There are typically several instances of a structure, otherwise programmers tend to prefer using a variable or somesuch for each field.

In Forth you can use raw address arithmetic to access fields of structures, but using field names and defining field access words with the defining words described in this section makes the code more readable.

6.12.1 Standard Structures

The Forth 2012 standard defines a number of words for defining fields and structures.

A typical example of defining a structure with several fields is:

```
0 \ offset of first field, 0 in the usual case
  field: intlist-next ( intlist -- addr1 )
  field: intlist-val  ( intlist -- addr2 )
constant intlist ( -- u )
```

An equivalent alternative way of defining this structure is:

```
begin-structure intlist ( -- u )
  field: intlist-next ( intlist -- addr1 )
  field: intlist-val  ( intlist -- addr2 )
end-structure
```

`Intlist` returns the size of the structure. The convention for the field names here is to prepend the structure name, so that you don't run into conflicts when several structures have `next` and `val` fields; in Forth, by default field names are in the same wordlist (i.e., the same name space) as the other words (including other field names), and trying to use the search order (see Section 6.19 [Word Lists], page 182) for avoiding conflicts is rather cumbersome (unless you use the scope recognizer see Section 6.17.4.1 [Default recognizers], page 171).

You can then use that to allocate an instance of that structure and then use the field words to access the fields of that instance:

```
intlist allocate throw constant my-intlist1
0 my-intlist1 intlist-next !
5 my-intlist1 intlist-val  !

intlist allocate throw constant my-intlist2
my-intlist1 my-intlist2 intlist-next !
7           my-intlist2 intlist-val !

: intlist-sum ( intlist -- n )
\ "intlist" is a pointer to the first element of a linked list
\ "n" is the sum of the intlist-val fields in the linked list
    0 BEGIN ( intlist1 n1 )
        over
    WHILE ( list1 n1 )
        over intlist-val @ +
        swap intlist-next @ swap
    REPEAT
    nip ;

my-intlist2 intlist-sum . \ prints "12"
```

The words for defining structures and fields of various types are:

`begin-structure` ("*name*" – *struct-sys 0*) facility-ext

`end-structure` (*struct-sys* +*n* –) facility-ext

end a structure started with `begin-structure`

`cfield:` (*u1* "*name*" – *u2*) facility-ext "c-field-colon"

Define a char-sized field

`field:` (*u1* "*name*" – *u2*) facility-ext "field-colon"

Define an aligned cell-sized field

`2field:` (*u1* "*name*" – *u2*) gforth-0.7 "two-field-colon"

Define an aligned double-cell-sized field

`ffield:` (*u1* "*name*" – *u2*) floating-ext "f-field-colon"

Define a faligned float-sized field

`sffield:` (*u1* "*name*" – *u2*) floating-ext "s-f-field-colon"

Define a sfaligned sfloat-sized field

`dffield:` (*u1* "*name*" – *u2*) floating-ext "d-f-field-colon"

Define a dfaligned dfloat-sized field

`wfield:` (*u1* "*name*" – *u2*) gforth-1.0 "w-field-colon"

Define a naturally aligned field for a 16-bit value.

`lfield:` (*u1* "*name*" – *u2*) gforth-1.0 "l-field-colon"

Define a naturally aligned field for a 32-bit value.

`xfield:` (*u1* "*name*" – *u2*) gforth-1.0 "x-field-colon"

Define a naturally aligned field for a 64-bit-value.

If you need something beyond these field types, you can use `+field` to define fields of arbitrary size. You have to ensure the correct alignment yourself in this case. E.g., if you want to put one struct inside another struct, you would do it with

```
0
  cfield:                 nested-foo
  aligned intlist +field nested-bar
constant nested
```

In this example the field `nested-bar` contains an intlist structure, so the size of `intlist` is passed to `+field`. An `intlist` must be cell-aligned (it contains cell fields), and this is achieved by aligning the current field offset with `aligned` before the field definition. Our recommendation is to always precede the usage of `+field` with an appropriate alignment word (except if character-alignment is good enough for the field); this ensures that the field will stay correctly aligned even if other fields are later inserted before the `+field`-defined field.

`+field` (*noffset1 nsize* "*name*" – *noffset2*) facility-ext "plus-field"

Defining word; defines *name* `( addr1 -- addr2 )`, where *addr2* is *addr1*+*noffset1*. *noffset2* is *noffset1*+*nsize*.

The first field is at the base address of a structure and the word for this field (e.g., `list-next`) actually does not change the address on the stack. You may be tempted to leave it away in the interest of run-time and space efficiency. This is not necessary, because Gforth and other Forth systems optimize this case: If you compile a first-field word, no code is generated. So, in the interest of readability and maintainability you should include the word for the field when accessing the field.

6.12.2 Value-Flavoured and Defer-Flavoured Fields

In addition to the variable-flavoured fields that produce an address (see Section 6.12.1 [Standard Structures], page 142), Gforth also provides value-flavoured fields. Like all fields, value-flavoured fields consume the start address of the struct, but they produce their value and you can apply `to`, `+to` and (if the field is `addressable:`, see Section 6.11.4 [Values], page 121) `addr` on them. E.g., we can do something like the `intlist` definition (see Section 6.12.1 [Standard Structures], page 142):

```
0
                value: intlist>next ( intlista -- intlista1 )
  addressable: value: intlist>val  ( intlista -- n )
constant intlista ( -- u )
```

This means that there are the following ways of accessing `intlist>val`:

```
intlist>val ( intlista -- n )
->intlist>val ( n intlista -- ) \ aka  to intlist>val
+>intlist>val ( n intlista -- ) \ aka +to intlist>val
addr intlist>val ( intlista -- addr )
```

And here's the earlier example (see Section 6.12.1 [Standard Structures], page 142) rewritten to use `intlista`:

```
intlista allocate throw constant my-intlista1
0 my-intlista1 to intlist>next
5 my-intlista1 to intlist>val

intlista allocate throw constant my-intlista2
my-intlista1 my-intlista2 to intlist>next
7            my-intlista2 to intlist>val

: intlista-sum ( intlista -- n )
\ "intlista" is a pointer to the first element of a linked list
\ "n" is the sum of the intlist>val fields in the linked list
    0 BEGIN ( intlista1 n1 )
        over
    WHILE ( list1 n1 )
        over intlist>val +
        swap intlist>next swap
    REPEAT
    nip ;

my-intlista2 intlista-sum . \ prints "12"
```

Depending on the type of the field, the value can be something different than a single cell.

`value:` (*u1* "*name*" – *u2*) gforth-experimental "value-colon"

Name is a value-flavoured field; in-memory-size: cell; on-stack: cell

`cvalue:` (*u1* "*name*" – *u2*) gforth-experimental "cvalue-colon"

Name is a value-flavoured field; in-memory-size: char; on-stack: unsigned cell

`wvalue:` (*u1* "*name*" – *u2*) gforth-experimental “wvalue-colon”

Name is a value-flavoured field; in-memory-size: 16 bits; on-stack: unsigned cell

`lvalue:` (*u1* "*name*" – *u2*) gforth-experimental “lvalue-colon”

Name is a value-flavoured field; in-memory-size: 32 bits; on-stack: unsigned cell

`scvalue:` (*u1* "*name*" – *u2*) gforth-experimental “scvalue-colon”

Name is a value-flavoured field; in-memory-size: char; on-stack: signed cell

`swvalue:` (*u1* "*name*" – *u2*) gforth-experimental “swvalue-colon”

Name is a value-flavoured field; in-memory-size: 16 bits; on-stack: signed cell

`slvalue:` (*u1* "*name*" – *u2*) gforth-experimental “slvalue-colon”

Name is a value-flavoured field; in-memory-size: 32 bits; on-stack: signed cell

`2value:` (*u1* "*name*" – *u2*) gforth-experimental “two-value-colon”

Name is a value-flavoured field; in-memory-size: 2 cells; on-stack: 2 cells; `+to` performs double-cell addition (`d+`).

`fvalue:` (*u1* "*name*" – *u2*) gforth-experimental “fvalue-colon”

Name is a value-flavoured field; in-memory-size: float; on-stack: float

`sfvalue:` (*u1* "*name*" – *u2*) gforth-experimental “sfvalue-colon”

Name is a value-flavoured field; in-memory-size: 32-bit float; on-stack: float

`dfvalue:` (*u1* "*name*" – *u2*) gforth-experimental “dfvalue-colon”

Name is a value-flavoured field; in-memory-size: 64-bit float; on-stack: float

`zvalue:` (*u1* "*name*" – *u2*) gforth-experimental “zvalue-colon”

Name is a value-flavoured field; in-memory-size: 2 floats; on-stack: 2 floats; `+to` performs componentwise addition.

`$value:` (*u1* "*name*" – *u2*) gforth-experimental “dollar-value-colon”

Name is a value-flavoured field; in-memory-size: cell; on-stack: c-addr u (see Section 6.9.5 [$tring words], page 96); `( c-addr u ) +to` *name* appends c-addr u to the string in the field.

Gforth also has field words for dealing with dynamically-sized arrays. A field for such an array contains just a cell that points to the actual data, and this cell has to be set to 0 before accessing the array the first time. When accessing the field (without operator, or with `to` or `+to`), there has to be the index and the structure address on the stack, with the structure address on top. Any further items consumed by `to` or `+to` are below the index on the stack. The array expands to the size given by the maximum access; any unset elements are 0; for `$value[]` accessing them produces a 0-length (i.e., empty) string.

Here is a usage example:

```
0
  value[]:  bla>x[]
  $value[]: bla>$y[]
constant bla

bla allocate throw constant mybla
mybla bla erase \ set all fields to 0
```

```
5 2 mybla to bla>x[] \ access at index 2
7 0 mybla to bla>x[] \ access at index 0
2 mybla bla>x[] . \ prints "5"
3 mybla bla>x[] . \ prints "0"
"foo" 2 mybla to bla>$y[]  \ access at index 2
"bla" 1 mybla to bla>$y[]  \ access at index 1
"bar" 2 mybla +to bla>$y[] \ access at index 2
0 mybla bla>$y[] . . \ prints "0 0"
1 mybla bla>$y[] type \ prints "bla"
2 mybla bla>$y[] type \ prints "foobar"
```

`value[]:` (*u1* "*name*" – *u2*) gforth-experimental "value-left-bracket-right-bracket-colon"

Name is a value-flavoured array field; in-memory-size: cell; on-stack: cell

`$value[]:` (*u1* "*name*" – *u2*) gforth-experimental "dollar-value-left-bracket-right-bracket-colon"

Name is a value-flavoured array field; in-memory-size: cell; on-stack: c-addr u (see Section 6.9.5 [$tring words], page 96); `( c-addr u ) +to` *name* appends c-addr u to the string in the array element.

Finally, you can define defer-flavoured fields. Here is a usage example:

```
0
  addressable: defer: foo'bar
constant foo

foo allocate throw constant my-foo
:noname ." test" ; my-foo is foo'bar
my-foo foo'bar                  \ prints "test"
my-foo addr foo'bar @ execute   \ prints "test"
my-foo action-of foo'bar execute \ prints "test"
my-foo `foo'bar defer@ execute   \ prints "test"
:noname ." test1" ; my-foo `foo'bar defer!
my-foo foo'bar                  \ prints "test1"
```

`defer:` (*u1* "*name*" – *u2*) gforth-experimental "defer-colon"

Name is a defer-flavoured field

For documentation of `is`, `action-of`, `defer@`, `defer!`, see See Section 6.11.11 [Deferred Words], page 138. Note however, that when used on defer-flavoured fields, all these words consume the start address of the structure, unlike for words defined with `defer`.

6.12.3 Structure Extension

You can create a new structure starting with an existing structure and its fields. E.g., if we also want to define `floatlist`, we can factor out the `...-next` field into a general structure `list` without payload, and then define `intlist` and `floatlist` as extensions of `list`:[13]

```
0
  field: list-next ( list -- addr )
```

[13] This feature is also known as *extended records* in Oberon.

```
constant list ( -- u )

list
  field: intlist-val ( intlist -- addr )
constant intlist ( -- u )

list
  ffield: floatlist-val ( floatlist -- addr )
constant floatlist ( -- u )
```

Note that in this variant there is no `intlist-next` nor a `floatlist-next`, just a `list-next`; so when you use, e.g., a `floatlist`, the organization through extension of `list` is exposed. This may make it harder to refactor things, so you may prefer to also introduce synonyms `intlist-next` and `floatlist-next`.

If you prefer to use `begin-structure`...`end-structure`, you can do the equivalent definition as follows:

```
begin-structure list ( -- u )
  field: list-next ( list -- addr )
end-structure

list extend-structure intlist
  field: intlist-val  ( intlist -- addr )
end-structure

list extend-structure floatlist
  ffield: floatlist-val  ( floatlist -- addr )
end-structure
```

`extend-structure` (*n* "*name*" – *struct-sys n*) gforth-1.0

Start a new structure *name* as extension of an existing structure with size *n*.

6.12.4 Gforth structs

Gforth has had structs before the standard had them; they are a little different, and you can still use them. One benefit of the Gforth structs is that they propagate knowledge of alignment requirements, so if you build the `nested` structure (see Section 6.12.1 [Standard Structures], page 142), you do not need to look inside `intlist` to find out the proper alignment, and you also do not need to mention alignment at all. Instead, this example would look like:

```
struct
  cell% field intlist-next
  cell% field intlist-val
end-struct intlist%

struct
  char%    field nested-foo
  intlist% field nested-bar
end-struct nested%
```

The fields are variable-flavoured, i.e., they work in the same way as those defined with `field:`, `+field` etc.

A disadvantage of the Gforth structs is that, with the standard going for something else, you need to learn additional material to write and understand code that uses them. Another disadvantage of the Gforth structs is that they do not support value-flavoured or defer-flavoured fields. On the balance, in our opinion the disadvantages now outweigh the advantages, so we recommend using the standard structure words (see Section 6.12.1 [Standard Structures], page 142). Nevertheless, here is the documentation for Gforth's structs.

The `list` and `intlist` examples look like this with Gforth structs:

```
struct
  cell% field list-next
end-struct list%

list%
  cell% field intlist-val
end-struct intlist%
```

`Intlist%` contains information about size and alignment, and you use `%size` to get the size, e.g., for allocation:

```
intlist% %size allocate throw constant my-intlist1
```

A shorthand for that is

```
intlist% %alloc constant my-intlist1
```

The fields behave the same way, so the rest of the example works as with standard structures.

In addition to specifying single cells with `cell%`, you can also specify an array of, e.g., 10 cells like this:

```
cell% 10 * field bla-blub
\ equivalent to the standard:
\ aligned 10 cells +field bla-blub
```

You can use `cell% 10 *` not just with `field`, but also in other places where an alignment and size is expected, e.g., with `%alloc`.

`%align` (*align size* –) gforth-0.4 "percent-align"

Align the data space pointer to the alignment *align*.

`%alignment` (*align size* – *align*) gforth-0.4 "percent-alignment"

The alignment of the structure.

`%alloc` (*align size* – *addr*) gforth-0.4 "percent-alloc"

Allocate *size* address units with alignment *align*, giving a data block at *addr*; `throw` an ior code if not successful.

`%allocate` (*align size* – *addr ior*) gforth-0.4 "percent-allocate"

Allocate *size* address units with alignment *align*, similar to `allocate`.

`%allot` (*align size* – *addr*) gforth-0.4 "percent-allot"

Allot *size* address units of data space with alignment *align*; the resulting block of data is found at *addr*.

`cell%` (– *align size*) gforth-0.4 “cell-percent”

`char%` (– *align size*) gforth-0.4 “char-percent”

`dfloat%` (– *align size*) gforth-0.4 “d-float-percent”

`double%` (– *align size*) gforth-0.4 “double-percent”

describes a double cell (equivalent to `cell% 2*`).

`end-struct` (*align size* "*name*" –) gforth-0.2

Define a structure/type descriptor *name* with alignment *align* and size *size1* (*size* rounded up to be a multiple of *align*).
`name` execution: – *align size1*

`field` (*align1 offset1 align size* "*name*" – *align2 offset2*) gforth-0.2

Create a field *name* with offset *offset1*, and the type given by *align size*. *offset2* is the offset of the next field, and *align2* is the alignment of all fields.
`name` execution: *addr1* – *addr2*.
addr2=*addr1*+*offset1*

`float%` (– *align size*) gforth-0.4 “float-percent”

`sfloat%` (– *align size*) gforth-0.4 “s-float-percent”

`%size` (*align size* – *size*) gforth-0.4 “percent-size”

The size of the structure.

`struct` (– *align size*) gforth-0.2

An empty structure, used to start a structure definition.

6.13 User-defined Stacks

Gforth supports user-defined stacks. They are used for implementing features such as recognizer sequences, but you can also define stacks for your own purposes. And these stacks actually support inserting and deleting at both ends, so they are actually double-ended queues (deques). In addition, they support inserting and deleting in the middle.

In Gforth the stacks grow as necessary, but the interface is designed to also support resource-constrained systems that allocate fixed-size stacks, where exceeding the stack size results in an error. So you should provide the size parameter accordingly.

A stack is represented on the data stack by a cell.

`stack` (*n* – *stack*) gforth-experimental

Create an unnamed stack with at least *n* cells space.

`stack:` (*n* "*name*" –) gforth-experimental “stack-colon”

Create a named stack with at least *n* cells space.

`stack>` (*stack* – *x*) gforth-experimental “stack-from”

Pop item *x* from top of *stack*.

`>stack` (*x stack* –) gforth-experimental “to-stack”

Push *x* to top of *stack*.

`>back` (*x stack* –) gforth-experimental "to-back"

Insert *x* at the bottom of *stack*.

`back>` (*stack* – *x*) gforth-experimental "back-from"

Remove item *x* from bottom of *stack*.

`+after` (*x1 x2 stack* –) gforth-experimental "plus-after"

Insert *x1* below every occurrence *x2* in *stack*.

`-stack` (*x stack* –) gforth-experimental "minus-stack"

Delete every occurrence of *x* from anywhere in *stack*.

`set-stack` (*x1 .. xn n stack* –) gforth-experimental

Overwrite the contents of *stack* with *n* elements from the data stack, with *xn* becoming the top of *stack*.

`get-stack` (*stack* – *x1 .. xn n*) gforth-experimental

Push the contents of *stack* on the data stack, with the top element in *stack* being pushed as *xn*.

6.14 Interpretation and Compilation Semantics

In Gforth every named word has interpretation and compilation semantics, i.e., separate actions that are performed in various contexts.

In principle these semantics can be anything and completely independent of each other, but in practice they are usually connected, and words usually have default compilation semantics (compile the interpretation semantics) or immediate compilation semantics (perform the interpretation semantics); a few have other combinations of interpretation and compilation semantics (combined words).

The standard also discusses execution semantics, but it uses them only to define interpretation and/or compilation semantics, so they are not as essential as interpretation and compilation semantics. In particular, for every word in the standard that has both interpretation and execution semantics, they are the same. In Gforth (since 1.0), they are always the same, and this manual uses the terms interchangeably, usually preferring interpretation semantics. In the description of defining words, you see "*name* execution", which describes the interpretation/execution semantics of *name*.

Some named words also have some of `to`/`+to`/`action-of`/`is`/`addr` *`name`* semantics, but these are mostly discussed elsewhere (see Section 6.11.4 [Values], page 121, see Section 6.11.11 [Deferred Words], page 138, see Section 6.11.10.6 [Words with user-defined TO etc.], page 132)

6.14.1 Where are interpretation semantics used?

The most common use of the interpretation semantics of a word *w* is when *w* is text-interpreted in interpretation state, the default state of the text interpreter.

I.e., when you start Gforth and type

```
s" hello" type
```

the text interpreter performs the interpretation semantics of the words `s"` and `type`.

Also, when you get the execution token of a word *w* with `` `w ``, `' w` or `['] w` (see Section 6.15.1 [Execution token], page 156), the execution token represents the interpretation semantics.

When you get the execution token of the most recently defined word with `latestxt` (see Section 6.11.7 [Anonymous Definitions], page 124), that also refers to the interpretation semantics of the word.

Finally, `name>interpret` (see Section 6.15.2 [Name token], page 158) produces an execution token that represents the interpretation semantics of the word.

6.14.2 Where are compilation semantics used?

The most common use of the compilation semantics of a word *w* is when *w* is text-interpreted in compile state, the state right after starting a definition with, e.g., `:`.

```
: hello
  s" hello" type ;
```

In this example, the text interpreter performs the compilation semantics of `s"`, `type` and `;` (after first performing the interpretation semantics of `:`)

When you postpone a word, you also use the compilation semantics.

```
: compile-+ ( -- ) \ compiled code: ( n1 n2 -- n )
  POSTPONE + ;

: foo ( n1 n2 -- n )
  [ compile-+ ] ;

see foo
```

Here the `POSTPONE +` compiles the compilation semantics of `+` into `compile-+` (By contrast, just writing `+` in this place would result in *performing* the compilation semantics of `+`, and because this is a word with default compilation semantics, that would compile the *execution/interpretation* semantics of `+`). In the definition of `foo`, (the interpretation semantics of) `compile-+` is performed, which in turn performs the compilation semantics of `+`, i.e., it compiles `+` into `foo`.

The compilation semantics is represented by a compilation token (see Section 6.15.3 [Compilation token], page 160). You can get the compilation token of a word *w* with `` ``w name>compile ``, `comp' w`, or `[comp'] w`. The first form first gets the name token of *w* and then accesses the compilation token with `name>compile`.

6.14.3 Which semantics do existing words have?

For words built into Gforth, the documentation specifies the semantics.

Most words have default compilation semantics. For such words (e.g., `!`, see Section 6.8.5 [Memory Access], page 81) the documentation describes the interpretation/execution semantics without explicitly labeling it as such. The compilation semantics of these words is to compile the interpretation semantics into the current definition; the stack effect of performing the default compilation semantics is `( -- )`.

Some words have non-default compilation semantics. This is either indicated by labels for interpretation, compilation, and/or run-time in the stack effects (e.g., for `IF`, see Section 6.10.6 [Arbitrary control structures], page 112), or by having separate paragraphs for

interpretation, compilation, and/or run-time in the prose (e.g., for `S"`, see Section 6.9.3 [String and character literals], page 91).

You may wonder about the run-time semantics mentioned in the previous paragraphs. For some words (e.g., `if`), the compilation semantics compiles something that is not the interpretation/execution semantics. We (and the standard) describe the behaviour of the code that these words compile with under the label "run-time semantics"; if you see "run-time" in a word description (e.g., in its stack effect), that usually refers to run-time semantics that the compilation semantics of the word compiles.

Concerning the description of the various semantics, both the standard and this manual describe the interpretation/execution semantics of words with default semantics without preceding these semantics with a label (the label "execution" or "interpretation" would be appropriate). The compilation semantics of such words are the implied default compilation semantics (see Section 6.14.4 [What semantics do normal definitions have?], page 152).

For words that have some non-default semantics, the standard specifies the different semantics of the word in separate subsections, each preceded with a label ("interpretation:", "compilation:", and, if necessary, "execution:" or "run-time:"[14]). This manual often takes a more informal approach. The approach taken in this manual may be more accommodating for everyday use, while the standard approach is more precise for reasoning about details of the language.

6.14.4 What semantics do normal definitions have?

Most defining words normally produce words with default interpretation semantics and default compilation semantics; those that do not (e.g, `synonym` or `interpret/compile:`) are documented appropriately.

The interpretation semantics of the newly defined word *name* are described in the "*name* execution:" part of the description of the defining word. Things are a little more complicated for colon definitions (see Section 6.11.5 [Colon Definitions], page 123) and words using `create`...`does>` (see Section 6.11.10.2 [User-defined defining words using CREATE], page 126), but again, the description of what these words do is about the interpretation semantics.

For a word *w* with default compilation semantics, the compilation semantics are to compile the interpretation semantics. More formally: to append the interpretation semantics of *w* to the interpretation semantics of the current definition. As an example, consider the definition

```
: name ... w ... ;
```

Here the interpretation semantics of *w* is appended to the interpretation semantics of *name*.

6.14.5 How to define immediate words

You can change the compilation semantics of a word to be the same as the interpretation semantics with

`immediate` (–) core

[14] In some cases the standard leaves the subsection for interpretation or compilation semantics away, and leaves it to the default mechanism to derive those semantics from execution semantics.

Change the compilation semantics of the most recently defined word to be the same as its interpretation semantics.

A contrived example:

```
: [foo]
  ." foo" ; immediate

: bar
  [foo] ; \ prints "foo"
bar \ no output
```

The `immediate` causes `[foo]` to perform the interpretation semantics during the definition of `bar` rather than compiling them. A convention sometimes (but not always) used for immediate words is to have their names in brackets, e.g. `[']`.

A common use of `immediate` is to define macros (see Section 6.16.2 [Macros], page 162).

The text interpretation of a macro in interpret state is often a mistake, so you can turn the macro into a `compile-only` word with

`compile-only` (–) gforth-0.2

Mark the last definition as compile-only; as a result, the text interpreter and `'` will warn when they encounter such a word.

Example:

```
: endif
  postpone then ; immediate compile-only

: foo
  if ." true" endif ;

endif \ "warning: endif is compile-only"
```

The warning is followed by a stack underflow error because `then` wants to consume an *orig* (see Section 6.10.6 [Arbitrary control structures], page 112).

Note that compiling code while the text interpreter is in interpret state is not a problem in itself, even if a number of words are marked `compile-only`. A more serious problem is compiling code if the current definition is not an unfinished colon definition: there is no way to run the resulting code. Gforth warns about that even if a word is not marked compile-only or if you text-interpret it in compile state:

```
: compile-+
  postpone + ;

: foo [ compile-+ ] ; \ no warning; interpretation semantics of compile-+

compile-+ \ warning: Compiling outside a definition
if        \ warning: IF is compile-only
          \ warning: Compiling outside a definition
compile-+ \ warning: Compiling outside a definition
then      \ warning: THEN is compile-only
] if      \ warning: Compiling outside a definition
```

```
  +       \ warning: Compiling outside a definition
  then
[
```

Note that switching to compile state in the last four lines silences the "is compile-only" warnings, because in these lines the compilation semantics of the words is performed.

Why does `then` not produce "Compiling outside a definition" warnings in the example above? `Then` does not generate any code by itself, it just changes the target of the code compiled by the matching `if` or `ahead`.

`restrict` (–) gforth-0.2

A synonym for `compile-only`

6.14.6 How to define combined words

In a few cases (and most of those are a bad idea) you want to define a word that has some other combination of interpretation and compilation semantics than words with default compilation semantics or immediate words; we call such a word a combined word[15]). The following contrived example shows how you can define a combined word:

```
: foo ." foo" ;
: bar ." bar" ;
' foo ' bar interpret/compile: foobar1

foobar1     \ "foo"
] foobar1 [ \ "bar"
```

`interpret/compile:` (*int-xt comp-xt* "*name*" –) gforth-0.2 "interpret-slash-compile-colon"

Defines *name*.
Name execution: execute *int-xt*.
Name compilation: execute *comp-xt*.

There are two kinds of uses for combined words:

One use of combined words is parsing words that should be copy-pasteable between interpreted and compiled code; these words should parse at text-interpret time both in their interpretation and their compilation semantics (like an immediate word), but then should perform an action in their interpretation semantics and compile that action in their compilation semantics, like a normal word. An example is `."` in Gforth:

```
: ."-int ( 'ccc"' -- )  '"' parse type ;
: ."-comp ( 'ccc"' -- ) '"' parse postpone sliteral postpone type ;

' ."-int ' ."-comp interpret/compile: ."
( interpretation 'ccc"' -- ; compilation 'ccc"' -- ; run-time -- )

." foo"         \ "foo"
: foo ." foo" ;
foo             \ "foo"
```

[15] Some people call combined words "NDCS", but immediate words also have non-default compilation semantics

The parsing code is the same in both cases, the action `type` is directly executed in the interpretation semantics and compiled in the compilation semantics. The compilation semantics also contains `postpone sliteral` to transfer the parsed string from text-interpretation time to the run-time of the action. This kind of parse/literal/action split with the use of `postpone` is typical for the implementations of the compilation semantics of such parsing words, and the interpretation semantics consist just of the parse and the action parts.

We discourage the definition of additional combined words for copy-pasteability. They do not work as intended within `]]...[[` (see Section 6.16.2 [Macros], page 162) and their behaviour is also confusing in other contexts, e.g., when ticking or `postpone`ing such a word. A way to achieve copy-pasteability without needing to define combined words is recognizers (see Section 6.17.4 [Recognizers], page 171). `"foo" type` uses the string recognizer (see Section 6.17.4.1 [Default recognizers], page 171) and can be copied and pasted between interpreted code, compiled code and code inside `]]...[[` without problem.

On the other hand, combined words are still far better than `state`-smart words.[16]

The other kind of use of combined words is for words like `[:` (see Section 6.11.8 [Quotations], page 125). These are not parsing words, but `[:...;]` sequences should be copy-pasteable between interpreted and compiled code; the whole sequence pushes an xt at its run-time. At text-interpret time, it restores the state at the end to what it was at the start. Ideally we would find a clean way to implement all this without needing combined words, but for now the implementation is pretty messy, including combined words.

Some people also have the idea to use combined words for optimization. However, the resulting words do not work as intended with `[compile]` (see Section 6.16.2 [Macros], page 162). Gforth has a better mechanism for optimization: `set-optimizer` (see Section 6.11.10.7 [User-defined compile-comma], page 134).

Some people worry about the aesthetics of `interpret/compile:` and have proposed alternative syntaxes, and the following ones are supported in Gforth:

```
: foobar2
  ." foo" ;
[: ." bar" ;] set-compsem

foobar2     \ "foo"
] foobar2 [ \ "bar"

: foobar3
  ." foo" ;
compsem:
  ." bar" ;

foobar3     \ "foo"
] foobar3 [ \ "bar"

: foobar4
```

[16] `State`-smart words are immediate words that do `state`-dependent things at run-time. For a more detailed discussion of this topic, see M. Anton Ertl, *'`State`'-smartness—Why it is Evil and How to Exorcise it (`https://www.complang.tuwien.ac.at/papers/ertl98.ps.gz`)*, EuroForth '98.

```
  ." bar" ;
intsem:
  ." foo" ;

foobar4     \ "foo"
] foobar4 [ \ "bar"
```

You can use `where` (see Section 6.30.3 [Locating uses of a word], page 259) to find out how rarely which syntax is used in Gforth.

`set-compsem` (*xt* –) gforth-experimental

change compilation semantics of the last defined word

`compsem:` (–) gforth-experimental "comp-sem-colon"

Changes the compilation semantics of the current definition to perform the definition starting at the `compsem:`.

`intsem:` (–) gforth-experimental "int-sem-colon"

The current definition's compilation semantics are changed to perform its interpretation semantics. Then its interpretation semantics are changed to perform the definition starting at the `intsem:` (without affecting the compilation semantics). Note that if you then call `immediate`, the compilation semantics are changed to perform the word's new interpretation semantics.

6.15 Tokens for Words

This section describes the creation and use of tokens that represent words.

6.15.1 Execution token

An *execution token* (*xt*) represents some behaviour of a word. You can use `execute` to invoke the behaviour represented by the xt and `compile,` (see Section 6.16.2 [Macros], page 162) to compile it into the current definition. Other uses include deferred words (see Section 6.11.11 [Deferred Words], page 138).

In particular, there is *the* execution token of a word that represents its interpretation semantics (see Section 6.14 [Interpretation and Compilation Semantics], page 150).

For a named word *x*, you can use `` `x `` to get its execution token:

```
5 `. ( n xt )
execute ( )        \ "5"
: foo `. execute ;
5 foo              \ "5"
```

However, the tick-recognizer that recognizes the `` ` `` prefix is a Gforth extension, so you may prefer to use the Standard Forth words:

`'` ("*name*" – *xt*) core "tick"

xt represents *name*'s interpretation semantics.

`[']` (*compilation.* "*name*" – *; run-time.* – *xt*) core "bracket-tick"

xt represents *name*'s interpretation semantics.

These are parsing words (whereas `` `x `` is treated as a literal by a recognizer), and you may find the behaviour in interpreted and compiled code unintuitive:

```
5 ' .   ( n xt )
execute ( )          \ "5"

: foo ['] . ;
5 foo execute        \ "5"

: bar ' dup ;
5 bar . drop execute \ "5"
```

`'` parses at run-time, so if you put `'` in a colon definition, as in `bar`, it does not consume the next word in the colon definition, but the next word at run-time (i.e., the `.` in the invocation of `bar`). If you want to push the xt of *x* inside a colon definition, write `` `x `` or `['] x`.

Gforth's `` `x ``, `'` and `[']` warn when you use them on compile-only words, because such usage may be non-portable between different Forth systems.

You get the xt of the most recently defined word with `latestxt` (see Section 6.11.7 [Anonymous Definitions], page 124). For words defined using `noname`, this is the usual way of getting a token.

For words defined with `:noname`, the definition already pushes the xt, so you do not need to use `latestxt` for `:noname`-defined words.

```
:noname ." hello" ;
execute
```

An xt occupies one cell and can be manipulated like any other cell.

In Standard Forth the xt is just an abstract data type (i.e., defined by the operations that produce or consume it). The concrete implementation (since Gforth 1.0) is the body address (for old hands: PFA) of the word; in Gforth 0.7 and earlier, the xt was implemented as code field address (CFA, 2 cells before the PFA).

`execute` (*xt* –) core "execute"

Perform the semantics represented by the execution token, *xt*.

`execute-exit` (*compilation* – *; run-time xt nest-sys* –) gforth-1.0

Execute `xt` and return from the current definition, in a tail-call-optimized way: The return address `nest-sys` and the locals are deallocated before executing `xt`.

`perform` (*a-addr* –) gforth-0.2 "perform"

`@ execute`.

`[Noop]` is sometimes used as a placeholder execution token:

`[noop]` (–) gforth-experimental "bracket-noop"

Does nothing, both when executed and when compiled.

`noop` (–) gforth-0.2 "noop"

Does nothing. However, code generation does not optimize it away; use `[noop]` for that.

6.15.2 Name token

A *name token* (*nt*) represents a word, primarily a named word, but in Gforth since 1.0 unnamed words have a name token, too.

The name token is a cell-sized abstract data type that occurs as argument or result of the words below.

The difference between name token and execution token is that an execution token represents one behaviour, whereas a name token represents a word and all its properties, in particular its name and its behaviours (interpretation semantics, compilation semantics, possibly `TO` *word* semantics, etc.)

You get the nt of a word *x* with ``` ``x ``` (since Gforth 1.0) or with

`find-name` (*c-addr u – nt | 0*) gforth-0.2

Find the name *c-addr u* in the current search order. Return its *nt*, if found, otherwise 0.

`find-name-in` (*c-addr u wid – nt | 0*) gforth-1.0

Find the name *c-addr u* in the word list *wid*. Return its *nt*, if found, otherwise 0.

`latest` (*– nt|0*) gforth-0.6

If the most recent word defined in the current section has a name, *nt* is its name token; otherwise, return 0.

`latestnt` (*– nt|0*) gforth-1.0

nt is the name token of the most recent word (named or unnamed) defined in the current section.

`>name` (*xt – nt|0*) gforth-0.2 “to-name”

For most words (all words with the default implementation of `name>interpret`), `>name` is the inverse of `name>interpret`: for these words *nt* `name>interpret` produces *xt*. For the other words `name` produces an nt for which *nt* `default-name>int` produces *xt*. Returns 0 if *xt* is not an xt (using a heuristic check that has a small chance of misidentifying a non-xt as xt), or (before Gforth 1.0) if *xt* is of an unnamed word. As of Gforth 1.0, unnamed words have nts, too, and `>name` produces an nt for xts of unnamed words.

`xt>name` (*xt – nt*) gforth-1.0 “xt-to-name”

If *xt* is an execution token, produces the same *nt* as `>name`. Otherwise, *nt* is an arbitrary value.

You can get all the nts in a wordlist with

`traverse-wordlist` (*... xt wid – ...*) tools-ext

perform *xt* (... nt – f ...) once for every word *nt* in the wordlist *wid*, until *f* is false or the wordlist is exhausted. *xt* is free to use the stack underneath.

You can use the nt to access the interpretation and compilation semantics of a word, its name, and the next word in the wordlist:

`name>interpret` (*nt – xt*) tools-ext “name-to-interpret”

xt represents the interpretation semantics of the word *nt*.

`name>compile` (*nt – xt1 xt2*) tools-ext “name-to-compile”

xt1 xt2 is the compilation token for the word *nt* (see Section 6.15.3 [Compilation token], page 160).

`name>string` (*nt – addr u*) tools-ext “name-to-string”

addr count is the name of the word represented by *nt*.

`id.` (*nt* –) gforth-0.6 "i-d-dot"

Print the name of the word represented by *nt*.

`.id` (*nt* –) gforth-0.6 "dot-i-d"

F83 name for `id.`.

`compile-only?` (*nt* – *flag*) gforth-1.0 "compile-only-question"

true if *nt* is marked as compile-only.

`obsolete?` (*nt* – *flag*) gforth-1.0 "obsolete-question"

true if *nt* is obsolete, i.e., will be removed in a future version of Gforth.

`name>link` (*nt1* – *nt2* / *0*) gforth-1.0 "name-to-link"

For a word *nt1*, returns the previous word *nt2* in the same wordlist, or 0 if there is no previous word.

As a usage example, the following code lists all the words in `forth-wordlist` with non-default compilation semantics (including immediate words):

```
: ndcs-words ( wid -- )
  [: dup name>compile ['] compile, <> if over id. then 2drop true ;]
  swap traverse-wordlist ;

forth-wordlist ndcs-words
```

This code assumes that a word has default compilation semantics if the xt part of its compilation token is the xt of `compile,`.

Since Gforth 1.0 (but not in earlier versions or many other Forth systems), nameless words (see Section 6.11.7 [Anonymous Definitions], page 124) have nts and compilation semantics, and `name>string` works on them (producing a zero-length name). They are not in a wordlist, however. You can get the nt of a nameless word with `latestnt`.

Since Gforth 1.0, for most words the concrete implementation of their nt has the same numeric value as the xt that `name>interpret` produces for the nt. However for word *w* that is a synonym, alias, or is defined with `interpret/compile:`, `intsem:` etc., the xt produced by `name>interpret` has a different numeric value than the nt (and using `>name` on these xts will not produce the nt of *w*). Therefore, you cannot use xts and nts interchangeably, even if you are prepared to write code specific to Gforth 1.0.

The closest thing to the nt in classic Forth systems like fig-Forth is the name field address (NFA), but there are significant differences: in older Forth systems each word has a unique NFA, LFA, CFA and PFA (in this order, or LFA, NFA, CFA, PFA) and there are words for getting from one to the next. By contrast, in Gforth in general there is an n:1 relation between name tokens and the xt representing interpretation semantics; i.e., when you pass different nts to `name>interpret`, the result may be the same xt.

Another difference is that the NFA usually points to the start of the header, whereas the nt in Gforth 1.0 points to the body (and header fields are accessed with a negative offset).

Moreover, all of the header fields of the old systems correspond to fields in Gforth, but Gforth 1.0 has a few additional ones (see Section 6.34.1 [Header fields], page 299).

6.15.3 Compilation token

The compilation semantics of a word is represented by a *compilation token* consisting of two cells: *xt1 xt2*.

You get the compilation token of, e.g., `if` in a standard way with `name>compile`, e.g., `s" if" find-name name>compile`, but there are also parsing words to get the compilation token of a word:

`[COMP']` (*compilation* "*name*" – *; run-time – xt1 xt2*) gforth-0.2 "bracket-comp-tick"

Compilation token *xt1 xt2* represents *name*'s compilation semantics.

`COMP'` ("*name*" – *xt1 xt2*) gforth-0.2 "comp-tick"

Compilation token *xt1 xt2* represents *name*'s compilation semantics.

You can perform the compilation semantics represented by the compilation token with `execute`. `Execute`ing the compilation token consumes the whole compilation token and there is possibly an additional stack effect determined by the represented compilation semantics (e.g., `execute`ing the compilation token of `if` pushes an orig).

You can compile the compilation semantics represented by a compilation token with `postpone,`. I.e., ``` ``x name>compile postpone, ``` is equivalent to `postpone x`.

`postpone,` (*xt1 xt2* –) gforth-0.2 "postpone-comma"

Compile the compilation semantics represented by the compilation token *xt1 xt2*.

Implementation: The top cell *xt2* of a compilation token *xt1 xt2* is an execution token that represents either `execute` or `compile,`[17] *Xt1* is more specific to the represented word; for most words, it is the xt produced by `name>interpret`, but there are exceptions, e.g., for words defined with `interpret/compile:`.

6.16 Compiling words

In contrast to most other languages, Forth has no strict boundary between compilation and run-time. E.g., you can run arbitrary code between defining words (or for computing data used by defining words like `constant`). Moreover, `Immediate` (see Section 6.14 [Interpretation and Compilation Semantics], page 150) and `[...]` (see Section 6.16.1 [Literals], page 160) allow running arbitrary code while compiling a colon definition (exception: any dictionary space you allot must be in a different section, see Section 6.8.3 [Sections], page 78).

6.16.1 Literals

The simplest and most frequent example is to compute a literal during compilation. E.g., the following definition prints an array of strings, one string per line:

```
: .strings ( addr u -- ) \ gforth
    2* cells bounds U+DO
        cr i 2@ type
    2 cells +LOOP ;
```

With a simple-minded compiler like Gforth 0.7, this computes `2 cells` on every loop iteration. You can compute this value at compile time and compile it into the definition like this:

[17] Depending upon the compilation semantics of the word. If the word has default compilation semantics, *xt2* will represent `compile,`. Otherwise (e.g., for immediate words), *xt2* will represent `execute`.

```
: .strings ( addr u -- ) \ gforth
    2* cells bounds U+DO
        cr i 2@ type
    [ 2 cells ] literal +LOOP ;
```

`[` switches the text interpreter to interpret state (you will get an `ok` prompt if you type this example interactively and insert a newline between `[` and `]`), so it performs the interpretation semantics of `2 cells`; this computes a number. `]` switches the text interpreter back into compile state. It then performs `Literal`'s compilation semantics, which are to compile this number into the current word. You can decompile the word with `see .strings` to see the effect on the compiled code.[18]

You can also optimize the `2* cells` into `[ 2 cells ] literal *` in this way.[19]

`[` (–) core "left-bracket"

Enter interpretation state. Immediate word.

`]` (–) core "right-bracket"

Enter compilation state.

`Literal` (*compilation n – ; run-time – n*) core

Compilation semantics: (*n* –) compile the run-time semantics.
Run-time Semantics: (– *n*).
Interpretation semantics: not defined in the standard.

`lit,` (*x* –) gforth-1.0 "lit-comma"

This is a non-immediate variant of `literal`
Execution semantics: (*x* –) Compile the following semantics:
Compiled semantics: (– *x*).

`ALiteral` (*compilation addr – ; run-time – addr*) gforth-0.2

Works like `literal`, but (when used in cross-compiled code) tells the cross-compiler that the literal is an address.

`]L` (*compilation: n – ; run-time: – n*) gforth-0.5 "right-bracket-L"

equivalent to `] literal`

There are also words for compiling other data types than single cells as literals:

`2Literal` (*compilation w1 w2 – ; run-time – w1 w2*) double "two-literal"

Compilation semantics: (*w1 w2* –) compile the run-time semantics.
Run-time Semantics: (– *w1 w2*).
Interpretation semantics: not defined in the standard.

`2lit,` (*x1 x2* –) gforth-1.0 "two-lit-comma"

This is a non-immediate variant of `2literal`
Execution semantics: (*x1 x2* –) Compile the following semantics:
Compiled semantics: (– *x1 x2*).

`FLiteral` (*compilation r – ; run-time – r*) floating "f-literal"

---

[18] In Gforth 1.0, the compiler performs constant folding, and `see` of the original `.strings` will show the same effect.

[19] In Gforth 1.0, `2 cells *` is sufficient.

Compilation semantics: (*r* –) compile the run-time semantics.
Run-time Semantics: (– *r*).
Interpretation semantics: not defined in the standard.

`flit,` (*r* –) gforth-1.0 “flit-comma”

This is a non-immediate variant of `fliteral`
Execution semantics: (*r* –) Compile the following semantics:
Compiled semantics: (– *r*).

`SLiteral` (*Compilation c-addr1 u – ; run-time – c-addr2 u*) string

Compilation semantics: (*c-addr1 u* –) Copy the string described by *c-addr1 u* to *c-addr2 u* and compile the run-time semantics.
Run-time Semantics: (– *c-addr2 u*).
Interpretation semantics: not defined in the standard.

`slit,` (*c-addr1 u* –) gforth-1.0 “slit-comma”

This is a non-immediate variant of `sliteral`
Execution semantics: (*c-addr1 u* –) Copy the string described by *c-addr1 u* to *c-addr2 u* and compile the following semantics:
Compiled semantics: (– *c-addr2 u*).

You might be tempted to pass data from outside a colon definition to the inside on the data stack. The straightforward approach to do this does not work, because `:` pushes a colon-sys, making stuff below unaccessible. E.g., this does not work:

```
5 : foo literal ; \ error: "unstructured"
```

Instead, you have to pass the value in some other way, e.g., through the return stack:

```
5 >r : foo [ r> ] literal ;
```

The interpretive use of the return stack is Gforth-specific; the use of a variable also works on other Forth systems:

```
variable temp
5 temp !
: foo [ temp @ ] literal ;
```

6.16.2 Macros

`Literal` and friends compile data values into the current definition. You can also write words that compile other words into the current definition. E.g.,

```
: compile-+ ( -- ) \ compiled code: ( n1 n2 -- n )
  POSTPONE + ;

: foo ( n1 n2 -- n )
  [ compile-+ ] ;
1 2 foo .
```

This is equivalent to `: foo + ;` (`see foo` to check this). What happens in this example? `Postpone` compiles the compilation semantics of `+` into `compile-+`; later the text interpreter executes `compile-+` and thus the compilation semantics of `+`, which compile (the execution semantics of) `+` into `foo`.

`postpone` ("*name*" –) core

Compiles the compilation semantics of *name*.

Compiling words like `compile-+` are usually immediate (see Section 6.14.5 [How to define immediate words], page 152) so you do not have to switch to interpret state to execute them; modifying the last example accordingly produces:

```
: [compile-+] ( compilation: --; interpretation: -- )
  \ compiled code: ( n1 n2 -- n )
  POSTPONE + ; immediate

: foo ( n1 n2 -- n )
  [compile-+] ;
1 2 foo .
```

You will occasionally find the need to POSTPONE several words; putting POSTPONE before each such word is cumbersome, so Gforth provides a more convenient syntax: `]] ... [[`. This allows us to write `[compile-+]` as:

```
: [compile-+] ( compilation: --; interpretation: -- )
  ]] + [[ ; immediate
```

`]]` (–) gforth-0.6 "right-bracket-bracket"

Switch into postpone state: All words and recognizers are processed as if they were preceded by `postpone`. Postpone state ends when `[[` is recognized.

The unusual direction of the brackets indicates their function: `]]` switches from compilation to postponing (i.e., compilation of compilation), just like `]` switches from immediate execution (interpretation) to compilation. Conversely, `[[` switches from postponing to compilation, ananlogous to `[` which switches from compilation to immediate execution.

The real advantage of `]] ... [[` becomes apparent when there are many words to POSTPONE. E.g., the word `compile-map-array` (see Section 3.35 [Advanced macros Tutorial], page 40) can be written much shorter as follows:

```
: compile-map-array ( compilation: xt -- ; run-time: ... addr u -- ... )
\ at run-time, execute xt ( ... x -- ... ) for each element of the
\ array beginning at addr and containing u elements
  {: xt: xt :}
  ]] cells over + swap ?do
    i @ xt 1 cells +loop [[ ;

: sum-array ( addr u -- n )
  0 [ ' + compile-map-array ] ;
```

If you then say `see sum-array`, it shows the following code:

```
: sum-array
  #0 over + swap ?do
    i @ + #8 +LOOP
;
```

In addition to `]]...[[`, this example shows off some other features:

- It uses a defer-flavoured (defined with `xt:`) local `xt`; mentioning such a local inside `]]...[[` results in `compile,`ing the xt in the local, i.e., `]] xt [[` is equivalent to `action-of xt compile,`.

- Not used in the example, but related to the previous point: For a normal (value-flavoured) local, using it inside `]]...[[` compiles the value of the local, i.e., `]] x [[` is equivalent to `x ]] literal [[`.
- It uses the literal `1` inside `]]...[[`. This results in `postpone`ing the `1`, i.e. compiling it when `compile-map-array` is run. `]] 1 [[` is equivalent to `1 ]] literal [[`.
- When `compile-map-array` is run (while `sum-array` is compiled), `1 cells` is compiled and optimized into `#8` by Gforth's constant folding.

Note that parsing words such as `s\"` don't parse at postpone time and therefore not inside `]]...[[`. Instead of `s\" mystring\n"` you can use the string recognizer and write `"mystring\n"`, which works inside `]]...[[`. Likewise, the parsing word `[']` does not parse inside `]]...[[` while the recognizer notation starting with `` ` `` works inside `]]...[[`.

But if you prefer to use `s\"` (or have a parsing word that has no recognizer replacement), you can do it by switching back to compilation:

```
]] ... [[ s\" mystring\n" ]] sliteral ... [[
```

Definitions of `]]` and friends in Standard Forth are provided in `compat/macros.fs`.

Immediate compiling words are similar to macros in other languages (in particular, Lisp). The important differences to macros in, e.g., C are:

- You use the same language for defining and processing macros, not a separate preprocessing language and processor.
- Consequently, the full power of Forth is available in macro definitions. E.g., you can perform arbitrarily complex computations, or generate different code conditionally or in a loop (e.g., see Section 3.35 [Advanced macros Tutorial], page 40). This power is very useful when writing a parser generators or other code-generating software.
- Macros defined using `postpone` etc. deal with the language at a higher level than strings; name binding and other resolutions happen at macro definition time, so you can avoid the pitfalls of name collisions that can happen in C macros. Of course, Forth is a liberal language and also allows you to define text-based macros using `evaluate` (see below).

You may want the macro to compile a number into a word. The word to do it is `literal`, but you have to `postpone` it, so its compilation semantics take effect when the macro is executed, not when it is compiled:

```
: [compile-5] ( -- ) \ compiled code: ( -- n )
  5 POSTPONE literal ; immediate

: foo [compile-5] ;
foo .
```

A more convenient, but less portable way to write `[compile-5]` is:

```
: [compile-5] ( -- ) \ compiled code: ( -- n )
  ]] 5 [[ ; immediate
```

You may want to pass parameters to a macro, that the macro should compile into the current definition. If the parameter is a number, then you can use `postpone literal` (similar for other values).

If you want to pass a word that is to be compiled, the usual way is to pass an execution token and `compile,` it:

```
: twice ( xt -- ) \ compiled code: ... -- ...
  dup compile, compile, ;

: 2+ ( n1 -- n2 )
  [ ' 1+ twice ] ;
```

`compile,` (*xt* –) core-ext "compile-comma"

Append the semantics represented by *xt* to the current definition. When the resulting code fragment is run, it behaves the same as if *xt* is `execute`d.

A more convenient, but less portable way to write `twice` is:

```
: twice {: xt: xt -- :} \ compiled code: ... -- ...
  ]] xt xt [[ ;
```

An alternative that allows you to pass the compilation semantics as parameters is to use the compilation token (see Section 6.15.3 [Compilation token], page 160). The same example in this technique:

```
: ctwice ( ... ct -- ... ) \ compiled code: ... -- ...
  2dup 2>r execute 2r> execute ;

: 2+ ( n1 -- n2 )
  [ ``1+ name>compile ctwice ] ;
```

In particular, you want to pass the compilation token and `execute` it in case of words without interpretation semantics or with non-default and non-immediate compilation semantics (i.e., not for `1+`).

In the example above `2>r` and `2r>` ensure that `ctwice` works even if the executed compilation semantics has an effect on the data stack.

You can also define complete definitions with these words; this provides an alternative to using `does>` (see Section 6.11.10 [User-defined Defining Words], page 125). E.g., instead of

```
: curry+ ( n1 "name" -- )
    CREATE ,
DOES> ( n2 -- n1+n2 )
    @ + ;
```

you could define

```
: curry+ ( n1 "name" -- )
  \ name execution: ( n2 -- n1+n2 )
  >r : r> POSTPONE literal POSTPONE + POSTPONE ; ;

-3 curry+ 3-
see 3-
```

The sequence `>r : r>` is necessary, because `:` puts a colon-sys on the data stack that makes everything below it unaccessible.

A more convenient, but less portable way to define `curry+` is:

```
: curry+ ( n1 "name" -- )
```

```
\ name execution: ( n2 -- n1+n2 )
{: n1 :} : ]] n1 + ; [[ ;
```

This way of writing defining words is sometimes more, sometimes less convenient than using `does>` (see Section 6.11.10.5 [Advanced does> usage example], page 130). One advantage of this method is that it can be optimized better, because the compiler knows that the value compiled with `literal` is fixed, whereas the data associated with a `create`d word can be changed. But if you use `does>`/`set-does>` with `set-optimizer`, the resulting code will be at least as good, and the defined word will consume less memory.

`[compile]` (*compilation* "*name*" – *; run-time ? – ?*) core-ext "bracket-compile"

Legacy word. Use `postpone` instead. Works like `postpone` if *name* has non-default compilation semantics. If *name* has default compilation semantics (i.e., is a normal word), compiling `[compile]` *name* is equivalent to compiling *name* (i.e. `[compile]` is redundant in this case).

`in-colon-def?` (– *flag*) gforth-experimental "in-colon-def-question"

flag is true if and only if there is an active colon definition to which `compile,` and friends would append code.

Some people argue in favour of `evaluate`-based macros like:

```
: my2dup s" over over" evaluate ; immediate
```

We do not recommend using this technique, because it moves a number of decisions to macro-use time, when the setup may be different than you intended:

- The recognizer order may be different. This will usually not make a difference, but in case of doubt, you can use `rec-meta` to reduce potential ambiguity.
- The search order and the contents of individual wordlists may be different. You can use `rec-scope` to eliminate the search order dependency, but there is no easy way to eliminate the changes in the wordlists, except to avoid `evaluate`-based macros altogether.
- `State` may be different, and if this influences the behaviour of the macro, it will be `state`-smart (see [state-smartness], page 155).
- `Base` may be different, potentially changing the meaning of multi-digit numbers. Use number prefixes to avoid this problem.

An advantage of an `evaluate`-based macros is that you may be able to use it interpretively, whereas `postpone`-based macros are useful only inside colon definitions and other macros. Our recommended alternative is to instead use an inline definition (see Section 6.11.6 [Inline Definitions], page 123) or define a colon definition and an optimizer for it (e.g., see the `my2dup` examples in Section 6.11.10.7 [User-defined compile-comma], page 134).

6.17 The Text Interpreter

The text interpreter processes Forth source code, and performs the semantics of the words in the source code. The text interpreter works on Forth source code coming from the standard input (the *user input device* in Forth standard terminology), from a file (through `included` and friends), from `evaluate`, or from a `loaded` block.

The text interpreter is also called the outer interpreter, in contrast to the inner interpreter of traditional Forth implementations which interprets the threaded code that is the output of the Forth compiler on these implementations; many Forth systems nowadays compile to native code and have no inner interpreter. Concerning Gforth's approach, see [Engine], page 349.

The text interpreter works line by line (using `refill`). Therefore, most parsing words (e.g., `'` and `s"`) only try to find their parsed arguments in the current line; if a word parses across line boundaies, that is stated explicitly in the documentation of the word. However, a block is interpreted like one line, their division into "lines" is just for display purposes; `Evaluate` interprets the string as a whole. The current line/block/string is available to programs through `source`, and its content is valid until the current line is changed; it has to be treated as read-only region.

Within a line, the text interpreter parses for white-space-delimited words (with `parse-name`). The text interpreter's position within the line is stored in `>in`, and by changing `>in`, a program can change where the text interpreter or parsing word continues to parse.

The text interpreter then tries to recognize the word with one of the recognizers in the system recognizer sequence. On success, this produces a *translation* (a translation token and additional data), which represents the recognized word on the stack, as far as the text interpreter is concerned (see Section 6.17.4.3 [Defining recognizers], page 176).

It then performs the interpretation, compilation, or postponing action of the translation, depending on the state of the text interpreter: Is it inside `]]...[[`? If not, what is the value of `state`? E.g., when a word from the search order is recognized, and the text interpreter is in compile state, the compilation semantics of this word will be performed. This translation action may perform additional parsing, e.g., when the recognizer has recognized a parsing word. If no recognizer recognizes the word, the text interpreter `throw`s -13 ("undefined word").

The text interpreter then continues to parse, recognize, and translate words. At the end of the line, it `refill`s and continues with the next line. At the end of an input source (e.g., at the end of a file), the text interpreter returns to its caller, and that caller (e.g., `included`) restores the previous input stream before returning itself.

If an exception is not caught earlier, the same unnesting of input streams and the associated calls are performed. If not `catch` intervenes, the `throw` is eventually caught by the system. In Gforth, the system then prints an error message with a backtrace of both the call stack and the stack of nested input streams (see Chapter 7 [Error messages], page 307). In interactive mode, Gforth then calls a text interpreter for the user input device, while in scripting mode (while processing OS command-line arguments, see [Scripting mode], page 9) Gforth then terminates with a non-zero exit code.

You can read about this in more detail in Section 6.17.1 [Input Sources], page 168.

`source` (– *c-addr u*) core "source"

c-addr u is the input buffer, i.e., the line/block/`evaluate`d-string currently processed by the text interpreter

`>in` (– *addr*) core "to-in"

Addr contains the offset into `source` where the text interpreter or parsing words parse next. A word can write to *addr* in order to change what is parsed next. *Addr* can be different for different tasks, and for different input streams within a task.

`tib` (– *addr*) gforth-obsolete "t-i-b"

Addr is the start of the input buffer. OBSOLETE: This word has been de-standardized in Forth-2012. `source` supersedes the function of this word.

`#tib` (– *addr*) gforth-obsolete "number-t-i-b"

Addr is the address of a cell containing the number of characters in the terminal input buffer. *Addr* can be different for different tasks, and for different input streams within a task. OBSOLETE: This word has been de-standardized in Forth-2012. `source` supersedes the function of this word.

6.17.1 Input Sources

By default, the text interpreter processes input from the user input device (the keyboard) when Forth starts up. The text interpreter can process input from any of these sources:

- The user input device – the keyboard.
- A file, using the words described in Section 6.22.1 [Forth source files], page 196.
- A text string, using `evaluate`; a variant of this is when Gforth processes an OS command line argument of the form '`-e` *`string`*'.
- A block, see Section 6.23 [Blocks], page 202.

A program can identify the current input device from the values of `source-id` and `blk`.

`evaluate` (*... addr u – ...*) core,block

Save the current input source. Store `-1` in `source-id` and `0` in `blk`. Set `>IN` to `0` and make the string *c-addr u* the input source and input buffer. Tex-interpret the input buffer. When the parse area is empty, restore the input source.

`source-id` (– *0* | *-1* | *fileid*) core-ext,file "source-i-d"

Return 0 (the input source is the user input device), -1 (the input source is a string being processed by `evaluate`) or a *fileid* (the input source is the file specified by *fileid*).

`blk` (– *addr*) block "b-l-k"

addr contains the current block number (or 0 if the current input source is not a block).

`save-input` (– *x1 .. xn n*) core-ext

The *n* entries *xn - x1* describe the current state of the input source specification, in some platform-dependent way that can be used by `restore-input`.

`restore-input` (*x1 .. xn n – flag*) core-ext

Attempt to restore the input source specification to the state described by the *n* entries *xn - x1*. *flag* is true if the restore fails. In Gforth 1.0, it is possible to save and restore between different active input streams. Note that closing the input streams must happen in the reverse order as they have been opened, but as long as they are both active, everything is allowed. These Gforth versions only produce non-zero flags as results of `catch`ing some exception, and the *flag* itself is the `throw`n ball and can be re`throw`n.

`query` (–) gforth-obsolete

Make the user input device the input source. Receive input into the Terminal Input Buffer. Set `>IN` to zero. OBSOLETE: This Forth-94 word has been de-standardized in Forth-2012. It is superceeded by `accept`.

6.17.2 Interpret/Compile states

A standard program is not permitted to change `state` explicitly. However, it can change `state` implicitly, using the words `[` and `]`. When `[` is executed it switches `state` to interpret state, and therefore the text interpreter starts interpreting. When `]` is executed it switches `state` to compile state and therefore the text interpreter starts compiling. The most common usage for these words is for switching into interpret state and back from within a colon definition; this technique can be used to compile a literal (for an example, see Section 6.16.1 [Literals], page 160) or for conditional compilation (for an example, see Section 6.17.3 [Interpreter Directives], page 169).

`state` (– *a-addr*) core,tools-ext

Don't use `state`! `State` is the state of the text interpreter, and ordinary words should work independently of it; in particular, `state` does not tell you whether the interpretation semantics or compilation semantics of a word are being performed. See `State`-*smartness–Why it is evil and how to exorcise it*. For an alternative to `state`-smart words, see Section 6.14.6 [How to define combined words], page 154.
A-addr is the address of a cell containing the compilation state flag. 0 => interpreting, -1 => compiling. A standard program must not store into `state`, but instead use `[` and `]`.

6.17.3 Interpreter Directives

These words are usually used in interpret state; typically to control which parts of a source file are processed by the text interpreter. There are only a few Standard Forth words, but Gforth supplements these with a rich set of immediate control structure words to compensate for the fact that the non-immediate versions can only be used in compile state (see Section 6.10 [Control Structures], page 103). Typical usage:

```
[undefined] \G [if]
  : \G source >in ! drop ; immediate
[endif]
```

So if the system does not define `\G`, compile replacement code (with reduced functionality).

`[IF]` (*flag* –) tools-ext "bracket-if"

If flag is `TRUE` do nothing (and therefore execute subsequent words as normal). If flag is `FALSE`, parse and discard words from the parse area (refilling it if necessary using `REFILL`) including nested instances of `[IF]`.. `[ELSE]`.. `[THEN]` and `[IF]`.. `[THEN]` until the balancing `[ELSE]` or `[THEN]` has been parsed and discarded. Immediate word.

`[ELSE]` (–) tools-ext "bracket-else"

Parse and discard words from the parse area (refilling it if necessary using `REFILL`) including nested instances of `[IF]`.. `[ELSE]`.. `[THEN]` and `[IF]`.. `[THEN]` until the balancing `[THEN]` has been parsed and discarded. `[ELSE]` only gets executed if the balancing `[IF]` was `TRUE`; if it was `FALSE`, `[IF]` would have parsed and discarded the `[ELSE]`, leaving the subsequent words to be executed as normal. Immediate word.

`[THEN]` (–) tools-ext "bracket-then"

Do nothing; used as a marker for other words to parse and discard up to. Immediate word.

`[ENDIF]` (–) gforth-0.2 "bracket-end-if"

Do nothing; synonym for `[THEN]`

`[defined]` ("<*spaces*>*name*" – *flag*) tools-ext "bracket-defined"

returns true if name is found in current search order. Immediate word.

`[undefined]` ("<*spaces*>*name*" – *flag*) tools-ext "bracket-undefined"

returns false if name is found in current search order. Immediate word.

`[IFDEF]` ("<*spaces*>*name*" –) gforth-0.2 "bracket-if-def"

If name is found in the current search-order, behave like `[IF]` with a `TRUE` flag, otherwise behave like `[IF]` with a `FALSE` flag. Immediate word.

`[IFUNDEF]` ("<*spaces*>*name*" –) gforth-0.2 "bracket-if-un-def"

If name is not found in the current search-order, behave like `[IF]` with a `TRUE` flag, otherwise behave like `[IF]` with a `FALSE` flag. Immediate word.

`[?DO]` (*n-limit n-index* –) gforth-0.2 "bracket-question-do"

`[DO]` (*n-limit n-index* –) gforth-0.2 "bracket-do"

`[LOOP]` (–) gforth-0.2 "bracket-loop"

`[+LOOP]` (*n* –) gforth-0.2 "bracket-question-plus-loop"

`[FOR]` (*n* –) gforth-0.2 "bracket-for"

`[NEXT]` (*n* –) gforth-0.2 "bracket-next"

`[I]` (*run-time* – *n*) gforth-0.2 "bracket-i"

At run-time, `[I]` pushes the loop index of the text-interpretation-time `[do]` iteration. If you want to process the index at text-interpretation time, use `INT-[I]` or perform the interpretation semantics of `[I]`.

`INT-[I]` (– *n*) gforth-1.0 "int-bracket-i"

Push the loop index of the `[do]` iteration at text-interpretation time.

`[BEGIN]` (–) gforth-0.2 "bracket-begin"

`[UNTIL]` (*flag* –) gforth-0.2 "bracket-until"

`[AGAIN]` (–) gforth-0.2 "bracket-again"

`[WHILE]` (*flag* –) gforth-0.2 "bracket-while"

`[REPEAT]` (–) gforth-0.2 "bracket-repeat"

You can use `#line` to change Gforth's idea about the current source line number and source file. This is useful in cases where the Forth file is generated from some other source code file, and you want to get, e.g. error messages etc. that refer to the original source code; then the Forth-code generator needs to insert `#line` lines in the Forth code wherever appropriate.

`#line` ("*u*" "[*"file"*]" –) gforth-1.0 "number-line"

Set the line number to *u* and (if present) the file name to *file*. Consumes the rest of the line.

6.17.4 Recognizers

The recognizer concepts factor the central part of the text interpreter: The processing of one word after its name has been parsed.

The words and descriptions in this section are those that have found consensus in the standardization committee and will probably be standardized. There are also additional words implemented by Gforth and described in this section. Currently this manual does not tell which words are planned to be standardized and which are not. If you want to know that, see *Recognizer committee proposal 2025-09-11 (*`https://forth-standard.org/proposals/recognizer-committee-proposal-2025-09-11?hideDiff#rep`

Most programs just use the text interpreter as-is, and you can skip this chapter completely in this case, but if you are curious, see Section 6.17.4.1 [Default recognizers], page 171. The next level of recognizer usage is to change which of the existing recognizers are used and in what order (see Section 6.17.4.2 [Recognizer order], page 175).

You may also want to define a new recognizer; a recognizer produces an on-stack representation of the recognized string, called a *translation*, which contains a *translation token*. You can use predefined translation tokens for defining your recognizer (see Section 6.17.4.3 [Defining recognizers], page 176), or first define a new tranlation token (see Section 6.17.4.4 [Defining translation tokens], page 178). Finally, you may want to process the translation (see Section 6.17.4.5 [Performing translation actions], page 180).

6.17.4.1 Default recognizers

Type `recs` to find out with which recognizers are currently being used by Gforth. When invoked in a colon definition after defining a local, the output of `recs` is (at the time of this writing):

```
rec-name ( rec-local search-order ( Forth Forth Root ) ) rec-scope rec-
number rec-float rec-complex rec-string rec-to rec-dtick rec-tick rec-body
rec-env rec-meta
```

Here the notation *name (name1 ... namen)* indicates that *name* is a recognizer sequence that contains the recognizers *name1 ... namen*.

The recognizers in this sequence are:

`rec-name` Recognizes locals and words in the search order.

`rec-local`
: Recognizes locals.

`search-order`
: Recognizes words in the search order. This is shown as recognizer sequence, because the wordlists (see Section 6.19 [Word Lists], page 182) themselves are also recognizers: They implement the recognizer interface (see Section 6.17.4.3 [Defining recognizers], page 176) in addition to working with `find-name-in`.

`rec-scope`
: Recognizes *`voc1`*`:`*`voc2`*`:..`*`vocn`*`:`*`word`*, where *voc1* is a vocabulary in the search order, *voc2* is a vocabulary found in *voc1*, and so on, until *word* is found in *vocn*. The result behaves as if *word* had been found directly in the search order. Example: `environment:max-n`.

`rec-number`
: Single-cell integers (`#-15`, `$-f`), characters (`'A'`), and double-cell integers `#-15.`, with or without number prefixes (see Section 5.1 [Integer and character literals], page 54).

`rec-float`
: Floating-point numbers (`1e`, see Section 5.2 [Floating-point number and complex literals], page 54)

`rec-complex`
: Complex numbers (`1e+2ei`, see Section 5.2 [Floating-point number and complex literals], page 54)

`rec-string`
: Strings (`"abc"`, see Section 5.3 [String and environment variable literals], page 56).

`rec-to`
: Recognizes `->`*v* (equivalent to `to` *v*), `+>`*v* (equivalent to `+to` *v*), and `'>`*v* (equivalent to `addr` *v*), where *v* is a value-flavoured word (see Section 6.11.4 [Values], page 121). Also recognizes `@>`*d* (equivalent to `action-of` *d*), and `=>`*d* (equivalent to `is` *d*), where *d* is a defer-flavoured word (see Section 6.11.11 [Deferred Words], page 138).

`rec-dtick`
: Recognizes ``` ``*word* ``` and produces the name token of *word* (see Section 5.4 [Literals for tokens and addresses], page 56).

`rec-tick`
: Recognizes ``` `*word* ``` and produces the execution token of *word* (see Section 5.4 [Literals for tokens and addresses], page 56).

`rec-body`
: Recognizes `<`*word*`>` for the body address of *word* and `<`*word*`+`*num*`>` for an offset *num* from the body address of *word* (see Section 5.4 [Literals for tokens and addresses], page 56).

`rec-env`
: Recognizes `${`*env*`}` for the string contained at run-time in the environment variable *env* see Section 5.3 [String and environment variable literals], page 56).

`rec-meta`
: Recognizes *rec*`?`*string*, e.g., `float?1.`. `Rec-`*rec* is a recognizer found in the search order (e.g., `rec-float`), and this recognizer then tries to recognize *string* (e.g, `1.`), and the result becomes the result of `rec-meta`. This may be useful in cases where you want to use a specific recognizer, e.g., to deal with conflicts.

The order of the recognizers is significant, because they are tried from left to right, and the first recognizer that recognizes a word is actually used. E.g., if you define a local `b`, it will supersede Gforth's predefined word `b`.

In most cases, however, recognizers are designed to avoid matching the same strings as other recognizers. E.g., `rec-env` (the environment variable recognizer) requires braces to avoid a conflict with the number recognizer when recognizing environment variables like `ADD`; i.e., `rec-env` recognizes `${ADD}`, while `rec-number` recognizes `$ADD`.

There are a few cases where Gforth's recognizers can recognize the same string, however:

- Word names can be anything, so they can conflict with any other recognizer (and locals and the search order are searched before other recognizers).

However, there are no conflicts of Gforth-defined words with decimal numbers prefixed with `#` or hex numbers prefixed with `$`, so it is a good practice to use these prefixes (it's also a good idea to make sure that the right `base` is used). An older practice (before number prefixes were introduced) was to prefix hex numbers with `0`.

In the code bases we have looked at, starting words with `'` (quote aka tick) is much more common than starting them with `` ` `` (backquote aka backtick), so the recognizers for the xt and the nt use `` ` `` to reduce the number of conflicts.

- Both the integer recognizer `rec-number` and the floating-point recognizer `rec-float` recognize, e.g., `1.`. Because `rec-number` is (by default) earlier, `1.` is recognized as a double-cell integer. If you change the recognizer order to use `rec-float` first, `1.` is recognized as a floating-point number, but loading code written in Standard Forth may behave in a non-standard way.

  In any case, it's a good practice to avoid that conflict in your own code as follows: Always write double-cell integers with a number prefix, e.g., `#1.`; and always write floating-point numbers with an `e`, e.g., `1e`.
- We have seen a few word names that start with `->`. You can avoid a conflict by using `to myvalue` or `to?->myvalue` (the latter works with `postpone`).

Note that most Forth systems do not support all the recognizers we describe above, but `rec-name rec-number rec-float` are relatively common (even if a system uses a hard-coded text interpreter instead of the flexible recognizer system).

You can use `locate` (see Section 6.30.2 [Locating source code definitions], page 258) to determine which recognizer recognizes a piece of source code. E.g.:

```
locate float?1.
```

will show that `rec-meta` recognizes `float?1.`. However, if the recognizer recognizes a dictionary word (e.g., the scope recognizer), `locate` will show that word.

Wordlists are also recognizers, as can be seen by the search order being shown as recognizer sequence containing the wordlists, . A wordlist recognizes the words that it contains. Just `execute` the wordlist-id, and it will behave as a recognizer:

```
"dup" forth-wordlist execute
```

produces the same result as

```
"dup" rec-name
```

Actually, `rec-name` searches all wordlists in the search order; with the default search order, that finds the `dup` in `forth-wordlist`.

`recs` (–) gforth-experimental

Print the system recognizer order, with the first-searched recognizer leftmost. For recognizer sequences, first the name is printed, then '`(`', then the content of the sequence, then '`)`'. For a deferred word, the name of the deferred word is shown, not that of the recognizer inside; if it contains a recognizer sequence, the name of the deferred word and the contents of the sequence are shown.

All recognizers produce the translation `translate-none` if they do not recognize the string. This is not mentioned every time in the following descriptions.

`rec-name` (*c-addr u – translation*) gforth-experimental

Recognizes (see Section 6.17.4.3 [Defining recognizers], page 176) a visible local or a visible named word. If successful, *translation* represents the text-interpretation semantics (interpreting, compiling, postponing) of that word (see `translate-name`).

`rec-local` (*c-addr u – translation*) gforth-experimental

Recognizes (see Section 6.17.4.3 [Defining recognizers], page 176) a visible local. If successful, *translation* represents pushing the value of the local at run-time (for details see Section 6.26.1 [Gforth locals], page 217, and see Section 6.16.2 [Macros], page 162).

`rec-scope` (*c-addr u – translation*) gforth-experimental

Recognizes (see Section 6.17.4.3 [Defining recognizers], page 176) *`vocabulary:word`*, where *vocabulary* is found in the search order. Otherwise the behaviour is like that of `rec-name`. The general form can have several vocabularies preceding *word*, separated by `:`; the first (leftmost) vocabulary is found in the search order, the second in the first, etc. *word* is looked up in the rightmost vocabulary.

`rec-number` (*c-addr u – translation*) gforth-experimental

Recognizes (see Section 6.17.4.3 [Defining recognizers], page 176) a single or double number (without or with prefix), or a character. If successful, *translation* represents pushing that number at run-time (see `translate-cell` and `translate-dcell`). If and only if `.-is-dcell?` is true, strings without prefix that contain a dot are recognized as double numbers.

`.-is-dcell?` (*– flag*) gforth-experimental "dot–is-dcell-question"

If this user flag is true (default), `rec-number` recognizes numbers without prefix that contain a decimal point as double-cell numbers. Otherwise `rec-number` does not recognize the number, and, if present, `rec-float` will recognize it as a floating-point number.
`to .-is-dcell?` run-time: *(x –)* If x=0 change the value of `.-is-dcell?` to false, otherwise to true.

`rec-float` (*c-addr u – translation*) gforth-experimental

Recognizes (see Section 6.17.4.3 [Defining recognizers], page 176) a floating-point number, *translation* represents pushing that number at run-time (see `translate-float`). For recognizing a string as a float, Gforth requires decimal `base`; it also requires the string to contain an exponent (`e` followed by an optional sign and 0 or more exponent digits) or a decimal point (the syntax with the decimal point only is shadowed by the double-cell syntax by default, see `rec-number`); in Gforth there can also be an SI prefix (e.g., `M`) instead of the decimal point, but then no `e` is allowed. Examples: `1234e`, `1234.`, `1.234e3`, `12340e-1`, `1k234`.

`rec-complex` (*c-addr u – translation*) gforth-experimental

A complex number has the format `a+bi`, where *a* and *b* are floating point numbers including their signs. If *c-addr u* is a complex number, translation represents pushing that number at run-time (see `translate-complex`).

`rec-string` (*c-addr u – translation*) gforth-experimental

A string starts and ends with `"` and may contain escaped characters, including `\"` (see Section 6.9.3 [String and character literals], page 91). If *c-addr u* is the start of a string, the translation represents parsing the rest of the string, if necessary, converting the escaped characters, and pushing the string at run-time (`translate-string` `scan-translate-string`).

`rec-to` (*c-addr u – translation*) gforth-experimental

Recognizes (see Section 6.17.4.3 [Defining recognizers], page 176) `->v` (`TO v`), `+>v` (`+TO v`), `'>v` (`ADDR v`), `@>d` (`ACTION-OF d`) and `=>d` (`IS d`), where *v* is a value-flavoured word and *d* is a defer-flavoured word. If successful, *translation* represents performing the operation on *v*/*d* at run-time.

`rec-dtick` (*c-addr u – translation*) gforth-experimental

Recognizes (see Section 6.17.4.3 [Defining recognizers], page 176) ``*word*. If successful, *translation* represents pushing the name token of *word* at run-time (see `translate-cell`). Example: ``S" gives the nt of S".

`rec-tick` (*c-addr u – translation*) gforth-experimental

Recognizes (see Section 6.17.4.3 [Defining recognizers], page 176) `*word*. If successful, *translation* represents pushing the execution token of *word* at run-time (see `translate-cell`). Example: `dup gives the xt of dup.

`rec-body` (*addr u – translation*) gforth-experimental

Recognizes (see Section 6.17.4.3 [Defining recognizers], page 176) `<*word*>` and `<*word*+*number*>`. If successful, *translation* represents pushing the sum of the body address *word* and *number* (0 if absent) at run-time (see `translate-cell`).

`rec-env` (*c-addr u – translation*) gforth-1.0

Recognizes (see Section 6.17.4.3 [Defining recognizers], page 176) `${*envvar*}`. If successful, *translation* represents passing *envvar* to `getenv` at run-time (see `translate-env`). Example: `${HOME}` gives the home directory.

`rec-meta` (*addr u – xt translate-to* | *0*) gforth-1.0

Recognizes (see Section 6.17.4.3 [Defining recognizers], page 176) *myrec*`?`*mystring*. Produces the result of passing *mystring* to `rec-`*myrec*.
Example: `hex num?cafe` will be recognized as number even if a word `cafe` is in the search order.
Example: `float?123.` will be recognized as float.

6.17.4.2 Recognizer order

You may prefer to use a different recognizer sequence, but with (some of the) existing recognizers. You can use the following words for that:

`rec-forth` (*c-addr u – translation*) gforth-experimental

The system recognizer: `rec-forth` is a `defer`red word that contains a recognizer (sequence). The system's text interpreter calls `rec-forth`.

`rec-sequence:` (*xtu .. xt1 u* `"`*name*`"` –) gforth-experimental “rec-sequence-colon”

Define a recognizer sequence *name*. *xtu..xt1* are xts of recognizers, and are the initial contents of the recognizer sequence, with *xt1* searched first. The order of operands is inspired by `get-order` and `set-order`.
name execution: (c-addr u – translation)
Execute the first xt in the recognizer sequence *name*. If the resulting translation has a translation token other than `translate-none`, this is the result of *name* and no further recognizers are tried. Otherwise, the stacks are restored to the initial state (*c-addr u*), and

the next xt is tried. If all xts produce `translate-none`, *translation* is `translate-none`. *name* is a recognizer itself, which makes recognizer sequences nestable.

`get-recs` (*recs-xt – xtu .. xt1 u*) gforth-experimental

Recs-xt is the execution token of a recognizer sequence. *xt1* is the first recognizer searched by this sequence, *xtu* is the last one. If *recs-xt* refers to a deferred word, perform `defer@ get-recs`.

`set-recs` (*xtu .. xt1 u recs-xt –*) gforth-experimental

rec-xt is the execution token of a recognizer sequence. Replace the contents of this sequence with *xtu...xt_1*, where *xt1* is searched first, and *xtu* is searched last. If *recs-xt* refers to a deferred word, perform `defer@ set-recs`.

You probably don't want to create a new recognizer sequence every time you want to change the system recognizer sequence. There are several ways to change an existing recognizer sequence:

- Put one or more `defer`red words in a recognizer sequence, and change the recognizer in this word later. If you do not want such a deferred word to recognize anything for now, put `rec-none` in it.
- Use `set-recs` on the recognizer sequence, possibly after getting the current sequence with `get-recs` and modifying it.
- The body of a recognizer sequence is a `stack` (see Section 6.13 [User-defined Stacks], page 149), and you can use the words for manipulating stacks on it. In particular, if you add a recognizer with `>stack`, that recognizer will be tried first; if you add it with `>back`, it will be tried last.

`rec-none` (*c-addr u – translate-none*) gforth-experimental

This recognizer recognizes nothing. It can be useful as a placeholder.

Here is an example of adding `rec-none` as last recognizer to the system recognizers:

```
' rec-none action-of rec-forth get-recs 1+ action-of rec-forth set-recs

\ alternative
' rec-none action-of rec-forth >body >back
```

6.17.4.3 Defining recognizers

A *recognizer* is a Forth word with the stack effect `( c-addr u -- translation )`. *c-addr u* describes the string to be recognized. The returned *translation* is an abstract, but transparent data type: On the top of stack, there is a single-cell *translation token*. If the recognizer does not recognize the string, it returns the translation token `translate-none`. If it does recognize the string, it returns a translation with a different translation token. The translation token also identifies how many other stack items the translation contains and how the translation will be processed later.

E.g., when you perform

```
"5" rec-number
```

it pushes `5 translate-cell` on the stack, which is a translation with the translation token `translate-cell`.

You typically write a recognizer as ordinary colon definition that examines the string in some way, and then pushes the appropriate translation. E.g., a simple variant of `rec-tick` can be implemented as follows:

```
: rec-tick ( addr u -- translation )
    2dup "`" string-prefix? if
        1 /string find-name dup if
            name>interpret translate-cell exit then
        drop translate-none exit then
    rec-none ;
```

The only appropriate use of a translation is to pass it to one of the words for performing translation actions (see Section 6.17.4.5 [Performing translation actions], page 180).

A number of translation tokens already exist in Gforth and can be used in a recognizer you write. If none of them is appropriate for your recognizer, read the next section about defining your own translation tokens.

The system-defined translation-token words are documented as removing some stack items and pushing a complete translation on the stack, e.g., for `translate-cell ( x -- translation )`. This makes the documentation uniform and avoids cumbersome descriptions. However, actually the current translation-token words just push a cell-sized translation token on the stack (for `translate-cell: ( -- translate-cell )`), and, combined with the additional stack items (for `translate-cell: ( x -- x translate-cell )`), the result is a translation (for `translate-cell: ( x -- translation )`).

The text interpreter passes the output of the recognizer to a translation action (see Section 6.17.4.5 [Performing translation actions], page 180). Every translation action removes the translation from the stack, then may perform additional parsing, and finally performs the interpreting run-time of the translation token, or the compiling run-time, or the postponing run-time.

For each system-defined translation token we specify the interpreting run-time explicitly. Unless otherwise specified, the compiling run-time compiles the interpreting run-time. Unless otherwise specified, the postponing run-time compiles the compiling run-time.

In the `rec-tick` example above, if the recognizer recognizes, say, `` `dup ``, it returns *xt-dup translate-cell*. If the text interpreter then performs the compiling action, that action first removes this translation (these two cells), and compiles code that pushes *xt-dup*.

`translate-name` (*nt – translation*) gforth-experimental

Interpreting run-time: `( ... -- ... )`
Perform the interpretation semantics of *nt*.
Compiling run-time: `( ... -- ... )`
Perform the compilation semantics of *nt*.

`translate-cell` (*x – translation*) gforth-experimental

Interpreting run-time: `( -- x )`

`translate-dcell` (*xd – translation*) gforth-experimental

Interpreting run-time: `( -- xd )`

`translate-float` (*r – translation*) gforth-experimental

Interpreting run-time: `( -- r )`

`translate-complex` (*r1 r2 – translation*) gforth-experimental

Interpreting run-time: (-- *r1 r2*)

`translate-string` (*c-addr1 u1 – translation*) gforth-experimental

Interpreting run-time: (-- *c-addr2 u2*)
c-addr2 u2 is the result of translating the \-escapes in *c-addr1 u1*.

`scan-translate-string` (*c-addr1 u1 'ccc"' – translation*) gforth-experimental

Every translation action also parses until the first non-escaped ". The string *c-addr u* and the parsed input are concatenated, then the \-escapes are translated, giving *c-addr2 u2*.
Interpreting run-time: (-- *c-addr2 u2*)

`translate-env` (*c-addr1 u1 – translation*) gforth-experimental

Interpreting run-time: (-- *c-addr2 u2*)
c-addr2 u2 is the content of the environment variable with name *c-addr1 u1*.

`translate-to` (*n xt – translation*) gforth-experimental

xt belongs to a value-flavoured (or defer-flavoured) word, *n* is the index into the `to-table:` for *xt* (see Section 6.11.10.6 [Words with user-defined TO etc.], page 132).
Interpreting run-time: (... -- ...)
Perform the to-action with index *n* in the `to-table:` of *xt*. Additional stack effects depend on *n* and *xt*.

One way to write a recognizer *r* is to call a recognizer (for the whole input of *r* or a substring) that recognizes more strings (e.g., `rec-forth`), and then look at the result to see if something was recognized that *r* actually deals with.

E.g., the actual implementation of `rec-tick` passes its input without the prefix '`' to `rec-forth` and checks whether the resulting translation-token is *nt translate-name*, then converts *nt* to *xt*, and replaces *translate-name* with *translate-cell*. The benefit of this approach compared to our example implementation above is that, e.g., `` `environment:max-n `` works, where `rec-scope` recognizes `environment:max-n`.

The specific check for an nt used in `rec-tick` is `rec-forth-nt?`; it is implemented on top of the more general `rec-filter`.

`rec-filter` (*c-addr u xt: filter xt: rec – translation*) gforth-experimental

Execute *rec* (*c-addr u -- translation1*); *translation1* is then examined with *filter* (*translation1 -- translation1 f*). If *f* is non-zero, *translation* is *translation1*, otherwise *translation* is *translate-none*.

`rec-forth-nt?` (*c-addr u – nt | 0*) gforth-experimental "rec-forth-nt-question"

If `rec-forth` produces a result *nt* `translate-name`, return *nt*, otherwise 0.

6.17.4.4 Defining translation tokens

Before you define a translation token, you should think about the interpreting run-time, compiling run-time, and postponing run-time that the corresponding actions should perform for translations with this translation token. Also, you should think about whether the translation actions should perform additional scanning (like `scan-translate-string`).

You then need to define three words, one for the interpreting action, one for the compiling action, and one for the postponing action.

Each of these words will eventually be called after removing the translation token from the stack (but the remainder of the translation is still on the stack(s)). It should remove this remainder, perform the additional scanning (if any), and then perform the appropriate run-time.

Once you have these three words, you can define a translation token by passing the xts of these words to

`translate:` (*int-xt comp-xt post-xt* "*name*" –) gforth-experimental "translate-colon"

Defines *name*, a translation token (see Section 6.17.4.3 [Defining recognizers], page 176).
name execution: (– *translation-token*)
name interpreting action: (*... translation* – *...*)
Remove the translation token from the stack and execute *int-xt*.
name compiling action: (*... translation* – *...*)
Remove the translation token from the stack and execute *comp-xt*.
name postponing action: (*translation* –)
Remove the translation token from the stack and execute *post-xt*.

To make this a little more concrete, here is an implementation for `translate-cell`:

```
' noop                       ( x -- x )                              \ int-xt
' lit,                       ( compilation: x -- ; run-time: -- x ) \ comp-xt
:noname lit, postpone lit, ; ( postponing: x -- ;  run-time: -- x ) \ post-xt
translate: translate-cell
```

If a recognizer recognizes something as a single-cell literal *x*, it pushes *x* and then calls `translate-cell`. Later the text interpreter (or `postpone` or some other consumer of translations) removes the translation token and executes one of the three xts above (depending on what translation action is desired).

When the interpretation semantics is needed, *int-xt* is executed, and *x* stays on the stack.

For the compilation semantics, *x* is compiled into the current definition as literal.

For postponing, more time levels are involved: at text-interpretation time (when the recognizer runs and the translation token action is performed) the current definition is *d1*. When *d1* runs, the current definition is *d2*[20]. The *post-xt* of the `translate-cell` implementation above first compiles *x* into *d1* and also compiles `lit,` into *d1* (that's the `postpone lit,` part). When *d1* runs, it pushes *x* and then the `lit,` compiles *x* into *d2*.

Many literal translation tokens follow this scheme.

A translation token that is quite different is `translate-name`. Here's an implementation:

```
: name-intsem ( ... nt -- ... )
  name>interpret execute-exit ;
: name-compsem ( ... nt -- ... )
  name>compile execute-exit ;
: name-compcompsem ( nt -- )
  lit, postpone name-compsem ;
' name-intsem ' name-compsem ' name-compcompsem translate: translate-name
```

`Name-intsem` performs the interpretation semantics of *nt*, by getting the xt of the interpretation semantics and executing it. Here `execute-exit` is used, in order for return-stack

[20] If there is no current definition when something is compiled, Gforth outputs a warning.

words to work (that's a Gforth 1.0 feature). Also, in Gforth 1.0 all words have interpretation semantics, so the result of `name-interpret` is not tested for 0.

`Name-compsem` performs the compilation semantics of *nt*.

`Name-compcompsem` compiles the compilation semantics of *nt*. This is achieved by compiling *nt* and `name-compsem` into the current definition *d1*. When *d1* runs, the result performs the compilation semantics of *nt* at that time.

6.17.4.5 Performing translation actions

There are the following words for performing the various translation token actions:

`interpreting` (*... translation – ...*) gforth-experimental

Perform the interpreting action of *translation*. For a system-defined translation token, first remove *translation* from the stack, then possibly perform additional scanning specified for the translation token, and finally perform the interpreting run-time of the translation token. For a user-defined translation token, remove it from the stack and execute its *int-xt*.

`compiling` (*... translation – ...*) gforth-experimental

Perform the compiling action of *translation*. For a system-defined translation token, first remove *translation* from the stack, then possibly perform additional scanning specified for the translation token, and finally perform the compiling run-time of the translation token. For a user-defined translation token, remove it from the stack and execute its *comp-xt*.

`postponing` (*... translation –*) gforth-experimental

Perform the postponing action of *translation*. For a system-defined translation token, first remove *translation* from the stack, then possibly perform additional scanning specified for the translation token, and finally perform the postponing run-time of the translation token. For a user-defined translation token, remove it from the stack and execute its *post-xt*.

`?rec-found` (*translation – translation*) gforth-experimental "question-rec-found"

`throw`s -13 (undefined word) if *translation* is `translate-none`.

Their typical use is in a text interpreter. A simple text interpreter could look like this:

```
: myinterpret ( -- )
  \ refill happens outside
  begin
    parse-name dup while
      forth-recognize ?rec-found state @ if compiling else interpreting then
  repeat
  2drop ;
```

This text interpreter itself does not deal with postponing; `]]` can be implemented as a text interpreter that performs the postponing:

```
: ]] ( -- )
  \ works only within a line
  begin
    parse-name dup 0= abort" [[ missing"
    2dup "[[" str= 0= while
      forth-recognize ?rec-found postponing
   repeat
   2drop ; immediate
```

6.17.5 Text Interpreter Hooks

`before-line` (–) gforth-1.0

Deferred word called before the text interpreter parses the next line

`before-word` (–) gforth-0.7

Deferred word called before the text interpreter parses the next word

`line-end-hook` (–) gforth-0.7

Deferred word called at every end-of-line when text-interpreting from a file.

6.18 The Input Stream

The text interpreter reads from the input stream, which can come from several sources (see Section 6.17.1 [Input Sources], page 168). Some words, in particular defining words, but also words like `'`, read parameters from the input stream instead of from the stack.

Such words are called parsing words, because they parse the input stream. Parsing words are hard to use in other words, because it is hard to pass program-generated parameters through the input stream. They also usually have an unintuitive combination of interpretation and compilation semantics when implemented naively, leading to various approaches that try to produce a more intuitive behaviour (see [Combined words], page 154).

It should be obvious by now that parsing words are a bad idea. If you want to implement a parsing word for convenience, also provide a factor of the word that does not parse, but takes the parameters on the stack. To implement the parsing word on top if it, you can use the following words:

`parse` (*xchar* `"`*ccc*`<`*xchar*`>"` – *c-addr u*) core-ext,xchar-ext

Parse *ccc*, delimited by *xchar*, in the parse area. *c-addr u* specifies the parsed string within the parse area. If the parse area was empty, *u* is 0.

`string-parse` (*c-addr1 u1* `"`*ccc*`<`*string*`>"` – *c-addr2 u2*) gforth-1.0

Parse *ccc*, delimited by the string *c-addr1 u1*, in the parse area. *c-addr2 u2* specifies the parsed string within the parse area. If the parse area was empty, *u2* is 0.

`parse-name` (`"`*name*`"` – *c-addr u*) core-ext

Get the next word from the input buffer

`parse-word` (– *c-addr u*) gforth-obsolete

old name for `parse-name`; this word has a conflicting behaviour in some other systems.

`name` (– *c-addr u*) gforth-obsolete

old name for `parse-name`

`word` (*char* `"<`*chars*`>`*ccc*`<`*char*`>`– *c-addr*) core

We recommend to use `parse-name` instead of `bl word` and, for other delimiters, `parse` instead of `word`.
Skip leading delimiters. Parse *ccc*, delimited by *char*, in the parse area. *c-addr* is the address of a transient region containing the parsed string in counted-string format (see Section 6.9.8 [Counted string words], page 102. If the parse area is empty or contains no characters other than delimiters, the resulting string has zero length. A program may replace characters

within the counted string. OBSOLESCENT: the counted string has a trailing space that is not included in its length.

`refill` (– *flag*) core-ext,block-ext,file-ext

Attempt to fill the input buffer from the input source. When the input source is the user input device, attempt to receive input into the terminal input device. If successful (including lines of length 0), make the result the input buffer, set `>IN` to 0 and return true; otherwise return false.
When the input source is a text file, attempt to read the next line from the file. If successful, make the result the current input buffer, set `>IN` to 0 and return true; otherwise, return false.
When the input source is a block, add 1 to the value of `BLK` to make the next block the input source and current input buffer, and set `>IN` to 0; return true if the new value of `BLK` is a valid block number, false otherwise.

If you have to deal with a parsing word that does not have a non-parsing factor, you can use `execute-parsing` to pass a string to it:

`execute-parsing` (*... addr u xt – ...*) gforth-0.6

Make *addr u* the current input source, execute *xt* `( ... -- ... )`, then restore the previous input source.

Example:

```
5 s" foo" ' constant execute-parsing
\ equivalent to
5 constant foo
```

A definition of this word in Standard Forth is provided in `compat/execute-parsing.fs`.

If you want to run a parsing word on a file, the following word should help:

`execute-parsing-file` (*i*x fileid xt – j*x*) gforth-0.6

Make *fileid* the current input source, execute *xt* `( i*x -- j*x )`, then restore the previous input source.

6.19 Word Lists

A wordlist is a list of named words; you can add new words and look up words by name (and you can remove words in a restricted way with markers). Every named (and `reveal`ed) word is in one wordlist.

The text interpreter searches the wordlists present in the search order (a stack of wordlists, see Section 6.13 [User-defined Stacks], page 149), from the top to the bottom. Within each wordlist, the search starts conceptually at the newest word; i.e., if two words in a wordlist have the same name, the newer word is found.

New words are added to the *compilation wordlist* (aka current wordlist).

A word list is identified by a cell-sized word list identifier (*wid*) in much the same way as a file is identified by a file handle. The numerical value of the wid has no (portable) meaning, and might change from session to session.

The Standard Forth "Search order" word set is intended to provide a set of low-level tools that allow various different schemes to be implemented. Gforth also provides `vocabulary`,

a traditional Forth word. `compat/vocabulary.fs` provides an implementation in Standard Forth.

`forth-wordlist` (– *wid*) search

`Constant` – *wid* identifies the word list that includes all of the standard words provided by Gforth. When Gforth is invoked, this word list is the compilation word list and is at the top of the search order.

`definitions` (–) search

Set the compilation word list to be the same as the word list that is currently at the top of the search order.

`get-current` (– *wid*) search

wid is the identifier of the current compilation word list.

`set-current` (*wid* –) search

Set the compilation word list to the word list identified by *wid*.

`in-wordlist` (*wordlist* "*defining-word*" –) gforth-experimental

execute *defining-word* with *wordlist* as one-shot current directory. Example: `gui-wordlist in-wordlist : init-gl ... ;` will define `init-gl` in the `gui-wordlist` wordlist.

`in` ("*voc*" "*defining-word*" –) gforth-experimental

execute *defining-word* with *voc* as one-shot current directory. Example: `in gui : init-gl ... ;` will define `init-gl` in the `gui` vocabulary.

`get-order` (– *widn .. wid1 n*) search

Copy the search order to the data stack. The current search order has *n* entries, of which *wid1* represents the wordlist that is searched first (the word list at the top of the search order) and *widn* represents the wordlist that is searched last.

`set-order` (*widn .. wid1 n* –) search

If *n*=0, empty the search order. If *n*=-1, set the search order to the implementation-defined minimum search order (for Gforth, this is the word list `Root`). Otherwise, replace the existing search order with the *n* wid entries such that *wid1* represents the word list that will be searched first and *widn* represents the word list that will be searched last.

`wordlist` (– *wid*) search

Create a new, empty word list represented by *wid*.

`table` (– *wid*) gforth-0.2

Create a lookup table (case-sensitive, no warnings).

`cs-wordlist` (– *wid*) gforth-1.0

Create a case-sensitive wordlist.

`cs-vocabulary` ("*name*" –) gforth-1.0

Create a case-sensitive vocabulary

`>order` (*wid* –) gforth-0.5 “to-order”

Push *wid* on the search order.

`previous` (–) search-ext

Drop the wordlist at the top of the search order.

`also` (–) search-ext

Like `DUP` for the search order. Usually used before a vocabulary (e.g., `also Forth`); the combined effect is to push the wordlist represented by the vocabulary on the search order.

`Forth` (–) search-ext

Replace the *wid* at the top of the search order with the *wid* associated with the word list `forth-wordlist`.

`Only` (–) search-ext

Set the search order to the implementation-defined minimum search order (for Gforth, this is the word list `Root`).

`order` (–) search-ext

Print the search order and the compilation word list. The word lists are printed in the order in which they are searched (which is reversed with respect to the conventional way of displaying stacks). The compilation word list is displayed last.

`.voc` (*wid* –) gforth-0.2 "dot-voc"

print the name of the wordlist represented by *wid*. Can only print names defined with `vocabulary` or `wordlist constant`, otherwise prints '`address`'.

`find` (*c-addr* – *xt* `+-1` | *c-addr 0*) core,search

We recommend to use `find-name` instead of `find`. Search all word lists in the current search order for the definition named by the counted string at *c-addr*. If the definition is not found, return 0. If the definition is found, return 1 (if the definition has non-default compilation semantics) or -1 (if the definition has default compilation semantics). The *xt* returned in interpret state represents the interpretation semantics. The *xt* returned in compile state represented either the compilation semantics (for non-default compilation semantics) or the run-time semantics that the compilation semantics would `compile,` (for default compilation semantics). The Forth-2012 standard does not specify clearly what the returned *xt* represents (and also talks about immediacy instead of non-default compilation semantics), so this word is questionable in portable programs. If non-portability is ok, `find-name` and friends are better (see Section 6.15.2 [Name token], page 158).

`search-wordlist` (*c-addr count wid* – *0* | *xt* `+-1`) search

We recommend to use `find-name-in` instead of `search-wordlist`.
Search the word list identified by *wid* for the definition named by the string at *c-addr count*. If the definition is not found, return 0. If the definition is found return 1 (if the definition has non-default compilation semantics) or -1 (if the definition has default compilation semantics) together with the *xt*. In Gforth, the *xt* returned represents the interpretation semantics. Forth-2012 does not specify clearly what *xt* represents.

`words` (–) tools

Display a list of all of the definitions in the word list at the top of the search order.

`vlist` (–) gforth-0.2

Old (pre-Forth-83) name for `WORDS`.

`wordlist-words` (*wid* –) gforth-0.6

Display the contents of the wordlist wid.

`mwords` (*["pattern"]* –) gforth-1.0

List all words in the word list on top of the search order. If a pattern is given, show only the words matching *pattern*. Words are listed old to new (reverse from `words`). By default

`search` is used to determine matching. You can switch to globbing as used by shells for files with `' mword-filename-match is mword-match`.

`Root` (–) gforth-0.2

Replace the top of the search order by the root wordlist. This vocabulary makes up the minimum search order and contains only a few search-order words.

`Vocabulary` ("*name*" –) gforth-0.2

Create a definition "name" and associate a new word list *wid1* with it.
name execution: (–) replace the *wid2* at the top of the search order with *wid1*.

`seal` (–) gforth-0.2

Remove all word lists from the search order stack other than the word list that is currently on the top of the search order stack.

`vocs` (–) gforth-0.2

List vocabularies and wordlists defined in the system.

`current` (– *addr*) gforth-0.2

`Variable` – holds the *wid* of the compilation word list.

`context` (– *addr*) gforth-0.2

`context @` is the *wid* of the word list at the top of the search order.

`map-vocs` (*... xt – ...*) gforth-1.0

Perform xt (... wid – ...) for all wordlists (including tables and cs-wordlists) in the system.

6.19.1 Why use word lists?

Here are some reasons why people use wordlists:

- To prevent a set of words from being used outside the context in which they are valid. Two classic examples of this are an integrated editor (all of the edit commands are defined in a separate word list; the search order is set to the editor word list when the editor is invoked; the old search order is restored when the editor is terminated) and an integrated assembler (the op-codes for the machine are defined in a separate word list which is used when a `CODE` word is defined).
- To organize the words of an application or library into a user-visible set (in `forth-wordlist` or some other common wordlist) and a set of helper words used just for the implementation (hidden in a separate wordlist). This keeps `words`' output smaller, separates implementation and interface, and reduces the chance of name conflicts within the common wordlist.
- To prevent a name-space clash between multiple definitions with the same name. For example, when building a cross-compiler you might have a word `IF` that generates conditional code for your target system. By placing this definition in a different word list you can control whether the host system's `IF` or the target system's `IF` get used in any particular context by controlling the order of the word lists on the search order stack.

The downsides of using wordlists are:

- Debugging becomes more cumbersome, because you have to use the scope recognizer or put the wordlist in the search order in order to refer to a word in a wordlist. Neither `locate` nor (currently) `see` show the names of words with their wordlist or vocabulary.
- Name conflicts worked around with wordlists are still there, and you have to arrange the search order carefully to get the desired results; if you forget to do that, you get hard-to-find errors (as in any case where you read the code differently from the compiler; `see` can help seeing which of several possible words the name resolves to in such cases). `See` displays just the name of the words, not what wordlist they belong to, so it might be misleading. Using unique names is a better approach to avoid name conflicts.

6.19.2 Wordlist and vocabulary usage

If you have an existing wordlist in constant `foo-wordlist` and want to push it on the search order, here's how you do it:

```
foo-wordlist >order
... \ code that may use words from foo-wordlist
previous
\ now the search order is back to where we started
```

If instead you have a vocabulary `foo`, the same task can be accomplished as follows:

```
also foo
... \ code that may use words from foo
previous
\ now the search order is back to where we started
```

Also, if you need only one or a few words from `foo`, you can use the scope recognizer (`rec-scope`, see Section 6.17.4.1 [Default recognizers], page 171) and write `foo:`*`word`*. Currently the scope recognizer only uses vocabularies for the scope, not wordlists.

A common usage is to define implementation words (which would be private in other programming languages) in a separate wordlist, and (public) interface words in a wordlist for public words (usually `forth-wordlist`). This can be achieved as follows:

```
wordlist constant foo-wordlist \ the implementation wordlist
get-current ( wid )
foo-wordlist >order definitions
\ foo-wordlist is now visible and definitions go into it
... \ define implementation words
( wid ) set-current
\ foo-wordlist is visible, but definitions go into wid
... \ define interface words
previous
\ search order and current wordlist are back to where we started
```

The same can be done with vocabularies as follows:

```
vocabulary foo \ the implementation vocabulary
get-current ( wid )
also foo definitions
\ foo is now visible and definitions go into it
```

```
... \ define implementation words
( wid ) set-current
\ foo is visible, but definitions go into wid
... \ define interface words
previous
\ search order and current wordlist are back to where we started
```

If you want to define just one word in a given wordlist, you can do it as follows:

```
foo-wordlist in-wordlist : bar ... ;
```

For vocabularies, the corresponding usage is:

```
in foo : bar ... ;
```

6.20 Number conversion

This section is about conversion between numbers and strings. For conversion between integers and FP numbers, see Section 6.5.9 [Floating Point], page 69.

In addition to the number-to-string conversion words mentioned in this section, you can also convert numbers to strings by using a number output word (see Section 6.24.1 [Simple numeric output], page 205, see Section 6.24.2 [Floating-point output], page 206), and outputting the result in a string with `>string-execute` (see Section 6.9.4 [String words], page 93).

6.20.1 Base and integer decimal point

By default, the number base used for integer number conversion is given by the contents of the variable `base`. `Base` affects both input integer conversion (unless a number prefix is used, see Chapter 5 [Literals in source code], page 54) and output integer conversion (unless a base-specific word such as `h.` or `dec.` is used, see Section 6.24.1 [Simple numeric output], page 205, see Section 6.20.3 [Integer to string conversion], page 189). In Gforth, non-decimal `base` disables floating-point conversion.

Note that a lot of confusion can result from unexpected values of `base`. If you change `base` anywhere, make sure to save the old value and restore it afterwards; better yet, use `base-execute`, which does this for you.

In general I recommend keeping `base` decimal; For dealing with the popular non-decimal bases, use number prefixes for inputting them (see Chapter 5 [Literals in source code], page 54) and `h.` and `base-execute` for outputting them.

`base-execute` (*i*x xt u – j*x*) gforth-0.7

execute *xt* with the content of `BASE` being *u*, and restoring the original `BASE` afterwards.

`base` (– *a-addr*) core

User variable – *a-addr* is the address of a cell that stores the number base used by default for number conversion during input and output. Don't store to `base`, use `base-execute` instead.

`hex` (–) core-ext

Set `base` to $10 (hexadecimal). In many cases `base-execute` is a better alternative.

`decimal` (–) core

Set `base` to #10 (decimal). In many cases `base-execute` is a better alternative.

`dpl` (– *a-addr*) gforth-0.2 "d-p-l"

User variable – *a-addr* is the address of a cell that stores the position of the decimal point in the most recent input integer conversion. After the conversion of a number containing no decimal point, `dpl` is -1. After the conversion of `2341239.` it holds 0. After the conversion of 234123.9 it contains 1, and so forth.

Number conversion has a number of traps for the unwary:

- A . in or at the end of a number by default indicates a double number, not a floating-point number.
- Input FP conversion does not happen if `base` is non-decimal.
- You cannot determine the current number base using the code sequence `base @ .` – the number base is always 10 in the current number base. Instead, use something like `base @ dec.`
- There is a word `bin` but it does *not* set the number base! (see Section 6.22.2 [General files], page 197).
- Standard Forth requires the `.` of a double-precision number to be the final character in the string. Gforth allows the `.` to be anywhere.
- The input integer conversion process does not check for overflow. Instead, you get a value in the range of the result integer type (single or double cell) that is congruent with the input number modulo the number of values in the integer type (e.g., modulo 2^128 for doubles with 64-bit cells).

Words affected by `base` are `rec-number` (see Section 6.17.4.1 [Default recognizers], page 171), `>number` (see Section 6.20.2 [String to number conversion], page 188), and the words described in the sections Section 6.24.1 [Simple numeric output], page 205, and Section 6.20.3 [Integer to string conversion], page 189. `Rec-number` sets `dpl`.

6.20.2 String to number conversion

`>number` (*ud1 c-addr1 u1 – ud2 c-addr2 u2*) core "to-number"

Attempt to convert the character string *c-addr1 u1* to an unsigned number in the current number base. The double *ud1* accumulates the result of the conversion to form *ud2*. Conversion continues, left-to-right, until the whole string is converted or a character that is not convertable in the current number base is encountered (including `+` or `-`). For each convertable character, *ud1* is first multiplied by the value in `BASE` and then incremented by the value represented by the character. *c-addr2* is the location of the first unconverted character (past the end of the string if the whole string was converted). *u2* is the number of unconverted characters in the string. Overflow is not detected.

For higher-level integer-to-number conversion (with number prefixes, group separators (`_`), and double indicators (`.`), use `rec-number` (see Section 6.17.4.1 [Default recognizers], page 171).

`convert` (*ud1 c-addr1 – ud2 c-addr2*) gforth-obsolete

OBSOLETE: This word has been de-standardized in Forth-2012. It is superseded by `>number`.

`>float` (*c-addr u – f:... flag*) floating "to-float"

Actual stack effect: (c_addr u – r t | f). Attempt to convert the character string *c-addr u* to internal floating-point representation. If the string represents a valid floating-point number, *r* is placed on the floating-point stack and *flag* is true. Otherwise, *flag* is false. A string of blanks is a special case and represents the floating-point number 0.

You can alternatively use `rec-float` (see Section 6.17.4.1 [Default recognizers], page 171), which accepts a smaller set of input strings: `>float` is used for known-FP-data possibly coming from another program with funny output syntax and is therefore more liberal in accepting input, while the strings that the text interpreter passes to `rec-float` are not all intended to be interpreted as FP numbers, and therefore `rec-float` is more restrictive.

`>float1` (*c-addr u c – f:... flag*) gforth-1.0 "to-float1"

Actual stack effect: (c_addr u c – r t | f). Attempt to convert the character string *c-addr u* to internal floating-point representation, with *c* being the decimal separator. If the string represents a valid floating-point number, *r* is placed on the floating-point stack and *flag* is true. Otherwise, *flag* is false. A string of blanks is a special case and represents the floating-point number 0.

6.20.3 Integer to string conversion

Forth uses a technique called *pictured numeric output* for formatted printing of integers. In this technique, digits are extracted from the number (using the current output radix defined by `base`, see Section 6.20.1 [Base and integer decimal point], page 187), converted to ASCII codes and prepended to a string that is built in the pictured numeric output buffer, aka `hold` buffer (total size, including all nestings and the `word` buffer, is at most 104 bytes on a 32-bit machine, see Section 8.1.1 [Implementation-defined options], page 310). Arbitrary characters can be prepended to the string during the extraction process. The completed string is specified by an address and length and can be manipulated (`TYPE`ed, copied, modified) under program control.

All of the integer output words described in the previous section (see Section 6.24.1 [Simple numeric output], page 205) are implemented in Gforth using pictured numeric output.

Three important things to remember about pictured numeric output:

- It always operates on double-precision numbers; to display a single-precision number, convert it first (for ways of doing this see Section 6.5.2 [Double precision], page 61).
- It always treats the double-precision number as though it were unsigned. The examples below show ways of printing signed numbers.
- The string is built up from right to left; least significant digit first.

Standard Forth supports a single output buffer (aka `hold` area) that you empty and initialize with `<#` and for which you get the result string with `#>`.

Gforth additionally supports nested usage of this buffer, allowing, e.g., to nest output from the debugging tracer `~~` inside code dealing with the hold area: `<<#` starts a new nest, `#>` produces the result string, and `#>>` unnests: the hold area for the nest is reclaimed, and `#>` now produces the string for the next-outer nest. All of Gforth's higher-level numeric output words use `<<#` ... `#>` ... `#>>` and can be nested inside other users of the hold area.

`<#` (–) core "less-number-sign"

Initialise/clear the pictured numeric output string.

`<<#` (–) gforth-0.5 "less-less-number-sign"

Start a hold area that ends with `#>>`. Can be nested in each other and in `<#`. Note: if you do not match up the `<<#`s with `#>>`s, you will eventually run out of hold area; you can reset the hold area to empty with `<#`.

`#` (*ud1* – *ud2*) core "number-sign"

Used between `<<#` and `#>`. Prepend the least-significant digit (according to `base`) of *ud1* to the pictured numeric output string. *ud2* is *ud1/base*, i.e., the number representing the remaining digits.

`#s` (*ud* – *0 0*) core "number-sign-s"

Used between `<<#` and `#>`. Prepend all digits of *ud* to the pictured numeric output string. `#s` will convert at least one digit. Therefore, if *ud* is 0, `#s` will prepend a "0" to the pictured numeric output string.

`hold` (*char* –) core

Used between `<<#` and `#>`. Prepend the ASCII character *char* to the pictured numeric output string. Use `holds` for prepending non-ASCII characters.

`holds` (*addr u* –) core-ext

Used between `<<#` and `#>`. Prepend the string `addr u` to the pictured numeric output string.

`sign` (*n* –) core

Used between `<<#` and `#>`. If *n* (a *single* number) is negative, prepend "`-`" to the pictured numeric output string.

`#>` (*xd* – *addr u*) core "number-sign-greater"

Complete the pictured numeric output string by discarding *xd* and returning *addr u*; the address and length of the formatted string. A Standard program may modify characters within the string. Does not release the hold area; use `#>>` to release a hold area started with `<<#`, or `<#` to release all hold areas.

`#>>` (–) gforth-0.5 "number-sign-greater-greater"

Release the hold area started with `<<#`.

Here are some examples of using pictured numeric output:

```
: my-u. ( u -- )
  \ Simplest use of pns.. behaves like Standard u.
  0              \ convert to unsigned double
  <<#            \ start conversion
  #s             \ convert all digits
  #>             \ complete conversion
  TYPE SPACE     \ display, with trailing space
  #>> ;          \ release hold area

: cents-only ( u -- )
  0              \ convert to unsigned double
  <<#            \ start conversion
  # #            \ convert two least-significant digits
```

```
  #>              \ complete conversion, discard other digits
  TYPE SPACE      \ display, with trailing space
  #>> ;           \ release hold area

: dollars-and-cents ( u -- )
  0               \ convert to unsigned double
  <<#             \ start conversion
  # #             \ convert two least-significant digits
  '.' hold        \ insert decimal point
  #s              \ convert remaining digits
  '$' hold        \ append currency symbol
  #>              \ complete conversion
  TYPE SPACE      \ display, with trailing space
  #>> ;           \ release hold area

: my-. ( n -- )
  \ handling negatives.. behaves like Standard .
  s>d             \ convert to signed double
  swap over dabs \ leave sign byte followed by unsigned double
  <<#             \ start conversion
  #s              \ convert all digits
  rot sign        \ get at sign byte, append "-" if needed
  #>              \ complete conversion
  TYPE SPACE      \ display, with trailing space
  #>> ;           \ release hold area

: account. ( n -- )
  \ accountants don't like minus signs, they use parentheses
  \ for negative numbers
  s>d             \ convert to signed double
  swap over dabs \ leave sign byte followed by unsigned double
  <<#             \ start conversion
  2 pick          \ get copy of sign byte
  0< IF ')' hold THEN \ right-most character of output
  #s              \ convert all digits
  rot             \ get at sign byte
  0< IF '(' hold THEN
  #>              \ complete conversion
  TYPE SPACE      \ display, with trailing space
  #>> ;           \ release hold area
```

Here are some examples of using these words:

```
1 my-u. 1
hex -1 my-u. decimal FFFFFFFF
1 cents-only 01
1234 cents-only 34
```

```
2 dollars-and-cents $0.02
1234 dollars-and-cents $12.34
123 my-. 123
-123 my. -123
123 account. 123
-456 account. (456)
```

6.20.4 Floating-point to string conversion

`f>str-rdp` (*rf* +*nr* +*nd* +*np* – *c-addr nr*) gforth-0.6 "f-to-str-rdp"

Convert *rf* into a string at *c-addr nr*. The conversion rules and the meanings of *nr* +*nd np* are the same as for `f.rdp` (see Section 6.24.2 [Floating-point output], page 206). The result is in in the pictured numeric output buffer and will be destroyed by anything overwriting that buffer.

`f>buf-rdp` (*rf c-addr* +*nr* +*nd* +*np* –) gforth-0.6 "f-to-buf-rdp"

Convert *rf* into a string at *c-addr nr*. The conversion rules and the meanings of *nr nd np* are the same as for `f.rdp` (see Section 6.24.2 [Floating-point output], page 206).

There is also a primitive used for implementing the higher-level FP-to-string and FP output words:

`represent` (*r c-addr u – n f1 f2*) floating "represent"

Convert the decimal significand (aka mantissa) of *r* into a string in buffer *c-addr u*; *n* is the exponent, *f1* is true if *r* is negative, and *f2* is true if *r* is valid (a finite number in Gforth).

6.21 Environmental Queries

Forth-94 introduced the idea of "environmental queries" as a way for a program running on a system to determine certain characteristics of the system. The Standard specifies a number of strings that might be recognised by a system, and a way of querying them:

`environment?` (*c-addr u – false / ... true*) core "environment-query"

c-addr, u specify a counted string. If the string is not recognised, return a `false` flag. Otherwise return a `true` flag and some (string-specific) information about the queried string.

Note that, while the documentation for (e.g.) `ADDRESS-UNIT-BITS` shows that it returns one cell on the stack, querying it using `environment?` will return an additional item; the `true` flag that shows that the string was recognised; so for querying `ADDRESS-UNIT-BITS` the stack effect of `environment?` may be `( c-addr u -- n true )` or `( c-addr u -- false )`. The support of various environmental queries by systems (even on hardware with enough memory) is somewhat lackluster, so you have to take the possibility of `environment?` returning false seriously.

Several environmental queries deal with the system's limits:

`ADDRESS-UNIT-BITS` (– *n*) environment

Size of one address unit, in bits.

`MAX-CHAR` (– *u*) environment

Maximum value of any character in the character set (there is also `max-xchar`).

`/COUNTED-STRING` (– *n*) environment "slash-counted-string"

Maximum size of a counted string, in characters.

`/HOLD` (– *n*) environment "slash-hold"

Size of the pictured numeric string output buffer, in characters.

`/PAD` (– *n*) environment "slash-pad"

Size of the scratch area pointed to by `PAD`, in characters.

`CORE` (– *f*) environment

True if the complete core word set is present. Always true for Gforth.

`CORE-EXT` (– *f*) environment

True if the complete core extension word set is present. Always true for Gforth.

`FLOORED` (– *f*) environment

True if `/` etc. perform floored division

`MAX-N` (– *n*) environment

Largest usable signed integer.

`MAX-U` (– *u*) environment

Largest usable unsigned integer.

`MAX-D` (– *d*) environment

Largest usable signed double.

`MAX-UD` (– *ud*) environment

Largest usable unsigned double.

`return-stack-cells` (– *n*) environment

Maximum size of the return stack, in cells.

`stack-cells` (– *n*) environment

Maximum size of the data stack, in cells.

`floating-stack` (– *n*) environment

n is non-zero, showing that Gforth maintains a separate floating-point stack of depth *n*.

`#locals` (– *n*) environment "number-locals"

The maximum number of locals in a definition

`wordlists` (– *n*) environment

the maximum number of wordlists usable in the search order

`max-float` (– *r*) environment

The largest usable floating-point number (implemented as largest finite number in Gforth)

`XCHAR-ENCODING` (– *addr u*) environment

Returns a printable ASCII string that reperesents the encoding, and use the preferred MIME name (if any) or the name in `http://www.iana.org/assignments/character-sets` like "ISO-LATIN-1" or "UTF-8", with the exception of "ASCII", where we prefer the alias "ASCII".

`MAX-XCHAR` (– *xchar*) environment

Maximal value for xchar. This depends on the encoding.

`XCHAR-MAXMEM` (– *u*) environment

Maximal memory consumed by an xchar in address units

Several environmental queries are there for determining the presence of the Forth-94 version of a wordset; they all have the stack effect `( -- f )` if the string is present (so the `environment?` stack effect for these queries is `( c-addr u -- false / f true )`.

`block block-ext double double-ext exception exception-ext facility facility-ext file file-ext floating floating-ext locals locals-ext memory-alloc memory-alloc-ext tools tools-ext search-order search-order-ext string string-ext`

These wordset queries were rarely used and implemented, so Forth-2012 did not introduce a way to query for the Forth-2012 variants of the wordsets. Instead, the idea is that you use `[defined]` (see Section 6.17.3 [Interpreter Directives], page 169) for checking the presence of individual standard words instead.

Forth-200x (a group that works on the next standard; the documents that they produce are also called Forth-200x) defines extension queries for the extension proposals once they finish changing (CfV stage), so programs using these proposals can check whether a system has them, and maybe load the reference implementation (if one exists). If `environment?` finds such a query, then the corresponding proposal on `www.forth200x.org` is implemented on the system (but the absence tells you nothing, as usual with `environment?`). These queries have the stack effect `( -- )`, which means that for them `environment?` has the stack effect `( c-addr u -- false / true )`, which is more convenient than that of wordset queries. A number of these proposals have been incorporated into Forth-2012. The extension queries are also not particularly popular among Forth system implementors, so going for `[defined]` may be the better approach. Anyway, Gforth implements the following extension queries:

`X:2value X:buffer X:deferred X:defined X:ekeys X:escaped-strings X:extension-query X:fp-stack X:ftrunc X:fvalue X:locals X:n-to-r X:number-prefixes X:parse-name X:required X:s-escape-quote X:s-to-f X:structures X:synonym X:text-substitution X:throw-iors X:traverse-wordlist X:xchar`

In addition, Gforth implements the following Gforth-specific queries:

`gforth` (– *c-addr u*) gforth-environment

Counted string representing a version string for this version of Gforth (for versions>0.3.0). The version strings of the various versions are guaranteed to be ordered lexicographically.

`os-class` (– *c-addr u*) gforth-environment

Counted string representing a description of the host operating system.

`os-type` (– *c-addr u*) gforth-environment

Counted string equal to `"$host_os"`

The Standard requires that the header space used for environmental queries be distinct from the header space used for definitions.

Typically, a Forth system supports environmental queries by creating a set of definitions in a wordlist that is *only* used for environmental queries; that is what Gforth does. There is no Standard way of adding definitions to the set of recognised environmental queries, but

in Gforth and other systems that use the wordlist mechanism, the wordlist used to honour environmental queries can be manipulated just like any other word list.

`environment-wordlist` (– *wid*) gforth-0.2

wid identifies the word list that is searched by environmental queries (present in SwiftForth and VFX).

`environment` (–) gforth-0.6

A vocabulary for `environment-wordlist` (present in Win32Forth and VFX).

Here are some examples of using environmental queries:

```
s" address-unit-bits" environment? 0=
[IF]
     cr .( environmental attribute address-units-bits unknown... ) cr
[ELSE]
     drop \ ensure balanced stack effect
[THEN]

\ this might occur in the prelude of a standard program that uses THROW
s" exception" environment? [IF]
   0= [IF]
      : throw abort" exception thrown" ;
   [THEN]
[ELSE] \ we don't know, so make sure
   : throw abort" exception thrown" ;
[THEN]

s" gforth" environment? [IF] .( Gforth version ) TYPE
                        [ELSE] .( Not Gforth..) [THEN]

\ a program using v*
s" gforth" environment? [IF]
  s" 0.5.0" compare 0< [IF] \ v* is a primitive since 0.5.0
   : v* ( f_addr1 nstride1 f_addr2 nstride2 ucount -- r )
     >r swap 2swap swap 0e r> 0 ?DO
       dup f@ over + 2swap dup f@ f* f+ over + 2swap
     LOOP
     2drop 2drop ;
  [THEN]
[ELSE] \
  : v* ( f_addr1 nstride1 f_addr2 nstride2 ucount -- r )
  ...
[THEN]
```

Here is an example of adding a definition to the environment word list:

```
get-current environment-wordlist set-current
true constant block
true constant block-ext
set-current
```

You can see what definitions are in the environment word list like this:

```
environment-wordlist wordlist-words
```

6.22 Files

Gforth provides facilities for accessing files that are stored in the host operating system's file-system.

6.22.1 Forth source files

The simplest way to interpret the contents of a file is to use one of these two equivalent syntaxes:

```
include mysource.fs
s" mysource.fs" included
```

You usually want to include a file only if it is not included already (by, say, another source file). In that case, you can replace `include` with `require`:

```
require mysource.fs
s" mysource.fs" required
```

It is good practice to write your source files such that interpreting them does not change the stack. Source files designed in this way can be used with `required` and friends without complications. For example, assume that including `foo.fs` has the stack effect `( u -- u )` (*u* might be a configuration parameter, such as a buffer size). Then you can use it with `require`:

```
1024 require foo.fs drop
```

If `require` actually includes `foo.fs`, the 1024 will be on the stack at the end, just like if `require` does nothing. With such parameters to required files, you have to ensure that the first `require` fits for all uses (i.e., `require` the file early in the master load file).

Another alternative is to pass configuration parameters by defining words (often constants) for them.

`include-file` (*i*x wfileid – j*x*) file

Interpret (process using the text interpreter) the contents of the file *wfileid*.

`included` (*i*x c-addr u – j*x*) file

`include-file` the file whose name is given by the string *c-addr u*.

`included?` (*c-addr u – f*) gforth-0.2 "included-question"

True only if the file *c-addr u* is in the list of earlier included files. If the file has been loaded, it may have been specified as, say, `foo.fs` and found somewhere on the Forth search path. To return `true` from `included?`, you must specify the exact path to the file, even if that is `./foo.fs`

`include` (*...* "*file*" *– ...*) file-ext

`include-file` the file *file*.

`required` (*i*x addr u – i*x*) file-ext

`include-file` the file with the name given by *addr u*, if it is not `included` (or `required`) already. Currently this works by comparing the name of the file (with path) against the names of earlier included files.

`require` (*...* "*file*" *– ...*) file-ext

`include-file` *file* only if it is not included already.

`needs` (... "*name*" – ...) gforth-0.2

An alias for `require`; exists on other systems (e.g., Win32Forth).

`\\\` (–) gforth-1.0 "triple-backslash"

skip remaining source file

`.included` (–) gforth-0.5 "dot-included"

List the names of the files that have been `included`.

`sourcefilename` (– *c-addr u*) gforth-0.2

The name of the source file which is currently the input source. The result is valid only while the file is being loaded. If the current input source is no (stream) file, the result is an arbitrary string. In Gforth, the result is valid during the whole session (but not across `savesystem` etc.).

`sourceline#` (– *u*) gforth-0.2 "sourceline-number"

The line number of the line that is currently being interpreted from a (stream) file. The first line has the number 1. If the current input source is not a (stream) file, the result is an unspecified number.

A definition in Standard Forth for `required` is provided in `compat/required.fs`.

6.22.2 General files

Files are opened/created by name and type. The following file access methods (FAMs) are recognised:

`r/o` (– *fam*) file "r-o"

`r/w` (– *fam*) file "r-w"

`w/o` (– *fam*) file "w-o"

`bin` (*fam1* – *fam2*) file

`+fmode` (*fam1 rwxrwxrwx* – *fam2*) gforth-1.0 "plus-f-mode"

Rwxrwxrwx is a 9-bit number; e.g. `%110_010_000` specifies that the owner can read and write a file, the group can read a file, and others cannot access the file. When passing *fam2* to `create-file`, the created file will have the mode specified by *rwxrwxrwx*, modified by the umask. Default: `%110_110_110` i.e. RW for everyone (modified by umask).

When a file is opened/created, it returns a file identifier, *wfileid* that is used for all other file commands. All file commands also return a status value, *wior*, that is 0 for a successful operation and an implementation-defined non-zero value in the case of an error.

`open-file` (*c-addr u wfam* – *wfileid wior*) file "open-file"

`create-file` (*c-addr u wfam* – *wfileid wior*) file "create-file"

`close-file` (*wfileid* – *wior*) file "close-file"

`delete-file` (*c-addr u* – *wior*) file "delete-file"

`rename-file` (*c-addr1 u1 c-addr2 u2* – *wior*) file-ext "rename-file"

Rename file *c_addr1 u1* to new name *c_addr2 u2*

`read-file` (*c-addr u1 wfileid* – *u2 wior*) file "read-file"

Read *u1* characters from file *wfileid* into the buffer at *c_addr*. A non-zero *wior* indicates an error. *U2* indicates the length of the read data. End-of-file is not an error and is indicated by *u2*<*u1* and *wior*=0.

`read-line` (*c_addr u1 wfileid – u2 flag wior*) file

Reads a line from *wfileid* into the buffer at *c_addr u1*. Gforth supports all three common line terminators: LF, CR and CRLF. A non-zero *wior* indicates an error. A false *flag* indicates that `read-line` has been invoked at the end of the file. *u2* indicates the line length (without terminator): *u2*<*u1* indicates that the line is *u2* chars long; *u2*=*u1* indicates that the line is at least *u1* chars long, *u1* chars of the buffer have been filled with chars from the line, and the next slice of the line will be read with the next `read-line`. If the line is *u1* chars long, the first `read-line` returns *u2*=*u1* and the next read-line returns *u2*=0.

`key-file` (*fd – key*) gforth-0.4

Read one character *n* from *wfileid*. This word disables buffering for *wfileid*. If you want to read characters from a terminal in non-canonical (raw) mode, you have to put the terminal in non-canonical mode yourself (using the C interface); the exception is `stdin`: Gforth automatically puts it into non-canonical mode.

`key?-file` (*wfileid – f*) gforth-0.4 "key-q-file"

f is true if at least one character can be read from *wfileid* without blocking. If you also want to use `read-file` or `read-line` on the file, you have to call `key?-file` or `key-file` first (these two words disable buffering).

`file-eof?` (*wfileid – flag*) gforth-0.6 "file-eof-query"

Flag is true if the end-of-file indicator for *wfileid* is set.

`write-file` (*c-addr u1 wfileid – wior*) file "write-file"

`write-line` (*c-addr u wfileid – ior*) file

`emit-file` (*c wfileid – wior*) gforth-0.2 "emit-file"

`flush-file` (*wfileid – wior*) file-ext "flush-file"

`file-status` (*c-addr u – wfam wior*) file-ext "file-status"

`file-position` (*wfileid – ud wior*) file "file-position"

`reposition-file` (*ud wfileid – wior*) file "reposition-file"

`file-size` (*wfileid – ud wior*) file "file-size"

`resize-file` (*ud wfileid – wior*) file "resize-file"

`slurp-file` (*c-addr1 u1 – c-addr2 u2*) gforth-0.6

c-addr1 u1 is the filename, *c-addr2 u2* is the file's contents. You should `free` *c-addr2* when you no longer need the contents of the file.

`slurp-fid` (*fid – c-addr u*) gforth-0.6

C-addr u is the content of the file *fid*. You should `free` *c-addr* when you no longer need the contents of the file.

`stdin` (*– wfileid*) gforth-0.4 "stdin"

The standard input file of the Gforth process.

`stdout` (*– wfileid*) gforth-0.2 "stdout"

The standard output file of the Gforth process.

`stderr` (– *wfileid*) gforth-0.2 “stderr”

The standard error output file of the Gforth process.

6.22.3 Redirection

You can redirect the output of `type` and `emit` and all the words that use them (all output words that don't have an explicit target file) to an arbitrary file with the `outfile-execute`, used like this:

```
: some-warning ( n -- )
    cr ." warning# " . ;

: print-some-warning ( n -- )
    ['] some-warning stderr outfile-execute ;
```

After `some-warning` is executed, the original output direction is restored; this construct is safe against exceptions. Similarly, there is `infile-execute` for redirecting the input of `key` and its users (any input word that does not take a file explicitly).

`outfile-execute` (*... xt file-id – ...*) gforth-0.7

execute *xt* with the output of `type` etc. redirected to *file-id*.

`outfile-id` (– *file-id*) gforth-0.2

File-id is used by `emit`, `type`, and any output word that does not take a file-id as input. By default `outfile-id` produces the process's `stdout`, unless changed with `outfile-execute`.

`infile-execute` (*... xt file-id – ...*) gforth-0.7

execute *xt* with the input of `key` etc. redirected to *file-id*.

`infile-id` (– *file-id*) gforth-0.4

File-id is used by `key`, `?key`, and anything that refers to the "user input device". By default `infile-id` produces the process's `stdin`, unless changed with `infile-execute`.

If you do not want to redirect the input or output to a file, you can also make use of the fact that `key`, `emit` and `type` are deferred words (see Section 6.11.11 [Deferred Words], page 138). However, in that case you have to worry about the restoration and the protection against exceptions yourself; also, note that for redirecting the output in this way, you have to redirect both `emit` and `type`.

6.22.4 Directories

You can split a file name into a directory and base component:

`basename` (*c-addr1 u1 – c-addr2 u2*) gforth-0.7

Given a file name *c-addr1 u1*, *c-addr2 u2* is the part of it with any leading directory components removed.

`dirname` (*c-addr1 u1 – c-addr1 u2*) gforth-0.7

C-addr1 u2 is the directory name of the file name *c-addr1 u1*, including the final `/`. If *caddr1 u1* does not contain a `/`, *u2*=0.

You can open and read directories similar to files. Reading gives you one directory entry at a time; you can match that to a filename (with wildcards).

`open-dir` (*c-addr u – wdirid wior*) gforth-0.5 “open-dir”

Open the directory specified by *c-addr, u* and return *wdirid* for futher access to it.

`read-dir` (*c-addr u1 wdirid – u2 flag wior*) gforth-0.5 “read-dir”

Attempt to read the next entry from the directory specified by *wdirid* to the buffer of length *u1* at address *c-addr*. If the attempt fails because there is no more entries, *ior*=0, *flag*=0, *u2*=0, and the buffer is unmodified. If the attempt to read the next entry fails because of any other reason, return *ior*<>0. If the attempt succeeds, store file name to the buffer at *c-addr* and return *ior*=0, *flag*=true and *u2* equal to the size of the file name. If the length of the file name is greater than *u1*, store first *u1* characters from file name into the buffer and indicate "name too long" with *ior*, *flag*=true, and *u2*=*u1*.

`close-dir` (*wdirid – wior*) gforth-0.5 “close-dir”

Close the directory specified by *dir-id*.

`filename-match` (*c-addr1 u1 c-addr2 u2 – flag*) gforth-0.5 “match-file”

match the file name *c-addr1 u1* with the pattern *c-addr2 u2*. Patterns match char by char except for the special characters ’*’ and ’?’, which are wildcards for several (’*’) or one (’?’) character.

`get-dir` (*c-addr1 u1 – c-addr2 u2*) gforth-0.7 “get-dir”

Store the current directory name in the buffer specified by *c-addr1, u1*; if there is sufficient space, *c-addr2*=*c-addr1* and *u2* is the length of the directory name. If the buffer size is not sufficient, return 0 0

`set-dir` (*c-addr u – wior*) gforth-0.7 “set-dir”

Change the current directory to *c-addr, u*. Return an error if this is not possible

`=mkdir` (*c-addr u wmode – wior*) gforth-0.7 “equals-mkdir”

Create directory *c-addr u* with mode *wmode* modified by umask. *wmode* is a 9-bit number rwxrwxrwx (see `+fmode`, see Section 6.22.2 [General files], page 197).

`mkdir-parents` (*c-addr u mode – ior*) gforth-0.7

create the directory *c-addr u* and all its parents with mode *mode* (modified by umask). *mode* is a 9-bit number rwxrwxrwx (see `+fmode`, see Section 6.22.2 [General files], page 197).

6.22.5 Search Paths

If you specify an absolute filename (i.e., a filename starting with `/` or `~`, or with `:` in the second position (as in ‘`C:...`’)) for `included` and friends, that file is included just as you would expect.

If the filename starts with `./`, this refers to the directory that the present file was `included` from. This allows files to include other files relative to their own position (irrespective of the current working directory or the absolute position). This feature is essential for libraries consisting of several files, where a file may include other files from the library. It corresponds to `#include "..."` in C. If the current input source is not a file, `.` refers to the directory of the innermost file being included, or, if there is no file being included, to the current working directory.

For relative filenames (not starting with `./`), Gforth uses a search path similar to Forth's search order (see Section 6.19 [Word Lists], page 182). It tries to find the given filename in the directories present in the path, and includes the first one it finds. There are separate search paths for Forth source files and general files. If the search path contains the directory `.`, this refers to the directory of the current file, or the working directory, as if the file had been specified with `./`.

Use `~+` to refer to the current working directory (as in the `bash`).

`absolute-file?` (*addr u – flag*) gforth-1.0 "absolute-file-question"

A filename is absolute if it starts with a / or a `~` (`~` expansion), or if it is in the form ./*, extended regexp: `^[/~]|./`, or if it has a colon as second character (`"C:..."`). Paths simply containing a / are not absolute!

6.22.5.1 Source Search Paths

The search path is initialized when you start Gforth (see Section 2.1 [Invoking Gforth], page 4). You can display it and change it using `fpath` in combination with the general path handling words.

`fpath` (*– path-addr*) gforth-0.4

`.fpath` (–) gforth-0.4 "dot-fpath"

Display the contents of the Forth search path.

`file>fpath` (*addr1 u1 – addr2 u2*) gforth-1.0 "file-to-fpath"

Searches for a file with the name *c-addr1 u1* in the `fpath`. If successful, *c-addr u2* is the absolute file name or the file name relative to the current working directory. Throws an exception if the file cannot be opened.

Here is an example of using `fpath` and `require`:

```
fpath path= /usr/lib/forth/|./
require timer.fs
```

6.22.5.2 General Search Paths

Your application may need to search files in several directories, like `included` does. To facilitate this, Gforth allows you to define and use your own search paths, by providing generic equivalents of the Forth search path words:

`open-path-file` (*addr1 u1 path-addr – wfileid addr2 u2 0 | ior*) gforth-0.2

Look in path *path-addr* for the file specified by *addr1 u1*. If found, the resulting path and an (read-only) open file descriptor are returned. If the file is not found, *ior* is what came back from the last attempt at opening the file (in the current implementation).

`file>path` (*c-addr1 u1 path-addr – c-addr2 u2*) gforth-1.0 "file-to-path"

Searches for a file with the name *c-addr1 u1* in path stored in *path-addr*. If successful, *c-addr u2* is the absolute file name or the file name relative to the current working directory. Throws an exception if the file cannot be opened.

`clear-path` (*path-addr* –) gforth-0.5

Set the path *path-addr* to empty.

`also-path` (*c-addr len path-addr* –) gforth-0.4

add the directory *c-addr len* to *path-addr*.

`.path` (*path-addr* –) gforth-0.4 "dot-path"

Display the contents of the search path *path-addr*.

`path+` (*path-addr* `"`*dir*`"` –) gforth-0.4 "path-plus"

Add the directory *dir* to the search path *path-addr*.

`path=` (*path-addr* `"`*dir1*`|`*dir2*`|`*dir3*`"` –) gforth-0.4 "path-equals"

Make a complete new search path; the path separator is `|`.

Here's an example of creating a custom search path:

```
variable mypath \ no special allocation required, just a variable
mypath path= /lib|/usr/lib \ assign initial directories
mypath path+ /usr/local/lib \ append directory
mypath .path \ output:"/lib /usr/lib /usr/local/lib"
```

Search file and show resulting path:

```
s" libm.so" mypath open-path-file throw type close-file \ output:"/lib/libm.so"
```

6.23 Blocks

Traditionally, Forth has been used on systems without an operating system[21], and in particular without a file system. Forth provides a mechanism, called *blocks*, for accessing mass storage on such systems.

A block is a 1024-byte data area, which can be used to hold data or Forth source code. No structure is imposed on the contents of the block. A block is identified by its number; blocks are numbered contiguously from 1 to an implementation-defined maximum.

A typical system that used blocks but no operating system might use a single floppy-disk drive for mass storage, with the disks formatted to provide 256-byte sectors. Blocks would be implemented by assigning the first four sectors of the disk to block 1, the second four sectors to block 2 and so on, up to the limit of the capacity of the disk. The disk would not contain any file system information, just the set of blocks.

On systems that do provide file services, blocks are typically implemented by storing a sequence of blocks within a single *blocks file*. The size of the blocks file will be an exact multiple of 1024 bytes, corresponding to the number of blocks it contains. This is the mechanism that Gforth uses.

Gforth's current implementation of blocks is not thread-safe, so blocks should only be used by one task (see Section 6.31.1 [Pthreads], page 271).

Only one blocks file can be open at a time. If you use block words without having specified a blocks file, Gforth defaults to the blocks file `blocks.fb`. Gforth uses the Forth search path when attempting to locate a blocks file (see Section 6.22.5.1 [Source Search Paths], page 201).

When you access blocks, Gforth uses a number of *block buffers* as intermediate storage. The behaviour of the block buffers is analogous to that of a cache. Each block buffer has three states:

- Unassigned

[21] This is called *native*, whereas Forth running on top of an OS is called *hosted*.

- Assigned-clean
- Assigned-dirty

Initially, all block buffers are *unassigned*. In order to access a block, the block (specified by its block number) must be assigned to a block buffer.

The assignment of a block to a block buffer is performed by `block` or `buffer`. Use `block` when you wish to modify the existing contents of a block. Use `buffer` don't care about the existing contents of the block and just want to overwrite it.

Once a block has been assigned to a block buffer using `block` or `buffer`, that block buffer becomes the *current block buffer*. Data may only be manipulated (read or written) within the current block buffer.

When the contents of the current block buffer has been modified, it is necessary, *before calling* `block` *or* `buffer` *again*, to either abandon the changes (by doing nothing) or mark the block as changed (assigned-dirty), using `update`. Using `update` does not change the blocks file; it simply changes a block buffer's state to *assigned-dirty*. The block will be written implicitly when it's buffer is needed for another block, or explicitly by `flush` or `save-buffers`.

`Flush` writes all *assigned-dirty* blocks back to the blocks file on disk. Leaving Gforth with `bye` also performs a `flush`.

When a block is used for storing source code, it is traditional to display the contents as 16 lines each of 64 characters (with `list`). A block provides a single, continuous stream of input (for example, it acts as a single parse area) – there are no end-of-line characters within a block, and no end-of-file character at the end of a block. There are two consequences of this:

- The last character of one line A is contiguous with the first character of the following line B, so you need to either put at least one blank at the end of A, or at the start of B in order to separate the last word of A from the first word of B.
- The word \ – comment to end of line – requires special treatment; in the context of a block it causes all characters until the end of the current 64-character "line" to be ignored.

In Gforth, when you use `block` with a non-existent block number, the current blocks file will be extended to the appropriate size and the block buffer will be initialised with spaces.

Gforth includes a simple block editor (type `use blocked.fb 0 list` for details) but doesn't encourage the use of blocks; the mechanism is only provided for backward compatibility.

Common techniques that are used when working with blocks include:

- A screen editor that allows you to edit blocks without leaving the Forth environment.
- Shadow screens; where every code block has an associated block containing comments (for example: code in odd block numbers, comments in even block numbers). Typically, the block editor provides a convenient mechanism to toggle between code and comments.
- Load blocks; a single block (typically block 1) contains a number of `thru` commands which `load` the whole of the application.

See Frank Sergeant's Pygmy Forth to see just how well blocks can be integrated into a Forth programming environment.

`open-blocks` (*c-addr u* –) gforth-0.2

C-addr u is the name of a file; `open-blocks` opens this file as the current blocks file.

`use` (`"`*file*`"` –) gforth-0.2

Open *file* as the current blocks file.

`get-block-fid` (– *wfileid*) gforth-0.2

Return the file-id of the current blocks file. If no blocks file has been opened, use `blocks.fb` as the default blocks file.

`block-offset` (– *addr*) gforth-0.5

User variable containing the number of the first block (default since 0.5.0: 0). Block files created with Gforth versions before 0.5.0 have the offset 1. If you use these files you can: `1 offset !`; or add 1 to every block number used; or prepend 1024 characters to the file.

`block` (*u* – *addr*) block

The contents of block *u* are found at *addr* (and the following 1023 bytes). *Addr* is valid until there is another call to `block` or `buffer` (possibly inside another block-access word). If the block is not yet in a buffer, `block` reads it from mass storage.

`buffer` (*u* – *addr*) block

`Addr` (and the following 1023 bytes) are the buffer of block *u*; this memory area is initialized arbitrarily. *Addr* is valid until there is another call to `block` or `buffer` (possibly inside another block-access word). The subtle difference between `buffer` and `block` mean that you should only use `buffer` if you don't care about the previous contents of block *u*.

`update` (–) block

Mark the state of the current block buffer as assigned-dirty.

`updated?` (*u* – *f*) gforth-0.2 "updated-question"

If and only if there is a buffer for block *u* and it has been `updated`, return true.

`save-buffers` (–) block

Transfer the contents of each `updated` block buffer to mass storage, then mark all block buffers as assigned-clean.

`empty-buffers` (–) block-ext

Mark all block buffers as unassigned; if any had been marked as assigned-dirty (by `update`), the changes to those blocks will be lost.

`flush` (–) block

Perform the functions of `save-buffers` then `empty-buffers`.

`list` (*u* –) block-ext

Display block *u* as 16 numbered lines, each of 64 characters.

`scr` (– *a-addr*) block-ext "s-c-r"

User variable containing the block number of the block most recently processed by `list`.

`load` (*i*x u* – *j*x*) block

Text-interpret block *u*. Block 0 cannot be `load`ed.

`thru` (*i*x n1 n2* – *j*x*) block-ext

`load` the blocks *n1* up to and including *n2* in sequence.

`+load` (*i*x n – j*x*) gforth-0.2 "plus-load"

Used within a block to load the block specified as the current block `+` *n*.

`+thru` (*i*x n1 n2 – j*x*) gforth-0.2 "plus-thru"

Used within a block to load the range of blocks specified as the current block `+` *n1* up to and including the current block `+` *n2*.

`-->` (–) gforth-0.2 "dash-dash-greater-than"

If this symbol is encountered whilst loading block *n*, discard the remainder of the block and load block *n+1*. Used for chaining multiple blocks together as a single loadable unit. Not recommended, because it destroys the independence of loading. Use `thru` (which is standard) or `+thru` instead.

`block-included` (*a-addr u* –) gforth-0.2

Use within a block that is to be processed by `load`. Save the current blocks file specification, open the blocks file specified by *a-addr u* and `load` block 1 from that file (which may in turn chain or load other blocks). Finally, close the blocks file and restore the original blocks file.

6.24 Other I/O

6.24.1 Simple numeric output

The simplest output functions are those that display integers from the data stack. Numbers are displayed in the base (aka radix) stored in `base` (see Section 6.20.1 [Base and integer decimal point], page 187). For outputting integers with additional formatting requirements see Section 6.20.3 [Integer to string conversion], page 189.

`.` (*n* –) core "dot"

Display (the signed single number) *n* in free-format, followed by a space.

`dec.` (*n* –) gforth-0.2 "dec-dot"

Display *n* as a signed decimal number, followed by a space.

`h.` (*u* –) gforth-1.0 "h-dot"

Display *u* as an unsigned hex number, prefixed with a `"$"` and followed by a space.

`hex.` (*u* –) gforth-0.2 "hex-dot"

Display *u* as an unsigned hex number, prefixed with a `$` and followed by a space. Another name for this word is `h.`, which is present in several other systems, but not in Gforth before 1.0.

`u.` (*u* –) core "u-dot"

Display (the unsigned single number) *u* in free-format, followed by a space.

`.r` (*n1 n2* –) core-ext "dot-r"

Display *n1* right-aligned in a field *n2* characters wide. If more than *n2* characters are needed to display the number, all digits and, if necessary, the sign "-", are displayed.

`u.r` (*u n* –) core-ext "u-dot-r"

Display *u* right-aligned in a field *n* characters wide. If more than *n* characters are needed to display the number, all digits are displayed.

`dec.r` (*u n* –) gforth-0.5 "dec-dot-r"

Display *u* as a unsigned decimal number in a field *n* characters wide.

`d.` (*d* –) double "d-dot"

Display (the signed double number) *d* in free-format. followed by a space.

`ud.` (*ud* –) gforth-0.2 "u-d-dot"

Display (the signed double number) *ud* in free-format, followed by a space.

`d.r` (*d n* –) double "d-dot-r"

Display *d* right-aligned in a field *n* characters wide. If more than *n* characters are needed to display the number, all digits and, if necessary, the sign "-", are displayed.

`ud.r` (*ud n* –) gforth-0.2 "u-d-dot-r"

Display *ud* right-aligned in a field *n* characters wide. If more than *n* characters are needed to display the number, all digits are displayed.

6.24.2 Floating-point output

Floating-point output is always displayed using decimal base (even if `BASE` is different). The words starting with `e.` output the FP number in a format that the text interpreter accepts as input.

`f.` (*r* –) floating-ext "f-dot"

Display (the floating-point number) *r* without exponent, followed by a space.

`fe.` (*r* –) floating-ext "f-e-dot"

Display *r* using engineering notation (with exponent dividable by 3), followed by a space.

`fs.` (*r* –) floating-ext "f-s-dot"

Display *r* using scientific notation (with exponent), followed by a space.

`fp.` (*r* –) floating-ext "f-p-dot"

Display *r* using SI prefix notation (with exponent dividable by 3, converted into SI prefixes if available), followed by a space.

`e.` (*f: r* –) gforth-1.0 "e-dot"

Print *r* with the least number of mantissa digits such that the result, when converted with `>float`, produces *r* (for finite *r*). Caveat: `e.` only works as specified if `represent` (which calls the C library's `ecvt_r()`) produces the closest mantissa for the given buffer length; that is the case on not-too-old versions of glibc.

`e.p` (*r u* –) gforth-experimental "e-dot-p"

Print *r* with up to *u* mantissa digits, leaving away trailing mantissa digits. One additional mantissa digit is displayed if this can avoid showing an exponent.

`e.exact` (*r* –) gforth-experimental "e-dot-exact"

Print *r* at full length, with possibly more than 700 mantissa digits. Caveat: `e.exact` only provides the exact output if `represent` does (which calls the C library's `ecvt_r()`); that is the case on not-too-old versions of glibc.

Examples of printing the number 1234.5678E23 in the different floating-point output formats are shown below.

```
1234.5678E23
fdup f.      \ 123456780000000000000000000.
fdup fe.     \ 123.456780000000E24
fdup fs.     \ 1.23456780000000E26
fdup fp.     \ 123.456780000000Y
fdup e.      \ 1.2345678e26 ok
fdup 5 e.p   \ 1.2346e26 ok
fdup e.exact \ 123456780000000004618977280e ok
fdrop
```

Note that `1234.5678E23` cannot be represented exactly as IEEE double-precision (binary64) FP number (Gforth's float format), so there is a rounding error when converting that string to a float, and the output of `e.exact` shows the exact value of the resulting number. All the other output words produce a result that rounds the mantissa to fit in a certain number of digits (and possibly extends that with trailing zeros).

The length of the output of `f.` is influenced by:

`precision` (– *u*) floating-ext

u is the number of significant digits currently used by `F.` `FE.` and `FS.`

`set-precision` (*u* –) floating-ext

Set the number of significant digits currently used by `F.` `FE.` and `FS.` to *u*.

You can control the output in more detail with:

`f.rdp` (*rf* +*nr* +*nd* +*np* –) gforth-0.6 "f-dot-rdp"

Print float *rf* formatted. The total width of the output is *nr*. For fixed-point notation, the number of digits after the decimal point is +*nd* and the minimum number of significant digits is *np*. `Set-precision` has no effect on `f.rdp`. Fixed-point notation is used if the number of siginicant digits would be at least *np* and if the number of digits before the decimal point would fit. If fixed-point notation is not used, exponential notation is used, and if that does not fit, asterisks are printed. We recommend using *nr*>=7 to avoid the risk of numbers not fitting at all. We recommend *nr*>=*np*+5 to avoid cases where `f.rdp` switches to exponential notation because fixed-point notation would have too few significant digits, yet exponential notation offers fewer significant digits. We recommend *nr*>=*nd*+2, if you want to have fixed-point notation for some numbers; the smaller the value of *np*, the more cases are shown in fixed-point notation (cases where few or no significant digits remain in fixed-point notation). We recommend *np*>*nr*, if you want to have exponential notation for all numbers.

To give you a better intuition of how they influence the output, here are some examples of parameter combinations; in each line the same number is printed, in each column the same parameter combination is used for printing:

```
    12 13 0    7 3 4   7 3 0   7 3 1   7 5 1   7 7 1   7 0 2  4 2 1
|-1.234568E-6|-1.2E-6| -0.000|-1.2E-6|-1.2E-6|-1.2E-6|-1.2E-6|****|
|-1.234568E-5|-1.2E-5| -0.000|-1.2E-5|-.00001|-1.2E-5|-1.2E-5|****|
|-1.234568E-4|-1.2E-4| -0.000|-1.2E-4|-.00012|-1.2E-4|-1.2E-4|****|
|-1.234568E-3|-1.2E-3| -0.001| -0.001|-.00123|-1.2E-3|-1.2E-3|****|
```

```
|-1.234568E-2|-1.2E-2| -0.012| -0.012|-.01235|-1.2E-2|-1.2E-2|-.01|
|-1.234568E-1|-1.2E-1| -0.123| -0.123|-.12346|-1.2E-1|-1.2E-1|-.12|
|-1.2345679E0| -1.235| -1.235| -1.235|-1.23E0|-1.23E0|-1.23E0|-1E0|
|-1.2345679E1|-12.346|-12.346|-12.346|-1.23E1|-1.23E1|   -12.|-1E1|
|-1.2345679E2|-1.23E2|-1.23E2|-1.23E2|-1.23E2|-1.23E2|  -123.|-1E2|
|-1.2345679E3|-1.23E3|-1.23E3|-1.23E3|-1.23E3|-1.23E3| -1235.|-1E3|
|-1.2345679E4|-1.23E4|-1.23E4|-1.23E4|-1.23E4|-1.23E4|-12346.|-1E4|
|-1.2345679E5|-1.23E5|-1.23E5|-1.23E5|-1.23E5|-1.23E5|-1.23E5|-1E5|
```

6.24.3 Miscellaneous output

`cr` (–) core "c-r"

Output a newline (of the favourite kind of the host OS). Note that due to the way the Forth command line interpreter inserts newlines, the preferred way to use `cr` is at the start of a piece of text; e.g., `cr ." hello, world"`.

`space` (–) core

Display one space.

`spaces` (*u* –) core

Display *u* spaces.

`out` (– *addr*) gforth-1.0

Addr contains a number that tries to give the position of the cursor within the current line on the user output device: It resets to 0 on `cr`, increases by the number of characters by `type` and `emit`, and decreases on `backspaces`. Unfortunately, it does not take into account tabs, multi-byte characters, or the existence of Unicode characters with width 0 and 2, so it only works for simple cases.

`.\"` (*compilation 'ccc"' – ; run-time* –) gforth-0.6 "dot-backslash-quote"

Like `."`, but translates C-like \-escape-sequences (see `S\"`).

`."` (*compilation 'ccc"' – ; run-time* –) core "dot-quote"

Compilation: Parse a string *ccc* delimited by a `"` (double quote). At run-time, display the string. Interpretation semantics for this word are undefined in standard Forth. Gforth's interpretation semantics are to display the string.

`.(` (*compilation&interpretation 'ccc<close-paren>'* –) core-ext "dot-paren"

Compilation and interpretation semantics: Parse a string *ccc* delimited by a `)` (right parenthesis). Display the string. This is often used to display progress information during compilation; see examples below.

If you don't want to worry about whether to use `.( hello)` or `." hello"`, you can write `"hello" type`, which gives you what you usually want (but is less portable to other Forth systems).

As an example, consider the following program:

```
.( text-1)     \ prints "text-1"
: my-word
  ." text-2"
  .( text-3)   \ prints "text-3"
  "text-4" type
```

```
;

my-word        \ prints "text-2text-4"

." text-5"     \ prints "text-5"
"text-6" type  \ prints "text-6"
```

If you want to understand why this code behaves this way, the explanations for this concrete example are:

- Messages `text-1` and `text-3` are displayed because `.(` is an immediate word; it behaves in the same way whether it is used inside or outside a colon definition (see Section 6.14 [Interpretation and Compilation Semantics], page 150).
- Message `text-2` is not displayed during the definition of `my-word`, because the text interpreter performs the compilation semantics for `."`. Later the interpretation semantics of `my-word` are performed, and they perform the run-time semantics of `."`.
- Message `text-4` is not displayed during the definition of `my-word`, because `"text-4"` (see `rec-string`, `translate-string`) performs the compiling run-time for the string, which compiles the interpreting run-time (see Section 6.17.4.3 [Defining recognizers], page 176) into `my-word`, and `type` compiles the interpretation/execution semantics of `type` into `my-word`. Later the interpretation semantics of `my-word` are performed, and they perform the compiled interpreting run-time and interpretation semantics.
- Message `text-5` is displayed because of Gforth's added interpretation semantics for `."`.
- Message `text-6` is displayed because `"text-6" type` is interpreted.

6.24.4 Displaying characters and strings

`type` (*c-addr u* –) core

If *u*>0, display *u* characters from a string starting with the character stored at *c-addr*.

`xemit` (*xc* –) xchar "x-emit"

Display extended char *xc*.

`emit` (*c* –) core

Display the byte *c*; for ASCII characters, `emit` is equivalent to `xemit`; for multi-byte characters, additional `emit` or `type` calls may be needed to display a complete character.

`typewhite` (*c-addr u* –) gforth-0.2

Like `type`, but white space is printed instead of the characters. For now, only works correctly for strings of ASCII characters.

6.24.5 Terminal output

If you are outputting to a terminal, you may want to control the positioning of the cursor:

`at-xy` (*x y* –) facility "at-x-y"

Put the curser at position *x y*. The top left-hand corner of the display is at 0 0.

`at-deltaxy` (*dx dy* –) gforth-0.7

With the current position at *x y*, put the cursor at *x*+*dx y*+*dy*.

In order to know where to position the cursor, it is often helpful to know the size of the screen:

`form` (– *nlines ncols*) gforth-0.2

And sometimes you want to use:

`page` (–) facility

Clear the screen

Note that on non-terminals you should use `#ff emit`, not `page`, to get a form feed.

6.24.5.1 Color output

The following words are used to create (semantic) colorful output; further output is produced in the color and style given by the word; the actual color and style depends on the theme (see Section 6.24.5.2 [Color themes], page 211) and on your terminal setup.[22]

`default-color` (–) gforth-1.0

Future terminal output will use the system-default color

`error-color` (–) gforth-1.0

Future terminal output will use the error color (red)

`error-hl-inv` (–) gforth-1.0

Future terminal output will be shown in the inverted error color (background color on red)

`error-hl-ul` (–) gforth-1.0

Future terminal output will be shown in the error color (red) and underlined.

`warning-color` (–) gforth-1.0

Future terminal output will use the color for warnings (`light-mode`: blue, `dark-mode`: yellow)

`info-color` (–) gforth-1.0

Future terminal output will use the color for informative output (`light-mode`: green, `dark-mode`: cyan)

`success-color` (–) gforth-1.0

Future terminal output will use the color for success (green)

`input-color` (–) gforth-1.0

Future terminal output will use the color for user-input (`light-mode`: bold black, `dark-mode`: bold white, `magenta-input`: magenta)

`status-color` (–) gforth-1.0

Future terminal output will be shown in the color for the interpret-state status bar (inverted blue)

`compile-color` (–) gforth-1.0

Future terminal output will be shown in the color for the compile-state status bar (inverted magenta)

`postpone-color` (–) gforth-1.0

Future terminal output will be shown in the color for the postpone-state status bar (inverted red)

[22] `https://www.complang.tuwien.ac.at/anton/xterm-colors/`

6.24.5.2 Color themes

Depending on whether you use a light or dark background, some colors tend to lead to too-low contrast. Therefore, Gforth supports adjusting its colors to the background with `light-mode` (for light background) and `dark-mode` for dark background.

Gforth tries to select the right mode automatically, but that does not always work. In order to avoid having to set the theme every time you enter Gforth, you can set the environment variable `GFORTH_INIT` before starting Gforth. Possible values are: `light`, `dark`, `uncolored`, or `auto` (default).

In addition, you can specify an input theme (separated from the mode theme by a space in `GFORTH_INIT` if both are present): `magenta` or `default` (default).

`light-mode` (–) gforth-1.0

Color theme for white background: sets the colors for future uses of color output words (see Section 6.24.5.1 [Color output], page 210).

`dark-mode` (–) gforth-1.0

Color theme for black background: sets the colors for future uses of color output words (see Section 6.24.5.1 [Color output], page 210).

`uncolored-mode` (–) gforth-1.0

"Color theme" that does not set colors: all the color output words (see Section 6.24.5.1 [Color output], page 210) just set the default colors or (for status bar colors) the inverted default colors. This mode does not set colors, but uses the default ones.

`magenta-input` (–) gforth-1.0

Future use of `input-color` will result in further output in magenta, which may be easier to recognize in presentations than just bold.

`default-input` (–) gforth-1.0

Future use of `input-color` will result in further output in bold foreground color (the default setting).

6.24.6 Single-key input

If you want to get a single byte, you can use `key`; to check whether a character is available for `key`, you can use `key?`.

`key` (– *c*) core

Receive (but do not display) one byte *c*.

`key-ior` (– *c*|*ior*) gforth-1.0

Receive (but do not display) one byte *c*. In case of an error or interrupt, return the negative *ior* instead.

`key?` (– *flag*) facility "key-question"

If a byte is available for receiving with `key`, return true, otherwise false.

`xkey` (– *xc*) xchar "x-key"

Reads an extended character *xc* xchar from the terminal without printing it. This will discard all input events until all bytes of *xc* have been received.

`xkey?` (– *flag*) xchar "x-key-query"

Ideally this word would return true if a complete extended char is available for input, otherwise false. Unfortunately, currently also a partial extended character results in returning true.

If you want to process a mix of printable and non-printable characters, you can do that with `ekey` and friends. `Ekey` produces a keyboard event that you have to convert into a character with `ekey>char` or into a key identifier with `ekey>fkey`.

Typical code for using EKEY looks like this:

```
ekey ekey>xchar if ( xc )
  ... \ do something with the character
else ekey>fkey if ( key-id )
  case
    k-up                                    of ... endof
    k-f1                                    of ... endof
    k-left k-shift-mask or k-ctrl-mask or of ... endof
    ...
  endcase
else ( keyboard-event )
  drop \ just ignore an unknown keyboard event type
then then
```

`ekey` (– *u*) facility-ext "e-key"

Receive a keyboard event *u* (encoding implementation-defined).

`ekey>xchar` (*u* – *u false* | *xc true*) xchar-ext "e-key-to-x-char"

Convert keyboard event *u* into extended char *xc*. If that is possible, return *xt* and true, otherise *u* and *false*.

`ekey>char` (*u* – *u false* | *c true*) facility-ext "e-key-to-char"

Convert keyboard event *u* into the ASCII char *c*. If that is possible, return *c* and true, otherise *u* and false. Instead of `ekey>char`, use `ekey>xchar` if available.

`ekey>fkey` (*u1* – *u2 f*) facility-ext "e-key-to-f-key"

If u1 is a keyboard event in the special key set, convert keyboard event *u1* into key id *u2* and return true; otherwise return *u1* and false.

`ekey?` (– *flag*) facility-ext "e-key-question"

If a keyboard event is available for receiving with `ekey`, return true, otherwise false.

The key identifiers for cursor keys are:

`k-left` (– *u*) facility-ext

`k-right` (– *u*) facility-ext

`k-up` (– *u*) facility-ext

`k-down` (– *u*) facility-ext

`k-home` (– *u*) facility-ext

aka Pos1

`k-end` (– *u*) facility-ext

`k-prior` (– *u*) facility-ext

aka PgUp

`k-next` (– *u*) facility-ext

aka PgDn

`k-insert` (– *u*) facility-ext

`k-delete` (– *u*) facility-ext

the `DEL` key on my xterm, not backspace

The key identifiers for function keys (aka keypad keys) are:

`k-f1` (– *u*) facility-ext "k-f-1"

`k-f2` (– *u*) facility-ext "k-f-2"

`k-f3` (– *u*) facility-ext "k-f-3"

`k-f4` (– *u*) facility-ext "k-f-4"

`k-f5` (– *u*) facility-ext "k-f-5"

`k-f6` (– *u*) facility-ext "k-f-6"

`k-f7` (– *u*) facility-ext "k-f-7"

`k-f8` (– *u*) facility-ext "k-f-8"

`k-f9` (– *u*) facility-ext "k-f-9"

`k-f10` (– *u*) facility-ext "k-f-10"

`k-f11` (– *u*) facility-ext "k-f-11"

`k-f12` (– *u*) facility-ext "k-f-12"

Note that `k-f11` and `k-f12` are not as widely available.

You can combine these key identifiers with masks for various shift keys:

`k-shift-mask` (– *u*) facility-ext

`k-ctrl-mask` (– *u*) facility-ext

`k-alt-mask` (– *u*) facility-ext

There are a number of keys that have ASCII values, and therefore are unlikely to be reported as special keys, but the combination of these keys with shift keys may be reported as a special key:

`k-enter` (– *u*) gforth-1.0

`k-backspace` (– *u*) gforth-1.0

`k-tab` (– *u*) gforth-1.0

Moreover, there are the following key codes for keys and other events:

`k-winch` (– *u*) gforth-1.0

This key code may be generated when the user changes the window size; if you have cached the value returned by `form`, this indicates that you should update your cache.

`k-pause` (– *u*) gforth-1.0

`k-mute` (– *u*) gforth-1.0

`k-volup` (– *u*) gforth-1.0

`k-voldown` (– *u*) gforth-1.0

`k-sel` (– *u*) gforth-1.0

`k-eof` (– *u*) gforth-1.0

Note that, even if a Forth system has `ekey>fkey` and the key identifier words, the keys are not necessarily available or it may not necessarily be able to report all the keys and all the possible combinations with shift masks. Therefore, write your programs in such a way that they are still useful even if the keys and key combinations cannot be pressed or are not recognized.

Examples: Older keyboards often do not have an F11 and F12 key. If you run Gforth in an xterm, the xterm catches a number of combinations (e.g., `Shift-Up`), and never passes it to Gforth. Finally, Gforth currently does not recognize and report combinations with multiple shift keys (so the `shift-ctrl-left` case in the example above would never be entered).

Gforth recognizes various keys available on ANSI terminals (in MS-DOS you need the ANSI.SYS driver to get that behaviour); it works by recognizing the escape sequences that ANSI terminals send when such a key is pressed. If you have a terminal that sends other escape sequences, you will not get useful results on Gforth. Other Forth systems may work in a different way.

Gforth also provides a few words for outputting names of function keys:

`fkey.` (*u* –) gforth-1.0 "fkey-dot"

Print a string representation for the function key *u*. *U* must be a function key (possibly with modifier masks), otherwise there may be an exception.

`simple-fkey-string` (*u1* – *c-addr u*) gforth-1.0

c-addr u is the string name of the function key *u1*. Only works for simple function keys without modifier masks. Any *u1* that does not correspond to a simple function key currently produces an exception.

6.24.7 String input from the terminal

For ways of storing character strings in memory see Section 6.9.1 [String representations], page 88.

Words for inputting one line from the keyboard:

`accept` (*c-addr* +*n1* – +*n2*) core

Get a string of up to *n1* characters from the user input device and store it at *c-addr*. *n2* is the length of the received string. The user indicates the end by pressing `RET`. Gforth supports all the editing functions available on the Forth command line (including history and word completion) in `accept`.

`edit-line` (*c-addr n1 n2* – *n3*) gforth-0.6

edit the string with length *n2* in the buffer *c-addr n1*, like `accept`.

Obsolete words:

`expect` (*c-addr* +*n* –) gforth-obsolete

Receive a string of at most +*n* characters, and store it in memory starting at *c-addr*. The string is displayed. Input terminates when the <return> key is pressed or +*n* characters have been received. The normal Gforth line editing capabilites are available. The length of the string is stored in `span`; it does not include the <return> character. OBSOLETE: This word has been de-standardized in Forth-2012. Use `accept` instead.

`span` (– *addr*) gforth-obsolete

The variable at *addr* contains the length of the last string received by `expect`. OBSOLETE: This word has been de-standardized in Forth-2012. Use `accept` instead.

6.24.8 Pipes

In addition to using Gforth in pipes created by other processes (see Section 2.7 [Gforth in pipes], page 12), you can create your own pipe with `open-pipe`, and read from or write to it.

`open-pipe` (*c-addr u wfam – wfileid wior*) gforth-0.2 “open-pipe”

c-addr u is the name/path of an OS-level program. If *wfam* is `r/o`, the standard output of the program is piped into the Gforth process and can be read from *wfileid*. If *wfam* is `w/o`, data written to *wfileid* is piped as standard input into the program. *wior* is 0 if and only if opening the pipe succeeded.

`close-pipe` (*wfileid – wretval wior*) gforth-0.2 “close-pipe”

Closes a pipe *wfileid* opened with `open-pipe`. If *wior* is 0, *wretval* is the exit code of the command invoked by `open-pipe`.

If you write to a pipe, Gforth can throw a `broken-pipe-error`; if you don’t catch this exception, Gforth will catch it and exit, usually silently (see Section 2.7 [Gforth in pipes], page 12). Since you probably do not want this, you should wrap a `catch` or `try` block around the code from `open-pipe` to `close-pipe`, so you can deal with the problem yourself, and then return to regular processing.

`broken-pipe-error` (– *n*) gforth-0.6

the error number for a broken pipe

6.24.9 CSV Reader

Comma-separated values (CSV) are a popular text format to interchange data. Gforth provides words for reading CSV files (with all features, including newlines in quoted strings).

`read-csv` (*c-addr u xt* –) gforth-experimental

Read the CSV file with the name given by *c-addr u* and execute *xt* for every field found. *Xt* `( c-addr2 u2 field line -- )` is called once for each field; *c-addr2 u2* is the decoded field content, *field* is the field number (starting with 0), and *line* is the line number (starting with 1).

`csv-separator` (– *c*) gforth-experimental

CSV field separator (default is ‘,’, hence the name `"comma-separated"`); this is a value and can be changed with `to csv-separator`.

`csv-quote` (– *c*) gforth-experimental

CSV quote character (default is `"`); this is a value and can be changed with `to csv-quote`.

`.quoted-csv` (*c-addr u* –) gforth-experimental “dot-quoted-csv”

print a field in CSV format, i.e., with enough quotes that `read-csv` will produce *c-addr u* when encountering the output of `.quoted-csv`.

6.25 OS command line arguments

The usual way to pass arguments to Gforth programs on the command line is via the `-e` option, e.g.

```
gforth -e "123 456" foo.fs -e bye
```

However, you may want to interpret the command-line arguments directly. In that case, you can access the (image-specific) command-line arguments through `next-arg`:

`next-arg` (– *addr u*) gforth-0.7

get the next argument from the OS command line, consuming it; if there is no argument left, return `0 0`.

Here's an example program `echo.fs` for `next-arg`:

```
: echo ( -- )
    begin
        next-arg 2dup 0 0 d<> while
            type space
    repeat
    2drop ;

echo cr bye
```

This can be invoked with

```
gforth echo.fs hello world
```

and it will print

```
hello world
```

The next lower level of dealing with the OS command line are the following words:

`arg` (*u* – *addr count*) gforth-0.2

Return the string for the *u*th command-line argument; returns `0 0` if the access is beyond the last argument. `0 arg` is the program name with which you started Gforth. The next unprocessed argument is always `1 arg`, the one after that is `2 arg` etc. All arguments already processed by the system are deleted. After you have processed an argument, you can delete it with `shift-args`.

`shift-args` (–) gforth-0.7

`1 arg` is deleted, shifting all following OS command line parameters to the left by 1, and reducing `argc @`. This word can change `argv @`.

Finally, at the lowest level Gforth provides the following words:

`argc` (– *addr*) gforth-0.2

`Variable` – the number of command-line arguments (including the command name). Changed by `next-arg` and `shift-args`.

`argv` (– *addr*) gforth-0.2

`Variable` – a pointer to a vector of pointers to the command-line arguments (including the command-name). Each argument is represented as a C-style zero-terminated string. Changed by `next-arg` and `shift-args`.

6.26 Locals

Local variables can make Forth programming more enjoyable and Forth programs easier to read.

Gforth implements an extended version of the standard Forth locals.

6.26.1 Gforth locals

Gforth locals are a superset of standard locals. Locals can be defined with

```
{: local1 local2 ... -- comment :}
```

or

```
{: local1 local2 ... :}
```

or

```
{: local1 local2 ... | ulocal0 ulocal1 -- comment :}
```

E.g.,

```
: max {: n1 n2 -- n3 :}
 n1 n2 > if
   n1
 else
   n2
 endif ;
```

The locals definition `{: n1 n2 -- n3 :}` consumes two cells at run-time, and, as one would expect, puts the top-of-stack in `n2` and the second stack item in `n1`. The part behind the `--` is just a comment. The occurences of the locals `n1` and `n2` push the respective value on the stack.

The similarity of locals definitions with stack comments is intended. A locals definition often replaces the stack comment of a word. The order of the locals corresponds to the order in a stack comment and everything after the `--` is really a comment.

The name of the local may be preceded by a type specifier, e.g., `F:` for a floating point value:

```
: CX* {: F: Ar F: Ai F: Br F: Bi -- Cr Ci :}
\ complex multiplication
 Ar Br f* Ai Bi f* f-
 Ar Bi f* Ai Br f* f+ ;
```

Gforth currently supports cells (`W:`, `W^`), doubles (`D:`, `D^`), floats (`F:`, `F^`), characters (`C:`, `C^`), and xts (`xt:`) in several flavours:

value-flavoured
: (see Section 6.11.4 [Values], page 121) You define a value-flavoured local with no type specifier (equivalent to `W:`), with `W:`, `D:`, `F:` or `C:`. When later using the name of a value-flavoured local on its own, it pushes the value of the local. You can also change a value-flavoured local `x` with `TO x`, `->x`, `+TO x` or `+>x`. Also, if you put `addressable:` in front of the locals definition, you can get the address of `x` with `ADDR x` or `'>x` (see Section 6.26.1.3 [How long do locals live?], page 223).

variable-flavoured

(see Section 6.11.2 [Variables], page 120) A variable-flavoured local (defined with `W^`, `D^`, `F^`, `C^`) produces its address (see Section 6.26.1.3 [How long do locals live?], page 223). E.g., the standard word `emit` can be defined in terms of `type` like this:

```
: emit {: C^ char* -- :}
    char* 1 type ;
```

In addition, if you define a local whose name ends in a `[`, used only without type specifier, the result is a variable-flavoured local, with the size given between the local name and `]`. The size is given by a piece of Forth code that is interpreted and must have the stack effect `( -- u )`, where u is the size of the local in bytes. If such a local occurs in front of `|`, it is initialized by copying u bytes from an address given on the stack. If such a local occurs behind `|`, it is uninitialized. The square bracket `[` is part of the defined name. E.g., the `emit` example above can also be coded as:

```
: emit ( c -- )
    {: | char[ 1 chars ] :}
    char[ c! char[ 1 type ;
```

defer-flavoured

(see Section 6.11.11 [Deferred Words], page 138) You define a defer-flavoured local `x` with `XT: x`, and the stack item that you store there (e.g., when the local is initialized from the stack at its definition) must be an execution token. When you later use `x` on its own, the execution token stored in the local is `execute`d. You can get the xt out of `x` with `ACTION-OF x` or `@>x` and change the execution token stored in `x` with `IS x` or `=>x`. If you defined `x` with `addressable: xt: x`, you can use `addr x` or `'>x` to get the address where the xt is stored (see Section 6.26.1.3 [How long do locals live?], page 223).

E.g., the standard word `execute` can be defined with a defer-flavoured local like this:

```
: execute {: xt: x -- :}
  x ;
```

A local without type specifier is a `W:` local, unless its name ends in `[`; in that case it's a variable-flavoured local. You can also leave away the `w:` if you use `addressable:`.

All flavours of locals are initialized with values from the data or (for FP locals) FP stack, with the exception being locals defined behind `|`: Gforth initializes the locals behind `|` to 0 or `0e`, with the uninitialized `[`-ending exception noted above; some Forth systems leave all locals behind `|` uninitialized.

Gforth allows defining locals everywhere in a colon definition, but as a consequence, some locals are not visible (and live) everywhere in the colon definition (see Section 6.26.1.2 [Where are locals visible by name?], page 220).

6.26.1.1 Locals definitions words

This section documents the words used for defining locals. Note that the run-times for the words (like `W:`) that define a local are performed from the rightmost defined local to the leftmost defined local, such that the rightmost local gets the top of stack.

`{:` (– *hmaddr u wid state 0*) local-ext "open-brace-colon"

Start locals definitions.

`--` (*hmaddr u wid 0 ... –*) local-ext "dash-dash"

During a locals definitions with `{:` everything from `--` to `:}` is ignored. This is typically used when you want to make a locals definition serve double duty as a stack effect description.

`|` (–) local-ext "bar"

Locals defined behind `|` are not initialized from the stack; so the run-time stack effect of the locals definitions after `|` is `( -- )`.

`:}` (*hmaddr u wid 0 xt1 ... xtn –*) gforth-1.0 "colon-close-brace"

Ends locals definitions.

`{` (– *hmaddr u wid 0*) gforth-0.2 "open-brace"

Start locals definitions. The Forth-2012 standard name for this word is `{:`.

`}` (*hmaddr u wid state 0 xt1 ... xtn –*) gforth-0.2 "close-brace"

Ends locals definitions. The Forth-2012 standard name for this word is `:}`.

`locals|` (*...* `"`*name ...*`"` –) local-ext "locals-bar"

Don't use '`locals| this read can't you|`'! Use `{: you can read this :}` instead.! A portable and free `{:` implementation can be found in `compat/xlocals.fs`.

`W:` (*compilation* `"`*name*`"` – *a-addr xt; run-time x –*) gforth-0.2 "w-colon"

Define local *name* with the initial value *x*.
name execution: *(– x1)* push the current value of *name*.
`to` ***name*** run-time: *(x2 –)* change the value of *name* to *x2*.
`+to` ***name*** run-time: *(n|u –)* increment the value of *name* by *n|u*.

`W^` (*compilation* `"`*name*`"` – *a-addr xt; run-time x –*) gforth-0.2 "w-caret"

Define local *name*, reserve a cell at *a-addr*, and store *x* there.
name execution: *(– a-addr)*.

`D:` (*compilation* `"`*name*`"` – *a-addr xt; run-time x1 x2 –*) gforth-0.2 "d-colon"

Define local *name* with the initial value *x1 x2*.
name execution: *(– x3 x4)* push the current value of *name*.
`to` ***name*** run-time: *(x5 x6 –)* change the value of *name* to *x5 x6*.
`+to` ***name*** run-time: *(d|ud –)* increment the value of *name* by *d|ud*.

`D^` (*compilation* `"`*name*`"` – *a-addr xt; run-time x1 x2 –*) gforth-0.2 "d-caret"

Define local *name*, reserve two cells at *a-addr*, and store *x1 x2* there.
name execution: *(– a-addr)*.

`C:` (*compilation* `"`*name*`"` – *a-addr xt; run-time c –*) gforth-0.2 "c-colon"

Define local *name* with the initial value *c*.
name execution: *(– c1)* push the current value of *name*.
`to` ***name*** run-time: *(c2 –)* change the value of *name* to *c2*.
`+to` ***name*** run-time: *(n|u –)* increment the value of *name* by *n|u*.

`C^` (*compilation* `"`*name*`"` – *a-addr xt; run-time c –*) gforth-0.2 "c-caret"

Define local *name*, reserve a cell at *c-addr*, and store *c* there.
name execution: *(– c-addr)*.

`F:` (*compilation* "*name*" – *a-addr xt; run-time r* –) gforth-0.2 “f-colon”

Define local *name* with the initial value *r*.
name execution: *(– r1)* push the current value of *name*.
`to` *name* run-time: *(r2 –)* change the value of *name* to *r2*.
`+to` *name* run-time: *(r3 –)* increment the value of *name* by *r3*.

`F^` (*compilation* "*name*" – *a-addr xt; run-time r* –) gforth-0.2 “f-caret”

Define local *name*, reserve a float at *f-addr*, and store *r* there.
name execution: *(– f-addr)*.

`Z:` (*compilation* "*name*" – *a-addr xt; run-time r1 r2* –) gforth-1.0 “z-colon”

Define local *name* with the initial value *r1 r2*.
name execution: *(– r3 r4)* push the current value of *name*.
`to` *name* run-time: *(r5 r6 –)* change the value of *name* to *r5 r6*.
`+to` *name* run-time: *(r7 r8 –)* increment the value of *name* by *r7*+*r8*i.

`Z^` (*compilation* "*name*" – *a-addr xt; run-time r1 r2* –) gforth-1.0 “z-caret”

Define local *name*, reserve two floats at *f-addr*, and store *r1 r2* there.
name execution: *(– f-addr)*.

`XT:` (*compilation* "*name*" – *a-addr xt; run-time xt1* –) gforth-1.0 “x-t-colon”

Define local *name*; set *name* to execute *xt1*.
name execution: execute the xt that *name* has most recently been set to execute.
`Is` *name* run-time: *(xt2 –)* set *name* to execute *xt2*.
`Action-of` *name* run-time: *(– xt3) xt3* is the xt that *name* has most recently been set to execute.

Apart from `{:` and `{`, none of these words are normally in the search order (they are in the vocabulary `locals-types`); on other Forth systems, they may not even be words (the standard documents `|`, `--` and `:}` under `{:`, not as separate word, and the other words are non-standard).

6.26.1.2 Where are locals visible by name?

Basically, the answer is that locals are visible where you would expect it in block-structured languages, and sometimes a little longer. If you want to restrict the scope of a local, enclose its definition in `SCOPE`...`ENDSCOPE`.

`scope` (*compilation – scope ; run-time* –) gforth-0.2

`endscope` (*compilation scope – ; run-time* –) gforth-0.2

These words behave like control structure words, so you can use them with `CS-PICK` and `CS-ROLL` to restrict the scope in arbitrary ways.

If you want a more exact answer to the visibility question, here's the basic principle: A local is visible in all places that can only be reached through the definition of the local[23]

[23] In compiler construction terminology, all places dominated by the definition of the local.

. In other words, it is not visible in places that can be reached without going through the definition of the local. E.g., locals defined in IF...THEN are visible until the THEN, locals defined in BEGIN...UNTIL are visible after the UNTIL (until, e.g., a subsequent ENDSCOPE).

The reasoning behind this solution is: We want to have the locals visible as long as it is meaningful. The user can always make the visibility shorter by using explicit scoping. In a place that can only be reached through the definition of a local, the meaning of a local name is clear. In other places it is not: How is the local initialized at the control flow path that does not contain the definition? Which local is meant, if the same name is defined twice in two independent control flow paths?

This should be enough detail for nearly all users, so you can skip the rest of this section. If you really must know all the gory details and options, read on.

In order to implement this rule, the compiler has to know which places are unreachable. It knows this automatically after AHEAD, AGAIN, EXIT and LEAVE; in other cases (e.g., after most THROWs), you can use the word UNREACHABLE to tell the compiler that the control flow never reaches that place. If UNREACHABLE is not used where it could, the only consequence is that the visibility of some locals is more limited than the rule above says. If UNREACHABLE is used where it should not (i.e., if you lie to the compiler), you can produce code whose behaviour is best determined by looking at the implementation (which may change).

UNREACHABLE (–) gforth-0.2

This word promises to the compiler that it can never be executed. One could put it, e.g., behind a -13 throw.

Another problem with this rule is that at BEGIN, the compiler does not know which locals will be visible on the incoming back-edge. All problems discussed in the following are due to this ignorance of the compiler (we discuss the problems using BEGIN loops as examples; the discussion also applies to ?DO and other loops). Perhaps the most insidious example is:

```
AHEAD
BEGIN
  x
[ 1 CS-ROLL ] THEN
  {: x :}
  ...
UNTIL
```

This should be legal according to the visibility rule. The use of x can only be reached through the definition; but the use appears textually before the definition.

From this example it is clear that the visibility rules cannot be fully implemented without major headaches in a one-pass compiler like Gforth's. Our implementation treats common cases as advertised and the exceptions are treated in a safe way: The compiler makes a reasonable guess about the locals visible after a BEGIN; if it is too pessimistic, the user will get a spurious error about the local not being defined; if the compiler is too optimistic, it will notice this later and issue a warning. In the case above the compiler would complain about x being undefined at its use. You can see from the obscure examples in this section that it takes quite unusual control structures to get the compiler into trouble, and even then it will often do fine.

If the BEGIN is reachable from above, the most optimistic guess is that all locals visible before the BEGIN will also be visible after the BEGIN. This guess is valid for all loops that

are entered only through the BEGIN, in particular, for normal BEGIN...WHILE...REPEAT and BEGIN...UNTIL loops and it is implemented in our compiler. When the branch to the BEGIN is finally generated by AGAIN or UNTIL, the compiler checks the guess and warns the user if it was too optimistic:

```
IF
  {: x :}
BEGIN
  \ x ?
[ 1 cs-roll ] THEN
  ...
UNTIL
```

Here, x lives only until the BEGIN, but the compiler optimistically assumes that it lives until the THEN. It notices this difference when it compiles the UNTIL and issues a warning. The user can avoid the warning, and make sure that x is not used in the wrong area by using explicit scoping:

```
IF
  SCOPE
  {: x :}
  ENDSCOPE
BEGIN
[ 1 cs-roll ] THEN
  ...
UNTIL
```

Since the guess is optimistic, there will be no spurious error messages about undefined locals.

If the BEGIN is not reachable from above (e.g., after AHEAD or EXIT), the compiler cannot even make an optimistic guess, as the locals visible after the BEGIN may be defined later.

It pessimistically assumes that all locals are visible that were visible at the latest place outside any control structure (i.e., where nothing is on the control-flow stack). This means that in:

```
: foo
  IF {: z :} THEN
  {: x :}
  AHEAD
    BEGIN
      ( * )
    [ 1 CS-ROLL ] THEN
    {: y :}
    ...
  UNTIL ;
```

At the place marked with (*), x is visible, but y is not (although, according to the reachability rule it should); z is not and should not be visible there.

However, you can use ASSUME-LIVE to make the compiler assume that the same locals are visible at the BEGIN as at the point where the top control-flow stack item was created.

ASSUME-LIVE (*compilation orig – orig*) gforth-0.2

Used immediatly before a BEGIN that is not reachable from above. Causes the BEGIN to assume that the same locals are live as right before the control flow word (typically ahead) that pushed the *orig*.

E.g.,

```
IF
  {: x :}
  AHEAD
    ASSUME-LIVE
    BEGIN
      x
    [ 1 CS-ROLL ] THEN
    ...
  UNTIL
THEN
```

Here x would not be visible at the use of x, because its definition is inside a control structure, but by using ASSUME-LIVE the programmer tells the compiler that the locals visible at the AHEAD should be visible at the BEGIN.

Other cases where the locals are defined before the BEGIN can be handled by inserting an appropriate CS-ROLL before the ASSUME-LIVE (and changing the control-flow stack manipulation behind the ASSUME-LIVE).

Cases where locals are defined in a BEGIN loop and should be visible in that loop before the definition can only be handled by rearranging the loop. E.g., the "most insidious" example above can be arranged into:

```
BEGIN
  {: x :}
  ... 0=
WHILE
  x
REPEAT
```

The ideas in this section have also been published in M. Anton Ertl, *Automatic Scoping of Local Variables (*https://www.complang.tuwien.ac.at/papers/ertl94l.ps.gz*)*, EuroForth '94.

6.26.1.3 How long do locals live?

Each local lives only as long as it is visible; afterwards its address is invalid, and programs that access it afterwards are erroneous; it's a good idea to use non-addressable value-flavoured or defer-flavoured locals to avoid such errors.

6.26.1.4 POSTPONEing locals

In Gforth you can postpone a local (or use a local inside]]...[[, see Section 6.16.2 [Macros], page 162). Given the normal lifetime of locals (see Section 6.26.1.3 [How long do locals live?], page 223), what does this mean? Consider:

```
: foo {: W: n D: d F: r C: c XT: xt -- :}
  POSTPONE n
  POSTPONE d
```

```
    POSTPONE r
    POSTPONE c
    POSTPONE xt
    0 ->n #0. ->d 0e ->r 0 ->c `abort =>xt
  ; immediate
```

This is equivalent to

```
: foo {: W: n D: d F: r C: c XT: xt -- :}
  n POSTPONE literal
  d POSTPONE 2literal
  r POSTPONE fliteral
  c POSTPONE literal
  ACTION-OF xt compile,
  0 ->n #0. ->d 0e -> r 0 ->c `abort =>xt
; immediate
```

I.e., when applied to locals, `postpone` compiles the value that the local has at the point of the `postpone`, not any later value of the local. This solves the problem that the local does not live indefinitely, and it also results in efficient code. But in general it does not produce the value that an infinite-lifetime local would have at the place where the local is compiled into by the `postpone` (what a Scheme programmer would expect). However, as long as you do not use `TO`, `+TO`, `IS` or `ADDR` on the same local as `POSTPONE`, the latter produces the result that a Scheme programmer would expect.

Concerning using locals without `TO` and friends, see Section 6.26.1.5 [Locals programming style], page 224.

Using `postpone` with variable-flavoured locals produces an error during compilation.

6.26.1.5 Locals programming style

The freedom to define locals anywhere has the potential to change programming styles dramatically. In particular, the need to use the return stack for intermediate storage vanishes. Moreover, all stack manipulations (except `PICK`s and `ROLL`s with run-time determined arguments) can be eliminated: If the stack items are in the wrong order, just write a locals definition for all of them; then write the items in the order you want.

This seems a little far-fetched and eliminating stack manipulations is unlikely to become a conscious programming objective. Still, the number of stack manipulations will be reduced dramatically if local variables are used liberally (e.g., compare `max` (see Section 6.26.1 [Gforth locals], page 217) with a traditional implementation of `max`).

This shows one potential benefit of locals: making Forth programs more readable. Of course, this benefit will only be realized if the programmers continue to honour the principle of factoring instead of using the added latitude to make the words longer.

Using `TO` can and should be avoided. Without `TO` and friends, every value-flavoured local has only a single assignment and many advantages of functional languages apply to Forth. I.e., programs are easier to analyse, to optimize and to read: It is clear from the definition what the local stands for, it does not turn into something different later.

E.g., a definition using `TO` might look like this:

```
: strcmp {: addr1 u1 addr2 u2 -- n :}
```

```
u1 u2 min 0 ?do
  addr1 c@ addr2 c@ - ?dup-if
    unloop exit then
  1 +TO addr1
  1 +TO addr2
loop
u1 u2 - ;
```

Here, `+TO` is used to update `addr1` and `addr2` at every loop iteration. `strcmp` is a typical example of the readability problems of using `TO` and friends: When you read the use of, e.g., `addr1` near the start of the loop, you think that `addr1` refers to the start of the string. Only near the end of the loop you realize that it is something else.

This can be avoided by defining two locals at the start of the loop that are initialized with the right value for the current iteration.

```
: strcmp {: addr1 u1 addr2 u2 -- n :}
 addr1 addr2
 u1 u2 min 0 ?do {: s1 s2 :}
   s1 c@ s2 c@ - ?dup-if
     unloop exit then
   s1 char+ s2 char+
 loop
 2drop
 u1 u2 - ;
```

Here it is clear from the start that `s1` has a different value in every loop iteration.

6.26.1.6 Locals implementation

Gforth uses a separate locals stack. The most compelling reason for this is that the return stack is not float-aligned; using a separate stack also eliminates the restrictions that other systems have because they use the return stack as locals stack (and that standard programs therefore also have). Like the other stacks, the locals stack grows toward lower addresses. A few primitives allow an efficient implementation; you should not use them directly, but they appear in the output of `see`, so they are documented here:

`@localn` (*noffset* – *w*) gforth-internal “fetch-local-n”

w is the content of the local at *noffset* from the locals pointer.

`f@localn` (*noffset* – *r*) gforth-internal “f-fetch-local-n”

r is the content of the *F:* local at *noffset* from the locals pointer.

`!localn` (*w noffset* –) gforth-internal “store-local-n”

Store *w* to the local at *noffset* from the locals pointer.

`lp+n` (*noffset* – *c-addr*) gforth-internal “lp-plus-n”

c-addr is the address of the local at *noffset* from the locals pointer.

`lp+!` (*noffset* –) gforth-internal “lp-plus-store”

Add *noffset* to the locals pointer; a negative *noffset* allocates, a positive *noffset* frees.

`>l` (*w* –) gforth-internal “to-l”

Push *w* on the locals stack.

`f>l` (*r* –) gforth-internal "f-to-l"

Push *r* on the locals stack.

See also `lp@`, `lp!` (see Section 6.7.5 [Stack pointer manipulation], page 75).

In addition to these primitives, some specializations of these primitives for commonly occurring inline arguments are provided for efficiency reasons, e.g., `@local0` as specialization of `0 @localn`.

Combinations of conditional branches and `lp+!#` like `?branch-lp+!#` (the locals pointer is only changed if the branch is taken) are provided for efficiency and correctness in loops.

Local variable names are stored in their own section `locals-headers` by `create-local`, the central factor in all the words that define a local. Locals names of a definition are deleted when the definition is complete.

A special feature of Gforth's dictionary is used to implement the definition of locals without type specifiers: every word list (aka vocabulary) has its own methods for searching etc. (see Section 6.19 [Word Lists], page 182). For the present purpose we defined a word list with a special search method: When it is searched for a word, it actually creates that word using `W:`. `{:` changes the search order to first search the word list containing `:}`, `W:` etc., and then the word list for defining locals without type specifiers. This implementation was designed before Gforth acquired recognizers; for a reimplementation we would use recognizers.

The lifetime rules support a stack discipline within a colon definition: The lifetime of a local is either nested with other locals lifetimes or it does not overlap them.

At `BEGIN`, `IF`, and `AHEAD` no code for locals stack pointer manipulation is generated. Between control structure words locals definitions can push locals onto the locals stack. `AGAIN` is the simplest of the other three control flow words. It has to restore the locals stack depth of the corresponding `BEGIN` before branching. The code looks like this:

current-locals-size dest-locals-size - `lp+!`
`branch` <begin>

`UNTIL` is a little more complicated: If it branches back, it must adjust the stack just like `AGAIN`. But if it falls through, the locals stack must not be changed. The compiler generates the following code:

`?branch-lp+!#` <begin> current-locals-size − dest-locals-size

The locals stack pointer is only adjusted if the branch is taken.

`THEN` can produce somewhat inefficient code:

current-locals-size dest-locals-size - `lp+!`
<orig target>:
orig-locals-size new-locals-size - `lp+!`

The second `lp+!#` adjusts the locals stack pointer from the level at the *orig* point to the level after the `THEN`. The first `lp+!#` adjusts the locals stack pointer from the current level to the level at the orig point, so the complete effect is an adjustment from the current level to the right level after the `THEN`. This effect happens e.g., if there is an `ELSE` and the code before the `ELSE` defines locals, and they have a different size than those after the `ELSE`. In general we recommend not to work around this shortcoming (except in performance-critical code). We intend to eliminate this shortcoming at some point.

In a conventional Forth implementation a dest control-flow stack entry is just the target address and an orig entry is just the address to be patched. Our locals implementation adds a word list to every orig or dest item. It is the list of locals visible (or assumed visible) at the point described by the entry. Our implementation also adds a tag to identify the kind of entry, in particular to differentiate between live and dead (reachable and unreachable) orig entries.

A few unusual operations have to be performed on locals word lists:

`common-list` (*list1 list2 – list3*) gforth-internal

`sub-list?` (*list1 list2 – f*) gforth-internal "sub-list-question"

`list-size` (*list – u*) gforth-internal

Several features of our locals word list implementation make these operations easy to implement: The locals word lists are organised as linked lists; the tails of these lists are shared, if the lists contain some of the same locals; and the address of a name is greater than the address of the names behind it in the list.

Another important implementation detail is the variable `dead-code`. It is used by `BEGIN` and `THEN` to determine if they can be reached directly or only through the branch that they resolve. `dead-code` is set by `UNREACHABLE`, `AHEAD`, `EXIT` etc., and cleared at the start of a colon definition, by `BEGIN` and usually by `THEN`.

Counted loops are similar to other loops in most respects, but `LEAVE` requires special attention: It performs basically the same service as `AHEAD`, but it does not create a control-flow stack entry. Therefore the information has to be stored elsewhere; traditionally, the information was stored in the target fields of the branches created by the `LEAVE`s, by organizing these fields into a linked list. Unfortunately, this clever trick does not provide enough space for storing our extended control flow information. Therefore, we introduce another stack, the leave stack. It contains the control-flow stack entries for all unresolved `LEAVE`s.

Local names are kept until the end of the colon definition, even if they are no longer visible in any control-flow path. In a few cases this may lead to increased space needs for the locals name area, but usually less than reclaiming this space would cost in code size.

6.26.2 Standard Forth locals

The Forth-2012 standard defines a syntax for locals that is a restricted version of Gforth's locals:

- Locals can only be cell-sized values (no type specifiers are allowed).
- Locals can be defined only outside control structures.
- Only one locals definition is allowed per colon definition.
- Locals can interfere with explicit usage of the return stack. For the exact (and long) rules, see the standard. If you don't use return stack accessing words in a definition using locals, you will be all right. The purpose of this rule is to make locals implementation on the return stack easier.
- The whole locals definition must be in one line.

The Standard Forth locals wordset itself consists of two words: `{:` and:

`(local)` (*addr u –*) local "paren-local-paren"

The Forth-2012 locals extension wordset also defines a syntax using `locals|`, but it is so awful that we strongly recommend not to use it, and the standardization committee standardized another, better syntax (the one using `{:`). We have implemented this syntax only to make porting to Gforth easy. The problem with this syntax is that the locals are defined in an order reversed with respect to the standard stack comment notation, making programs harder to read, and easier to misread and miswrite.

6.27 Object-oriented Forth

Gforth comes with the following packages for object-oriented programming: `objects.fs`, `oof.fs`, `mini-oof.fs`, and `mini-oof2.fs`; none of them is preloaded, so you have to `require` them before use. The most important differences between these packages (and others) are discussed in [Comparison with other object models], page 246. Mini-OOF2 is Gforth-specific, the other three are are written in Standard Forth and can be used with any other standard-compliant Forth system.

6.27.1 Why object-oriented programming?

Often we have to deal with several data structures (*objects*), that have to be treated similarly in some respects, but differently in others. Graphical objects are the textbook example: circles, triangles, dinosaurs, icons, and others, and we may want to add more during program development. We want to apply some operations to any graphical object, e.g., `draw` for displaying it on the screen. However, `draw` has to do something different for every kind of object.

We could implement `draw` as a big `CASE` control structure that executes the appropriate code depending on the kind of object to be drawn. This would be not be very elegant, and, moreover, we would have to change `draw` every time we add a new kind of graphical object (say, a spaceship).

What we would rather do is: When defining spaceships, we would tell the system: "Here's how you `draw` a spaceship; you figure out the rest".

This is the problem that all systems solve that (rightfully) call themselves object-oriented; the object-oriented packages presented here solve this problem (and not much else).

6.27.2 Object-Oriented Terminology

This section is mainly for reference, so you don't have to understand all of it right away. The terminology is mainly Smalltalk-inspired. In short:

class a data structure definition with some extras.

object an instance of the data structure described by the class definition.

instance variables
fields of the data structure.

selector (or *method selector*) a word (e.g., `draw`) that performs an operation on a variety of data structures (classes). A selector describes *what* operation to perform. In C++ terminology: a (pure) virtual function.

method the concrete definition that performs the operation described by the selector for a specific class. A method specifies *how* the operation is performed for a specific class.

selector invocation
a call of a selector. One argument of the call (the TOS (top-of-stack)) is used for determining which method is used. In Smalltalk terminology: a message (consisting of the selector and the other arguments) is sent to the object.

receiving object
the object used for determining the method executed by a selector invocation. In the `objects.fs` model, it is the object that is on the TOS when the selector is invoked. (*Receiving* comes from the Smalltalk *message* terminology.)

child class a class that has (*inherits*) all properties (instance variables, selectors, methods) from a *parent class*. In Smalltalk terminology: The subclass inherits from the superclass. In C++ terminology: The derived class inherits from the base class.

6.27.3 The `objects.fs` model

This section describes the `objects.fs` package. This material also has been published in M. Anton Ertl, *Yet Another Forth Objects Package* (`https://www.complang.tuwien.ac.at/forth/objects/objects.html`), Forth Dimensions 19(2), pages 37–43.

This section assumes that you have read Section 6.12 [Structures], page 141.

The techniques on which this model is based have been used to implement the parser generator, Gray, and have also been used in Gforth for implementing the various flavours of word lists (hashed or not, case-sensitive or not, special-purpose word lists for locals etc.).

Marcel Hendrix provided helpful comments on this section.

6.27.3.1 Properties of the `objects.fs` model

- It is straightforward to pass objects on the stack. Passing selectors on the stack is a little less convenient, but possible.
- Objects are just data structures in memory, and are referenced by their address. You can create words for objects with normal defining words like `constant`. Likewise, there is no difference between instance variables that contain objects and those that contain other data.
- Late binding is efficient and easy to use.
- It avoids parsing, and thus avoids problems with state-smartness and reduced extensibility; for convenience there are a few parsing words, but they have non-parsing counterparts. There are also a few defining words that parse. This is hard to avoid, because all standard defining words parse (except `:noname`); however, such words are not as bad as many other parsing words, because they are not state-smart.
- It does not try to incorporate everything. It does a few things and does them well (IMO). In particular, this model was not designed to support information hiding (although it has features that may help); you can use a separate package for achieving this.

- It is layered; you don't have to learn and use all features to use this model. Only a few features are necessary (see Section 6.27.3.2 [Basic Objects Usage], page 230, see Section 6.27.3.3 [The Objects base class], page 231, see Section 6.27.3.4 [Creating objects], page 231.), the others are optional and independent of each other.
- An implementation in Standard Forth is available.

6.27.3.2 Basic `objects.fs` Usage

You can define a class for graphical objects like this:

```
object class \ "object" is the parent class
  selector draw ( x y graphical -- )
end-class graphical
```

This code defines a class `graphical` with an operation `draw`. We can perform the operation `draw` on any `graphical` object, e.g.:

```
100 100 t-rex draw
```

where `t-rex` is a word (say, a constant) that produces a graphical object.

How do we create a graphical object? With the present definitions, we cannot create a useful graphical object. The class `graphical` describes graphical objects in general, but not any concrete graphical object type (C++ users would call it an *abstract class*); e.g., there is no method for the selector `draw` in the class `graphical`.

For concrete graphical objects, we define child classes of the class `graphical`, e.g.:

```
graphical class \ "graphical" is the parent class
  cell% field circle-radius

:noname ( x y circle -- )
  circle-radius @ draw-circle ;
overrides draw

:noname ( n-radius circle -- )
  circle-radius ! ;
overrides construct

end-class circle
```

Here we define a class `circle` as a child of `graphical`, with field `circle-radius` (which behaves just like a field (see Section 6.12 [Structures], page 141); it defines (using `overrides`) new methods for the selectors `draw` and `construct` (`construct` is defined in `object`, the parent class of `graphical`).

Now we can create a circle on the heap (i.e., `allocate`d memory) with:

```
50 circle heap-new constant my-circle
```

`heap-new` invokes `construct`, thus initializing the field `circle-radius` with 50. We can draw this new circle at (100,100) with:

```
100 100 my-circle draw
```

Note: You can only invoke a selector if the object on the TOS (the receiving object) belongs to the class where the selector was defined or one of its descendents; e.g., you can invoke `draw` only for objects belonging to `graphical` or its descendents (e.g., `circle`).

Immediately before `end-class`, the search order has to be the same as immediately after `class`.

6.27.3.3 The `object.fs` base class

When you define a class, you have to specify a parent class. So how do you start defining classes? There is one class available from the start: `object`. It is ancestor for all classes and so is the only class that has no parent. It has two selectors: `construct` and `print`.

6.27.3.4 Creating objects

You can create and initialize an object of a class on the heap with `heap-new` (... class – object) and in the dictionary (allocation with `allot`) with `dict-new` (... class – object). Both words invoke `construct`, which consumes the stack items indicated by `"..."` above.

If you want to allocate memory for an object yourself, you can get its alignment and size with `class-inst-size 2@` (class – align size). Once you have memory for an object, you can initialize it with `init-object` (... class object –); `construct` does only a part of the necessary work.

6.27.3.5 Object-Oriented Programming Style

This section is not exhaustive.

In general, it is a good idea to ensure that all methods for the same selector have the same stack effect: when you invoke a selector, you often have no idea which method will be invoked, so, unless all methods have the same stack effect, you will not know the stack effect of the selector invocation.

One exception to this rule is methods for the selector `construct`. We know which method is invoked, because we specify the class to be constructed at the same place. Actually, I defined `construct` as a selector only to give the users a convenient way to specify initialization. The way it is used, a mechanism different from selector invocation would be more natural (but probably would take more code and more space to explain).

6.27.3.6 Class Binding

Normal selector invocations determine the method at run-time depending on the class of the receiving object. This run-time selection is called *late binding*.

Sometimes it's preferable to invoke a different method. For example, you might want to use the simple method for `print`ing `object`s instead of the possibly long-winded `print` method of the receiver class. You can achieve this by replacing the invocation of `print` with:

```
[bind] object print
```

in compiled code or:

```
bind object print
```

in interpreted code. Alternatively, you can define the method with a name (e.g., `print-object`), and then invoke it through the name. Class binding is just a (often more convenient) way to achieve the same effect; it avoids name clutter and allows you to invoke methods directly without naming them first.

A frequent use of class binding is this: When we define a method for a selector, we often want the method to do what the selector does in the parent class, and a little more. There

is a special word for this purpose: `[parent]`; `[parent]` *`selector`* is equivalent to `[bind]` *`parent selector`*, where *`parent`* is the parent class of the current class. E.g., a method definition might look like:

```
:noname
  dup [parent] foo \ do parent's foo on the receiving object
  ... \ do some more
; overrides foo
```

In *Object-oriented programming in ANS Forth* (Forth Dimensions, March 1997), Andrew McKewan presents class binding as an optimization technique. I recommend not using it for this purpose unless you are in an emergency. Late binding is pretty fast with this model anyway, so the benefit of using class binding is small; the cost of using class binding where it is not appropriate is reduced maintainability.

While we are at programming style questions: You should bind selectors only to ancestor classes of the receiving object. E.g., say, you know that the receiving object is of class `foo` or its descendents; then you should bind only to `foo` and its ancestors.

6.27.3.7 Method conveniences

In a method you usually access the receiving object pretty often. If you define the method as a plain colon definition (e.g., with `:noname`), you may have to do a lot of stack gymnastics. To avoid this, you can define the method with `m: ... ;m`. E.g., you could define the method for `draw`ing a `circle` with

```
m: ( x y circle -- )
  ( x y ) this circle-radius @ draw-circle ;m
```

When this method is executed, the receiver object is removed from the stack; you can access it with `this` (admittedly, in this example the use of `m: ... ;m` offers no advantage). Note that I specify the stack effect for the whole method (i.e. including the receiver object), not just for the code between `m:` and `;m`. You cannot use `exit` in `m:...;m`; instead, use `exitm`.[24]

You will frequently use sequences of the form `this` *`field`* (in the example above: `this circle-radius`). If you use the field only in this way, you can define it with `inst-var` and eliminate the `this` before the field name. E.g., the `circle` class above could also be defined with:

```
graphical class
  cell% inst-var radius

m: ( x y circle -- )
  radius @ draw-circle ;m
overrides draw

m: ( n-radius circle -- )
  radius ! ;m
overrides construct
```

[24] Moreover, for any word that calls `catch` and was defined before loading `objects.fs`, you have to redefine it like I redefined `catch`: `: catch this >r catch r> to-this ;`

```
end-class circle
```

`radius` can only be used in `circle` and its descendent classes and inside `m:...;m`.

You can also define fields with `inst-value`, which is to `inst-var` what `value` is to `variable`. You can change the value of such a field with `[to-inst]`. E.g., we could also define the class `circle` like this:

```
graphical class
  inst-value radius

m: ( x y circle -- )
  radius draw-circle ;m
overrides draw

m: ( n-radius circle -- )
  [to-inst] radius ;m
overrides construct

end-class circle
```

6.27.3.8 Classes and Scoping

Inheritance is frequent, unlike structure extension. This exacerbates the problem with the field name convention (see Section 6.12.1 [Standard Structures], page 142): One always has to remember in which class the field was originally defined; changing a part of the class structure would require changes for renaming in otherwise unaffected code.

To solve this problem, I added a scoping mechanism (which was not in my original charter): A field defined with `inst-var` (or `inst-value`) is visible only in the class where it is defined and in the descendent classes of this class. Using such fields only makes sense in `m:`-defined methods in these classes anyway.

This scoping mechanism allows us to use the unadorned field name, because name clashes with unrelated words become much less likely.

Once we have this mechanism, we can also use it for controlling the visibility of other words: All words defined after `protected` are visible only in the current class and its descendents. `public` restores the compilation (i.e. `current`) word list that was in effect before. If you have several `protected`s without an intervening `public` or `set-current`, `public` will restore the compilation word list in effect before the first of these `protected`s.

6.27.3.9 Dividing classes

You may want to do the definition of methods separate from the definition of the class, its selectors, fields, and instance variables, i.e., separate the implementation from the definition. You can do this in the following way:

```
graphical class
  inst-value radius
end-class circle

... \ do some other stuff
```

```
circle methods \ now we are ready

m: ( x y circle -- )
  radius draw-circle ;m
overrides draw

m: ( n-radius circle -- )
  [to-inst] radius ;m
overrides construct

end-methods
```

You can use several `methods`...`end-methods` sections. The only things you can do to the class in these sections are: defining methods, and overriding the class's selectors. You must not define new selectors or fields.

Note that you often have to override a selector before using it. In particular, you usually have to override `construct` with a new method before you can invoke `heap-new` and friends. E.g., you must not create a circle before the `overrides construct` sequence in the example above.

6.27.3.10 Object Interfaces

In this model you can only call selectors defined in the class of the receiving objects or in one of its ancestors. If you call a selector with a receiving object that is not in one of these classes, the result is undefined; if you are lucky, the program crashes immediately.

Now consider the case when you want to have a selector (or several) available in two classes: You would have to add the selector to a common ancestor class, in the worst case to `object`. You may not want to do this, e.g., because someone else is responsible for this ancestor class.

The solution for this problem is interfaces. An interface is a collection of selectors. If a class implements an interface, the selectors become available to the class and its descendents. A class can implement an unlimited number of interfaces. For the problem discussed above, we would define an interface for the selector(s), and both classes would implement the interface.

As an example, consider an interface `storage` for writing objects to disk and getting them back, and a class `foo` that implements it. The code would look like this:

```
interface
  selector write ( file object -- )
  selector read1 ( file object -- )
end-interface storage

bar class
  storage implementation

... overrides write
... overrides read1
...
```

```
end-class foo
```

(I would add a word `read` *(file – object)* that uses `read1` internally, but that's beyond the point illustrated here.)

Note that you cannot use `protected` in an interface; and of course you cannot define fields.

In the Neon model, all selectors are available for all classes; therefore it does not need interfaces. The price you pay in this model is slower late binding, and therefore, added complexity to avoid late binding.

6.27.3.11 `objects.fs` Implementation

An object is a piece of memory, like one of the data structures described with `struct...end-struct`. It has a field `object-map` that points to the method map for the object's class.

The *method map*[25] is an array that contains the execution tokens (*xt*s) of the methods for the object's class. Each selector contains an offset into a method map.

`selector` is a defining word that uses `CREATE` and `DOES>`. The body of the selector contains the offset; the `DOES>` action for a class selector is, basically:

```
( object addr ) @ over object-map @ + @ execute
```

Since `object-map` is the first field of the object, it does not generate any code. As you can see, calling a selector has a small, constant cost.

A class is basically a `struct` combined with a method map. During the class definition the alignment and size of the class are passed on the stack, just as with `struct`s, so `field` can also be used for defining class fields. However, passing more items on the stack would be inconvenient, so `class` builds a data structure in memory, which is accessed through the variable `current-interface`. After its definition is complete, the class is represented on the stack by a pointer (e.g., as parameter for a child class definition).

A new class starts off with the alignment and size of its parent, and a copy of the parent's method map. Defining new fields extends the size and alignment; likewise, defining new selectors extends the method map. `overrides` just stores a new *xt* in the method map at the offset given by the selector.

Class binding just gets the *xt* at the offset given by the selector from the class's method map and `compile,`s (in the case of `[bind]`) it.

I implemented `this` as a `value`. At the start of an `m:...;m` method the old `this` is stored to the return stack and restored at the end; and the object on the TOS is stored `TO this`. This technique has one disadvantage: If the user does not leave the method via `;m`, but via `throw` or `exit`, `this` is not restored (and `exit` may crash). To deal with the `throw` problem, I have redefined `catch` to save and restore `this`; the same should be done with any word that can catch an exception. As for `exit`, I simply forbid it (as a replacement, there is `exitm`).

`inst-var` is just the same as `field`, with a different `DOES>` action:

```
@ this +
```

Similar for `inst-value`.

[25] This is Self terminology; in C++ terminology: virtual function table.

Each class also has a word list that contains the words defined with `inst-var` and `inst-value`, and its protected words. It also has a pointer to its parent. `class` pushes the word lists of the class and all its ancestors onto the search order stack, and `end-class` drops them.

An interface is like a class without fields, parent and protected words; i.e., it just has a method map. If a class implements an interface, its method map contains a pointer to the method map of the interface. The positive offsets in the map are reserved for class methods, therefore interface map pointers have negative offsets. Interfaces have offsets that are unique throughout the system, unlike class selectors, whose offsets are only unique for the classes where the selector is available (invocable).

This structure means that interface selectors have to perform one indirection more than class selectors to find their method. Their body contains the interface map pointer offset in the class method map, and the method offset in the interface method map. The `does>` action for an interface selector is, basically:

```
( object selector-body )
2dup selector-interface @ ( object selector-body object interface-offset )
swap object-map @ + @ ( object selector-body map )
swap selector-offset @ + @ execute
```

where `object-map` and `selector-offset` are first fields and generate no code.

As a concrete example, consider the following code:

```
interface
  selector if1sel1
  selector if1sel2
end-interface if1

object class
  if1 implementation
  selector cl1sel1
  cell% inst-var cl1iv1

' m1 overrides construct
' m2 overrides if1sel1
' m3 overrides if1sel2
' m4 overrides cl1sel2
end-class cl1

create obj1 object dict-new drop
create obj2 cl1    dict-new drop
```

The data structure created by this code (including the data structure for `object`) is shown in the figure (`objects-implementation.eps`), assuming a cell size of 4.

6.27.3.12 `objects.fs` Glossary

`bind` (*... "class" "selector" – ...*) objects

Execute the method for *selector* in *class*.

`<bind>` (*class selector-xt – xt*) objects "less-bind-to"

xt is the method for the selector *selector-xt* in *class*.

`bind'` (`"`*class*`" "`*selector*`"` – *xt*) objects "bind-tick"

xt is the method for *selector* in *class*.

`[bind]` (*compile-time:* `"`*class*`" "`*selector*`"` – *; run-time: ... object* – *...*) objects "left-bracket-bind-right-bracket"

Compile the method for *selector* in *class*.

`class` (*parent-class* – *align offset*) objects

Start a new class definition as a child of *parent-class*. *align offset* are for use by *field* etc.

`class->map` (*class* – *map*) objects "class–to-map"

map is the pointer to *class*'s method map; it points to the place in the map to which the selector offsets refer (i.e., where *object-map*s point to).

`class-inst-size` (*class* – *addr*) objects

Give the size specification for an instance (i.e. an object) of *class*; used as `class-inst-size 2@ ( class -- align size )`.

`class-override!` (*xt sel-xt class-map* –) objects "class-override-store"

xt is the new method for the selector *sel-xt* in *class-map*.

`class-previous` (*class* –) objects

Drop *class*'s wordlists from the search order. No checking is made whether *class*'s wordlists are actually on the search order.

`class>order` (*class* –) objects "class-to-order"

Add *class*'s wordlists to the head of the search-order.

`construct` (*... object* –) objects

Initialize the data fields of *object*. The method for the class *object* just does nothing: `( object -- )`.

`current'` (`"`*selector*`"` – *xt*) objects "current-tick"

xt is the method for *selector* in the current class.

`[current]` (*compile-time:* `"`*selector*`"` – *; run-time: ... object* – *...*) objects "left-bracket-current-right-bracket"

Compile the method for *selector* in the current class.

`current-interface` (– *addr*) objects

Variable: contains the class or interface currently being defined.

`dict-new` (*... class* – *object*) objects

`allot` and initialize an object of class *class* in the dictionary.

`end-class` (*align offset* `"`*name*`"` –) objects

name execution: `-- class`
End a class definition. The resulting class is *class*.

`end-class-noname` (*align offset* – *class*) objects

End a class definition. The resulting class is *class*.

`end-interface` (`"`*name*`"` –) objects

`name` execution: `-- interface`
End an interface definition. The resulting interface is *interface*.

`end-interface-noname` (– *interface*) objects

End an interface definition. The resulting interface is *interface*.

`end-methods` (–) objects

Switch back from defining methods of a class to normal mode (currently this just restores the old search order).

`exitm` (–) objects

`exit` from a method; restore old `this`.

`heap-new` (*... class – object*) objects

`allocate` and initialize an object of class *class*.

`implementation` (*interface* –) objects

The current class implements *interface*. I.e., you can use all selectors of the interface in the current class and its descendents.

`init-object` (*... class object* –) objects

Initialize a chunk of memory (*object*) to an object of class *class*; then performs `construct`.

`inst-value` (*align1 offset1* "*name*" – *align2 offset2*) objects

name execution: `-- w`
w is the value of the field *name* in `this` object.

`inst-var` (*align1 offset1 align size* "*name*" – *align2 offset2*) objects

name execution: `-- addr`
addr is the address of the field *name* in `this` object.

`interface` (–) objects

Start an interface definition.

`m:` (– *xt colon-sys; run-time: object* –) objects “m-colon”

Start a method definition; *object* becomes new `this`.

`:m` ("*name*" – *xt; run-time: object* –) objects “colon-m”

Start a named method definition; *object* becomes new `this`. Has to be ended with `;m`.

`;m` (*colon-sys –; run-time:* –) objects “semicolon-m”

End a method definition; restore old `this`.

`method` (*xt* "*name*" –) objects

`name` execution: `... object -- ...`
Create selector *name* and makes *xt* its method in the current class.

`methods` (*class* –) objects

Makes *class* the current class. This is intended to be used for defining methods to override selectors; you cannot define new fields or selectors.

`object` (– *class*) objects

the ancestor of all classes.

`overrides` (*xt* "*selector*" –) objects

replace default method for *selector* in the current class with *xt*. `overrides` must not be used during an interface definition.

`[parent]` (*compile-time:* "*selector*" – *; run-time: ... object* – *...*) objects "left-bracket-parent-right-bracket"

Compile the method for *selector* in the parent of the current class.

`print` (*object* –) objects

Print the object. The method for the class *object* prints the address of the object and the address of its class.

`protected` (–) objects

Set the compilation wordlist to the current class's wordlist

`public` (–) objects

Restore the compilation wordlist that was in effect before the last `protected` that actually changed the compilation wordlist.

`selector` ("*name*" –) objects

name execution: `... object -- ...`
Create selector *name* for the current class and its descendents; you can set a method for the selector in the current class with `overrides`.

`this` (– *object*) objects

the receiving object of the current method (aka active object).

`<to-inst>` (*w xt* –) objects "less-to-inst-to"

store *w* into the field *xt* in `this` object.

`[to-inst]` (*compile-time:* "*name*" – *; run-time: w* –) objects "left-bracket-to-inst-right-bracket"

store *w* into field *name* in `this` object.

`to-this` (*object* –) objects

Set `this` (used internally, but useful when debugging).

`xt-new` (*... class xt* – *object*) objects

Make a new object, using `xt ( align size -- addr )` to get memory.

6.27.4 The `oof.fs` model

This section describes the `oof.fs` package.

The package described in this section has been used in bigFORTH since 1991, and used for two large applications: a chromatographic system used to create new medicaments, and a graphic user interface library (MINOS).

You can find a description (in German) of `oof.fs` in *Object oriented bigFORTH* by Bernd Paysan, published in *Vierte Dimension* 10(2), 1994.

6.27.4.1 Properties of the `oof.fs` model

- This model combines object oriented programming with information hiding. It helps you writing large application, where scoping is necessary, because it provides class-oriented scoping.

- Named objects, object pointers, and object arrays can be created, selector invocation uses the "object selector" syntax. Selector invocation to objects and/or selectors on the stack is a bit less convenient, but possible.
- Selector invocation and instance variable usage of the active object is straightforward, since both make use of the active object.
- Late binding is efficient and easy to use.
- State-smart objects parse selectors. However, extensibility is provided using a (parsing) selector `postpone` and a selector `'`.
- An implementation in Standard Forth is available.

6.27.4.2 Basic `oof.fs` Usage

This section uses the same example as for `objects` (see Section 6.27.3.2 [Basic Objects Usage], page 230).

You can define a class for graphical objects like this:

```
object class graphical \ "object" is the parent class
  method draw ( x y -- )
class;
```

This code defines a class `graphical` with an operation `draw`. We can perform the operation `draw` on any `graphical` object, e.g.:

```
100 100 t-rex draw
```

where `t-rex` is an object or object pointer, created with e.g. `graphical : t-rex`.

How do we create a graphical object? With the present definitions, we cannot create a useful graphical object. The class `graphical` describes graphical objects in general, but not any concrete graphical object type (C++ users would call it an *abstract class*); e.g., there is no method for the selector `draw` in the class `graphical`.

For concrete graphical objects, we define child classes of the class `graphical`, e.g.:

```
graphical class circle \ "graphical" is the parent class
  cell var circle-radius
how:
  : draw ( x y -- )
    circle-radius @ draw-circle ;

  : init ( n-radius -- )
    circle-radius ! ;
class;
```

Here we define a class `circle` as a child of `graphical`, with a field `circle-radius`; it defines new methods for the selectors `draw` and `init` (`init` is defined in `object`, the parent class of `graphical`).

Now we can create a circle in the dictionary with:

```
50 circle : my-circle
```

`:` invokes `init`, thus initializing the field `circle-radius` with 50. We can draw this new circle at (100,100) with:

```
100 100 my-circle draw
```

Note: You can only invoke a selector if the receiving object belongs to the class where the selector was defined or one of its descendents; e.g., you can invoke `draw` only for objects belonging to `graphical` or its descendents (e.g., `circle`). The scoping mechanism will check if you try to invoke a selector that is not defined in this class hierarchy, so you'll get an error at compilation time.

6.27.4.3 The `oof.fs` base class

When you define a class, you have to specify a parent class. So how do you start defining classes? There is one class available from the start: `object`. You have to use it as ancestor for all classes. It is the only class that has no parent. Classes are also objects, except that they don't have instance variables; class manipulation such as inheritance or changing definitions of a class is handled through selectors of the class `object`.

`object` provides a number of selectors:

- `class` for subclassing, `definitions` to add definitions later on, and `class?` to get type informations (is the class a subclass of the class passed on the stack?).

  `object-class` ("*name*" –) oof

  `object-definitions` (–) oof

  `object-class?` (*o* – *flag*) oof "class-query"
- `init` and `dispose` as constructor and destructor of the object. `init` is invocated after the object's memory is allocated, while `dispose` also handles deallocation. Thus if you redefine `dispose`, you have to call the parent's dispose with `super dispose`, too.

  `object-init` (... –) oof

  `object-dispose` (–) oof
- `new`, `new[]`, `:`, `ptr`, `asptr`, and `[]` to create named and unnamed objects and object arrays or object pointers.

  `object-new` (– *o*) oof

  `object-new[]` (*n* – *o*) oof "new-array"

  `object-:` ("*name*" –) oof "define"

  `object-ptr` ("*name*" –) oof

  `object-asptr` (*o* "*name*" –) oof

  `object-[]` (*n* "*name*" –) oof "array"
- `::` and `super` for explicit scoping. You should use explicit scoping only for super classes or classes with the same set of instance variables. Explicitly-scoped selectors use early binding.

  `object-::` ("*name*" –) oof "scope"

  `object-super` ("*name*" –) oof
- `self` to get the address of the object

  `object-self` (– *o*) oof
- `bind`, `bound`, `link`, and `is` to assign object pointers and instance defers.

  `object-bind` (*o* "*name*" –) oof

  `object-bound` (*class addr* "*name*" –) oof

  `object-link` ("*name*" – *class addr*) oof

  `object-is` (*xt* "*name*" –) oof

- `'` to obtain selector tokens, `send` to invocate selectors form the stack, and `postpone` to generate selector invocation code.

  `object-'` ("*name*" – *xt*) oof "tick"

  `object-postpone` ("*name*" –) oof
- `with` and `endwith` to select the active object from the stack, and enable its scope. Using `with` and `endwith` also allows you to create code using selector `postpone` without being trapped by the state-smart objects.

  `object-with` (*o* –) oof

  `object-endwith` (–) oof

6.27.4.4 Class Declaration

- Instance variables

  `var` (*size* –) oof

  Create an instance variable
- Object pointers

  `ptr` (–) oof

  Create an instance pointer

  `asptr` (*class* –) oof

  Create an alias to an instance pointer, cast to another class.
- Instance defers

  `defer` (–) oof

  Create an instance defer
- Method selectors

  `early` (–) oof

  Create a method selector for early binding.

  `method` (–) oof

  Create a method selector.
- Class-wide variables

  `static` (–) oof

  Create a class-wide cell-sized variable.
- End declaration

  `how:` (–) oof "how-to"

  End declaration, start implementation

  `class;` (–) oof "end-class"

  End class declaration or implementation

6.27.5 The `mini-oof.fs` model

Gforth's third object oriented Forth package is a 12-liner. It uses a mixture of the `objects.fs` and the `oof.fs` syntax, and reduces to the bare minimum of features. This is based on a posting of Bernd Paysan in comp.lang.forth.

6.27.5.1 Basic `mini-oof.fs` Usage

There is a base class (`class`, which allocates one cell for the object pointer) plus seven other words: to define a method, a variable, a class; to end a class, to resolve binding, to allocate an object and to compile a class method.

`object` (– *a-addr*) mini-oof

object is the base class of all objects.

`method` (*m v* "*name*" – *m' v*) mini-oof2

Define a selector *name*; increments the number of selectors *m* (in bytes).

`var` (*m v size* "*name*" – *m v'*) mini-oof2

define an instance variable with *size* bytes by the name *name*, and increments the amount of storage per instance *m* by *size*.

`class` (*class* – *class methods vars*) mini-oof2

start a class definition with superclass *class*, putting the size of the methods table and instance variable space on the stack.

`end-class` (*class methods vars* "*name*" –) mini-oof2

finishs a class definition and assigns a name *name* to the newly created class. Inherited methods are copied from the superclass.

`defines` (*xt class* "*name*" –) mini-oof

Bind *xt* to the selector *name* in class *class*.

`new` (*class* – *o*) mini-oof

Create a new incarnation of the class *class*.

`::` (*class* "*name*" –) mini-oof "double-colon"

Compile the method for the selector *name* of the class *class* (not immediate!).

6.27.5.2 Mini-OOF Example

A short example shows how to use this package. This example, in slightly extended form, is supplied as `moof-exm.fs`

```
object class
  method init
  method draw
end-class graphical
```

This code defines a class `graphical` with an operation `draw`. We can perform the operation `draw` on any `graphical` object, e.g.:

```
100 100 t-rex draw
```

where `t-rex` is an object or object pointer, created with e.g. `graphical new Constant t-rex`.

For concrete graphical objects, we define child classes of the class `graphical`, e.g.:

```
graphical class
  cell var circle-radius
end-class circle \ "graphical" is the parent class
```

```
:noname ( x y -- )
  circle-radius @ draw-circle ; circle defines draw
:noname ( r -- )
  circle-radius ! ; circle defines init
```

There is no implicit init method, so we have to define one. The creation code of the object now has to call init explicitly.

```
circle new Constant my-circle
50 my-circle init
```

It is also possible to add a function to create named objects with automatic call of `init`, given that all objects have `init` on the same place:

```
: new: ( .. o "name" -- )
    new dup Constant init ;
80 circle new: large-circle
```

We can draw this new circle at (100,100) with:

```
100 100 my-circle draw
```

6.27.5.3 `mini-oof.fs` Implementation

Object-oriented systems with late binding typically use a "vtable"-approach: the first variable in each object is a pointer to a table, which contains the methods as function pointers. The vtable may also contain other information.

So first, let's declare selectors:

```
: method ( m v "name" -- m' v ) Create  over , swap cell+ swap
  DOES> ( ... o -- ... ) @ over @ + @ execute ;
```

During selector declaration, the number of selectors and instance variables is on the stack (in address units). `method` creates one selector and increments the selector number. To execute a selector, it takes the object, fetches the vtable pointer, adds the offset, and executes the method *xt* stored there. Each selector takes the object it is invoked with as top of stack parameter; it passes the parameters (including the object) unchanged to the appropriate method which should consume that object.

Now, we also have to declare instance variables

```
: var ( m v size "name" -- m v' ) Create  over , +
  DOES> ( o -- addr ) @ + ;
```

As before, a word is created with the current offset. Instance variables can have different sizes (cells, floats, doubles, chars), so all we do is take the size and add it to the offset. If your machine has alignment restrictions, put the proper `aligned` or `faligned` before the variable, to adjust the variable offset. That's why it is on the top of stack.

We need a starting point (the base object) and some syntactic sugar:

```
Create object  1 cells , 2 cells ,
: class ( class -- class selectors vars ) dup 2@ ;
```

For inheritance, the vtable of the parent object has to be copied when a new, derived class is declared. This gives all the methods of the parent class, which can be overridden, though.

```
: end-class  ( class selectors vars "name" -- )
```

```
Create  here >r , dup , 2 cells ?DO ['] noop , 1 cells +LOOP
cell+ dup cell+ r> rot @ 2 cells /string move ;
```

The first line creates the vtable, initialized with `noop`s. The second line is the inheritance mechanism, it copies the xts from the parent vtable.

We still have no way to define new methods, let's do that now:

```
: defines ( xt class "name" -- ) ' >body @ + ! ;
```

To allocate a new object, we need a word, too:

```
: new ( class -- o )  here over @ allot swap over ! ;
```

Sometimes derived classes want to access the method of the parent object. There are two ways to achieve this with Mini-OOF: first, you could use named words, and second, you could look up the vtable of the parent object.

```
: :: ( class "name" -- ) ' >body @ + @ compile, ;
```

Nothing can be more confusing than a good example, so here is one. First let's declare a text object (called `button`), that stores text and position:

```
object class
  cell var text
  cell var len
  cell var x
  cell var y
  method init
  method draw
end-class button
```

Now, implement the two methods, `draw` and `init`:

```
:noname ( o -- )
 >r r@ x @ r@ y @ at-xy  r@ text @ r> len @ type ;
 button defines draw
:noname ( addr u o -- )
 >r 0 r@ x ! 0 r@ y ! r@ len ! r> text ! ;
 button defines init
```

To demonstrate inheritance, we define a class `bold-button`, with no new data and no new selectors:

```
button class
end-class bold-button

: bold   27 emit ." [1m" ;
: normal 27 emit ." [0m" ;
```

The class `bold-button` has a different draw method to `button`, but the new method is defined in terms of the draw method for `button`:

```
:noname bold [ button :: draw ] normal ; bold-button defines draw
```

Finally, create two objects and apply selectors:

```
button new Constant foo
s" thin foo" foo init
page
```

```
foo draw
bold-button new Constant bar
s" fat bar" bar init
1 bar y !
bar draw
```

6.27.6 Mini-OOF2

Mini-OOF2 is very similar to Mini-OOF in many respects, but differs significantly in a few aspects. In particular, Mini-OOF2 has a current object variable, and uses the primitives `>o` and `o>` to manipulate that object stack. All method invocations and instance variable accesses refer to the current object.

`>o` (*c-addr* – *r:c-old*) new "to-o"

Set the current object to *c_addr*, the previous current object is pushed to the return stack

`o>` (*r:c-addr* –) new "o-restore"

Restore the previous current object from the return stack

To ease passing an object pointer to method invocation or instance variable accesses, the additional recognizer `rec-moof2` is activated.

`rec-moof2` (*c-addr u* – *nt translate-moof2* | *translate-none*) mini-oof2 "rec-moof-two"

This recognizer recognizes `.word` (if *word* is visible) and the translation is equivalent to `>o word o>`.

To assign methods to selectors, use *xt class* `is` *selector*, so no `defines` necessary. For early binding of methods, instead of `::` you use `defers`:

```
[ class ] defers selector
```

Instead of writing

```
:noname code ; class is selector
```

you can also use the syntactic sugar

```
class :method selector code ;
```

`:method` (*class* "*name*" –) gforth-experimental "colon-method"

define a noname that is assigned to the deferred word *name* in *class* at `;`.

6.27.7 Comparison of object models

Many object-oriented Forth extensions have been proposed (*A survey of object-oriented Forths* (SIGPLAN Notices, April 1996) by Bradford J. Rodriguez and W. F. S. Poehlman lists 17).

A number of Forth object-oriented models were inspired by Neon (see *Object-oriented programming in ANS Forth* (Forth Dimensions, March 1997) by Andrew McKewan) but this model has a number of limitations[26]:

- It uses a `selector object` syntax, which makes it unnatural to pass objects on the stack. By contrast, all models discussed in this manual have objects on the stack, leading to a `object selector` syntax.

[26] A longer version of this critique can be found in *On Standardizing Object-Oriented Forth Extensions* (Forth Dimensions, May 1997) by Anton Ertl.

- It requires that the selector parses the input stream (at compile time); this leads to reduced extensibility and to bugs that are hard to find. In `oof.fs` there is also parsing of the input stream, in the other models this does not happen for regular selector invocations.
- It allows using every selector on every object; this eliminates the need for interfaces, but makes it harder to create efficient implementations. The models included with Gforth typically only support using selectors on objects of certain classes, with surprising and usually unpleasant results if you use a selector on other objects (or other things).

Another difference between object models is whether they have an active object such as `this` in `objects.fs`. `mini-oof.fs` does not have an active object, and uses the top-of-stack as the object for instance variables and the selector.

And if they have an active object, there are differences in what sets or changes it, and what make use of the active object.

- In `objects.fs` `this` is only changed on method invocation (i.e., by `m: ... ;m`; note the possible separation between methods and selectors in `objects.fs`), and restored when leaving the method. Only instance variables use `this`, while selectors take their object from the stack.
- In `mini-oof2.fs` only `>o ... o>` changes the active object, and both selectors and instance variables use the active object as the object for which the selector is invoked, or the instance variable accessed. However, `mini-oof2.fs` has a recognizer that recognizes `.`*`word`* and executes/compiles/postpones code equivalent to `>o` *`word`* `>o`, so you can use it like `objects.fs` by always invoking selectors with a prefixed '`.`', and never invoke instance variables in that way.
- In `oof.fs` selector/method invocation changes the active object, like in `object.fs`, but it also has `>o ... o>` like `mini-oof2.fs` (but without a helping recognizer).

In Does late binding have to be slow? (`http://www.forth.org/oopf.html`) (Forth Dimensions 18(1) 1996, pages 31-35) Andras Zsoter describes a model that makes heavy use of an active object (like `this` in `objects.fs`): The active object is not only used for accessing all fields, but also specifies the receiving object of every selector invocation (like in `mini-oof2.fs`; you have to change the active object explicitly with `{ ... }` (similar to how you use `>o ... o>` in `oof.fs` and `mini-oof2.fs`). A Standard Forth implementation of this model is available through `http://www.forth.org/oopf.html`.

The `oof.fs` model combines information hiding and overloading resolution (by keeping names in various word lists) with object-oriented programming. It sets the active object implicitly on method entry, but also allows explicit changing (with `>o...o>` or with `with...endwith`). It uses parsing and state-smart objects and classes for resolving overloading and for early binding: the object or class parses the selector and determines the method from this. If the selector is not parsed by an object or class, it performs a call to the selector for the active object (late binding), like Zsoter's model. Fields are always accessed through the active object. The big disadvantage of this model is the parsing and the state-smartness, which reduces extensibility and increases the opportunities for subtle bugs; essentially, you are only safe if you never tick or `postpone` an object or class (Bernd disagrees, but I (Anton) am not convinced).

6.28 Closures

Gforth provides flat closures (called closures in the following). Closures are similar to quotations (see Section 6.11.8 [Quotations], page 125), but the execution token (xt) that represents a closure does not just refer to code, but also to data. Running the code of a closure definition creates a closure data structure (also referred to as "closure"), that is represented by an execution token. The closure data structure needs to be allocated somewhere, and in Gforth this memory is managed explicitly.

As an example, consider a word that sums up the results of a function `( n -- r)` across a range of input values:

```
: sum {: limit start xt -- r :}
  0e limit start ?do
    i xt execute f+
  loop ;
```

You can add up the values of the function 1/n for n=1..10 with:

```
11 1 [: s>f -1e f** ;] sum f.
```

Yes, you can do it shorter and more efficiently with `1/f`, but bear with me. If you want to add up 1/n^2, you can write

```
11 1 [: s>f -2e f** ;] sum f.
```

Now if you want to deal with additional exponents and these exponents are known at compile time, you can create a new quotation for every exponent. But you may prefer to provide an exponent and produce an xt without having to write down a quotation every time. If the value of the exponent is only known at run-time, producing such an xt is possible in Forth, but even more involved, and consumes dictionary memory (with limited deallocation options). Closures come to the rescue:

```
: 1/n^r ( r -- xt; xt execution: n -- r1 )
  fnegate [f:h ( n -r ) s>f fswap f** ;] ;

11 1 3e   1/n^r dup >r sum f. r> free-closure
11 1 0.5e 1/n^r dup >r sum f. r> free-closure
```

When `1/n^r` runs, it creates a closure that incorporates a floating-point number (indicated by the `f` in `[f:h`), in particular the value -r. It also references the code between `[f:h` and `;]`. The memory for the closure comes from the heap, i.e. `allocate`d memory (indicated by the `h` in `[f:h`). `1/n^r` produces an xt representing this closure. This xt is then passed to `sum` and `execute`d there.

When the closure is executed (in `sum`), -r is pushed (in addition to the n that has already been pushed before the `execute`) and the code of the closure is run.

The code above shows a pure-stack closure (no locals involved). Pure-stack closures start with a word with the naming scheme `[`*T*`:`*A* where the type *T* can be `n` (cell), `d` (double-cell), or `f` (FP). The allocator *A* can be `l` (local), `d` (dictionary), `h` (heap), or `h1: Allocate` the closure on the heap and `free` it after the first execution; this is used for passing data to another task with `send-event` (see Section 6.31.1.5 [Message queues], page 275). A pure-stack closure consumes one *T* from a stack at closure creation time (when the code containing the closure definition is run), and pushes an xt. After creating the closure, execution continues behind the `;]`.

When the xt is executed (directly with `execute` or indirectly through, e.g., `compile,` or a deferred word), it pushes the stack item that was consumed at closure creation time and then runs the code inside the closure definition (up to the `;]`). You can deallocate heap-allocated closures with

`free-closure` (*xt* –) gforth-1.0

Free the heap-allocated closure *xt*.

Like a quotation, a (flat) closure cannot access locals of the enclosing definition(s).

The words for starting pure-stack closure definitions are:

`[n:l` (*compilation – colon-sys; run-time: n – xt ; xt execution: – n*) gforth-1.0 "open-bracket-n-colon-l"

`[d:l` (*compilation – colon-sys; run-time: d – xt ; xt execution: – d*) gforth-1.0 "open-bracket-d-colon-l"

`[f:l` (*compilation – colon-sys; run-time: r – xt ; xt execution: – r*) gforth-1.0 "open-bracket-r-colon-l"

`[n:d` (*compilation – colon-sys; run-time: n – xt ; xt execution: – n*) gforth-1.0 "open-bracket-n-colon-d"

`[d:d` (*compilation – colon-sys; run-time: d – xt ; xt execution: – d*) gforth-1.0 "open-bracket-d-colon-d"

`[f:d` (*compilation – colon-sys; run-time: r – xt ; xt execution: – r*) gforth-1.0 "open-bracket-r-colon-d"

`[n:h` (*compilation – colon-sys; run-time: n – xt ; xt execution: – n*) gforth-1.0 "open-bracket-n-colon-h"

`[d:h` (*compilation – colon-sys; run-time: d – xt ; xt execution: – d*) gforth-1.0 "open-bracket-d-colon-h"

`[f:h` (*compilation – colon-sys; run-time: r – xt ; xt execution: – r*) gforth-1.0 "open-bracket-r-colon-h"

`[n:h1` (*compilation – colon-sys; run-time: n – xt ; xt execution: – n*) gforth-1.0 "open-bracket-n-colon-h1"

`[d:h1` (*compilation – colon-sys; run-time: d – xt ; xt execution: – d*) gforth-1.0 "open-bracket-d-colon-h1"

`[f:h1` (*compilation – colon-sys; run-time: r – xt ; xt execution: – r*) gforth-1.0 "open-bracket-r-colon-h1"

If you want to pass more than one stack item from closure creation to execution time, defining more such words becomes unwieldy, and the code inside the closure definition might have to juggle many stack items, so Gforth does not provide such additional words. Instead, Gforth offers flat closures that define locals. Here's the example above, but using locals-defining closures:

```
: 1/n^r ( r -- xt; xt execution: n -- r1 )
  fnegate [{: f: -r :}h s>f -r f** ;] ;
```

The number, types, and order of the locals are used for specifying how many and which stack items are consumed at closure creation time. At closure execution time these values become the values of the locals. The locals definition ends with a word with a naming

scheme `:}A`, where *A* specifies where the closure is allocated: `l` (local), `d` (dictionary), `h` (heap), or `h1` (heap, `free` on first execution).

Note that the locals are still strictly local to one execution of the xt, and any changes to the locals (e.g., with `to`) do not change the values stored in the closure; i.e., in the next execution of the closure the locals will be initialized with the values that closure creation consumed.

`[{:` (*compilation – hmaddr u latest wid 0 ; instantiation ... – xt*) gforth-1.0 "start-closure"

Starts a closure. Closures started with `[{:` define locals for use inside the closure. The locals-definition part ends with `:}l`, `:}h`, `:}h1`, `:}d` or `:}xt`. The rest of the closure definition is Forth code. The closure ends with `;]`. When the closure definition is encountered during execution (closure creation time), the values going into the locals are consumed, and an execution token (xt) is pushed on the stack; when that execution token is executed (with `execute`, through `compile,` or a deferred word), the code in the closure is executed (closure execution time). If the xt of a closure is executed multiple times, the values of the locals at the start of code execution are those from closure-creation time, unaffected by any locals-changes in earlier executions of the closure.

`:}l` (*hmaddr u latest latestnt wid state 0 a-addr1 u1 ... –*) gforth-1.0 "close-brace-locals"

Ends a closure's locals definition. The closure will be allocated on the locals stack.

`:}d` (*hmaddr u latest latestnt wid state 0 a-addr1 u1 ... –*) gforth-1.0 "colon-close-brace-d"

Ends a closure's locals definition. The closure will be allocated in the dictionary.

`:}h` (*hmaddr u latest latestnt wid state 0 a-addr1 u1 ... –*) gforth-1.0 "colon-close-brace-h"

Ends a closure's locals definition. At the run-time of the surrounding definition this allocates the closure on the heap; you are then responsible for deallocating it with `free-closure`.

`:}h1` (*hmaddr u latest latestnt wid state 0 a-addr1 u1 ... –*) gforth-1.0 "colon-close-brace-h-one"

Ends a closure's locals definition. The closure is deallocated after the first execution, so this is a one-shot closure, particularly useful in combination with `send-event` (see Section 6.31.1.5 [Message queues], page 275).

`:}xt` (*hmaddr u latest latestnt wid state 0 a-addr1 u1 ... –*) gforth-1.0 "colon-close-brace-x-t"

Ends a closure's locals definition. The closure will be allocated by the xt on the stack, so the closure's run-time stack effect is `( ... xt-alloc -- xt-closure )`.

`>addr` (*... xt – addr*) gforth-internal "to-addr"

Obtain the address *addr* of the `addressable:` value-flavoured word *xt*. For some value-flavoured words, additional inputs may be consumed.

If you look at closures in other languages (e.g., Scheme), they are quite different: data is passed by accessing and possibly changing locals of enclosing definitions (lexical scoping). Gforth's closures are based on the flat closures used in the implementation of Scheme, so by writing the code appropriately (see the following subsections) you can do the same things with Gforth's closures as with lexical-scoping closures.

In our programming we have not missed lexical scoping, except when trying to convert code (usually textbook examples) coming from another language. I.e., in our experience

flat closures are as useful and similarly convenient as lexical scoping. For comparison, if Gforth supported lexical scoping instead of flat closures, the definition of `1/n^r` might look as follows:

```
\ this does not work in Gforth:
: 1/n^r ( r -- xt; xt execution: n -- r1 )
  fnegate {: -r :} [:h s>f -r f** ;] ;
```

But if you want to know how to convert lexical scoping to Gforth's flat closures, the following subsections explain it.

6.28.1 How do I read outer locals?

As long as you only read the value of locals, you can duplicate them as needed, so a way to convert an access to an outer local for flat closures is to just pass the values on the stack to the closures and define them again as locals there. Here's an example: Consider the following code for a hypothetical Gforth with a quotation-like syntax for lexical-scoping closures:

```
\ does not work; [:d would dictionary-allocate the closure
: ...
  ... {: a b :} ...
  [:d ...
    ... {: c d :} ...
    [:d ... a b c d ... ;]
    ...
  ;]
  ... ;
```

you can convert it to flat closures as follows:

```
: ...
  ... {: a b :} ...
  a b [{: a b :}d ...
    ... {: c d :} ...
    a b c d [{: a b c d :}d
      ... a b c d ... ;]
    ...
  ;]
  ... ;
```

Only those locals that are read in the closure need to be passed in.

This process is called *closure conversion* in the programming language implementation literature.

6.28.2 How do I write outer locals?

A local instance that is written and read must exist at only one location, its home location. The address of this home location is only read and can be duplicated and passed around. A textbook example might look like this in a hypothetical Gforth with lexical-scoping and explicit dictionary allocation:

```
\ does not work
: counter ( -- xt-inc xt-val )
```

```
  0 {: n :}d
  [:d n 1+ to n ;]
  [:d n ;]
;
\ for usage example see below
```

Instead, you allocate the home location, and pass its address around:

```
: counter ( -- xt-inc xt-val )
  align here {: np :} 0 , \ home location
  np [{: np :}d 1 np +! ;]
  np [{: np :}d np @ ;]
;
\ usage example
counter \ first instance
dup execute . \ prints 0
over execute
over execute
dup execute . \ prints 2
counter \ second instance
over execute
dup execute . \ prints 1
2swap \ work on first instance again
dup execute . \ prints 2
```

This introduction of a home location is called *assignment conversion* in the programming language implementation literature.

You can also use pure-stack closures in this case:

```
: counter ( -- xt-inc xt-val )
  align here {: np :} 0 , \ home location
  np [n:d 1 swap +! ;]
  np [n:d @ ;]
;
\ same usage
```

Instead of dictionary allocation you can also `allocate` on the heap. For local allocation of the home location you can use variable-flavoured locals (see Section 6.26.1 [Gforth locals], page 217), but of course then the closures must not be used after leaving the definition in which the home location is defined. E.g.

```
: counter-example ( -- )
  0 {: w^ np :} \ home location
  np [n:d 1 swap +! ;]
  np [n:d @ ;]
  dup execute cr .
  over execute
  over execute
  dup execute cr .
  2drop
;
```

```
counter-example \ prints 0 and 2
```

There is actually rarely a reason to use home locations at all, because what the textbook examples do with closures and writable locals can be done in Gforth more directly with structs (see Section 6.12 [Structures], page 141) or objects (see Section 6.27 [Object-oriented Forth], page 228), or in the counter example, simply with `create`:

```
: counter ( "name" -- )
  create 0 , ;
: counter-inc ( addr -- )
  1 swap +! ;
: counter-val ( addr -- )
  @ ;
\ usage example
counter a
a counter-val . \ prints 0
a counter-inc
a counter-inc
a counter-val . \ prints 2
counter b
b counter-inc
b counter-val . \ prints 1
a counter-val . \ prints 2
```

Still, for dictionary and heap allocation Gforth has a home-location definition syntax based on the locals-definition syntax. Here's a heap-allocation version of `counter` using closures and the locals-like home-location syntax:

```
: counter ( -- handle xt-inc xt-val )
  0 <{: w^ np :}h
  np [n:h 1 swap +! ;]
  np [n:h @ ;]
  ;> -rot ;
\ usage example
counter \ first instance
dup execute . \ prints 0
over execute
over execute
dup execute . \ prints 2
counter \ second instance
over execute
dup execute . \ prints 1
free-closure free-closure free throw \ back to first instance
dup execute . \ prints 2
free-closure free-clouse free throw
```

Here `<{:` starts a locals scope (similar to a closure itself), then you define (variable-flavoured) locals. `:}h` (or `:}d`) finishes the locals definition. Now (and up to `;>`) you can use the names of the defined locals. Finally, `;>` ends the scope and pushes the start address of the allocated home-location block (also when using `:}d` for dictionary allocation), for `free`ing the home-location block later.

We have produced no uses of `<{:` and `;>` in the first 6 years that they were present in (development) Gforth. We think that the reason is that one prefers structs or objects for modifiable data. Therefore, we intend to remove these words in the future. If you want to see them preserved, contact us and make a case for them.

`<{:` (*compilation – colon-sys ; run-time –*) gforth-obsolete "start-homelocation"

Starts defining a home location block.

`;>` (*compilation colon-sys – ; run-time – addr*) gforth-obsolete "end-homelocation"

Ends defining a home location; *addr* is the start address of the home-location block (used for deallocation).

6.29 Regular Expressions

Regular expressions are pattern matching algorithms for strings found in many contemporary languages. You can add regular expression functionality to Gforth with `require regexp.fs`.

However, the regexp support does not use any of the established regexp syntaxes; instead, the syntax is described below. Here's an example of how it is used:

```
charclass [bl-]   blanks +class '-' +char

: telnum ( addr u -- flag )
    (( {{ ` (  \( \d \d \d \) ` ) || \( \d \d \d \) }}  \s
    \( \d \d \d \) [bl-] c?
    \( \d \d \d \d \) )) ;

: ?tel ( addr u -- )
  telnum IF
    '(' emit \1 type ." ) " \2 type '-' emit \3 type ."  succeeded"
  ELSE
    \0 type ."  failed"
  THEN ;

s" (555) 555-5555abc" ?tel \ prints "(555) 555-5555 succeeded"
s" (555) 555-555abc"  ?tel \ prints "(555) 555-555abc failed"
```

More examples can be found in `test/regexp-test.fs`.

Your regular expression code is enclosed in `((` and `))`.

`((` (*addr u –*) regexp-pattern "paren-paren"

start regexp block

`))` (*– flag*) regexp-pattern "close-paren-close-paren"

end regexp block

Pattern matching in regular expressions have character sets as elements, so a number of functions allow you to create and modify character sets (called `charclass`). All characters here are bytes, so this doesn't extend to unicode characters.

`charclass` (–) regexp-cg

Create a charclass

`+char` (*char* –) regexp-cg "plus-char"

add a char to the current charclass

`-char` (*char* –) regexp-cg

remove a char from the current charclass

`..char` (*start end* –) regexp-cg "dot-dot-char"

add a range of chars to the current charclass

`+chars` (*addr u* –) regexp-cg "plus-chars"

add a string of chars to the current charclass

`+class` (*class* –) regexp-cg "plus-class"

union of charclass *class* and the current charclass

`-class` (*class* –) regexp-cg

subtract the charclass *class* from the current charclass

There are predefined charclasses and tests for them, and generic checks. If a check fails, the next possible alternative of the regular expression is tried, or a loop is terminated.

`c?` (*addr class* –) regexp-pattern "c-question"

check *addr* for membership in charclass *class*

`-c?` (*addr class* –) regexp-pattern "-c-question"

check *addr* for not membership in charclass *class*

`\d` (*addr* – *addr'*) regexp-pattern "backslash-d"

check for digit

`\s` (*addr* – *addr'*) regexp-pattern "backslash-s"

check for blanks

`.?` (*addr* – *addr'*) regexp-pattern "dot-question"

check for any single charachter

`-\d` (*addr* – *addr'*) regexp-pattern "–backslash-d"

check for not digit

`-\s` (*addr* – *addr'*) regexp-pattern "–backslash-s"

check for not blank

`` ` `` (`"`*char*`"` –) regexp-pattern "backtick"

check for particular char

`` `? `` (`"`*char*`"` –) regexp-pattern "backtick-question"

`` -` `` (`"`*char*`"` –) regexp-pattern "–backtick"

check for particular char

You can certainly also check for start and end of the string, and for whole string constants.

`\^` (*addr* – *addr*) regexp-pattern "backslash-caret"

check for string start

`\$` (*addr* – *addr*) regexp-pattern "backslash-dollar"

check for string end

`str=?` (*addr1 addr u – addr2*) regexp-pattern "str-equals-question"

check for a computed string on the stack (possibly a backreference)

`="` (`<`*string*`>"` –) regexp-pattern "equals-quote"

check for string

Loops that check for repeated character sets can be greedy or non-greedy.

`{**` (*addr – addr addr*) regexp-pattern "begin-greedy-star"

greedy zero-or-more pattern

`**}` (*sys* –) regexp-pattern "end-greedy-star"

end of greedy zero-or-more pattern

`{++` (*addr – addr addr*) regexp-pattern "begin-greedy-plus"

greedy one-or-more pattern

`++}` (*sys* –) regexp-pattern "end-greedy-plus"

end of greedy one-or-more pattern

`{*` (*addr – addr addr*) regexp-pattern "begin-non-greedy-star"

non-greedy zero-or-more pattern

`*}` (*addr addr' – addr'*) regexp-pattern "end-non-greedy-star"

end of non-greedy zero-or-more pattern

`{+` (*addr – addr addr*) regexp-pattern "begin-non-greedy-plus"

non-greedy one-or-more pattern

`+}` (*addr addr' – addr'*) regexp-pattern "end-non-greedy-plus"

end of non-greedy one-or-more pattern

Example: Searching for a substring really is a non-greedy match of anything in front of it.

`//` (–) regexp-pattern "slash-slash"

search for string

Alternatives are written with

`{{` (*addr – addr addr*) regexp-pattern "begin-alternatives"

Start of alternatives

`||` (*addr addr – addr addr*) regexp-pattern "next-alternative"

separator between alternatives

`}}` (*addr addr – addr*) regexp-pattern "end-alternatives"

end of alternatives

You can use up to 9 variables named `\1` to `\9` to refer to matched substrings

`\(` (*addr – addr*) regexp-pattern "backslash-paren"

start of matching variable; variables are referred as `\\`1–9

`\)` (*addr – addr*) regexp-pattern "backslash-close-paren"

end of matching variable

`\0` (– *addr u*) regexp-pattern "backslash-zero"

the whole string

Certainly, you can also write code to replace patterns you found.

`s>>` (*addr – addr*) regexp-replace "s-to-to"

Start replace pattern region

`>>` (*addr – addr*) regexp-replace "to-to"

Start arbitrary replacement code, the code shall compute a string on the stack and pass it to `<<`

`<<` (*run-addr addr u – run-addr*) regexp-replace "less-less"

Replace string from start of replace pattern region with *addr u*

`<<"` (`"`*string*`<">"` –) regexp-replace "less-less-quote"

Replace string from start of replace pattern region with *string*

`s//` (*addr u – ptr*) regexp-replace "s-slash-slash"

start search/replace loop

`//s` (*ptr –*) regexp-replace "slash-slash-s"

search end

`//o` (*ptr addr u – addr' u'*) regexp-replace "slash-slash-o"

end search/replace single loop

`//g` (*ptr addr u – addr' u'*) regexp-replace "slash-slash-g"

end search/replace all loop

The classical implementation for this pattern matching is a backtracking algorithm, which is also necessary if you want to have features like backreferencing. Gforth implements regular expressions by providing a language to define backtracking programs for pattern matching. Basic element is the control structure `FORK ... JOIN`, which is a forward call within a word, and therefore allows to code a lightweight try and fail control structure.

`FORK` (*compilation – orig ; run-time f –*) gforth-0.7

AHEAD-like control structure: calls the code after JOIN.

`JOIN` (*orig –*) gforth-0.7

THEN-like control structure for FORK

You can program any sort of arbitrary checks yourself by computing a flag and `?LEAVE` when the check fails.

6.30 Programming Tools

See also Chapter 11 [Emacs and Gforth], page 327.

6.30.1 Text interpreter status

In interactive usage Gforth shows the status after processing each line. The status consists of the "prompt" behind the end of the output of the words on the line and optionally a status bar at the bottom (on by default if the terminal is a color terminal).

The prompt shows the current text interpreter state (`ok` (interpretation), `compiling`, or `postponing`) and the number of items on the data stack (nothing shown if empty) and the

FP stack (nothing shown if empty). E.g., if there are two items on the data stack and one item on the FP stack, `2 f:1` is shown.

The status bar is in blue for interpretation, magenta for compilation, and red for postponing state. By default it shows the contents of `base` in decimal (not shown when the base is decimal), the number of items on the data stack and the top items of the data stack, the number of items on the FP stack, and the top items on the FP stack, the search order and the current wordlist. What is shown can be changed by changing the contents of `status-xts`, but that is currently not documented, apart from turning the status bar off completely:

`-status` (–) gforth-1.0 “"minus-status"”

Turn off the status bar at the bottom of the screen

`+status` (–) gforth-1.0 “plus-status”

Turn on the status bar at the bottom of the screen

6.30.2 Locating source code definitions

Many programming systems are organized as an integrated development environment (IDE) where the editor is the hub of the system, and allows building and running programs. If you want that, Gforth has it, too (see Chapter 11 [Emacs and Gforth], page 327).

However, several Forth systems have a different kind of IDE: The Forth command line is the hub of the environment; you can view the source from there in various ways, and call an editor if needed.

Gforth also implements such an IDE. It mostly follows the conventions of SwiftForth where they exist, but implements features beyond them.

An advantage of this approach is that it allows you to use your favourite editor: set the environment variable `EDITOR` to your favourite editor, and the editing commands will call that editor; Gforth invokes some GUI editors in the background (so you do not need to finish editing to continue with your Forth session), terminal editors in the foreground (default for editors not known to Gforth is foreground). If you have not set `EDITOR`, the default editor is `vi`.

`locate` ("*name*" –) gforth-1.0

Show the source code of the word *name*, set the current location there, and then call `fancy-after-l` for navigating the displayed source.

`fancy-after-l` (*c-addr1 u1 lineno1* – *c-addr2 u2 lineno2*) gforth-internal

Wait for keyboard input. If any other key than one of those mentioned below is pressed (in particular, if `ESC` or `newline` is pressed), `fancy-after-l` terminates; you can then reenter the `fancy-after-l` mode with `l`.
`cursor-up`, `Ctrl-p`, or `k`: Show one more line at the top.
`Cursor-down`, `ctrl-n`, or `j`: Show one more line at the bottom.
`Page-up` or `ctrl-u`: Show half of the terminal height more lines at the top.
`Page-down` or `ctrl-d`: Show half of the terminal height more lines at the bottom.
`Ctrl-b`: Scroll up by a full terminal height of source code, with only one line of overlap.
`Space`: Scroll down by a full terminal height of source code, with only one line of overlap.
`Cursor-right` or `l`: Switch to the next `where`, `browse` or backtrace result.

`Cursor-left` or `h`: Switch to the previous `where`, `browse` or backtrace result.
`Ctrl-l`: Redisplay the currently shown source code.

`xt-locate` (*nt/xt* –) gforth-1.0

Show the source code of the word *xt*, set the current location there, and then call `fancy-after-l` for navigating the displayed source.

The *current location* is set by a number of other words in addition to `locate`. Also, when an error happens while loading a file, the location of the error becomes the current location.

A number of words work with the current location:

`l` (–) gforth-1.0

Display source code lines at the current location, then call `fancy-after-l` for navigating the displayed source.

`n` (–) gforth-1.0

Display lines behind the current location, or behind the last `n` or `b` output (whichever was later).

`b` (–) gforth-1.0

Display lines before the current location, or before the last `n` or `b` output (whichever was later).

`g` (–) gforth-1.0

Enter the editor at the current location, or at the start of the last `n` or `b` output (whichever was later).

You can control how many lines `l`, `n` and `b` show by changing the values:

`before-locate` (– *u*) gforth-1.0

number of lines shown before current location (default 3).

`after-locate` (– *u*) gforth-1.0

number of lines shown after current location (default 12).

Finally, you can directly go to the source code of a word in the editor with

`edit` ("*name*" –) gforth-1.0

Enter the editor at the location of "name"

You can see the definitions of similarly-named words with

`browse` ("*subname*" –) gforth-1.0

Show all places where a word with a name that contains *subname* is defined (`mwords`-like, see Section 6.19 [Word Lists], page 182). You can then use `ww`, `nw` or `bw` (see Section 6.30.3 [Locating uses of a word], page 259) to inspect specific occurrences more closely.

6.30.3 Locating uses of a word

`where` ("*name*" –) gforth-1.0

Show all places where *name* is used (text-interpreted). You can then use `ww`, `nw` or `bw` to inspect specific occurrences more closely. Gforth's `where` does not show the definition of *name*; use `locate` for that.

`ww` (*u* –) gforth-1.0

Show the source code of the `where` or `browse` result with index *u* (indices are shown at the right edge of each line in `where` or `browse` output), set the current location there, and then call `fancy-after-l` for navigating the displayed source (including switching to the next or previous result).

`nw` (–) gforth-1.0

Show the source code of the next `where` or `browse` result and set the current location there. The "next result" wraps around to the first result if you reach the end. Then call `fancy-after-l` for navigating the displayed source (including switching to the next or previous result).

`bw` (–) gforth-1.0

Show the source code of the previous `where` or `browse` result and set the current location there. The "previous result" wraps around to the last result if you reach the beginning. Then call `fancy-after-l` for navigating the displayed source (including switching to the next or previous result).

`gg` (–) gforth-1.0

The next `ww`, `nw`, `bw`, `tt`, `bt`, or `nt` (but not `locate`, `edit`, `l` or `g`) puts it result in the editor (like `g`) and does not call `fancy-after-l`. Use `gg gg` to make this permanent rather than one-shot.

`ll` (–) gforth-1.0

The next `ww`, `nw`, `bw`, `tt`, `bt`, or `nt` (but not `locate`, `edit`, `l` or `g`) displays in the Forth system (like `l`). Use `ll ll` to make this permanent rather than one-shot.

`whereg` ("*name*" –) gforth-1.0

Like `where`, but puts the output in the editor. In Emacs, you can then use the compilation-mode commands (see Section "Compilation Mode" in *GNU Emacs Manual*) to inspect specific occurrences more closely.

`short-where` (–) gforth-1.0

Set up `where` to use a short file format (default).

`expand-where` (–) gforth-1.0

Set up `where` to use a fully expanded file format (to pass to e.g. editors).

`prepend-where` (–) gforth-1.0

Set up `where` to show the file on a separate line, followed by `where` lines without file names (like SwiftForth).

The data we have on word usage also allows us to show which words have no uses:

`unused-words` (–) gforth-1.0

list all words without usage

6.30.4 Locating backtrace entry source

An uncaught exception shows a backtrace (actually a return stack dump, see Chapter 7 [Error messages], page 307). If the return stack entry is a return address, the immediately preceding location produced the call that the return address would have returned from, and is shown as word in the backtrace. You can inspect the source code of such a call with the following words:

`tt` (*u* –) gforth-1.0

Show the source code corresponding to the backtrace line with index *u* (indices are shown in the backtrace after the file name and position information and before the return stack content and its interpretation as return address), set the current location there, and then call `fancy-after-l` for navigating the displayed source (including switching to the next or previous result).

`nt` (–) gforth-1.0

Show the source code corresponding to the next backtrace line and set the current location there. The "next backtrace line wraps around to the first one if you reach the end. Then call `fancy-after-l` for navigating the displayed source (including switching to the next or previous result).

`bt` (–) gforth-1.0

Show the source code corresponding to the previous backtrace line and set the current location there. The "previous backtrace line" wraps around to the last one if you reach the beginning. Then call `fancy-after-l` for navigating the displayed source (including switching to the next or previous result).

6.30.5 Examining compiled code

And finally, `see` and friends show compiled code. Some of the things in the source code are not present in the compiled code (e.g., formatting and comments), but this is useful to see what threaded code or native code is produced by macros and Gforth's optimization features.

`see` ("<*spaces*>*name*" –) tools

Locate *name* using the current search order. Display the definition of *name*. Since this is achieved by decompiling the definition, the formatting is mechanised and some source information (comments, interpreted sequences within definitions etc.) is lost.

`xt-see` (*xt* –) gforth-0.2

Decompile the definition represented by *xt*.

`simple-see` ("*name*" –) gforth-0.6

Decompile the colon definition *name*, showing a line for each cell, and try to guess a meaning for the cell, and show that.

`xt-simple-see` (*xt* –) gforth-1.0

Decompile the colon definition *xt* like `simple-see`

`simple-see-range` (*addr1 addr2* –) gforth-0.6

Decompile code in [*addr1*,*addr2*) like `simple-see`

`see-code` ("*name*" –) gforth-0.7

Like `simple-see`, but also shows the dynamic native code for the inlined primitives. For static superinstructions, it shows the primitive sequence instead of the first primitive (the other primitives of the superinstruction are shown, too). For primitives for which native code is generated, it shows the number of stack items in registers at the beginning and at the end (e.g., `1->1` means 1 stack item is in a register at the start and at the end). For each primitive or superinstruction with native code, the inline arguments and component primitives are shown first, then the native code.

`xt-see-code` (*xt* –) gforth-1.0

Decompile the colon definition *xt* like `see-code`.

`see-code-range` (*addr1 addr2* –) gforth-0.7

Decompile code in [*addr1*,*addr2*) like `see-code`.

As an example, consider the following contrived definition:

```
: foo dup >r drop 5 fsin ;
```

This is not particularly useful, but it demonstrates the various variants in Gforth's code generation. Compiling this on `gforth-fast` on AMD64 and then using `see-code foo` outputs:

```
<foo>          dup >r     1->1
<foo+$8>       >r    1->1
<foo+$10>      drop    1->0
   $7FBC47873933: mov    %r13,-0x8(%r14)
   $7FBC47873937: sub    $0x8,%r14
<foo+$18>      lit    0->1
<foo+$20>      #5
<foo+$28>      fsin    1->1
   $7FBC4787393B: mov    0x20(%rbx),%r13
   $7FBC4787393F: add    $0x28,%rbx
   $7FBC47873943: mov    (%rbx),%rax
   $7FBC47873946: jmp    *%rax
<foo+$30>      ;s    1->1
   $7FBC47873948: mov    (%r14),%rbx
   $7FBC4787394B: add    $0x8,%r14
   $7FBC4787394F: mov    (%rbx),%rax
   $7FBC47873952: jmp    *%rax
```

This shows two sequences: The threaded code at `<foo>...<foo+$30>`, and the dynamically-generated native code at `$7FBC47873933...$7FBC47873952`. These two sequences are shown in an interleaved way, ideally such that we see the threaded code of a primitive or static superinstruction (including immediate arguments and slots for the remaining primitives of a static superinstruction) is followed by the native code for the primitive or static superinstruction. In some cases, Gforth deviates from this ideal, as we will see. But the important thing to remember is that the two sequences are at different places in memory and are not interleaved there. You can easily see that by noticing that there is no room for the native code between the threaded-code addresses, nor is there room for the threaded code between the native-code addresses.

In the example, you first see a threaded-code cell for a static superinstruction with the components `dup` and `>r`, starting and ending with one data stack item in a register (`1->1`); this is followed by the the cell for the `>r` component of the superinstruction; the latter cell is not used, but is there, because static superinstructions are formed only after the threaded-code layout has been determined. Next, you see the threaded code for `drop`, which starts with one data stack item in registers and ends with no data stack items in registers (`1->0`), i.e., all data stack items are in memory.

Next, the dynamically generated native code for the superinstruction `dup >r` is shown.

There is no native code executed for `drop` in the `1->0` case, so no native code is shown for `drop`, and that's also the reason why the threaded code for `drop` is shown before the native

code for `dup >r`; `see-code` is not clever enough to show the `drop` behind that native-code fragment.

If you want to understand the native code shown here: the return-stack pointer is in `%r14`, and the first data stack register is `%r13` (i.e., the top of stack resides there if there is one data stack item in a register). The threaded-code instruction pointer (IP) is in `%rbx` (used by `lit` and `;s` in this example). Note that the register assignments vary between builds, so you may see a different register assignment for this code.

Next, the threaded code for `lit` is shown as well as its stack states (`0->1`), followed by the cell of the immediate argument `5` of `lit`.

Then the threaded code for `fsin` is shown. There is no dynamically-generated native code for `fsin`, and `see-code` does not show the static native code for this word (you can look at it with `see fsin`), that's why no native code is shown for `fsin`, and its threaded code is shown adjacent to the previous threaded code.

The next line is the native code for the `lit`, which is adjacent to the last shown native code. The code for `lit` loads the 5 residing at `<foo+$20>` with an access that uses IP (which still points to `<foo>` at this point) with an offset of `$20` (shown as `0x20`).

Because the (static) native code for `fsin` is elsewhere, code for jumping to that native code has to be generated: First, IP is brought up-to-date with the instruction at `$7FBC4787393F` and after that IP points to `<foo+$28>`. The instruction at `$7FBC47873943` loads the code address of `fsin` from IP, and the next instruction jumps to that code address, executing `fsin`. These two instructions constitute a direct-threaded code dispatch (NEXT1).

Next, we see the threaded code of `;s` (the primitive compiled for `;` in `foo`), and then the dynamically generated native code for `;s`: It loads IP from the return stack and updates the return stack pointer. Because it performs control flow, `;s` has to end in a direct-threaded code dispatch.

6.30.6 Examining data

The following words inspect the stack non-destructively:

`...` (*x1 .. xn* – *x1 .. xn*) gforth-1.0 "dot-dot-dot"

Display the number of items on the data stack, followed by the items (but not more than specified by `maxdepth-.s`; TOS is the right-most item. For each item, and in some cases, for a group of items, `...` tries to guess the type and displays it accordingly (smart `.s`).

`.s` (–) tools "dot-s"

Display the number of items on the data stack, followed by the top items (but not more than specified by `maxdepth-.s`; TOS is the right-most item. Each item is displayed as a number.

`f.s` (–) gforth-0.2 "f-dot-s"

Display the number of items on the floating-point stack, followed by a list of the items (but not more than specified by `maxdepth-.s`; TOS is the right-most item.

`f.s-precision` (– *u*) gforth-1.0 "f-dot-s-precision"

A `value`. *U* is the field width for f.s output. Other precision details are derived from that value.

`maxdepth-.s` (– *addr*) gforth-0.2 "maxdepth-dot-s"

A variable containing 9 by default. `.s` and `f.s` display at most that many stack items.

There is a word `.r` but it does *not* display the return stack! It is used for formatted numeric output (see Section 6.24.1 [Simple numeric output], page 205).

The following words work on the stack as a whole, either by determining the depth or by clearing them:

`depth` (– +*n*) core "depth"

+*n* is the number of values that were on the data stack before +*n* itself was placed on the stack.

`fdepth` (– +*n*) floating "f-depth"

+*n* is the current number of (floating-point) values on the floating-point stack.

`clearstack` (*...* –) gforth-0.2 "clear-stack"

remove and discard all/any items from the data stack.

`fclearstack` (*r0 .. rn* –) gforth-1.0 "f-clearstack"

clear the floating point stack

`clearstacks` (*...* –) gforth-0.7 "clear-stacks"

empty data and FP stack

The following words inspect memory.

`?` (*a-addr* –) tools "question"

Display the contents of address a-*addr* in the current number base.

`dump` (*addr u* –) tools "dump"

Display *u* lines of memory starting at address *addr*. Each line displays the contents of 16 bytes. When Gforth is running under an operating system you may get `Invalid memory address` errors if you attempt to access arbitrary locations.

6.30.7 Forgetting words

Forth allows you to forget words (and everything that was allotted in the dictionary after them) in a LIFO manner.

`marker` ("<*spaces*> *name*" –) core-ext

Create a definition, *name* (called a *mark*) whose execution semantics are to remove itself and everything defined after it.

The most common use of this feature is during program development: when you change a source file, forget all the words it defined and load it again (since you also forget everything defined after the source file was loaded, you have to reload that, too). Note that effects like storing to variables and destroyed system words are not undone when you forget words. With a system like Gforth, that is fast enough at starting up and compiling, I find it more convenient to exit and restart Gforth, as this gives me a clean slate.

Here's an example of using `marker` at the start of a source file that you are debugging; it ensures that you only ever have one copy of the file's definitions compiled at any time:

```
[IFDEF] my-code
    my-code
[THEN]
```

```
marker my-code
init-included-files

\ .. definitions start here
\ .
\ .
\ end
```

6.30.8 Debugging

Languages with a slow edit/compile/link/test development loop tend to require sophisticated tracing/stepping debuggers to facilate debugging.

A much better (faster) way in fast-compiling languages is to add printing code at well-selected places, let the program run, look at the output, see where things went wrong, add more printing code, etc., until the bug is found.

The simple debugging aids provided in `debugs.fs` are meant to support this style of debugging.

The word `~~` prints debugging information (by default the source location and the stack contents). It is easy to insert. If you use Emacs it is also easy to remove (`C-x ~` in the Emacs Forth mode to query-replace them with nothing). The deferred words `printdebugdata` and `.debugline` control the output of `~~`. The default source location output format works well with Emacs' compilation mode, so you can step through the program at the source level using `C-x `` (the advantage over a stepping debugger is that you can step in any direction and you know where the crash has happened or where the strange data has occurred).

`~~` (–) gforth-0.2 "tilde-tilde"

Prints the source code location of the `~~` and the stack contents with `.debugline`.

`printdebugdata` (–) gforth-0.2 "print-debug-data"

`.debugline` (*nfile nline* –) gforth-0.6 "print-debug-line"

Print the source code location indicated by *nfile nline*, and additional debugging information; the default `.debugline` prints the additional information with `printdebugdata`.

`debug-fid` (– *file-id*) gforth-1.0 "*File-id*"

Output file for debugging words. By default it is the process's `stderr`.

`~~` (and assertions) will usually print the wrong file name if a marker is executed in the same file after their occurrence. They will print '`*somewhere*`' as file name if a marker is executed in the same file before their occurrence.

`once` (–) gforth-1.0

do the following up to THEN only once

`~~bt` (–) gforth-1.0 "tilde-tilde-bt"

print stackdump and backtrace

`~~1bt` (–) gforth-1.0 "tilde-tilde-one-bt"

print stackdump and backtrace once

`???` (–) gforth-0.2 "question-question-question"

Open a debuging shell

`WTF??` (–) gforth-1.0 "WTF-question-question"

Open a debugging shell with backtrace and stack dump

`!!FIXME!!` (–) gforth-1.0 “store-store-FIXME-store-store”

word that should never be reached

`replace-word` (*xt1 xt2* –) gforth-1.0

make xt2 do xt1, both need to be colon definitions

`~~Variable` ("*name*" –) gforth-1.0 “tilde-tilde-Variable”

Define *name* and reserve a cell at *addr*.
name execution: `( -- addr )` Show a `~~` tracer with *addr* `@` on the top of the data stack. In all other respects *name* behaves like a normal variable.

`~~Value` (*n* "*name*" –) gforth-1.0 “tilde-tilde-Value”

Define *name* with the initial value *w*; like `value` but with tracers on the `to` and `+to` actions.
name execution: *(– w2)* push the current value of *name*.
`to` *name* run-time: *(w3 –)* Show a `~~` tracer with the xt of *name* on top of *w3*, then change the value of *name* to *w3*.
`+to` *name* run-time: *(n|u –)* Show a `~~` tracer with the xt of *name* on top of *n|u*, then increment the value of *name* by *n|u*.

`+ltrace` (–) gforth-1.0 “plus-ltrace”

turn on line tracing

`-ltrace` (–) gforth-1.0

turn off line tracing

`#loc` (*nline nchar* "*file*" –) gforth-1.0 “number-loc”

set next word’s location to *nline nchar* in "*file*"

6.30.9 Assertions

It is a good idea to make your programs self-checking, especially if you make an assumption that may become invalid during maintenance (for example, that a certain field of a data structure is never zero). Gforth supports *assertions* for this purpose. They are used like this:

```
assert( flag )
```

The code between `assert(` and `)` should compute a flag, that should be true if everything is alright and false otherwise. It should not change anything else on the stack. The overall stack effect of the assertion is `( -- )`. E.g.

```
assert( 1 1 + 2 = ) \ what we learn in school
assert( dup 0<> ) \ assert that the top of stack is not zero
assert( false ) \ this code should not be reached
```

The need for assertions is different at different times. During debugging, we want more checking, in production we sometimes care more for speed. Therefore, assertions can be turned off, i.e., the assertion becomes a comment. Depending on the importance of an assertion and the time it takes to check it, you may want to turn off some assertions and keep others turned on. Gforth provides several levels of assertions for this purpose:

`assert0(` (–) gforth-0.2 “assert-zero”

Important assertions that should always be turned on.

`assert1(` (–) gforth-0.2 “assert-one”

Normal assertions; turned on by default.

`assert2(` (–) gforth-0.2 “assert-two”

Debugging assertions.

`assert3(` (–) gforth-0.2 “assert-three”

Slow assertions that you may not want to turn on in normal debugging; you would turn them on mainly for thorough checking.

`assert(` (–) gforth-0.2 “assert-paren”

Equivalent to `assert1(`

`)` (–) gforth-0.2 “close-paren”

End an assertion. Generic end, can be used for other similar purposes

The variable `assert-level` specifies the highest assertions that are turned on. I.e., at the default `assert-level` of one, `assert0(` and `assert1(` assertions perform checking, while `assert2(` and `assert3(` assertions are treated as comments.

The value of `assert-level` is evaluated at compile-time, not at run-time. Therefore you cannot turn assertions on or off at run-time; you have to set the `assert-level` appropriately before compiling a piece of code. You can compile different pieces of code at different `assert-level`s (e.g., a trusted library at level 1 and newly-written code at level 3).

`assert-level` (– *a-addr*) gforth-0.2

All assertions above this level are turned off.

If an assertion fails, a message compatible with Emacs' compilation mode is produced and the execution is aborted (currently with `ABORT"`. If there is interest, we will introduce a special throw code. But if you intend to `catch` a specific condition, using `throw` is probably more appropriate than an assertion).

Assertions (and `~~`) will usually print the wrong file name if a marker is executed in the same file after their occurrence. They will print ‘`*somewhere*`’ as file name if a marker is executed in the same file before their occurrence.

Definitions in Standard Forth for these assertion words are provided in `compat/assert.fs`.

6.30.10 Singlestep Debugger

The singlestep debugger works only with the engine `gforth-itc`.

When you create a new word there's often the need to check whether it behaves correctly or not. You can do this by typing `dbg badword`. A debug session might look like this:

```
: badword 0 DO i . LOOP ;  ok
2 dbg badword
: badword
Scanning code...

Nesting debugger ready!
```

```
400D4738  8049BC4 0             -> [ 2 ] 00002 00000
400D4740  8049F68 DO            -> [ 0 ]
400D4744  804A0C8 i             -> [ 1 ] 00000
400D4748 400C5E60 .             -> 0 [ 0 ]
400D474C  8049D0C LOOP          -> [ 0 ]
400D4744  804A0C8 i             -> [ 1 ] 00001
400D4748 400C5E60 .             -> 1 [ 0 ]
400D474C  8049D0C LOOP          -> [ 0 ]
400D4758  804B384 ;             ->  ok
```

Each line displayed is one step. You always have to hit return to execute the next word that is displayed. If you don't want to execute the next word in a whole, you have to type *n* for `nest`. Here is an overview what keys are available:

`RET` Next; Execute the next word.

n Nest; Single step through next word.

u Unnest; Stop debugging and execute rest of word. If we got to this word with nest, continue debugging with the calling word.

d Done; Stop debugging and execute rest.

s Stop; Abort immediately.

Debugging large application with this mechanism is very difficult, because you have to nest very deeply into the program before the interesting part begins. This takes a lot of time.

To do it more directly put a `BREAK:` command into your source code. When program execution reaches `BREAK:` the single step debugger is invoked and you have all the features described above.

If you have more than one part to debug it is useful to know where the program has stopped at the moment. You can do this by the `BREAK" string"` command. This behaves like `BREAK:` except that string is typed out when the "breakpoint" is reached.

`dbg` (`"`*name*`"` –) gforth-0.2

`break:` (–) gforth-0.4 "break-colon"

`break"` (*'ccc*`"` *'* –) gforth-0.4 "break-quote"

6.30.11 Code Coverage and Execution Frequency

If you run extensive tests on your code, you often want to figure out if the tests exercise all parts of the code. This is called (test) coverage. The file `coverage.fs` contains tools for measuring the coverage as well as execution frequency.

Code coverage inserts counting code in every basic block (straight-line code sequence) loaded after `coverage.fs`. Each time that code is run, it increments the counter for that basic block. Later you can show the source file with the counts inserted in these basic blocks.

A simple use of the coverage tools is as follows:

```
gforth coverage.fs fib.fs -e "main .coverage bye"
```

This shows the source program annotated with execution counts (and code that is never executed in red). A more sophisticated usgae is:

```
gforth coverage.fs source.fs tests1.fs -e "save-cov bye"
gforth coverage.fs source.fs -e "load-cov" tests2.fs -e "save-cov bye"
gforth coverage.fs source.fs -e "load-cov" tests3.fs -e "bw-cover annotate-cov bye"
```

This runs the tests in three steps, saving and restoring the coverage information from one step to the next. Eventually the execution counts are put in copies of the source file(s), with an appended extension `.cov` (e.g., `source.fs.cov`). The counts are shown in parentheses (due to `bw-cover`) rather than in color (default).

`.coverage` (–) gforth-experimental "dot-coverage"

Show code with execution frequencies.

`annotate-cov` (–) gforth-experimental

For every file with coverage information, produce a `.cov` file that has the execution frequencies inserted. We recommend to use `bw-cover` first (with the default `color-cover` you get escape sequences in the files).

By default, the counts are shown in colour (using ANSI escape sequences), but you can use `bw-cover` to show them in parenthesized form without escape sequences.

`bw-cover` (–) gforth-1.0

Print execution counts in parentheses (source-code compatible).

`color-cover` (–) gforth-1.0

Print execution counts in colours (default).

You can save and reload the coverage counters in binary format, to aggregate coverage counters across several test runs of the same program.

`save-cov` (–) gforth-experimental

Save coverage counters.

`load-cov` (–) gforth-experimental

Load coverage counters.

In addition, there are the following words that one may want to use in special circumstances.

`nocov[` (–) gforth-1.0 "nocov-bracket"

(Immediate) Turn coverage off temporarily.

`]nocov` (–) gforth-1.0 "bracket-nocov"

(Immediate) End of temporary turned off coverage.

`cov+` (–) gforth-experimental "cov-plus"

(Immediate) Place a coverage counter here.

`?cov+` (*flag* – *flag*) gforth-experimental "question-cov-plus"

(Immediate) A coverage counter for a flag; in the coverage output you see three numbers behind `?cov`: The first is the number of executions where the top-of-stack was non-zero; the second is the number of executions where it was zero; the third is the total number of executions.

`cov%` (–) gforth-experimental "cov-percent"

Print the percentage of basic blocks loaded after `coverage.fs` that are executed at least once.

`.cover-raw` (–) gforth-experimental "dot-cover-raw"

Print raw execution counts.

6.30.12 `ans-report.fs`: Report the words used, sorted by wordset

If you want to label a Forth program as Standard Program, you must document which wordsets the program uses.

The `ans-report.fs` tool makes it easy for you to determine which words from which wordset and which non-standard words your application uses. You simply have to include `ans-report.fs` before loading the program you want to check. After loading your program, you can get the report with `print-ans-report`. A typical use is to run this as batch job like this:

```
gforth ans-report.fs myprog.fs -e "print-ans-report bye"
```

The output looks like this (for `compat/control.fs`):

```
The program uses the following words
from CORE :
: POSTPONE THEN ; immediate ?dup IF 0=
from BLOCK-EXT :
\
from FILE :
(
```

`ans-report.fs` reports both Forth-94 and Forth-2012 wordsets. For words that are in both standards, it reports the wordset without suffix (e.g., `CORE-EXT`). For Forth-2012-only words, it reports the wordset with a `-2012` suffix (e.g., `CORE-EXT-2012`); and likewise for the words that are Forth-94-only (i.e., that have been removed in Forth-2012).

6.30.12.1 Caveats

Loading `ans-report.fs` disables the reporting of shadowed words (a quick fix to avoid recording the shadowed words as being used).

Note that `ans-report.fs` just checks which words are used, not whether they are used in a standard-conforming way!

Some words are defined in several wordsets in the standard. `ans-report.fs` reports them for only one of the wordsets, and not necessarily the one you expect. It depends on usage which wordset is the right one to specify. E.g., if you only use the compilation semantics of `S"`, it is a Core word; if you also use its interpretation semantics, it is a File word.

If you load `ans-report.fs` on the OS command line, then enter interactive mode and call `print-ans-report` only afterwards, the word `bootmessage` is reported, most likely spuriously. To avoid that, call `print-ans-report` from the OS command line, or `require ans-report.fs` interactively.

6.30.13 Stack depth changes during interpretation

Sometimes you notice that, after loading a file, there are items left on the stack. The tool `depth-changes.fs` helps you find out quickly where in the file these stack items are coming from.

The simplest way of using `depth-changes.fs` is to include it before the file(s) you want to check, e.g.:

```
gforth depth-changes.fs my-file.fs
```

This will compare the stack depths of the data and FP stack at every empty line (in interpretation state) against these depths at the last empty line (in interpretation state). If the depths are not equal, the position in the file and the stack contents are printed with `~~` (see Section 6.30.8 [Debugging], page 265). This indicates that a stack depth change has occurred in the paragraph of non-empty lines before the indicated line. It is a good idea to leave an empty line at the end of the file, so the last paragraph is checked, too.

Checking only at empty lines usually works well, but sometimes you have big blocks of non-empty lines (e.g., when building a big table), and you want to know where in this block the stack depth changed. You can check all interpreted lines with

```
gforth depth-changes.fs -e "' all-lines is depth-changes-filter" my-file.fs
```

This checks the stack depth at every end-of-line. So the depth change occurred in the line reported by the `~~` (not in the line before).

Note that, while this offers better accuracy in indicating where the stack depth changes, it will often report many intentional stack depth changes (e.g., when an interpreted computation stretches across several lines). You can suppress the checking of some lines by putting backslashes at the end of these lines (not followed by white space), and using

```
gforth depth-changes.fs -e "' most-lines is depth-changes-filter" my-file.fs
```

6.31 Multitasker

Gforth offers two multitaskers: a traditional, cooperative round-robin multitasker (undocumented), and a multitasker based on POSIX threads (pthreads) which allows to run several threads concurrently on multi-core machines (documented here). The pthread-based multitasker is now marked as experimental feature, as standardization of Forth multitaskers will likely result in changes here and there.

6.31.1 Pthreads

Many of the following words are the same as in the traditional cooperative multi-tasker. However, while in the cooperative multi-tasker you can be sure that any sequence of words between two task-switching words (e.g., `pause`, `ms`, maybe I/O words) is atomic, POSIX threads can switch tasks everywhere (pre-emptive multitasking) or (worse) run two tasks at the same time on different cores, often with unintuitive memory ordering by the hardware.

So, with pthreads, you need to use various ways to perform atomic accesses or use some mutual exclusion mechanism when dealing with written-to memory shared between tasks. Alternatively, you can use message queues, which avoid the need to deal with shared memory.

6.31.1.1 Basic multi-tasking

Tasks can be created with `newtask` or `newtask4` with a given amount of stack space (either all the same or each stack's size specified). The *task* returned by the following words is a single cell that serves as reference to the task.

`newtask` (*stacksize* – *task*) gforth-experimental

creates *task*; each stack (data, return, FP, locals) has size *stacksize*.

`task` (*ustacksize* `"`*name*`"` –) gforth-experimental

creates a task *name*; each stack (data, return, FP, locals) has size *ustacksize*.
name execution: (– *task*)

`newtask4` (*u-data u-return u-fp u-locals* – *task*) gforth-experimental "newtask-four"

creates *task* with data stack size *u-data*, return stack size *u-return*, FP stack size *u-fp* and locals stack size *u-locals*.

If you don't know which stack sizes to use for the task, you can use the size(s) of the main task:

`stacksize` (– *u*) gforth-experimental

u is the data stack size of the main task.

`stacksize4` (– *u-data u-return u-fp u-locals*) gforth-experimental "stacksize-four"

Pushes the data, return, FP, and locals stack sizes of the main task.

A task is created in an inactive state. To let it run, you have to activate it with one of the following words:

`initiate` (*xt task* –) gforth-experimental

Let *task* execute *xt*. Upon return from the *xt*, the task terminates itself (VFX compatible). Use one-time executable closures (see `:}h1`, see Section 6.28 [Closures], page 248) to pass arbitrary paramenters to a task.

You can also do creation and initiation in one step:

`execute-task` (*xt* – *task*) gforth-experimental

Create a new task *task* with the same stack sizes as the main task. Let *task* execute *xt*. Upon return from the *xt*, the task terminates itself.

Apart from terminating by running to the end, a task can terminate itself with `kill-task`. Other tasks can terminate it with `kill`.

`kill-task` (–) gforth-experimental

Terminate the current task.

`kill` (*task* –) gforth-experimental

Terminate *task*.

Tasks can also temporarily stop themselves or be stopped:

`stop` (–) gforth-experimental

stops the current task, and waits for events (which may restart it)

`stop-ns` (*timeout* –) gforth-experimental

Stop with timeout (in nanoseconds), better replacement for ms

`stop-dns` (*dtimeout* –) gforth-experimental

Stop with dtimeout (in nanoseconds), better replacement for ms

`thread-deadline` (*d* –) gforth-experimental

stop until absolute time *d* in nanoseconds, base is 1970-1-1 0:00 UTC, but you usually will want to base your deadlines on a time you get with `ntime`.

Using `stop-dns` is easier to code, but if you want your task to wake up at regular intervals rather than some time after it finished its last piece of work, the way to go is to work with deadlines.

`halt` (*task* –) gforth-experimental

`Stop` *task* (no difference from `sleep`)

`sleep` (*task* –) gforth-experimental

`Stop` *task* (no difference from `halt`)

A task restarts when the timeout is over or when another task wakes it with:

`wake` (*task* –) gforth-experimental

Wake *task*

`restart` (*task* –) gforth-experimental

Wake *task* (no difference from `wake`)

There is also:

`pause` (–) gforth-experimental

voluntarily switch to the next waiting task (`pause` is the traditional cooperative task switcher; in the pthread multitasker, you don't need `pause` for cooperation, but you still can use it e.g. when you have to resort to polling for some reason). This also checks for events in the queue.

6.31.1.2 Task-local data

In Forth every task has essentially the same task-local data, called "user" area (early Forth systems were multi-user systems and there often was one user per task). The *task* result of, e.g. `newtask` is the start address of its user area. Each task gets the user data defined by the system (e.g., `base`). You can define additional user data with:

`User` ("*name*" –) gforth-0.2

Name is a user variable (1 cell).
Name execution: (– *addr*)
Addr is the address of the user variable in the current task.

`AUser` ("*name*" –) gforth-0.2

Name is a user variable containing an address (this only makes a difference in the cross-compiler).

`uallot` (*n1* – *n2*) gforth-0.3

Reserve *n1* bytes of user data. *n2* is the offset of the start of the reserved area within the user area.

`UValue` ("*name*" –) gforth-1.0

Name is a user value.
Name execution: (– *x*)

`UDefer` ("*name*" –) gforth-1.0

Name is a task-local deferred word.
Name execution: (... – ...)

There are also the following words for dealing with user data.

`up@` (– *a-addr*) new "up-fetch"

Addr is the start of the user area of the current task (*addr* also serves as the *task* identifier of the current task).

`user'` ("*name*" – *u*) gforth-experimental "user-tick"

U is the offset of the user variable *name* in the user area of each task.

`'s` (*addr1 task* – *addr2*) gforth-experimental "tick-s"

With *addr1* being an address in the user data of the current task, *addr2* is the corresponding address in *task*'s user data.

The pictured numeric output buffer is also task-local, but other areas like dictionary or `PAD` are shared.

6.31.1.3 Mutexes

Pthreads tasks run concurrently, so they need a mechanism to coordinate access to resources (while that is not necessary for cooperative multitasking).

Mutexes can only be locked by one thread at one time, all other threads have to wait until the mutex is unlocked.

`mutex` ("*name*" –) gforth-experimental

create a named mutex *name*, initially unlocked.
name execution: (– *mutex*)

`lock` (*mutex* –) gforth-experimental

If *mutex* is currently locked, wait until it is unlocked. Lock *mutex*.

`unlock` (*mutex* –) gforth-experimental

Unlock *mutex*.

You can ensure that you always unlock a mutex after locking it by using:

`critical-section` (*xt mutex* –) gforth-experimental

If *mutex* is currently locked, wait until it is unlocked. Lock *mutex*, then execute *xt*. After *xt* finishes (regularly, or through an exception), unlock *mutex*.

6.31.1.4 Atomic memory accesses and memory ordering

Atomic hardware operations perform the whole operation, without any other task seeing an intermediate state. These operations can be used to synchronize tasks without using slow OS primitives, but compared to the non-atomic sequences of operations they tend to be slow. Atomic operations only work correctly on aligned addresses, even on hardware that otherwise does not require alignment.

`atomic!@` (*w1 a-addr* – *w2*) gforth-experimental "atomic-store-fetch"

Fetch *w2* from a_*addr*, then store *w1* there, combined into an atomic operation.

`atomic+!@` (*u1 a-addr* – *u2*) gforth-experimental "atomic-plus-store-fetch"

Fetch *w2* from *a_addr*, then increment this location by *u1*. This atomic operation is commonly known as fetch-and-add.

`atomic?!@` (*unew uold a-addr – uprev*) gforth-experimental "atomic-question-store-fetch"

Fetch *uprev* from *a_addr*, compare it to *uold*, and if equal, store *unew* there. This atomic operation is commonly known as compare-and-swap.

There are also the non-atomic `!@` and `+!@` (otherwise the same behaviour, see Section 6.8.5 [Memory Access], page 81).

Another hardware operation is the memory barrier. Unfortunately modern hardware often can reorder memory operations relative to other memory operations (as seen by a different core), and the memory barrier suppresses this reordering for one point in the execution of the task.

`barrier` (–) gforth-experimental "barrier"

All memory operations before the barrier are performed before any memory operation after the barrier.

6.31.1.5 Message queues

Gforth's message queues are a variant of the actor model.

The sending task tells the receiving task to execute an xt with the stack effect `( -- )` (an *event* in the name of the words below; the actor model would call these xts *messages*), and when the receiving task is ready, it will execute the xt, possibly after other xts from its message queue.

The execution order between xts from different tasks is arbitrary, the order between xts from the same task is the sending order.

In many cases you do not just want to pass the xts of existing words, but also parameters. You can construct execute-once closures (defined using `:}h1`, see Section 6.28 [Closures], page 248) to achieve that, e.g., with

```
: .-in-task ( n task -- )
  >r [{: n :}h1 n . ;] r> send-event ;

5 my-task .-in-task \ my-task prints 5
```

`send-event` (*xt task* –) gforth-experimental

Inter-task communication: send *xt* `( -- )` to *task*. *task* executes the xt at some later point in time. To pass parameters, construct a one-shot closure that contains the parameters (see Section 6.28 [Closures], page 248) and pass the xt of that closure.

In order to execute xts received from other tasks, perform one of the following words in the receiving task:

`?events` (–) gforth-experimental "question-events"

Execute all event xts in the current task's message queue, one xt at a time.

`event-loop` (–) gforth-experimental

Wait for event xts and execute these xts when they arrive, one at a time. Return to waiting if no event xts are in the queue. This word never returns.

Alternatively, when a task is `stop`ped, it is also ready for receiving xts, and receiving an xt will not just execute the xt, but also continue execution after the `stop`.

6.31.2 Cilk

Gforth's Cilk is a framework for dividing work between multiple tasks running on several cores, inspired by the programming language of the same name. Use `require cilk.fs` if you want to use Cilk.

The idea is that you identify subproblems that can be solved in parallel, and the framework assigns worker tasks to these subproblems. In particular, you use one of the `spawn` words for each subtask. Eventually you need to wait with `cilk-sync` for the subproblems to be solved.

Currently all the spawning has to happen from one task, and `cilk-sync` waits for all subproblems to complete, so using the current Gforth Cilk for recursive algorithms is not straightforward.

Do not divide the subproblems too finely, in order to avoid overhead; how fine is too fine depends on how uniform the run-time for the subproblems is, but for problems with substantial run-time, having 5*`cores` subproblems is probably a good starting point.

`cores` (– *u*) cilk

A value containing the number of worker tasks to use. By default this is the number of hardware threads (with SMT/HT), if we can determine that, otherwise 1. If you want to use a different number, change `cores` before calling `cilk-init`.

`cilk-init` (–) cilk

Start the worker tasks if not already done.

`spawn` (*xt* –) cilk

Execute *xt* (–) in a worker task. Use one-time executable closures (defined using `:}h1`, see Section 6.28 [Closures], page 248) to pass arbitrary data from the spawner to the code running in the worker.
E.g.: `( n r ) [{: n f: r :}h1 code ;] spawn`

`spawn1` (*x xt* –) cilk "spawn-one"

Execute *xt* (x –) in a worker task.

`spawn2` (*x1 x2 xt* –) cilk "spawn-two"

Execute *xt* (x1 x2 –) in a worker task.

`cilk-sync` (–) cilk

Wait for all subproblems to complete.

`cilk-bye` (–) cilk

Terminate all workers.

6.32 C Interface

Gforth's C interface works by compiling a wrapper library that contains C functions which take parameters from the Forth stacks and calls the C functions. This wrapper library is compiled by the C compiler. Compilation results are cached, so that Gforth only needs to rerun the C compilation if the wrapper library has to change. This build process is automatic, and done at the end of a interface declaration. Gforth uses libtool and GCC for that process.

The C interface is now mostly complete, callbacks have been added, but for structs, we use Forth2012 structs, which don't have independent scopes. The offsets of those structs are extracted from header files with a SWIG plugin.

6.32.1 Existing C interface libraries

Before you can call a C function, you have to declare it in some way. To spare you the effort of writing the declarations yourself (see Section 6.32.3 [Declaring C Functions], page 278), Gforth comes with a number of files that contain definitions for many of the commonly-used C functions available in Unix-based systems. They are typically in the files `/usr/local/share/gforth/1.0/unix` (possibly with `/usr/local` replaced with some other installation directory, such as `/usr`, and `1.0` replaced with the Gforth version).

To find whether a function you are interested in has already been declared, you can search the `.fs` files in that directory. E.g., to find the function `localtime_r()`, you can use

```
grep ' localtime_r ' /usr/local/share/gforth/1.0/unix/*.fs
```

and you will find it defined in `unix/time.fs` with:

```
c-function localtime_r localtime_r a a -- a ( time-addr tmaddr -- tmaddr )
```

Here you see `localtime_r` twice, first as Forth name, then as C name. You can now include that file from Gforth with

```
require unix/time.fs
```

and then call `localtime_r` (the Forth name of the C function) in Forth (see Section 6.32.2 [Calling C Functions], page 277).

The documentation for a C function also applies to the Forth word for the C function.

Some of the files containing the function definitions were written manually (in that case you see a copyright notice at the top), and in some cases contain useful Forth words (that are, however, usually not documented in this manual) in addition to the C functions.

Other files were generated by automatically converting `.h` files. In that case you see a comment "\ This file has been generated [...]" at the top of the file. These files declare Forth words for all the constants, structures, fields and functions that could be converted from the `.h` file.

6.32.2 Calling C functions

Once a C function has been declared (see Section 6.32.1 [Existing C interface libraries], page 277, see Section 6.32.3 [Declaring C Functions], page 278), you can call it as follows: You push the arguments on the stack(s), and then call the word for the C function. The arguments have to be pushed in the same order as the arguments appear in the C documentation (i.e., the first argument is deepest on the stack). Integer and pointer arguments have to be pushed on the data stack, floating-point arguments on the FP stack; these arguments are consumed by the called C function.

On returning from the C function, the return value, if any, resides on the appropriate stack: an integer return value is pushed on the data stack, an FP return value on the FP stack, and a void return value results in not pushing anything. Note that most C functions have a return value, even if that is often not used in C; in Forth, you have to `drop` this return value explicitly if you do not use it.

The C interface automatically converts between the C type and the Forth type as necessary, on a best-effort basis (in some cases, there may be some loss).

As an example, consider the POSIX function `lseek()`:

```
off_t lseek(int fd, off_t offset, int whence);
```

This function takes three integer arguments, and returns an integer argument, so a Forth call for setting the current file offset to the start of the file could look like this:

```
fd @ 0 SEEK_SET lseek -1 = if
  ... \ error handling
then
```

You might be worried that an `off_t` does not fit into a cell, so you could not pass larger offsets to lseek, and might get only a part of the return values. In that case, in your declaration of the function (see Section 6.32.3 [Declaring C Functions], page 278) you should declare it to use double-cells for the off_t argument and return value, and maybe give the resulting Forth word a different name, like `dlseek`; the result could be called like this:

```
fd @ 0. SEEK_SET dlseek -1. d= if
  ... \ error handling
then
```

Passing and returning structs or unions is currently not supported by our interface[27].

Calling functions with a variable number of arguments (*variadic* functions, e.g., `printf()`) is only supported by having you declare one function-calling word for each argument pattern, and calling the appropriate word for the desired pattern.

6.32.3 Declaring C Functions

Before you can call `lseek` or `dlseek`, you have to declare it.

In Gforth 1.0, C function declarations have to be enclosed in a C library definition (see Section 6.32.5 [Defining library interfaces], page 280), e.g.

```
c-library lseek-lib
...
end-c-library
```

The declaration of a C function consists of two parts:

The C part
: is the C declaration of the function, or more typically and portably, a C-style `#include` of a file that contains the declaration of the C function.

The Forth part
: declares the Forth types of the parameters and the Forth word name corresponding to the C function.

For the words `lseek` and `dlseek` mentioned earlier, the declarations are:

```
c-library lseek-lib
\c #define _FILE_OFFSET_BITS 64
\c #include <sys/types.h>
```

[27] If you know the calling convention of your C compiler, you usually can call such functions in some way, but that way is usually not portable between platforms, and sometimes not even between C compilers.

```
\c #include <unistd.h>
c-function lseek lseek n n n -- n
c-function dlseek lseek n d n -- d
0 constant SEEK_SET \ getting the value from the .h files is complicated
end-c-library
```

The C part of the declarations is prefixed by `\c`, and the rest of the line is ordinary C code. You can use as many lines of C declarations as you like, and they are visible for all further function declarations.

The Forth part declares each interface word with `c-function`, followed by the Forth name of the word, the C name of the called function, and the stack effect of the word. The stack effect contains an arbitrary number of types of parameters, then `--`, and then exactly one type for the return value. The possible types are:

`n`	single-cell signed integer
`u`	single-cell unsigned integer
`a`	address (single-cell)
`d`	double-cell signed integer
`ud`	double-cell unsigned integer
`r`	floating-point value
`func`	C function pointer
`void`	no value (used as return type for void functions)
`s`	String: On the Forth side, represented as *c-addr u* pair, on the C side represented as zero-terminated string.
`ws`	wide string: On the Forth side, represented as *c-addr u* pair describing an UTF-8 string; on the C side, represented as a zero-terminated string of `wchar_t` (i.e., UTF-16 or UTF-32, depending on the platform).
`t`	tuple
`0`	NULL pointer (sentinel)
`...`	varargs (programmable)

To deal with variadic C functions, you can declare one Forth word for every pattern you want to use, e.g.:

```
\c #include <stdio.h>
c-function printf-nr printf s n r -- n
c-function printf-rn printf s r n -- n
```

Note that with C functions declared as variadic (or if you don't provide a prototype), the C interface has no C type to convert to, so no automatic conversion happens, which may lead to portability problems in some cases. You can add the C type cast in curly braces after the Forth type. This also allows to pass e.g. structs to C functions, which in Forth cannot live on the stack.

```
c-function printfll printf s n{(long long)} -- n
```

```
c-function pass-struct pass_struct a{*(struct foo *)} -- n
```

This typecasting is not available to return values, as C does not allow typecasts for lvalues.

`\c` ("*rest-of-line*" –) gforth-0.7 "backslash-c"

One line of C declarations for the C interface

`c-function` ("*forth-name*" "*c-name*" "{*type*}" "—" "*type*" –) gforth-0.7

Define a Forth word *forth-name*. *Forth-name* has the specified stack effect and calls the C function `c-name`.

`c-value` ("*forth-name*" "*c-name*" "—" "*type*" –) gforth-1.0

Define a Forth word *forth-name*. *Forth-name* has the specified stack effect and gives the C value of `c-name`.

`c-variable` ("*forth-name*" "*c-name*" –) gforth-1.0

Define a Forth word *forth-name*. *Forth-name* returns the address of `c-name`.

In order to work, this C interface invokes GCC at run-time and uses dynamic linking. If these features are not available, there are other, less convenient and less portable C interfaces in `lib.fs` and `oldlib.fs`. These interfaces are mostly undocumented and mostly incompatible with each other and with the documented C interface; you can find some examples for the `lib.fs` interface in `lib.fs`.

6.32.4 Calling C function pointers from Forth

If you come across a C function pointer (e.g., in some C-constructed structure) and want to call it from your Forth program, you could use the structures as described above by defining a macro. Or you use `c-funptr`.

`c-funptr` ("*forth-name*" <{>"*c-typecast*"<}> "{*type*}" "—" "*type*" –) gforth-1.0

Define a Forth word *forth-name*. *Forth-name* has the specified stack effect plus the called pointer on top of stack, i.e. `( {type} ptr -- type )` and calls the C function pointer `ptr` using the typecast or struct access `c-typecast`.

Let us assume that there is a C function pointer type `func1` defined in some header file `func1.h`, and you know that these functions take one integer argument and return an integer result; and you want to call functions through such pointers. Just define

```
\c #include <func1.h>
c-funptr call-func1 {((func1)ptr)} n -- n
```

and then you can call a function pointed to by, say `func1a` as follows:

```
-5 func1a call-func1 .
```

The Forth word `call-func1` is similar to `execute`, except that it takes a C `func1` pointer instead of a Forth execution token, and it is specific to `func1` pointers. For each type of function pointer you want to call from Forth, you have to define a separate calling word.

6.32.5 Defining library interfaces

As outlines earlier (see Section 6.32.3 [Declaring C Functions], page 278), you have to wrap C function declarations in a C library definition.

The effect of defining a C library is that the names of the generated files will contain that name, and when you use the interface a second time, it will use the existing files instead of

generating and compiling them again, saving you time. The generated file contains a hash of the source code, so changing the declarations will cause a new compilation. Normally these files are cached in `$HOME/.gforth/`*architecture*`/libcc-named`, so if you experience problems or have other reasons to force a recompilation, you can delete the files there.

Note that you should use `c-library` before everything else having anything to do with that library, as it resets some setup stuff. The idea is that the typical use is to put each `c-library`...`end-c-library` unit in its own file, and to be able to include these files in any order. All other words dealing with the C interface are hidden in the vocabulary `c-lib`, which is put on top o the search stack by `c-library` and removed by `end-c-library`.

Note that the library name is not allocated in the dictionary and therefore does not shadow dictionary names. It is used in the file system, so you have to use naming conventions appropriate for file systems. The name is also used as part of the C symbols, but characters outside the legal C symbol names are replaced with underscores. Also, do not call a function you declare after `c-library` before you perform `end-c-library`!

A major benefit of these named library interfaces is that, once they are generated, the tools used to generated them (in particular, the C compiler and libtool) are no longer needed, so the interface can be used even on machines that do not have the tools installed. The build system of Gforth can even cross-compile these libraries, so that the libraries are available for platforms on which build tools aren't installed.

`c-library-name` (*c-addr u* –) gforth-0.7

Start a C library interface with name *c-addr u*.

`c++-library-name` (*c-addr u* –) gforth-1.0 "c-plus-plus–library-name"

Start a C++ library interface with name *c-addr u*.

`c-library` ("*name*" –) gforth-0.7

Parsing version of `c-library-name`

`c++-library` ("*name*" –) gforth-1.0 "c-plus-plus–library"

Parsing version of `c++-library-name`

`end-c-library` (–) gforth-0.7

Finish and (if necessary) build the latest C library interface.

6.32.6 C compiler and linker options

Sometimes you need to pass additional options to the C compiler for compiling or linking a library.

For calling some C functions, you need to link with a specific OS-level library that contains that function. E.g., the `sin` function requires linking a special library by using the command line switch `-lm`. In our C interface you do the equivalent thing by calling `add-lib` as follows:

```
c-library math
s" m" add-lib
\c #include <math.h>
c-function sin sin r -- r
end-c-library
```

The `add-lib` call adds the `-lm` to the linker command line, such that the library `libm.so` (or somesuch) is linked with the new library; you can add as many as you need. Finally

you declare the function as shown above. All words inside the `c-library`...`end-c-library` can use functions from this library (or these libraries).

`add-lib` (*c-addr u* –) gforth-0.7

Add `-l`*string* to the linker call for the current library, where *string* is represented by *c-addr u*. I.e., link with `lib`*string*`.so` or somesuch.

`add-libpath` (*c-addr u* –) gforth-0.7

Add `-L`*string* to the linker call for the current library, where *string* is represented by *c-addr u*. I.e., add *string* to the directories that are searched for C libraries specified with `add-lib`.

`add-framework` (*c-addr u* –) gforth-1.0

Add `-framework` *string* (see Section "Darwin Options" in *Using the GNU Compiler Collection*) to the compiler call for the current library, where *string* is represented by *c-addr u*.

`add-incdir` (*c-addr u* –) gforth-1.0

Add `-I`*string* to the compiler call for the current library, where *string* is represented by *c-addr u*. I.e., add *string* to the directory where the C compiler searches for `.h` files.

`add-cflags` (*c-addr u* –) gforth-1.0

Add the string *c-addr u* to the currently used flags for C compilation.

`add-ldflags` (*c-addr u* –) gforth-1.0

Add the string *c-addr u* to the currently used flags for C linking.

6.32.7 Callbacks

In some cases you have to pass a function pointer to a C function, i.e., the library wants to call back to your application (and the pointed-to function is called a callback function). You can pass the address of an existing C function (that you get with `lib-sym`, see Section 6.32.9 [Low-Level C Interface Words], page 283), but if there is no appropriate C function, you probably want to define the function as a Forth word. Then you need to generate a callback as described below:

Callbacks use a staged approach: First you define a callback instantiator for a certain stack effect; then you use the instantiator to produce a C function pointer from an xt (the prototype of the C function is derived from the Forth stack effect); and finally you can use the resulting C function pointer in a context that needs it.

You can generate a callback instantiator with:

`c-callback` ("*forth-name*" "{*type*}" "—" "*type*" –) gforth-1.0

Define a callback instantiator with the given signature. The callback instantiator *forth-name* `( xt -- addr )` takes an *xt*, and returns the *addr*ess of the C function handling that callback.

`c-callback-thread` ("*forth-name*" "{*type*}" "—" "*type*" –) gforth-1.0

Define a callback instantiator with the given signature. The callback instantiator *forth-name* `( xt -- addr )` takes an *xt*, and returns the *addr*ess of the C function handling that callback. This callback is safe when called from another thread

Each instantiator can be called `callback#` times, and thus can generate that number of callbacks, with the relevant value of `callback#` being the one at the time when the

instantiator is defined. I.e., you can determine that number for an instantiator by changing the value `callback#` before defining the instantiator. In other words, you need to follow a static allocation discipline for the callbacks.

`callback#` (– *n*) gforth-experimental "callback-number"

N is the number of instantiations that the next callback instantiator can create. This number can be changed with `to callback#`, and affects the callback instantiators defined after this change.

Here is an example of a callback use with all stages:

```
c-library callback-test
\c typedef void (*rr2v)(double,double);
\c void testrr2v(rr2v f) {f(0.5,1.5);} /* function that calls a callback function f */
c-function testrr2v testrr2v a -- void
c-callback rr2v-callback r r -- void   \ define callback instantiator
end-c-library
: printrr fswap f. f. ;
' printrr rr2v-callback constant printrr-c \ generate a callback instance
printrr-c testrr2v \ use the callback function pointer by passing it to testrr2v
```

6.32.8 How the C interface works

For every Forth word declared with `c-function`, the C interface generates a wrapper function in C that takes the Forth data from the Forth stacks, and calls the target C function with these data as arguments. After the C function returns, the return value is pushed on the appropriate stack.

During this process, type conversion may happen between the Forth type and the C type of an argument or return value. E.g. on a 64-bit platform, if the Forth argument is (64-bit) `n`, and the C argument is (32-bit) int, the `n` will be cast into the int (chopping off the 32 most-significant bits), while for a return value the same relation will lead into a sign-extension of the 32-bit int into a 64-bit `n`.

The `\c` lines are literally included in the C code (but without the `\c`), and provide the necessary declarations so that the C compiler knows the C types and has enough information to perform the conversion.

These wrapper functions are eventually compiled and dynamically linked into Gforth, and then they can be called.

The libraries added with `add-lib` are used in the compile command line to specify dependent libraries with `-l`*lib*, causing these libraries to be dynamically linked when the wrapper function is linked.

6.32.9 Low-Level C Interface Words

`open-lib` (*c-addr1 u1* – *c-addr2*) gforth-0.4 "open-lib"

Open the (typically shared) library with the name given by *c-addr1 u1* and return the handle *c-addr2* (to be used by `lib-sym`).

`lib-sym` (*c-addr1 u1 c-addr2* – *c-addr3*) gforth-0.4 "lib-sym"

C-addr3 is the address of the symbol *c-addr1 u1* in the library *c_addr2* (opened with `open-lib`). If the library does not contain the symbol, *c-addr3* is 0.

`lib-error` (– *c-addr u*) gforth-0.7 "lib-error"

Error message for last failed `open-lib` or `lib-sym`.

`call-c` (... *w* – ...) gforth-0.2 "call-c"

Call the C function pointed to by *w*. The C function has to access the stack itself. The stack pointers are exported into a ptrpair structure passed to the C function, and returned in that form.

6.32.10 Automated interface generation using SWIG

SWIG, the Simple Wrapper Interface Generator, is used to create C interfaces for a lot of programming languages. The SWIG version extended with a Forth module can be found on github (`https://github.com/GeraldWodni/swig`).

6.32.10.1 Basic operation

C-headers are parsed and converted to Forth-Sourcecode which uses the previously describe C interface functions.

6.32.10.2 Detailed operation:

1. Select a target, in this example we are using `example.h`
2. Create an interface file for the header. This can be used to pass options, switches and define variables. In the simplest case it just instructs to translate all of `example.h`:

```
%module example
%insert("include")
{
    #include "example.h"
}
%include "example.h"
```

3. Use SWIG to create a `.fsi-c` file.
   `swig -forth -stackcomments -use-structs -enumcomments -o example-fsi.c example.i`.
   FSI stands "Forth Source Independent" meaning it can be transferred to any host having a C-compiler. SWIG is not required past this point.
4. On the target machine compile the `.fsi-c` file to a `.fsx` (x stands for executable)
   `gcc -o example.fsx example-fsi.c`
   The compilation will resolve all constants to the values on the target.
5. The last step is to run the executable and capture its output to a `.fs` "Forth Source" file.
   `./example.fsx -gforth > example.fs`
   This code can now be used on the target platform.

6.32.10.3 Examples

You can find some examples in SWIG's Forth Example section (`https://github.com/GeraldWodni/swig/tree/master/Examples/forth`).

A lot of interface files can be found in Forth Posix C-Interface (`https://github.com/GeraldWodni/posix`) and Forth C-Interface Modules (`https://github.com/GeraldWodni/forth-c-interfaces`).

Contribution to the Forth C-Interface Module repository (`https://github.com/GeraldWodni/forth-c-interfaces`) is always welcome.

6.32.11 Migrating from Gforth 0.7

In Gforth 1.0 and following, you can use `\c`, `c-function` and `add-lib` only inside `c-library`...`end-c-library`. `add-lib` now always starts from a clean slate inside a `c-library`, so you don't need to use `clear-libs` in most cases.

If you have a program that uses these words outside `c-library`...`end-c-library`, just wrap them in `c-library`...`end-c-library`. You may have to add some instances of `add-lib`, however.

6.33 Assembler and Code Words

6.33.1 Definitions in assembly language

Gforth provides ways to implement words in assembly language (using `abi-code`...`end-code`), and also ways to define defining words that specify the run-time behaviour of the defined words in assembly language (using `;abi-code`, similar to `does>`, but with assembly language after `;abi-code`).

However, the machine-independent nature of Gforth poses a few problems: First of all, Gforth runs on several architectures, so it can provide no standard assembler. It does provide assemblers for several of the architectures it runs on, though. Moreover, you can use a system-independent assembler in Gforth, or compile machine code directly with `,` and `c,`.

A problem with the traditional words `code` and `;code` is that the virtual machine registers of Gforth (the stack pointers and the virtual machine instruction pointer) depend on the installation and engine. Also, which registers are free to use also depends on the installation and engine. So any code written to run in the context of the Gforth virtual machine is essentially limited to the installation and engine it was developed for (it may run elsewhere, but you cannot rely on that).

Fortunately, you can define `abi-code` and `;abi-code` words in Gforth that are portable to any Gforth running on a platform with the same calling convention (ABI); typically this means portability to the same architecture/OS combination, sometimes crossing OS boundaries).

`assembler` (–) tools-ext

A vocubulary: Replaces the wordlist at the top of the search order with the assembler wordlist.

`init-asm` (–) gforth-0.2

Pushes the assembler wordlist on the search order.

`abi-code` ("*name*" – *colon-sys*) gforth-1.0 “abi-code”

Start a native code definition that is called using the platform's ABI conventions corresponding to the C-prototype:

```
Cell *function(Cell *sp, Float **fpp);
```

The FP stack pointer is passed in by providing a reference to a memory location containing the FP stack pointer and is passed out by storing the changed FP stack pointer there (if necessary).

`;abi-code` (–) gforth-1.0 "semicolon-abi-code"

Ends the colon definition, but at run-time also changes the last defined word *X* (which must be a `create`d word) to call the following native code using the platform's ABI convention corresponding to the C prototype:

```
Cell *function(Cell *sp, Float **fpp, Address body);
```

The FP stack pointer is passed in by providing a reference to a memory location containing the FP stack pointer and is passed out by storing the changed FP stack pointer there (if necessary). The parameter *body* is the body of *X*.

`end-code` (*colon-sys* –) gforth-0.2 "end-code"

End a code definition. Note that you have to assemble the return from the ABI call (for `abi-code`) or the dispatch to the next VM instruction (for `code` and `;code`) yourself.

`code` ("*name*" – *colon-sys*) tools-ext

Start a native code definition that runs in the context of the Gforth virtual machine (engine). Such a definition is not portable between Gforth installations, so we recommend using `abi-code` instead of `code`. You have to end a `code` definition with a dispatch to the next virtual machine instruction.

`;code` (*compilation. colon-sys1* – *colon-sys2*) tools-ext "semicolon-code"

The code after `;code` becomes the behaviour of the last defined word (which must be a `create`d word). The same caveats apply as for `code`, so we recommend using `;abi-code` instead.

`flush-icache` (*c-addr u* –) gforth-0.2 "flush-icache"

Make sure that the instruction cache of the processor (if there is one) does not contain stale data at *c-addr* and *u* bytes afterwards. `END-CODE` performs a `flush-icache` automatically. Caveat: `flush-icache` might not work on your installation; this is usually the case if direct threading is not supported on your machine (take a look at your `machine.h`) and your machine has a separate instruction cache. In such cases, `flush-icache` does nothing instead of flushing the instruction cache.

If `flush-icache` does not work correctly, `abi-code` words etc. will not work (reliably), either.

The typical usage of these words can be shown most easily by analogy to the equivalent high-level defining words:

```
: foo                                abi-code foo
   <high-level Forth words>                <assembler>
;                                    end-code

: bar                                : bar
   <high-level Forth words>             <high-level Forth words>
   CREATE                               CREATE
      <high-level Forth words>             <high-level Forth words>
   DOES>                                ;abi-code
```

```
        <high-level Forth words>           <assembler>
;                                       end-code
```

For using `abi-code`, take a look at the ABI documentation of your platform to see how the parameters are passed (so you know where you get the stack pointers) and how the return value is passed (so you know where the data stack pointer is returned). The ABI documentation also tells you which registers are saved by the caller (caller-saved), so you are free to destroy them in your code, and which registers have to be preserved by the called word (callee-saved), so you have to save them before using them, and restore them afterwards. For some architectures and OSs we give short summaries of the parts of the calling convention in the appropriate sections. More reverse-engineering oriented people can also find out about the passing and returning of the stack pointers through `see abi-call`.

Most ABIs pass the parameters through registers, but some (in particular the most common 386 (aka IA-32) calling conventions) pass them on the architectural stack. The common ABIs all pass the return value in a register.

Other things you need to know for using `abi-code` is that both the data and the FP stack grow downwards (towards lower addresses) in Gforth, with `1 cells` size per cell, and `1 floats` size per FP value.

Here's an example of using `abi-code` on the 386 architecture:

```
abi-code my+ ( n1 n2 -- n )
4 sp d) ax mov \ sp into return reg
ax )    cx mov \ tos
4 #     ax add \ update sp (pop)
cx    ax ) add \ sec = sec+tos
ret            \ return from my+
end-code
```

An AMD64 variant of this example can be found in Section 6.33.6 [AMD64 Assembler], page 291.

Here's a 386 example that deals with FP values:

```
abi-code my-f+ ( r1 r2 -- r )
8 sp d) cx mov  \ load address of fp
cx )    dx mov  \ load fp
.fl dx )   fld  \ r2
8 #     dx add  \ update fp
.fl dx )   fadd \ r1+r2
.fl dx )   fstp \ store r
dx    cx ) mov  \ store new fp
4 sp d) ax mov  \ sp into return reg
ret             \ return from my-f+
end-code
```

And an AMD64 example of using `;abi-code`:

```
: myconst ( n "name" -- )
  create ,
;abi-code ( -- n )
  \ data stack pointer in rdi, body address of "name" in rdx
  -8 di d) ax lea \ rax = rdi-8 \ compute new data stack pointer
```

```
  dx ) cx mov     \ rcx = *rdx  \ get the value of the constant
  cx ax ) mov     \ *rax = rcx  \ and store it on the data stack
  \ data stack pointer in rax
  ret
end-code
5 myconst x
x .
```

6.33.2 Common Assembler

The assemblers in Gforth generally use a postfix syntax, i.e., the instruction name follows the operands.

The operands are passed in the usual order (the same that is used in the manual of the architecture). Since they all are Forth words, they have to be separated by spaces; you can also use Forth words to compute the operands.

The instruction names usually end with a `,`. This makes it easier to visually separate instructions if you put several of them on one line; it also avoids shadowing other Forth words (e.g., `and`).

Registers are usually specified by number; e.g., (decimal) `11` specifies registers R11 and F11 on the Alpha architecture (which one, depends on the instruction). The usual names are also available, e.g., `s2` for R11 on Alpha.

Control flow is specified similar to normal Forth code (see Section 6.10.6 [Arbitrary control structures], page 112), with `if,`, `ahead,`, `then,`, `begin,`, `until,`, `again,`, `cs-roll`, `cs-pick`, `else,`, `while,`, and `repeat,`. The conditions are specified in a way specific to each assembler.

The rest of this section is of interest mainly for those who want to define `code` words (instead of the more portable `abi-code` words).

Note that the register assignments of the Gforth engine can change between Gforth versions, or even between different compilations of the same Gforth version (e.g., if you use a different GCC version). If you are using `CODE` instead of `ABI-CODE`, and you want to refer to Gforth's registers (e.g., the stack pointer or TOS), I recommend defining your own words for referring to these registers, and using them later on; then you can adapt to a changed register assignment.

The most common use of these registers is to end a `code` definition with a dispatch to the next word (the `next` routine). A portable way to do this is to jump to `' noop >code-address` (of course, this is less efficient than integrating the `next` code and scheduling it well). When using `ABI-CODE`, you can just assemble a normal subroutine return (but make sure you return the data stack pointer).

Another difference between Gforth versions is that the top of stack is kept in memory in `gforth` and, on most platforms, in a register in `gforth-fast`. For `ABI-CODE` definitions, any stack caching registers are guaranteed to be flushed to the stack, allowing you to reliably access the top of stack in memory.

6.33.3 Common Disassembler

You can disassemble a `code` word with `see` (see Section 6.30.8 [Debugging], page 265). You can disassemble a section of memory with

`discode` (*addr u* –) gforth-0.2

hook for the disassembler: disassemble u bytes of code at addr

By default, `discode` calls one of the following words (not all of them are available on all installations):

- `disasm2`, which calls `libopcodes` from the binutils. This is usually the default disassembler, if Gforth is built as we recommend.
- `disasm`, a manually written disassembler (only available on some architectures). This is often the default disassembler if `disasm2` is not available.

  The Forth disassembler generally produces output that can be fed into the assembler (i.e., same syntax, etc.). It also includes additional information in comments. In particular, the address of the instruction is given in a comment before the instruction.
- `disasm-gdb`, which calls `gdb` to disassemble code. Unfortunately, this is not a very reliable disassembler, and can hang. Use *`Ctrl-c`* if that happens. This disassembler is used by default if neither `disasm2` nor `disasm` is available.
- `dump` shows the machine code as hex dump. This disassembler is used if none of the others are available.

You can override the default by changing `discode`, e.g., with

```
' disasm is discode
```

`See` may display more or less than the actual code of a `code` word, because the recognition of the end of the code is unreliable. You can use `discode` if it did not display enough. `See` may display more, if the code word is not immediately followed by a named word. If you have something else there, you can follow the word with `align latest ,` to ensure that the end is recognized.

`disasm2` (*c-addr u* –) gforth-1.0 “disasm-two”

Disassemble code block starting at *c-addr* with *u* bytes length using `libopcodes` from GNU binutils. This word is only defined if Gforth was built with `libopcodes` support.

`intel-syntax` (–) gforth-1.0

Change `disasm2` to output Intel syntax. This word is only defined on AMD64 and IA-32 machines and only if Gforth was built with `libopcodes` support.

`at&t-syntax` (–) gforth-1.0 “at-and-t-syntax”

Change `disasm2` to output AT&T syntax. This word is only defined on AMD64 and IA-32 machines and only if Gforth was built with `libopcodes` support.

`disasm` (*c-addr u* –) gforth-0.5

Disassemble code block starting at *c-addr* with *u* bytes length using a hand-written disassembler. This word is only defined on architectures for which a hand-written disassembler exists.

`disasm-gdb` (*addr u* –) gforth-0.7

Disassemble code block starting at *c-addr* with *u* bytes length by calling `gdb`. This word works only if `gdb` is installed, can attach to its parent, and uses the right syntax. If it hangs, use *`Ctrl-c`*.

6.33.4 386 Assembler

The 386 assembler included in Gforth was written by Bernd Paysan, it's available under GPL, and originally part of bigFORTH.

The 386 disassembler included in Gforth was written by Andrew McKewan and is in the public domain.

The disassembler displays code in an Intel-like prefix syntax.

The assembler uses a postfix syntax with AT&T-style parameter order (i.e., destination last).

The assembler includes all instruction of the Athlon, i.e. 486 core instructions, Pentium and PPro extensions, floating point, MMX, 3Dnow!, but not ISSE. It's an integrated 16- and 32-bit assembler. Default is 32 bit, you can switch to 16 bit with .86 and back to 32 bit with .386.

There are several prefixes to switch between different operation sizes, `.b` for byte accesses, `.w` for word accesses, `.d` for double-word accesses. Addressing modes can be switched with `.wa` for 16 bit addresses, and `.da` for 32 bit addresses. You don't need a prefix for byte register names (`AL` et al).

For floating point operations, the prefixes are `.fs` (IEEE single), `.fl` (IEEE double), `.fx` (extended), `.fw` (word), `.fd` (double-word), and `.fq` (quad-word). The default is `.fx`, so you need to specify `.fl` explicitly when dealing with Gforth FP values.

The MMX opcodes don't have size prefixes, they are spelled out like in the Intel assembler. Instead of move from and to memory, there are PLDQ/PLDD and PSTQ/PSTD.

The registers lack the 'e' prefix; even in 32 bit mode, eax is called ax. Immediate values are indicated by postfixing them with `#`, e.g., `3 #`. Here are some examples of addressing modes in various syntaxes:

```
Gforth          Intel (NASM)   AT&T (gas)      Name
.w ax           ax             %ax             register (16 bit)
ax              eax            %eax            register (32 bit)
3 #             offset 3       $3              immediate
1000 #)         byte ptr 1000  1000            displacement
bx )            [ebx]          (%ebx)          base
100 di d)       100[edi]       100(%edi)       base+displacement
20 ax *4 i#)    20[eax*4]      20(,%eax,4)     (index*scale)+displacement
di ax *4 i)     [edi][eax*4]   (%edi,%eax,4)   base+(index*scale)
4 bx cx di)     4[ebx][ecx]    4(%ebx,%ecx)    base+index+displacement
12 sp ax *2 di) 12[esp][eax*2] 12(%esp,%eax,2) base+(index*scale)+displacement
```

You can use `L)` and `LI)` instead of `D)` and `DI)` to enforce 32-bit displacement fields (useful for later patching).

Some example of instructions are:

```
ax bx mov             \ move ebx,eax
3 # ax mov            \ mov eax,3
100 di d) ax mov      \ mov eax,100[edi]
4 bx cx di) ax mov    \ mov eax,4[ebx][ecx]
.w ax bx mov          \ mov bx,ax
```

The following forms are supported for binary instructions:

```
<reg> <reg> <inst>
<n> # <reg> <inst>
<mem> <reg> <inst>
<reg> <mem> <inst>
<n> # <mem> <inst>
```

The shift/rotate syntax is:

```
<reg/mem> 1 # shl \ shortens to shift without immediate
<reg/mem> 4 # shl
<reg/mem> cl shl
```

Precede string instructions (`movs` etc.) with `.b` to get the byte version.

The control structure words `IF UNTIL` etc. must be preceded by one of these conditions: `vs vc u< u>= 0= 0<> u<= u> 0< 0>= ps pc < >= <= >`. (Note that most of these words shadow some Forth words when `assembler` is in front of `forth` in the search path, e.g., in `code` words). Currently the control structure words use one stack item, so you have to use `roll` instead of `cs-roll` to shuffle them (you can also use `swap` etc.).

Based on the Intel ABI (used in Linux), `abi-code` words can find the data stack pointer at `4 sp d)`, and the address of the FP stack pointer at `8 sp d)`; the data stack pointer is returned in `ax`; `Ax`, `cx`, and `dx` are caller-saved, so you do not need to preserve their values inside the word. You can return from the word with `ret`, the parameters are cleaned up by the caller.

For examples of 386 `abi-code` words, see Section 6.33.1 [Assembler Definitions], page 285.

6.33.5 Alpha Assembler

The Alpha assembler and disassembler were originally written by Bernd Thallner.

The register names `a0`–`a5` are not available to avoid shadowing hex numbers.

Immediate forms of arithmetic instructions are distinguished by a `#` just before the `,`, e.g., `and#,` (note: `lda,` does not count as arithmetic instruction).

You have to specify all operands to an instruction, even those that other assemblers consider optional, e.g., the destination register for `br,`, or the destination register and hint for `jmp,`.

You can specify conditions for `if,` by removing the first `b` and the trailing `,` from a branch with a corresponding name; e.g.,

```
11 fgt if, \ if F11>0e
  ...
endif,
```

`fbgt,` gives `fgt`.

6.33.6 AMD64 (x86_64) Assembler

The AMD64 assembler is a slightly modified version of the 386 assembler, and as such shares most of the syntax. Two new prefixes, `.q` and `.qa`, are provided to select 64-bit operand and address sizes respectively. 64-bit sizes are the default, so normally you only have to use the other prefixes. Also there are additional register operands `R8-R15`.

The registers lack the 'e' or 'r' prefix; even in 64 bit mode, `rax` is called `ax`. Additional register operands are available to refer to the lowest-significant byte of all registers: `R8L`-`R15L`, `SPL`, `BPL`, `SIL`, `DIL`.

The Linux-AMD64 calling convention is to pass the first 6 integer parameters in rdi, rsi, rdx, rcx, r8 and r9 and to return the result in rax and rdx; to pass the first 8 FP parameters in xmm0–xmm7 and to return FP results in xmm0–xmm1. So `abi-code` words get the data stack pointer in `di` and the address of the FP stack pointer in `si`, and return the data stack pointer in `ax`. The other caller-saved registers are: r10, r11, xmm8-xmm15. This calling convention reportedly is also used in other non-Microsoft OSs.

Windows x64 passes the first four integer parameters in rcx, rdx, r8 and r9 and return the integer result in rax. The other caller-saved registers are r10 and r11.

On the Linux platform, according to `https://uclibc.org/docs/psABI-x86_64.pdf` page 21 the registers AX CX DX SI DI R8 R9 R10 R11 are available for scratch.

The addressing modes for the AMD64 are:

```
\ running word A produces a memory error as the registers are not initialised ;-)
ABI-CODE A  ( -- )
    500        #               AX  MOV     \ immediate
        DX              AX  MOV     \ register
        200             AX  MOV     \ direct addressing
        DX  )           AX  MOV     \ indirect addressing
    40  DX  D)          AX  MOV     \ base with displacement
        DX  CX      I)  AX  MOV     \ scaled index
        DX  CX  *4  I)  AX  MOV     \ scaled index
    40  DX  CX  *4  DI) AX  MOV     \ scaled index with displacement

        DI              AX  MOV     \ SP Out := SP in
                            RET
END-CODE
```

Here are a few examples of an AMD64 `abi-code` words:

```
abi-code my+  ( n1 n2 -- n3 )
\ SP passed in di, returned in ax,  address of FP passed in si
8 di d) ax lea        \ compute new sp in result reg
di )    dx mov        \ get old tos
dx    ax ) add        \ add to new tos
ret
end-code

\ Do nothing
ABI-CODE aNOP  ( -- )
       DI  )       AX      LEA          \ SP out := SP in
                           RET
END-CODE

\ Drop TOS
ABI-CODE aDROP  ( n -- )
   8   DI  D)      AX      LEA          \ SPout := SPin - 1
                           RET
```

```
END-CODE
\ Push 5 on the data stack
ABI-CODE aFIVE   ( -- 5 )
   -8  DI  D)      AX      LEA          \ SPout := SPin + 1
   5   #           AX  )   MOV          \ TOS := 5
                           RET
END-CODE
\ Push 10 and 20 into data stack
ABI-CODE aTOS2  ( -- n n )
   -16 DI  D)      AX      LEA          \ SPout := SPin + 2
   10  #       8   AX  D)  MOV          \ TOS - 1 := 10
   20  #           AX  )   MOV          \ TOS := 20
                           RET
END-CODE
\ Get Time Stamp Counter as two 32 bit integers
\ The TSC is incremented every CPU clock pulse
ABI-CODE aRDTSC   ( -- TSCl TSCh )
                           RDTSC        \ DX:AX := TSC
   $FFFFFFFF #     AX      AND          \ Clear upper 32 bit AX
  0xFFFFFFFF #     DX      AND          \ Clear upper 32 bit DX
       AX          R8      MOV          \ Temporarily save AX
   -16 DI  D)      AX      LEA          \ SPout := SPin + 2
       R8      8   AX  D)  MOV          \ TOS-1 := saved AX = TSC low
       DX          AX  )   MOV          \ TOS := Dx = TSC high
                           RET
END-CODE
\ Get Time Stamp Counter as 64 bit integer
ABI-CODE RDTSC   ( -- TSC )
                           RDTSC        \ DX:AX := TSC
   $FFFFFFFF #     AX      AND          \ Clear upper 32 bit AX
   32  #           DX      SHL          \ Move lower 32 bit DX to upper 32 bit
       AX          DX      OR           \ Combine AX with DX in DX
   -8  DI  D)      AX      LEA          \ SPout := SPin + 1
       DX          AX  )   MOV          \ TOS := DX
                           RET
END-CODE
VARIABLE V

\ Assign 4 to variable V
ABI-CODE V=4 ( -- )
       BX                  PUSH         \ Save BX, used by gforth
   V   #           BX      MOV          \ BX := address of V
   4   #           BX )    MOV          \ Write 4 to V
       BX                  POP          \ Restore BX
       DI  )       AX      LEA          \ SPout := SPin
                           RET
```

```
END-CODE
VARIABLE V

\ Assign 5 to variable V
ABI-CODE V=5 ( -- )
   V   #           CX      MOV          \ CX := address of V
   5   #           CX )    MOV          \ Write 5 to V
   DI )            AX      LEA          \ SPout := SPin
                           RET
END-CODE
ABI-CODE TEST2  ( -- n n )
   -16 DI  D)  AX          LEA          \ SPout := SPin + 2
   5   #       CX          MOV          \ CX := 5
   5   #       CX          CMP
   0= IF
       1   #   8   AX  D)      MOV      \ If CX = 5 then TOS - 1 := 1  <--
   ELSE
       2   #   8   AX  D)      MOV      \ else TOS - 1 := 2
   THEN
   6   #       CX          CMP
   0= IF
       3   #       AX  )       MOV      \ If CX = 6 then TOS := 3
   ELSE
       4   #       AX  )       MOV      \ else TOS := 4  <--
   THEN
                           RET
END-CODE
\ Do four loops. Expect : ( 4 3 2 1 -- )
ABI-CODE LOOP4  ( -- n n n n )
       DI          AX      MOV          \ SPout := SPin
   4   #           DX      MOV          \ DX := 4  loop counter
   BEGIN
       8   #           AX      SUB      \ SP := SP + 1
           DX          AX  )   MOV      \ TOS := DX
       1   #           DX      SUB      \ DX := DX - 1
   0= UNTIL
                           RET
END-CODE
```

Here's a AMD64 example that deals with FP values:

```
abi-code my-f+  ( r1 r2 -- r )
\ SP passed in di, returned in ax,  address of FP passed in si
si )       dx mov         \ load fp
8 dx d)  xmm0 movsd       \ r2
dx )     xmm0 addsd       \ r1+r2
xmm0  8 dx d) movsd       \ store r
8 #      si ) add         \ update fp
```

```
di          ax mov          \ sp into return reg
ret
end-code
```

6.33.7 ARM A32 assembler

Source code: `arch/arm/asm.fs`

The ARM A32 assembler includes all instruction of ARM architecture version 4, and the BLX instruction from architecture 5. It does not have support for T32 (Thumb) instructions. It also lacks support for any co-processors.

The assembler uses a postfix syntax with the same operand order as used in the ARM Architecture Reference Manual. Mnemonics are suffixed by a comma.

Registers are specified by their names `r0` through `r15`, with the aliases `pc`, `lr`, `sp`, `ip` and `fp` provided for convenience. Note that `ip` refers to the"intra procedure call scratch register" (`r12`) and does not refer to an instruction pointer. `sp` refers to the ARM ABI stack pointer (`r13`) and not the Forth stack pointer.

Condition codes can be specified anywhere in the instruction, but will be most readable if specified just in front of the mnemonic. The 'S' flag is not a separate word, but encoded into instruction mnemonics, ie. just use `adds,` instead of `add,` if you want the status register to be updated.

The following table lists the syntax of operands for general instructions:

Gforth	normal assembler	description
123 #	#123	immediate
r12	r12	register
r12 4 #LSL	r12, LSL #4	shift left by immediate
r12 r1 LSL	r12, LSL r1	shift left by register
r12 4 #LSR	r12, LSR #4	shift right by immediate
r12 r1 LSR	r12, LSR r1	shift right by register
r12 4 #ASR	r12, ASR #4	arithmetic shift right
r12 r1 ASR	r12, ASR r1	... by register
r12 4 #ROR	r12, ROR #4	rotate right by immediate
r12 r1 ROR	r12, ROR r1	... by register
r12 RRX	r12, RRX	rotate right with extend by 1

Memory operand syntax is listed in this table:

Gforth	normal assembler	description
r4]	[r4]	register
r4 4 #]	[r4, #+4]	register with immediate offset
r4 -4 #]	[r4, #-4]	with negative offset
r4 r1 +]	[r4, +r1]	register with register offset
r4 r1 -]	[r4, -r1]	with negated register offset
r4 r1 2 #LSL -]	[r4, -r1, LSL #2]	with negated and shifted offset
r4 4 #]!	[r4, #+4]!	immediate preincrement
r4 r1 +]!	[r4, +r1]!	register preincrement
r4 r1 -]!	[r4, +r1]!	register predecrement
r4 r1 2 #LSL +]!	[r4, +r1, LSL #2]!	shifted preincrement
r4 -4]#	[r4], #-4	immediate postdecrement

```
r4 r1 ]+          [r4], r1              register postincrement
r4 r1 ]-          [r4], -r1             register postdecrement
r4 r1 2 #LSL ]-   [r4], -r1, LSL #2     shifted postdecrement
' xyz >body [#]   xyz                   PC-relative addressing
```

Register lists for load/store multiple instructions are started and terminated by using the words `{` and `}` respectively. Between braces, register names can be listed one by one or register ranges can be formed by using the postfix operator `r-r`. The `^` flag is not encoded in the register list operand, but instead directly encoded into the instruction mnemonic, ie. use `^ldm,` and `^stm,`.

Addressing modes for load/store multiple are not encoded as instruction suffixes, but instead specified like an addressing mode, Use one of `DA`, `IA`, `DB`, `IB`, `DA!`, `IA!`, `DB!` or `IB!`.

The following table gives some examples:

```
Gforth                          normal assembler
r4 ia  { r0 r7 r8 }  stm,       stmia    r4, {r0,r7,r8}
r4 db!  { r0 r7 r8 }  ldm,      ldmdb    r4!, {r0,r7,r8}
sp ia!  { r0 r15 r-r }  ^ldm,   ldmfd    sp!, {r0-r15}^
```

Control structure words typical for Forth assemblers are available: `if, ahead, then, else, begin, until, again, while, repeat, repeat-until,`. Conditions are specified in front of these words:

```
r1 r2 cmp,    \ compare r1 and r2
eq if,        \ equal?
   ...          \ code executed if r1 == r2
then,
```

Example of a definition using the ARM assembler:

```
abi-code my+ ( n1 n2 --  n3 )
   \ arm abi: r0=SP, r1=&FP, r2,r3,r12 saved by caller
   r0 IA!  { r2 r3 }  ldm,     \ pop r2 = n2, r3 = n1
   r3  r2  r3         add,     \ r3 = n1+n1
   r3  r0 -4 #]!      str,     \ push r3
   pc  lr             mov,     \ return to caller, new SP in r0
end-code
```

6.33.8 LoongArch assembler

Source code: `arch/loongarch64/asm.fs`. You can find the list of instructions (and their formats) in `arch/loongarch64/insts.fs`.

Registers are `r0..r31`, `f0..f31`, `fcsr0...fcsr31`, `fcc0..fcc7` and `?caf ?saf ?clt ?slt ?ceq ?seq ?cle ?sle ?cun ?sun ?cult ?sult ?cueq ?sueq ?cule ?sule ?cne ?sne ?cor ?sor ?cune ?sune`.

Mnemonics don't end with ",", so you need to worry about collisions with Forth word names (e.g, `and`). To use immediates, you need to use the immediate name (e.g., `addi.d` instead of `add.d`).

This assembler does not support structured control structure words.

6.33.9 MIPS assembler

The MIPS assembler was originally written by Christian Pirker.

Currently the assembler and disassembler covers most of the MIPS32 architecture and doesn't support FP instructions.

The register names `$a0`–`$a3` are not available to avoid shadowing hex numbers. Use register numbers `$4`–`$7` instead.

Nothing distinguishes registers from immediate values. Use explicit opcode names with the `i` suffix for instructions with immediate argument. E.g. `addiu,` in place of `addu,`.

Where the architecture manual specifies several formats for the instruction (e.g., for `jalr,`),use the one with more arguments (i.e. two for `jalr,`). When in doubt, see `arch/mips/testasm.fs` for an example of correct use.

Branches and jumps in the MIPS architecture have a delay slot. You have to fill it manually (the simplest way is to use `nop,`), the assembler does not do it for you (unlike `as`). Even `if,`, `ahead,`, `until,`, `again,`, `while,`, `else,` and `repeat,` need a delay slot. Since `begin,` and `then,` just specify branch targets, they are not affected. For branches the argument specifying the target is a relative address. Add the address of the delay slot to get the absolute address.

Note that you must not put branches nor jumps (nor control-flow instructions) into the delay slot. Also it is a bad idea to put pseudo-ops such as `li,` into a delay slot, as these may expand to several instructions. The MIPS I architecture also had load delay slots, and newer MIPSes still have restrictions on using `mfhi,` and `mflo,`. Be careful to satisfy these restrictions, the assembler does not do it for you.

Some example of instructions are:

```
$ra  12 $sp  sw,          \ sw    ra,12(sp)
$4    8 $s0  lw,          \ lw    a0,8(s0)
$v0  $0  lui,             \ lui   v0,0x0
$s0  $s4  $12  addiu,     \ addiu s0,s4,0x12
$s0  $s4  $4  addu,       \ addu  s0,s4,$a0
$ra  $t9  jalr,           \ jalr  t9
```

You can specify the conditions for `if,` etc. by taking a conditional branch and leaving away the `b` at the start and the `,` at the end. E.g.,

```
4 5 eq if,
  ... \ do something if $4 equals $5
then,
```

The calling conventions for 32-bit MIPS machines is to pass the first 4 arguments in registers `$4..$7`, and to use `$v0-$v1` for return values. In addition to these registers, it is ok to clobber registers `$t0-$t8` without saving and restoring them.

If you use `jalr,` to call into dynamic library routines, you must first load the called function's address into `$t9`, which is used by position-indirect code to do relative memory accesses.

Here is an example of a MIPS32 `abi-code` word:

```
abi-code my+  ( n1 n2 -- n3 )
  \ SP passed in $4, returned in $v0
```

```
  $t0  4 $4  lw,          \ load n1, n2 from stack
  $t1  0 $4  lw,
  $t0  $t0  $t1  addu,    \ add n1+n2, result in $t0
  $t0  4 $4  sw,          \ store result (overwriting n1)
  $ra  jr,                \ return to caller
  $v0  $4  4  addiu,      \ (delay slot) return uptated SP in $v0
end-code
```

6.33.10 PowerPC assembler

The PowerPC assembler and disassembler were contributed by Michal Revucky.

This assembler does not follow the convention of ending mnemonic names with a ",", so some mnemonic names shadow regular Forth words (in particular: `and or xor fabs`); so if you want to use the Forth words, you have to make them visible first, e.g., with `also forth`.

Registers are referred to by their number, e.g., `9` means the integer register 9 or the FP register 9 (depending on the instruction).

Because there is no way to distinguish registers from immediate values, you have to explicitly use the immediate forms of instructions, i.e., `addi,`, not just `add,`.

The assembler and disassembler usually support the most general form of an instruction, but usually not the shorter forms (especially for branches).

6.33.11 RISC-V assembler

Source code: `arch/riscv/asm.fs`. The instructions are listed in `arch/riscv/inst32.fs` and (compressed instructions) `arch/riscv/inst16.fs`; note that the compressed instructions are listed there without their `c.` prefix, but that prefix is required by the assembler.

Both raw register names (`x0`...`x31`, `f0`...`f31`) and symbolic names (e.g., `a7`, `fa7`) are available. Note that some register names form valid hex numbers, so we recommend using the prefix syntax `$fa7` instead of using `hex` base.

Fences are called `|w |r |rw |o |ow |or |orw |i |iw |ir |irw |io |iow |ior |iorw`.

Instruction names do not end with ",", so there are conflicts with Forth words like `and`. You need to use the immediate form of an instruction (e.g., `addi` instead of just `add`) explicitly. Only the general form of the instruction is supported, not specialized names (e.g, there is `jalr`, but not `jr`; there is, however `c.jr`).

There is no automatic selection of compressed instructions. Instead, you have to call them by their name, with a preceding `c.`, e.g., `c.addi` for the compressed form of `addi`.

Note that the assembler does not check whether the used registers and immediate values fit in the instruction, and just produces some truncated form if they don't. This is especially relevant for compressed instructions, where immediate value sizes are even smaller, and some of which only work with some registers.

6.33.12 Other assemblers

In addition to the assemblers mentioned above, there are the following asm.fs files:

```
arch/4stack/asm.fs
arch/6502/asm.fs
```

```
arch/8086/asm.fs
arch/avr/asm.fs
arch/c165/asm.fs
arch/h8/asm.fs
arch/lm32/asm.fs
arch/m68k/asm.fs
arch/misc/asm.fs
arch/nibz40/asm.fs
arch/r8c/asm.fs
arch/shboom/asm.fs
```

These architectures may be supported by Gforth EC (embedded control), but that's not documented here (and only sporadically elsewhere, with quite a bit documentation for the r8c port). If you want to use one of these assemblers, use the source!

If you want to contribute another assembler/disassembler, please check at `https://cgit.git.savannah.gnu.org/cgit/gforth.git/tree/arch` if there is such an assembler already. If you are writing them from scratch, please use a similar syntax style as the one we use (i.e., postfix, commas at the end of the instruction names, see Section 6.33.2 [Common Assembler], page 288); make the output of the disassembler be valid input for the assembler, and keep the style similar to the style we used.

Hints on implementation: The most important part is to have a good test suite that contains all instructions. Once you have that, the rest is easy. For actual coding you can take a look at `arch/mips/disasm.fs` to get some ideas on how to use data for both the assembler and disassembler, avoiding redundancy and some potential bugs. You can also look at that file (and see Section 6.11.10.5 [Advanced does> usage example], page 130) to get ideas how to factor a disassembler.

Start with the disassembler, because it's easier to reuse data from the disassembler for the assembler than the other way round.

For the assembler, take a look at `arch/alpha/asm.fs`, which shows how simple it can be.

6.34 Carnal words

These words deal with the mechanics of Gforth (in Forth circles called "carnal knowledge" of a Forth system), but we consider them stable enough to document them.

6.34.1 Header fields

In Gforth 1.0 we switched to a new word header layout. For a detailed description, read: Bernd Paysan and M. Anton Ertl. *The new Gforth header (`http://www.euroforth.org/ef19/papers/paysan.pdf`)*. In 35th EuroForth Conference, pages 5-20, 2019. Since this paper was published, xt and nt have been changed to point to the parameter field, like the body, and "vt" (virtual table) has been renamed to "hm" (header methods), but otherwise it is still up-to-date.

This section explains just the data structure and the words used to access it. A header has the following fields:

```
name
```

```
>f+c
>link
>cfa
>namehm
>body
```

Currently Gforth has the names shown above for getting from the xt/nt/body to the field, but apart from the standard `>body` they are not stable Gforth words. Instead, we provide access words. Note that the documented access words have survived the reorganization of the header layout.

Some of the words expect an nt, some expect an xt. Given that both nt and xt point to the body of a word, what is the difference? For most words, the xt and nt use the same header, and with nt=xt, they point to the same place. However, for a synonym (see Section 6.11.12 [Synonyms], page 141) there is a difference; consider the example

```
create x
synonym y x
synonym z y
```

In this case the nt of `z` points to the body of `z`, while the xt of `z` points to the body of `x`. Words defined with `alias` or `forward` (see Section 6.10.7 [Calls and returns], page 113) also have different nts and xts.

The name field is variable-length and is accessed with `name>string` (see Section 6.15.2 [Name token], page 158).

The `>f+c` field contains flags and the name length (count). You read the count with `name>string`, and the flags with `compile-only?` and `obsolete?` (see Section 6.15.2 [Name token], page 158).

The `>link` field contains a link to the previous word in the same word list. You can read it with `name>link` (see Section 6.15.2 [Name token], page 158).

The name, `>f+c` and `>link` fields are not present for `noname` words, but `name>string` and `name>link` work nevertheless, producing 0 0 and 0, respectively.

The `>cfa` field (aka code field) contains the code address used for `execute`ing the word; you can read it with `>code-address` and write it with `code-address!` (see Section 6.34.3 [Threading Words], page 303).

The `>namehm` field contains the address of the header methods table (see Section 6.34.2 [Header methods], page 300). You access it by performing or accessing header methods (see Section 6.34.2 [Header methods], page 300).

The `>body` (aka parameter) field contains data or threaded code specific to the word type; its length depends on the word type. E.g., for a `constant` it contains a cell with the value of the constant. You can access it through `>body` (see Section 6.11.10.4 [CREATE..DOES> details], page 129), but this is only standard for words you defined with `create`.

6.34.2 Header methods

The new Gforth word header is object-oriented and supports the following methods (method selectors):

```
.hm label method          overrider        field
          execute         set-execute      >cfa
```

```
opt:      opt-compile,    set-optimizer    >hmcompile,
to:       (to)            set-to           >hmto
extra:                                     >hmextra
>int:     name>interpret  set->int         >hm>int
>comp:    name>compile    set->comp        >hm>comp
>string:  name>string     set-name>string  >hm>string
>link:    name>link       set-name>link    >hm>link
```

Many of these words are not stable Gforth words, but Gforth has stable higher-level words that we mention below.

You can look at the header methods of a word with

`.hm` (*nt* –) gforth-1.0 "dot-h-m"

print the header methods of *nt*

Overrider (setter) words change the method implementation for the most recent definition. Quotations or closures restore the previous most recent definition when they are completed, so they are not considered most recent, and you can do things like:

```
: my2dup over over ;
[: drop ]] over over [[ ;] set-optimizer
```

The `execute` method is actually stored in the `>cfa` field in the header rather than in the header-methods table for performance reasons; also it is implemented through a native-code address, while the other methods are implemented by calling an xt. The high-level way to set this method is

`set-execute` (*ca* –) gforth-1.0

Changes the current word such that it jumps to the native code at *ca*. Also changes the `compile,` implementation to the most general (and slowest) one. Call `set-optimizer` afterwards if you want a more efficient `compile,` implementation.

To get a code address for use with `set-execute`, you can use words like `docol:` or `>code-address`, See Section 6.34.3 [Threading Words], page 303.

As an alternative to `set-execute`, there is also `set-does>` (see Section 6.11.10 [User-defined Defining Words], page 125), which takes an xt.

Moreover, there are the low-level `code-address!` and `definer!` (see Section 6.34.3 [Threading Words], page 303).

The `opt-compile,` method is what `compile,` does on most Gforth engines (`gforth-itc` uses `,` instead). You can define a more efficient implementation of `compile,` for the current word with `set-optimizer` (see Section 6.11.10.7 [User-defined compile-comma], page 134). Note that the end result must be equivalent to `postpone literal postpone execute`.

As an example of the use of `set-optimizer`, consider the following definition of `constant`:

```
: constant ( n "name" -- ; name: -- n )
  create ,
  ['] @ set-does>
;

5 constant five
```

```
: foo five ; see foo
```

The Forth system does not know that the value of a constant must not be changed, and just sees a `create`d word (which can be changed with `>body`), and `foo` first pushes the body address of `five` and then fetches from there. With `set-optimizer` the definition of `constant` can be optimized as follows:

```
: constant ( n "name" -- ; name: -- n )
  create ,
  ['] @ set-does>
  [: >body @ postpone literal ;] set-optimizer
;
```

Now `foo` contains the literal 5 rather than a call to `five`.

Note that `set-execute` and `set-does>` perform `set-optimizer` themselves in order to ensure that `execute` and `compile,` agree, so if you want to add your own optimizer, you should add it afterwards.

The `(to)` method and `set-to` are used for implementing `to` *`name`* semantics etc. (see Section 6.11.10.6 [Words with user-defined TO etc.], page 132).

The `>hmextra` field is used for cases where additional data needs to be stored in the header methods table. In particular, it stores the xt passed to `set-does>` (and `does>` calls `set-does>`) and the code address behind `;abi-code`.

The methods above all consume an xt, not an nt, but the override words work on the most recent definition. This means that if you use, e.g., `set-optimizer` on a synonym, the effect will probably not be what you intended: When `compile,`ing the xt of the word, the `opt-compile,` implementation of the original word will be used, not the freshly-set one of the synonym.

The following methods consume an nt.

The `name>interpret` method is implemented as noop for most words, except synonyms and similar words.

`set->int` (*xt* –) gforth-1.0 “set-to-int”

Sets the implementation of the `name>interpret ( nt -- xt2 )` method of the current word to *xt*.

The `name>compile` method produces the compilation semantics of the nt. By changing it with `set->comp`, you can change the compilation semantics, but it's not as simple as just pushing the xt of the desired compilation semantics, because of the stack effect of `name>compile`. Generally you should avoid changing the compilation semantics, and if you do, use a higher-level word like `immediate` or `interpret/compile:`, See [Combined words], page 154.

`set->comp` (*xt* –) gforth-1.0 “set-to-comp”

Sets the implementation of the `name>compile ( nt -- xt1 xt2 )` method of the current word to *xt*.

`immediate?` (*nt* – *flag*) gforth-1.0 “immediate-question”

true if the word *nt* has non-default compilation semantics (that's not quite according to the definition of immediacy, but many people mean that when they call a word “immediate”).

`Name>string` and `Name>link` are methods in order to make it possible to eliminate the name, `>f+c` and `link` fields from noname headers, but still produce meaningful results when using these words. You will typically not change the implementations of these methods except with `noname`, but we still have

`set-name>string` (*xt* –) gforth-1.0 "set-name-to-string"

Sets the implementation of the `name>string ( nt -- addr u )` method of the current word to *xt*.

`set-name>link` (*xt* –) gforth-1.0 "set-name-to-link"

Sets the implementation of the `name>link ( nt1 -- nt2|0 )` method of the current word to *xt*.

6.34.3 Threading Words

The terminology used here stems from indirect threaded Forth systems; in such a system, the XT of a word is represented by the CFA (code field address) of a word; the CFA points to a cell that contains the code address. The code address is the address of some machine code that performs the run-time action of invoking the word (e.g., the `dovar:` routine pushes the address of the body of the word (a variable) on the stack).

These words provide access to code fields, code addresses and other threading stuff in Gforth. It more or less abstracts away the differences between direct and indirect threading.

Up to and including Gforth 0.7, the code address (plus, for `does>`-defined words, the address returned by `>does-code`) was sufficient to know the type of the word. However, since Gforth-1.0 the behaviour or at least implementation of words like `compile,` and `name>compile` can be determined independently as described in Section 6.34.2 [Header methods], page 300.

To create a code field and at the same time initialize the header methods use `create-from` (see Section 6.11.10.8 [Creating from a prototype], page 136).

The following words were designed before the introduction of header methods, and are therefore not the best (and recommended) way to deal with different word types in Gforth.

In an indirect threaded Forth, you can get the code address of *name* with `' name @`; in Gforth you can get it with `' name >code-address`, independent of the threading method.

`threading-method` (– *n*) gforth-0.2 "threading-method"

0 if the engine is direct threaded. Note that this may change during the lifetime of an image.

`>code-address` (*xt* – *c_addr*) gforth-0.2 "to-code-address"

c-addr is the code address of the word *xt*.

`code-address!` (*c_addr xt* –) gforth-obsolete "code-address-store"

Change a code field with code address *c-addr* at *xt*.

The code addresses produced by various defining words are produced by the following words:

`docol:` (– *addr*) gforth-0.2 "docol-colon"

The code address of a colon definition.

`docon:` (– *addr*) gforth-0.2 "docon-colon"

The code address of a `CONSTANT`.

`dovar:` (– *addr*) gforth-0.2 “dovar-colon”

The code address of a `CREATE`d word.

`douser:` (– *addr*) gforth-0.2 “douser-colon”

The code address of a `USER` variable.

`dodefer:` (– *addr*) gforth-0.2 “dodefer-colon”

The code address of a `defer`ed word.

`dofield:` (– *addr*) gforth-0.2 “dofield-colon”

The code address of a `field`.

`dovalue:` (– *addr*) gforth-experimental “dovalue-colon”

The code address of a `value`. This will probably stop being different from `:docon` in a future version. If you want to know whether a word is a constant or a value, better test whether the `hmcompile,` field contains the xt of `constant,` or of `value,`.

`dodoes:` (– *addr*) gforth-0.6 “dodoes-colon”

The code address of a `DOES>`-defined word.

`doabicode:` (– *addr*) gforth-1.0 “doabicode-colon”

The code address of a `ABI-CODE` definition.

For a word *X* defined with `set-does>`, the code address points to `dodoes:`, and the `>hmextra` field of the header methods contains the xt of the word that is called after pushing the body address of *X*.

If you want to know whether a word is a `DOES>`-defined word, and what Forth code it executes, `>does-code` tells you that:

`>does-code` (*xt1* – *xt2*) gforth-0.2 “to-does-code”

If *xt1* is the execution token of a child of a `set-does>`-defined word, *xt2* is the xt passed to `set-does>`, i.e, the xt of the word that is executed when executing *xt1* (but first the body address of *xt1* is pushed). If *xt1* does not belong to a `set-does>`-defined word, *xt2* is 0.

You can use the resulting *xt2* with `set-does>` (preferred) to change the latest word or with

`does-code!` (*xt2 xt1* –) gforth-0.2 “does-code-store”

Change *xt1* to be a *xt2* `set-does>`-defined word.

to change an arbitrary word.

The following two words generalize `>code-address`, `>does-code`, `code-address!`, and `does-code!`:

`>definer` (*xt* – *definer*) gforth-0.2 “to-definer”

Definer is a unique identifier for the way the *xt* was defined. Words defined with different `does>`-codes have different definers. The definer can be used for comparison and in `definer!`.

`definer!` (*definer xt* –) gforth-obsolete “definer-store”

The word represented by *xt* changes its behaviour to the behaviour associated with *definer*.

`Code-address!`, `does-code!`, and `definer!` update the `opt-compile,` method to a somewhat generic compiler for that word type (in particular, primitives get the slow `general-compile,` method rather than the primitive-specific `peephole-compile,`).

6.35 Passing Commands to the Operating System

Gforth allows you to pass an arbitrary string to the host operating system shell (if such a thing exists) for execution.

`sh` (`"..."` –) gforth-0.2

Execute the rest of the command line as shell command(s). Afterwards, `$?` produces the exit status of the command.

`system` (*c-addr u* –) gforth-0.2

Pass the string specified by *c-addr u* to the host operating system for execution in a sub-shell. Afterwards, `$?` produces the exit status of the command. The value of the environment variable `GFORTHSYSTEMPREFIX` (or its default value) is prepended to the string (mainly to support using `command.com` as shell in Windows instead of whatever shell Cygwin uses by default; see Section 2.5 [Environment variables], page 11).

`sh-get` (*c-addr u – c-addr2 u2*) gforth-1.0

Run the shell command *addr u*; *c-addr2 u2* is the output of the command. The exit code is in `$?`, the output also in `sh$ 2@`.

`$?` (– *n*) gforth-0.2 “dollar-question”

`Value` – the exit status returned by the most recently executed `system` command.

`getenv` (*c-addr1 u1 – c-addr2 u2*) gforth-0.2 “getenv”

The string *c-addr1 u1* specifies an environment variable. The string *c-addr2 u2* is the host operating system's expansion of that environment variable. If the environment variable does not exist, *c-addr2 u2* specifies a string 0 characters in length.

6.36 Keeping track of Time

`ms` (*u* –) facility-ext

Wait *u* milliseconds.

`ns` (*ud* –) gforth-1.0

Wait *ud* nanoseconds.

`time&date` (– *nsec nmin nhour nday nmonth nyear*) facility-ext “time-and-date”

Report the current time of day. Seconds, minutes and hours are numbered from 0. Months are numbered from 1.

`>time&date&tz` (*dtime – nsec nmin nhour nday nmonth nyear fdst ndstoff c-addrtz utz*) gforth-1.0 “to-time-and-date”

Convert *dtime* in seconds since 1.1.1970 0:00Z to the current time of day in the current timezone. Seconds, minutes and hours are numbered from 0. Months are numbered from 1.

`utime` (– *dtime*) gforth-0.5 “utime”

Report the current time in microseconds since some epoch. Use `#1000000 um/mod nip` to convert to seconds

`ntime` (– *dtime*) gforth-1.0 “ntime”

Report the current time in nanoseconds since some epoch.

`cputime` (– *duser dsystem*) gforth-0.5 “cputime”

duser and dsystem are the respective user- and system-level CPU times used since the start of the Forth system (excluding child processes), in microseconds (the granularity may be much larger, however). On platforms without the getrusage call, it reports elapsed time (since some epoch) for duser and 0 for dsystem.

6.37 Miscellaneous Words

This section lists the Standard Forth words that are not documented elsewhere in this manual. Ultimately, they all need proper homes.

`quit` (*??* – *??*) core

Empty the return stack, make the user input device the input source, enter interpret state and start the text interpreter.

The following Standard Forth words are not currently supported by Gforth (see Chapter 8 [Standard conformance], page 309):

`EDITOR EMIT? FORGET`

7 Error messages

A typical Gforth error message looks like this:

```
in file included from *args*:1:1:
in file included from xxx-err1.fs:1:9:
xxx-err2.fs:3:1: error: Invalid memory address
>>>bar<<<
Backtrace:
xxx-err2.fs:1:15:              0 $7FA610EA7F68 @
                               1 $14
xxx-err2.fs:2:7:               2 $7FA610EA7FB0 foo
```

The line identifying the error is

```
xxx-err2.fs:3:1: error: Invalid memory address
```

It says that in file `xxx-err2.fs`, line 3, character 1, the error "Invalid memory address" happened; more precisely, the exception -9 was thrown there, and eventually was caught by Gforth's system exception handler (either because the program did not `catch` it or caught it and rethrew it).

The next line shows the text-interpreted line where the error happened, abd the word where the error happened is surrounded by `>>>` and `<<<`.

Before that you find the nesting of source file includes that led to the inclusion of `xxx-err2.fs`, with the outermost at the top and the innermost at the bottom. So in this case `xxx-err2.fs` was included from `xxx-err1.fs`, line 1, character 9, and that was included from an OS-level command-line argument to Gforth, indicated by `*args*`.

At the end of the error message you find the backtrace, actually a return stack dump. In our example, the middle line of the return stack dump just contains the number 20 (`$14`), and is not a return address. The number `1` before the `$14` is the index of the return stack entry.

The bottom line of the backtrace shows a return address, pushed by the call to `foo`; the backtrace also shows `foo` for this entry, at the end, and the source location of the call `xxx-err2.fs:2:7` at the beginning of the line.

The top line of the backtrace shows a pseudo-return address, as if the primitive `@` pushed a return address. This return address was pushed by the signal handler in order to produce this line in the backtrace, so you know where the exception came from.

So in our example we see that `bar` called `foo`, and `foo` pushed 20 on the return stack and then called `@` (and `@` threw an *Invalid memory address* exception).

You can see the source code of the backtrace location with `nt` (next backtrace entry, initially the one with index 0, i.e., the top of the return stack), `bt` (previous backtrace entry, initially the one with the biggest index, i.e., the bottom of the return stack), or *index* `tt` (the *index*ed backtrace entry; see Section 6.30.4 [Locating backtrace entry source], page 260).

Note that the backtrace is not perfect: We don't know which return stack entries are return addresses (so we may get false positives).

The return stack dump represents the return stack at the time when a specific `throw` was executed. In programs that make use of `catch`, it is not necessarily clear which `throw`

should be used for the return stack dump (e.g., consider one `throw` that indicates an error, which is caught, and during recovery another error happens; which `throw` should be used for the stack dump?). Gforth presents the return stack dump for the first `throw` after the last executed (not returned-to) `catch` or `nothrow`; this works well in the usual case. To get the right backtrace, you usually want to insert `nothrow` or `['] false catch 2drop` after a `catch` if the error is not rethrown.

The `gforth` engine is able to do a return stack dump for throws generated from primitives (e.g., invalid memory address, stack empty etc.); `gforth-fast` is only able to do a return stack dump from a directly called `throw` (including `abort"`). Given an exception caused by a primitive in `gforth-fast`, you will typically see no return stack dump at all; however, if the exception is caught by `catch` (e.g., for restoring some state), and then `throw`n again, the return stack dump will be for the first such `throw`.

`gforth-fast` also does not attempt to differentiate between division by zero and division overflow, because that costs time in every division.

8 Standard conformance

To the best of our knowledge, Gforth is a

ANS Forth (Forth-94) System and a Forth-2012 System

- providing the Core Extensions word set
- providing the Block word set
- providing the Block Extensions word set
- providing the Double-Number word set
- providing the Double-Number Extensions word set
- providing the Exception word set
- providing the Exception Extensions word set
- providing the Facility word set
- providing the Facility Extensions word set, except `EMIT?`
- providing the File Access word set
- providing the File Access Extensions word set
- providing the Floating-Point word set
- providing the Floating-Point Extensions word set
- providing the Locals word set
- providing the Locals Extensions word set
- providing the Memory-Allocation word set
- providing the Memory-Allocation Extensions word set
- providing the Programming-Tools word set
- providing the Programming-Tools Extensions word set, except `EDITOR` and `FORGET`
- providing the Search-Order word set
- providing the Search-Order Extensions word set
- providing the String word set
- providing the String Extensions word set
- providing the Extended-Character wordset

Gforth has the following environmental restrictions:

- While processing the OS command line, if an exception is not caught, Gforth exits with a non-zero exit code instead of performing QUIT.
- When a `throw` is performed after a `query`, Gforth does not always restore the input source specification in effect at the corresponding catch.

In addition, Standard Forth systems are required to document certain implementation choices. This chapter tries to meet these requirements for the Forth-94 standard. For the Forth-2012 standard, we decided to produce the additional documentation only if there is demand. So if you are really missing this documentation, please let us know.

In many cases, the following documentation gives a way to ask the system for the information instead of providing the information directly, in particular, if the information depends on the processor, the operating system or the installation options chosen, or if they are likely to change during the maintenance of Gforth.

8.1 The Core Words

8.1.1 Implementation Defined Options

(Cell) aligned addresses:
: processor-dependent. Gforth's alignment words perform natural alignment (e.g., an address aligned for a datum of size 8 is divisible by 8). Unaligned accesses may result in a `-23 THROW`.

`EMIT` *and non-graphic characters:*
: The character is output using the C library function (actually, macro) `putc`.

character editing of `ACCEPT` *and* `EXPECT`*:*
: This is modeled on the GNU readline library (see Section "Command Line Editing" in *The GNU Readline Library*) with Emacs-like key bindings. *`Tab`* deviates a little by producing a full word completion every time you type it (instead of producing the common prefix of all completions). See Section 2.4 [Command-line editing], page 10.

character set:
: The character set of your computer and display device. Gforth is 8-bit-clean (but some other component in your system may make trouble).

Character-aligned address requirements:
: All addresses are character-aligned, because `1 chars` produces 1.

character-set extensions and matching of names:
: Any character can be used in a name. Matching is case-insensitive (except in `table`s and `cs-wordlist`s) for ASCII characters and case-sensitive for other characters.

conditions under which control characters match a space delimiter:
: If `word` is called with the space character as a delimiter, all white-space characters (as identified by the C macro `isspace()`) are delimiters. `Parse`, on the other hand, treats space like other delimiters. `Parse-name`, which is used by the outer interpreter (aka text interpreter) by default, treats all white-space characters as delimiters.

format of the control-flow stack:
: The data stack is used as control-flow stack. The size of a control-flow stack item in cells is given by the constant `cs-item-size`. At the time of this writing, an item consists of a cell-sized stack-state descriptor (fourth), a (pointer to a) locals list (third), an address in the code (second), and a tag for identifying the item (TOS). The following tags are used: `defstart`, `live-orig`, `dead-orig`, `dest`, `do-dest`, `scopestart`.

conversion of digits `>` *35*
: The characters `[\]^_'` are the digits with the decimal value 36−41. There is no way to input many of the larger digits.

display after input terminates in `ACCEPT` *and* `EXPECT`*:*
: The cursor is moved to the end of the entered string. If the input is terminated using the *`Return`* key, a space is typed.

exception abort sequence of `ABORT"`*:*
: The error string is stored into the variable `abort-string` and a `-2 throw` is performed.

input line terminator:
: For interactive input, *`C-m`* (CR) and *`C-j`* (LF) terminate lines. One of these characters is typically produced when you type the *`Enter`* or *`Return`* key.

maximum size of a counted string:
: 255 characters.

maximum size of a parsed string:
: Given by the constant `/line`. Currently 255 characters.

maximum size of a definition name, in characters:
: `-1 8 rshift`; 72057594037927935 characters on 64-bit systems, 16777215 characters on 32-bit systems.

maximum string length for `ENVIRONMENT?`*, in characters:*
: Any string length, but only strings that can be names can be recognized.

method of selecting the user input device:
: Change `infile-id`, ideally with `infile-execute`. Or by changing `ip-vector` (undocumented).

method of selecting the user output device:
: Change `outfile-id`, e.g., with `outfile-execute`. Or change `type` and `emit`, e.g., with `>string-execute`. Or by changing `op-vector` (undocumented).

methods of dictionary compilation:
: What are we expected to document here?

number of bits in one address unit:
: 8

number representation and arithmetic:
: Binary. Negative integers are represented in two's-complement form. FP numbers are represented as IEEE DP FP numbers (binary64) on nearly all platforms (exception: ARM with soft floats).

ranges for integer types:
: For 32-bit platforms: u: 0...4294967295, n: -2147483648...2147483647, ud: 0...18446744073709551615, d: -9223372036854775808...9223372036854775807.

  For 64-bit platforms: u: 0...18446744073709551615, n: -9223372036854775808...9223372036854775
  ud: 0...340282366920938463463374607431768211455, d: -170141183460469231731687303715884105

  You can also make environmental queries for `MAX-N`, `MAX-U`, `MAX-D` and `MAX-UD`.

read-only data space regions:
: The whole Forth data space is writable.

size of buffer at `WORD`*:*
: `PAD HERE - .` . 104 characters on 32-bit machines, 176 on 64-bit machines. The buffer is shared with the pictured numeric output string. If overwriting `PAD` is acceptable, it is as large as the remaining dictionary space, although only as much can be sensibly used as fits in a counted string.

size of one cell in address units:
: `1 cells ..` 4 on 32-bit systems, 8 on 64-bit systems.

size of one character in address units:
: 1

size of the keyboard terminal buffer:
: `max#tib @`. This can vary.

size of the pictured numeric output buffer:
: `word-pno-size`

size of the scratch area returned by `PAD`*:*
: The remainder of dictionary space. `unused pad here - - ..`

system case-sensitivity characteristics:
: Dictionary searches are case-insensitive for ASCII characters and case-sensitive for other characters (except that everything is case-sensitive in `table`s and `cs-wordlist`s).

system prompt:
: `ok` in interpret state, `compiled` in compile state, `postponed` in postpone state.

division rounding:
: The ordinary division words `/ mod /mod */ */mod` perform floored division. If you write programs that need a specific division rounding, best use `fm/mod` or `sm/rem` for portability, but you can also use `/s`, `/f` etc.

values of `STATE` *when true:*
: -1.

values returned after arithmetic overflow:
: Arithmetic is performed modulo 2**bits-per-cell for single arithmetic and 4**bits-per-cell for double arithmetic (with appropriate mapping for signed types). Division by zero typically results in a `-55 throw` (Floating-point unidentified fault) or `-10 throw` (divide by zero). Integer division overflow can result in these throws, or in `-11 throw`; in `gforth-fast` division overflow and divide by zero may also result in returning bogus results without producing an exception.

whether the current definition can be found after `DOES>`*:*
: No.

8.1.2 Ambiguous conditions

a name is neither a word nor a number:
: If a recognizer matches (see `recs`, see Section 6.17.4.1 [Default recognizers], page 171, see Chapter 5 [Literals in source code], page 54), the text interpreter performs its interpreting, compiling, or postponing action (which one, depends on the state). If none matches, the text interpreter performs `-13 throw` (Undefined word).

a definition name exceeds the maximum length allowed:
: `-19 throw` (Word name too long)

addressing a region not inside the various data spaces of the forth system:
The stacks, data space, code space and header space are readable and writable. Machine code space is typically readable. Accessing other addresses gives results dependent on the operating system. On decent systems: `-9 throw` (Invalid memory address).

argument type incompatible with parameter:
This is usually not caught. Some words perform checks, e.g., the control flow words, and issue a `ABORT"` or `-12 THROW` (Argument type mismatch).

attempting to obtain the execution token of a word with undefined execution semantics:
You get an execution token that represents the interpretation semantics of the word. In some cases, the standard does not define the interpretation semantics nor the execution semantics (e.g., `if`). In those cases, you get an execution token for the compilation semantics. For words marked compile-only, the most common ways of obtaining the xt produce a warning.

dividing by zero:
On some platforms, this produces a `-10 throw` (Division by zero); on other systems, this typically results in a `-55 throw` (Floating-point unidentified fault). On some platforms, the `gforth-fast` engine produces some hardware-dependent value.

insufficient data stack or return stack space:
Depending on the operating system, the installation, and the invocation of Gforth, this is either checked by the memory management hardware, or it is not checked. If it is checked, you typically get a `-3 throw` (Stack overflow), `-5 throw` (Return stack overflow), or `-9 throw` (Invalid memory address) (depending on the platform and how you achieved the overflow) as soon as the overflow happens. If it is not checked, overflows typically result in mysterious illegal memory accesses, producing `-9 throw` (Invalid memory address) or `-23 throw` (Address alignment exception); they might also destroy the internal data structure of `ALLOCATE` and friends, resulting in various errors in these words.

insufficient space for loop control parameters:
Like other return stack overflows.

insufficient space in the dictionary:
If you try to allot (either directly with `allot`, or indirectly with `,`, `create` etc.) more memory than available in the dictionary, you get a `-8 throw` (Dictionary overflow). If you try to access memory beyond the end of the dictionary, the results are similar to stack overflows.

interpreting a word with undefined interpretation semantics:
Gforth defines interpretation semantics for all words; for words where the standard defines execution semantics (except `exit` and `leave`), the interpretation semantics are to perform the execution semantics. For words where the standard defines no interprtation semantics, but defined compilation semantics (plus `exit` and `leave`), the interpretation semantics are to perform the compilation semantics. Some words are marked as compile-only, and text-interpreting them gives a warning.

modifying the contents of the input buffer or a string literal:
: These are located in writable memory and can be modified, which may have bad consequences for later execution.

overflow of the pictured numeric output string:
: `-17 throw` (Pictured numeric output string overflow).

parsed string overflow:
: `PARSE` cannot overflow. `WORD` does not check for overflow.

producing a result out of range:
: Arithmetic is performed modulo 2**bits-per-cell for single arithmetic and 4**bits-per-cell for double arithmetic (with appropriate mapping for signed types). Division by zero typically results in a `-10 throw` (divide by zero) or `-55 throw` (floating point unidentified fault). Overflow on division may result in these errors or in `-11 throw` (result out of range). `Gforth-fast` may silently produce bogus results on division overflow or division by zero. `Convert` and `>number` currently overflow silently.

reading from an empty data or return stack:
: The data stack is checked by the outer (aka text) interpreter after every word executed. If it has underflowed, a `-4 throw` (Stack underflow) is performed. Apart from that, stacks may be checked or not, depending on operating system, installation, and invocation. If they are caught by a check, they typically result in `-4 throw` (Stack underflow), `-6 throw` (Return stack underflow) or `-9 throw` (Invalid memory address), depending on the platform and which stack underflows and by how much. Note that even if the system uses checking (through the MMU), your program may have to underflow by a significant number of stack items to trigger the reaction (the reason for this is that the MMU, and therefore the checking, works with a page-size granularity). If there is no checking, the symptoms resulting from an underflow are similar to those from an overflow. Unbalanced return stack errors can result in a variety of symptoms, including `-9 throw` (Invalid memory address) and Illegal Instruction (typically `-260 throw`).

unexpected end of the input buffer, resulting in an attempt to use a zero-length string as a name:
: `Create` and its descendants, as well as `'` and friends perform a `-16 throw` (Attempt to use zero-length string as a name) if they are invoked at the end of a line. However, it is possible to create zero-length names with `nextname`.

`>IN` *greater than input buffer:*
: The next invocation of a parsing word returns a string with length 0.

`RECURSE` *appears after* `DOES>`*:*
: Compiles a recursive call to the code after `DOES>`.

argument input source different than current input source for `RESTORE-INPUT`*:*
: Ideally, this should result in a `-12 THROW`. Currently it produces unintended results (up to and including terminating the Gforth session).

data space containing definitions gets de-allocated:
: Deallocation with `allot` is not checked, possibly resulting in system corruption. If you deallocate to the `section-start`, you get a `-8 throw` (dictionary overflow).

data space read/write with incorrect alignment:
: Processor-dependent. Typically results in a `-23 throw` (Address alignment exception). There are reportedly some processors with alignment restrictions that do not report violations, but just produce unintended results.

data space pointer not properly aligned, `,` *,* `C,`*:*
: Like other alignment errors.

less than u+2 stack items (`PICK` *and* `ROLL`*):*
: Like other stack underflows.

loop control parameters not available:
: Not checked. The counted loop words simply assume that the top of return stack items are loop control parameters and behave accordingly.

most recent definition does not have a name (`IMMEDIATE`*):*
: No effect.

name not defined by `VALUE` *used by* `TO`*:*
: If `to` is not supported by the word type: `-21 throw` (Unsupported operation).

name not found (`'`*,* `POSTPONE`*,* `[']`*,* `[COMPILE]`*):*
: `-13 throw` (Undefined word)

parameters are not of the same type (`DO`*,* `?DO`*,* `WITHIN`*):*
: Gforth behaves as if they were of the same type. I.e., you can predict the behaviour by interpreting all parameters as, e.g., signed.

`POSTPONE` *or* `[COMPILE]` *applied to* `TO`*:*
: Assume `: X POSTPONE TO ; IMMEDIATE`. `X` performs the compilation semantics of `TO`.

String longer than a counted string returned by `WORD`*:*
: Not checked. The string will be ok, but the count will, of course, contain only the least significant bits of the length.

u greater than or equal to the number of bits in a cell (`LSHIFT`*,* `RSHIFT`*):*
: Processor-dependent. Typical behaviours are returning 0; or using only the low bits of the shift count.

word not defined via `CREATE`*:*
: `>BODY` produces the PFA of the word no matter how it was defined.

  `DOES>` changes the execution semantics of the last defined word no matter how it was defined. E.g., `CONSTANT DOES>` is equivalent to `CREATE , DOES>`.

words improperly used outside `<#` *and* `#>`*:*
: Not checked. As usual, you can expect memory faults.

8.1.3 Other system documentation

nonstandard words using `PAD`*:*
None.

operator's terminal facilities available:
After processing the OS's command line, Gforth goes into interactive mode, and you can give commands to Gforth interactively. The actual facilities available depend on how you invoke Gforth.

program data space available:
`UNUSED .` gives the remaining dictionary space. The total dictionary space can be specified with the `-m` switch (see Section 2.1 [Invoking Gforth], page 4) when Gforth starts up.

return stack space available:
You can compute the total return stack space in cells with `s" RETURN-STACK-CELLS" environment? drop .`. You can specify it at startup time with the `-r` switch (see Section 2.1 [Invoking Gforth], page 4).

stack space available:
You can compute the total data stack space in cells with `s" STACK-CELLS" environment? drop .`. You can specify it at startup time with the `-d` switch (see Section 2.1 [Invoking Gforth], page 4).

system dictionary space required, in address units:
Type `here forthstart - .` after startup. At the time of this writing, this gives 696016 (bytes) on a 64-bit system.

8.2 The optional Block word set

8.2.1 Implementation Defined Options

the format for display by `LIST`*:*
First the screen number is displayed, then 16 lines of 64 characters, each line preceded by the line number.

the length of a line affected by `\`*:*
64 characters.

8.2.2 Ambiguous conditions

correct block read was not possible:
Typically results in a `throw` of some OS-derived value (between -512 and -2048). If the blocks file was just not long enough, blanks are supplied for the missing portion.

I/O exception in block transfer:
Typically results in a `throw` of some OS-derived value (between -512 and -2048).

invalid block number:
`-35 throw` (Invalid block number)

a program directly alters the contents of `BLK`*:*
: The input stream is switched to that other block, at the same position. If the storing to `BLK` happens when interpreting non-block input, the system will get quite confused when the block ends.

no current block buffer for `UPDATE`*:*
: `UPDATE` has no effect.

8.2.3 Other system documentation

any restrictions a multiprogramming system places on the use of buffer addresses:
: The current implementation of blocks is not thread-safe, so blocks should only be used by one task.

the number of blocks available for source and data:
: depends on your disk space.

8.3 The optional Double Number word set

8.3.1 Ambiguous conditions

d outside of range of n in `D>S`*:*
: The least significant cell of *d* is produced.

8.4 The optional Exception word set

8.4.1 Implementation Defined Options

`THROW`*-codes used in the system:*
: The codes -256–-511 are used for reporting signals. The mapping from OS signal numbers to throw codes is -256−*signal*.

  The codes -512–-2047 are used for OS errors (for file and memory allocation operations). The mapping from OS error numbers to throw codes is -512−`errno`. One side effect of this mapping is that undefined OS errors produce a message with a strange number; e.g., `-1000 THROW` results in `Unknown error 488` on my system.

  You can allocate an exception number in the range (-4095...-256) with `exception`. Gforth has allocated a number of such exceptions itself.

8.5 The optional Facility word set

8.5.1 Implementation Defined Options

encoding of keyboard events (`EKEY`*):*
: Keys corresponding to ASCII characters are encoded as ASCII characters. Other keys are encoded with the constants `k-left`, `k-right`, `k-up`, `k-down`, `k-home`, `k-end`, `k1`, `k2`, `k3`, `k4`, `k5`, `k6`, `k7`, `k8`, `k9`, `k10`, `k11`, `k12`, `k-winch`, `k-eof`.

duration of a system clock tick:
: System dependent. With respect to `MS`, the time is specified in nanoseconds. How well the OS and the hardware implement this, is another question.

repeatability to be expected from the execution of `MS`*:*
: System dependent. On Unix, a lot depends on load. If the system is lightly loaded, and the delay is short enough that Gforth does not get swapped out, the performance should be acceptable. Under MS-DOS and other single-tasking systems, it should be good.

8.5.2 Ambiguous conditions

`AT-XY` *can't be performed on user output device:*
: Largely terminal dependent. No range checks are done on the arguments. No errors are reported. You may see some garbage appearing, you may see simply nothing happen.

8.6 The optional File-Access word set

8.6.1 Implementation Defined Options

file access methods used:
: `R/O`, `R/W` and `BIN` work as you would expect. `W/O` translates into the C file opening mode `w` (or `wb`): The file is cleared, if it exists, and created, if it does not (with both `open-file` and `create-file`). Under Unix `create-file` creates a file with 666 permissions modified by your umask.

file exceptions:
: The file words do not raise exceptions (except, perhaps, memory access faults when you pass illegal addresses or file-ids).

file line terminator:
: System-dependent. Gforth outputs LF on Unix (including WSL(2)) and CRLF on Windows (if anybody ever builds a Windows version again). Gforth accepts LF, CR and CRLF on input.

file name format:
: System dependent. Gforth just uses the file name format of your OS.

information returned by `FILE-STATUS`*:*
: `FILE-STATUS` returns the most powerful file access mode allowed for the file: Either `R/O`, `W/O` or `R/W`. If the file cannot be accessed, `R/O BIN` is returned. `BIN` is applicable along with the returned mode.

input file state after an exception when including source:
: All files that are left via the exception are closed.

ior values and meaning:
: The *ior*s returned by the file and memory allocation words are intended as throw codes. They typically are in the range -512–-2047 of OS errors. The mapping from OS error numbers to *ior*s is -512−*errno*.

maximum depth of file input nesting:
: limited by the amount of return stack, locals/TIB stack, and the number of open files available. This should not give you troubles.

maximum size of input line:
: For file input, the maximum input line size is only limited by memory; the buffer is expanded to read in as long lines as the file contains. For terminal input, this auto-expansion does not happen, and the current size of the input buffer is used; `max#tib ?` shows the current input buffer size.

methods of mapping block ranges to files:
: By default, blocks are accessed in the file `blocks.fb` in the current working directory. The file can be switched with `open-blocks` or `use`.

number of string buffers provided by `S"`*:*
: The interpretation semantics `S" allocate`s the memory for the strings, so the number of string buffers is only limited by memory.

size of string buffer used by `S"`*:*
: `S" allocate`s the strings, as long as they are.

8.6.2 Ambiguous conditions

attempting to position a file outside its boundaries:
: `REPOSITION-FILE` is performed as usual: Afterwards, `FILE-POSITION` returns the value given to `REPOSITION-FILE`.

attempting to read from file positions not yet written:
: End-of-file, i.e., zero characters are read and no error is reported.

file-id is invalid (`INCLUDE-FILE`*):*
: An appropriate exception may be thrown, but a memory fault or other problem is more probable.

I/O exception reading or closing file-id (`INCLUDE-FILE`*,* `INCLUDED`*):*
: The operation that discovers the problem produces an *ior* and that *ior* is thrown.

named file cannot be opened (`INCLUDED`*):*
: The *ior* produced by `open-file` is thrown.

requesting an unmapped block number:
: There are no unmapped legal block numbers. On some operating systems, writing a block with a large number may overflow the file system and have an error message as consequence.

using `source-id` *when* `blk` *is non-zero:*
: `source-id` performs its function. Typically it will give the id of the source which loaded the block. (Better ideas?)

8.7 The optional Floating-Point word set

8.7.1 Implementation Defined Options

format and range of floating point numbers:
: The `double` type of C, which is IEEE DP FP (binary64) on most platforms (exception: ARM with soft floats).

results of `REPRESENT` *when float is out of range:*
: `Represent` produces results for all inputs. For infinities, it produces as much of "infinity" or "-infinity" as fits in the buffer. For NaNs, it produces as much of "NaN" as fits in the buffer. The remaining buffer is filled with spaces. For infinities and NaNs, *flag2* (valid) is false, otherwise true.

rounding or truncation of floating-point numbers:
: IEEE-FP-based (i.e., most) systems by default round to nearest, and break ties by rounding to even (i.e., such that the last bit of the mantissa is 0).

size of floating-point stack:
: `s" FLOATING-STACK" environment? drop .` gives the total size of the floating-point stack (in floats). You can specify this on startup with the command-line option `-f` (see Section 2.1 [Invoking Gforth], page 4).

width of floating-point stack:
: `1 floats`.

8.7.2 Ambiguous conditions

`df@` *or* `df!` *used with an address that is not double-float aligned:*
: Hardware-dependent. If the hardware does not support unaligned accesses, this typically results in a `-23 THROW` like other alignment violations.

`f@` *or* `f!` *used with an address that is not float aligned:*
: Hardware-dependent. If the hardware does not support unaligned accesses, this typically results in a `-23 THROW` like other alignment violations.

floating-point result out of range:
: An FP value is returned (so nothing is out of range), but it may be an infinity or a NaN.

`sf@` *or* `sf!` *used with an address that is not single-float aligned:*
: Hardware-dependent. If the hardware does not support unaligned accesses, this typically results in a `-23 THROW` like other alignment violations.

`base` *is not decimal (*`REPRESENT`*,* `F.`*,* `FE.`*,* `FS.`*):*
: Conversion between FP and string representations always use decimal, irrespective of `base`.

Both arguments are equal to zero (`FATAN2`*):*
: System-dependent. `FATAN2` is implemented using the C library function `atan2()`.

Using `FTAN` *on an argument r1 where cos(r1) is zero:*
: System-dependent. Anyway, typically the cos of *r1* will not be zero because of small errors and the tan will be a very large (or very small) but finite number.

d cannot be presented precisely as a float in `D>F`*:*
: The result is rounded to the nearest float.

dividing by zero:
: On IEEE FP based systems, prouces an infinity or a NaN.

exponent too big for conversion (`DF!`*,* `DF@`*,* `SF!`*,* `SF@`*):*
: On IEEE-FP based systems the number is converted into an infinity.

float`<1` *(*`FACOSH`*):*
: On IEEE-FP based systems this produces a NaN.

float`<=-1` *(*`FLNP1`*):*
: On IEEE-FP based systems this produces a NaN (or a negative infinity for *float*=-1).

float`<=0` *(*`FLN`*,* `FLOG`*):*
: On IEEE-FP based systems this produces a NaN (or a negative infinity for *float*=0).

float`<0` *(*`FASINH`*,* `FSQRT`*):*
: On IEEE-FP based systems, for `fsqrt` this gives a NaN, for `fasinh` some platforms produce a NaN, others a number.

`|`*float*`|>1` *(*`FACOS`*,* `FASIN`*,* `FATANH`*):*
: IEEE-FP based systems produce a NaN.

integer part of float cannot be represented by d in `F>D`*:*
: Platform-dependent; typically, some double number is produced and no error is reported.

string larger than pictured numeric output area (`f.`*,* `fe.`*,* `fs.`*):*
: These words do not use the pictured numeric output buffer, but a buffer big enough for the needs of these words.

8.8 The optional Locals word set

8.8.1 Implementation Defined Options

maximum number of locals in a definition:
: Gforth has a `locals-headers` section with 19KB free. How many locals fit in that section depends on the length of the locals names. On a 64-bit machine a local takes about 40 bytes plus the length of the name, rounded up to 8. So with typical use, several hundred locals fit.

8.8.2 Ambiguous conditions

executing a named local in interpretation state:
: Compiles the local into the current definition (just as in compile state); in addition text-interpreting a local in interpretation state gives an “is compile-only” warning.

name not defined by `VALUE` *or* `(LOCAL)` *(*`TO`*):*
: `-21 throw` (Unsupported operation)

8.9 The optional Memory-Allocation word set

8.9.1 Implementation Defined Options

values and meaning of ior:
: The *ior*s returned by the file and memory allocation words are intended as throw codes. They typically are in the range -512–-2047 of OS errors. The mapping from OS error numbers to *ior*s is -512−*errno*.

8.10 The optional Programming-Tools word set

8.10.1 Implementation Defined Options

ending sequence for input following `;CODE` *and* `CODE`*:*
: `END-CODE`

manner of processing input following `;CODE` *and* `CODE`*:*
: The `ASSEMBLER` vocabulary is pushed on the search order stack, and the input is processed by the text interpreter, (starting) in interpret state.

search order capability for `EDITOR` *and* `ASSEMBLER`*:*
: The Search-Order word set.

source and format of display by `SEE`*:*
: The source for `see` is the word header and possibly body data; for colon definitions `see` decompiles the threaded code; for primitives `see` disassembles the machine code.

8.10.2 Ambiguous conditions

deleting the compilation word list (`FORGET`*):*
: Not implemented.

fewer than u+1 items on the control-flow stack (`CS-PICK`*,* `CS-ROLL`*):*
: This typically results in an `abort"` with a descriptive error message (may change into a `-22 throw` (Control structure mismatch) in the future). You may also get a stack underflow error. If you are unlucky, this ambiguous condition is not caught.

name can't be found (`FORGET`*):*
: Not implemented.

name not defined via `CREATE`*:*
: `;CODE` behaves like `DOES>` in this respect, i.e., it changes the execution semantics of the last defined word no matter how it was defined.

`POSTPONE` *applied to* `[IF]`*:*
: After defining `: X POSTPONE [IF] ; IMMEDIATE`. `X` is equivalent to `[IF]`.

reaching the end of the input source before matching `[ELSE]` *or* `[THEN]`*:*
: While in a taken branch of an `[if]` or `[if]...[else]`, `[then]` is ignored, and it's absence is not noticed.

While in a non-taken-branch, the end of the input source is considered to be the end of the `[if]` construct, and the missing `[then]` is reported in a warning.

removing a needed definition (`FORGET`*):*
Not implemented.

8.11 The optional Search-Order word set

8.11.1 Implementation Defined Options

maximum number of word lists in search order:
Only limited by memory.

minimum search order:
`root root`.

8.11.2 Ambiguous conditions

changing the compilation word list (during compilation):
The word is entered into the word list that was the compilation word list at the start of the definition. Any changes to the name field (e.g., `immediate`) or the code field (e.g., when executing `DOES>`) are applied to the latest defined word (as reported by `latest` or `latestxt`), if possible, irrespective of the compilation word list.

search order empty (`previous`*):*
`-4 throw` (stack underflow). You can distinguish this from data stack underflows by having `throw` on the top of the backtrace and `stack>` close by.

too many word lists in search order (`also`*):*
The search order size in Gforth is only limited by memory, and when the `resize` fails, an appropriate exception is thrown. However, if you use `get-order`, you will probably get a data stack overflow at a much smaller size (by default, the data stack can hold 4K elements).

9 Should I use Gforth extensions?

As you read through the rest of this manual, you will see documentation for *Standard* words, and documentation for some appealing Gforth *extensions*. You might ask yourself the question: *"Should I restrict myself to the standard, or should I use the extensions?"*

The answer depends on the goals you have for the program you are working on:

- Is it just for yourself or do you want to share it with others?
- If you want to share it, do the others all use Gforth?
- If it is just for yourself, do you want to restrict yourself to Gforth?

If restricting the program to Gforth is ok, then there is no reason not to use extensions. It is still a good idea to keep to the standard where it is easy, in case you want to reuse these parts in another program that you want to be portable.

If you want to be able to port the program to other Forth systems, there are the following points to consider:

- Most Forth systems that are being maintained support Standard Forth. So if your program complies with the standard, it will be portable among many systems.
- A number of the Gforth extensions can be implemented in Standard Forth using public-domain files provided in the `compat/` directory. These are mentioned in the text in passing. There is no reason not to use these extensions, your program will still be Standard Forth compliant; just include the appropriate compat files with your program.
- The tool `ans-report.fs` (see Section 6.30.12 [Standard Report], page 270) makes it easy to analyse your program and determine what non-Standard words it relies upon. However, it does not check whether you use standard words in a non-standard way.
- Some techniques are not standardized by Standard Forth, and are hard or impossible to implement in a standard way, but can be implemented in most Forth systems easily, and usually in similar ways. Forth has a rich historical precedent for programmers taking advantage of implementation-dependent features of their tools (for example, relying on a knowledge of the dictionary structure). Sometimes these techniques are necessary to extract every last bit of performance from the hardware, sometimes they are just a programming shorthand.
- Does using a Gforth extension save more work than the porting this part to other Forth systems (if any) will cost?
- Is the additional functionality worth the reduction in portability and the additional porting problems?

In order to perform these considerations, you need to know what's standard and what's not. This manual generally states if something is non-standard, but the authoritative source is the standard document (`https://forth-standard.org/standard/words`). Appendix A of the Standard (*Rationale*) provides valuable insight into the thought processes of the standardization committee.

Note also that portability between Forth systems is not the only portability issue; there is also the issue of portability between different platforms (processor/OS combinations).

10 Integrating Gforth into C programs

Several people like to use Forth as scripting language for applications that are otherwise written in C, C++, or some other language.

The Forth system ATLAST provides facilities for embedding it into applications; unfortunately it has several disadvantages: most importantly, it is not based on Standard Forth, and it is apparently dead (i.e., not developed further and not supported). The facilities provided by Gforth in this area are inspired by ATLAST's facilities, so making the switch should not be hard.

We also tried to design the interface such that it can easily be implemented by other Forth systems, so that we may one day arrive at a standardized interface. Such a standard interface would allow you to replace the Forth system without having to rewrite C code.

You embed the Gforth interpreter by linking with the library `libgforth.a` or `libgforth.so` (give the compiler the option `-lgforth`, or for one of the other engines `-lgforth-fast`, `-lgforth-itc`, or `-lgforth-ditc`). All global symbols in this library that belong to the interface, have the prefix `gforth_`; if a common interface emerges, the functions may also be available through `#define`s with the prefix `forth_`.

You can include the declarations of Forth types, the functions and variables of the interface with `#include <gforth.h>`.

As an example, you can build a small program to call the gforth library as follows:

```
#include <gforth.h>

Cell main(int argc, char **argv, char **env)
{
  Cell retvalue=gforth_start(argc, argv);

  if(retvalue == -56) { /* throw-code for quit */
    gforth_bootmessage();     // show boot message
    retvalue = gforth_quit(); // run quit loop
  }
  gforth_cleanup();
  gforth_printmetrics();
  // gforth_free_dict(); // if you want to restart, do this

  return retvalue;
}
```

This was compiled as follows (from the gforth source and build directory, adjust the directories as appropriate):

```
gcc -static -I engine -I include -L engine/.libs xxx.c -lgforth -lm -lltdl
```

To interact with the Forth interpreter, there's `Xt gforth_find(Char * name)` and `Cell gforth_execute(Xt xt)`.

More documentation needs to be put here.

10.1 Types

`Cell`, `UCell`: cell (data and return stack element)

`Float`: float (FP stack element)

`Address`: Memory address (a cell)

`Xt`: execution token (a cell)

10.2 Variables

`gforth_SP`: data stack pointer.

`gforth_FP`: FP stack pointer.

10.3 Functions

```
void *gforth_engine(Xt *, stackpointers *);
Cell gforth_main(int argc, char **argv, char **env);
int gforth_args(int argc, char **argv, char **path, char **imagename);
ImageHeader* gforth_loader(char* imagename, char* path);
user_area* gforth_stacks(Cell dsize, Cell rsize, Cell fsize, Cell lsize);
void gforth_free_stacks(user_area* t);
void gforth_setstacks(user_area * t);
void gforth_free_dict();
Cell gforth_go(Xt* ip0);
Cell gforth_boot(int argc, char** argv, char* path);
void gforth_bootmessage();
Cell gforth_start(int argc, char ** argv);
Cell gforth_quit();
Xt gforth_find(Char * name);
Cell gforth_execute(Xt xt);
void gforth_cleanup();
void gforth_printmetrics();
void gforth_setwinch();
```

10.4 Signals

Gforth sets up signal handlers to catch exceptions and window size changes. This may interfere with your C program.

11 Emacs and Gforth

Gforth comes with `gforth.el`, an improved version of `forth.el` by Goran Rydqvist (included in the TILE package). The improvements are:

- A better handling of indentation.
- A custom hilighting engine for Forth-code.
- Comment paragraph filling (*M-q*)
- Commenting (*C-x *) and uncommenting (*C-u C-x *) of regions
- Removal of debugging tracers (*C-x ~*, see Section 6.30.8 [Debugging], page 265).
- Support of the `info-lookup` feature for looking up the documentation of a word.
- Support for reading and writing blocks files.

To get a basic description of these features, enter Forth mode and type *C-h m*.

In addition, Gforth supports Emacs quite well: The source code locations given in error messages, debugging output (from `~~`) and failed assertion messages are in the right format for Emacs' compilation mode (see Section "Running Compilations under Emacs" in *Emacs Manual*) so the source location corresponding to an error or other message is only a few keystrokes away (*C-x `* for the next error, *C-c C-c* for the error under the cursor).

Moreover, for words documented in this manual, you can look up the glossary entry quickly by using *C-h TAB* (`info-lookup-symbol`, see Section "Documentation Commands" in *Emacs Manual*). This feature requires Emacs 20.3 or later and does not work for words containing `:`.

11.1 Installing gforth.el

To make the features from `gforth.el` available in Emacs, add the following lines to your `.emacs` file:

```
(autoload 'forth-mode "gforth.el")
(setq auto-mode-alist (cons '("\\.fs\\'" . forth-mode)
                            auto-mode-alist))
(autoload 'forth-block-mode "gforth.el")
(setq auto-mode-alist (cons '("\\.fb\\'" . forth-block-mode)
                            auto-mode-alist))
(add-hook 'forth-mode-hook (function (lambda ()
   ;; customize variables here:
   (setq forth-indent-level 4)
   (setq forth-minor-indent-level 2)
   (setq forth-hilight-level 3)
   ;;; ...
)))
```

11.2 Emacs Tags

If you `require etags.fs`, a new `TAGS` file will be produced (see Section "Tags Tables" in *Emacs Manual*) that contains the definitions of all words defined afterwards. You can then find the source for a word using *M-.*. Note that Emacs can use several tags files

at the same time (e.g., one for the Gforth sources and one for your program, see Section "Selecting a Tags Table" in *Emacs Manual*). The TAGS file for the preloaded words is `$(datadir)/gforth/$(VERSION)/TAGS` (e.g., `/usr/local/share/gforth/0.2.0/TAGS`). To get the best behaviour with `etags.fs`, you should avoid putting definitions both before and after `require` etc., otherwise you will see the same file visited several times by commands like `tags-search`.

11.3 Hilighting

`gforth.el` comes with a custom source hilighting engine. When you open a file in `forth-mode`, it will be completely parsed, assigning faces to keywords, comments, strings etc. While you edit the file, modified regions get parsed and updated on-the-fly.

Use the variable 'forth-hilight-level' to change the level of decoration from 0 (no hilighting at all) to 3 (the default). Even if you set the hilighting level to 0, the parser will still work in the background, collecting information about whether regions of text are "compiled" or "interpreted". Those information are required for auto-indentation to work properly. Set 'forth-disable-parser' to non-nil if your computer is too slow to handle parsing. This will have an impact on the smartness of the auto-indentation engine, though.

Sometimes Forth sources define new features that should be hilighted, new control structures, defining-words etc. You can use the variable 'forth-custom-words' to make `forth-mode` hilight additional words and constructs. See the docstring of 'forth-words' for details (in Emacs, type *C-h v forth-words*).

'forth-custom-words' is meant to be customized in your `.emacs` file. To customize hilighing in a file-specific manner, set 'forth-local-words' in a local-variables section at the end of your source file (see Section "Variables" in *Emacs Manual*).

Example:

```
0 [IF]
   Local Variables:
   forth-local-words:
      ((("t:") definition-starter (font-lock-keyword-face . 1)
        "[ \t\n]" t name (font-lock-function-name-face . 3))
       ((";t") definition-ender (font-lock-keyword-face . 1)))
   End:
[THEN]
```

11.4 Auto-Indentation

`forth-mode` automatically tries to indent lines in a smart way, whenever you type `TAB` or break a line with *C-m*.

Simple customization can be achieved by setting 'forth-indent-level' and 'forth-minor-indent-level' in your `.emacs` file. For historical reasons `gforth.el` indents per default by multiples of 4 columns. To use the more traditional 3-column indentation, add the following lines to your `.emacs`:

```
(add-hook 'forth-mode-hook (function (lambda ()
   ;; customize variables here:
   (setq forth-indent-level 3)
```

```
    (setq forth-minor-indent-level 1)
)))
```

If you want indentation to recognize non-default words, customize it by setting 'forth-custom-indent-words' in your `.emacs`. See the docstring of 'forth-indent-words' for details (in Emacs, type *C-h v forth-indent-words*).

To customize indentation in a file-specific manner, set 'forth-local-indent-words' in a local-variables section at the end of your source file (see Section "Local Variables in Files" in *Emacs Manual*).

Example:

```
0 [IF]
   Local Variables:
   forth-local-indent-words:
      ((("t:") (0 . 2) (0 . 2))
       ((";t") (-2 . 0) (0 . -2)))
   End:
[THEN]
```

11.5 Blocks Files

`forth-mode` Autodetects blocks files by checking whether the length of the first line exceeds 1023 characters. It then tries to convert the file into normal text format. When you save the file, it will be written to disk as normal stream-source file.

If you want to write blocks files, use `forth-blocks-mode`. It inherits all the features from `forth-mode`, plus some additions:

- Files are written to disk in blocks file format.
- Screen numbers are displayed in the mode line (enumerated beginning with the value of 'forth-block-base')
- Warnings are displayed when lines exceed 64 characters.
- The beginning of the currently edited block is marked with an overlay-arrow.

There are some restrictions you should be aware of. When you open a blocks file that contains tabulator or newline characters, these characters will be translated into spaces when the file is written back to disk. If tabs or newlines are encountered during blocks file reading, an error is output to the echo area. So have a look at the '*Messages*' buffer, when Emacs' bell rings during reading.

Please consult the docstring of `forth-blocks-mode` for more information by typing *C-h v forth-blocks-mode*).

12 Image Files

An image file is a file containing an image of the Forth dictionary, i.e., compiled Forth code and data residing in the dictionary. By convention, we use the extension `.fi` for image files.

12.1 Image Licensing Issues

An image created with `gforthmi` (see Section 12.3.1 [gforthmi], page 331) or `savesystem` (see Section 12.4 [Non-Relocatable Image Files], page 333) includes the original image; i.e., according to copyright law it is a derived work of the original image.

Since Gforth is distributed under the GNU GPL, the newly created image falls under the GNU GPL, too. In particular, this means that if you distribute the image, you have to make all of the sources for the image available, including those you wrote. For details see Section D.2 [GNU General Public License (Section 3)], page 369.

If you create an image with `cross` (see Section 12.3.2 [cross.fs], page 333), the image contains only code compiled from the sources you gave it; if none of these sources is under the GPL, the terms discussed above do not apply to the image. However, if your image needs an engine (a gforth binary) that is under the GPL, you should make sure that you distribute both in a way that is at most a *mere aggregation*, if you don't want the terms of the GPL to apply to the image.

12.2 Image File Background

Gforth consists not only of primitives (in the engine), but also of definitions written in Forth. Since the Forth compiler itself belongs to those definitions, it is not possible to start the system with the engine and the Forth source alone. Therefore we provide the Forth code as an image file in nearly executable form: Integers, strings etc. are represented as-is, data addresses in in relocatable form (section index and offset from the start of the section), and tokens (integers) for code addresses of primitives.

When Gforth starts up, a C routine loads the image file into memory, relocates the addresses and generates native code for the primitive invocations and replaces the tokens with code addresses of this native code (or of the code for the primitives in the engine), then sets up the memory (stacks etc.) according to information in the image file, and (finally) starts executing Forth code.

The default image file is `gforth.fi` (in the `GFORTHPATH`). You can use a different image by using the `-i`, `--image-file` or `--appl-image` options (see Section 2.1 [Invoking Gforth], page 4), e.g.:

```
gforth-fast -i myimage.fi
```

Gforth image files have several limitations:

- Only the sections are in the image; in particular, this means, that an image file cannot represent `ALLOCATE`d memory chunks (and pointers to them). The contents of the stacks are not represented, either.
- Dynamically generated native code is not stored in the image, but is regenerated from the tokens for the primitives on loading.

- The only kinds of relocation supported are: adding the same offset to all cells that represent data addresses; and replacing primitive tokens with code addresses of code in the engine, or dynamically generated code.

  If any complex computations involving addresses are performed, the results cannot be represented in the image file. Several applications that use such computations come to mind:

  – Hashing addresses (or data structures which contain addresses) for table lookup. If you use Gforth's `table`s or `wordlist`s for this purpose, you will have no problem, because the hash tables for them are rebuilt automatically when the system is started. If you use your own hash tables, you will have to do something similar.
  – There's a cute implementation of doubly-linked lists that uses `XOR`ed addresses. You could represent such lists as singly-linked in the image file, and restore the doubly-linked representation on startup.[1]
  – On many architectures addresses are represented in machine code in some shifted or mangled form. You cannot put `CODE` words that contain absolute addresses in this form in a relocatable image file. Workarounds are representing the address in some relative form (e.g., relative to the CFA, which is present in some register), or loading the address from a place where it is stored in a non-mangled form.

12.3 Fully Relocatable Image Files

These image files have relocatable data addresses, and tokens for code addresses. They can be used with different binaries (e.g., with and without debugging) on the same machine, and even across machines with the same data formats (byte order, cell size, floating point format), and they work with dynamic native code generation. However, they are usually specific to the version of Gforth they were created with.

This is the preferred form of image files, and the files `gforth.fi` and `kernl*.fi` are fully relocatable.

There are two ways to create a fully relocatable image file:

12.3.1 `gforthmi`

You will usually use `gforthmi`. If you want to create an image *file* that contains everything you would load by invoking Gforth with `gforth` *args*, you simply say:

```
gforthmi file args
```

E.g., if you want to create an image `asm.fi` that has the file `asm.fs` loaded in addition to the usual stuff, you could do it like this:

```
gforthmi asm.fi asm.fs
```

`gforthmi` is implemented as a sh script and works like this: It produces two non-relocatable images for different addresses and then compares them. Its output reflects this: first you see the output (if any) of the two Gforth invocations that produce the non-relocatable image files, then you see the output of the comparing program, which looks similar to this:

```
code offset=-32541536048
```

[1] In my opinion, though, you should think thrice before using a doubly-linked list (whatever implementation).

```
   xt offset=-32541536080
label offset=-32541536112
section      start      size        dp
0    $7F3680DFE040   $800000   $22EB20
1    $7F368F285000     $5000      $318
...
section      start      size        dp
0    $7FEF133FF200   $800000   $22EB20
1    $7FEF21765000     $5000      $318
...
compare warnings (add section start to offset):
Compare section 0
    A9800     7F368F27849A     7FEF2175849A
   154E58 FFFFFFFFFF99CB10 FFFFFFFFFFBA3B10
...
stats: addr=1378F code=838E xts=12FA labels=24D errs=22
Compare section 1
stats: addr=3C code=0 xts=7 labels=0 errs=0
...
end warnings
```

First the various offsets for code addresses and xts (for generating tokens) are given, then information about the sections (relevant for data addresses); the start address of corresponding sections should be different (that is used for finding out what data is an address and what is not), size and dp (where `here` points to when in this section) for the corresponding sections should be the same.

Next, warnings resulting from comparing data addresses are shown: E.g.,

```
Compare section 0
    A9800     7F368F27849A     7FEF2175849A
```

tells that at offset `$A9800` in section 0, in one image file the data is `$7F368F27849A`, while in the other it is `$7FEF2175849A`.

The cells mentioned in the warnings cannot be represented correctly in the output image and result in unintended values after loading, so you should examine these places in the dictionary and verify that these cells are dead (i.e., not read before they are written).

One way to suppress these warnings is to zero these cells at the end of whatever code is performed by *args* in the command line above. Of course, you should be certain that the overwritten data is dead.

A frequent reason for such warnings is that an `allocate`d address is stored somewhere in the dictionary. In some cases you may want to preserve the data in allocated memory in an image. You can use `save-mem-dict` for that.

There is a deferred word (see Section 6.11.11 [Deferred Words], page 138) that is called before saving an image: `'image`. You can plug your own actions into this word, but make sure you call the action that was there before, too (e.g., with `defers`).

`'image` (–) gforth-experimental “tick-image”

deferred word executed before saving an image

You may want to reorganize the data from the in-image form back to the run-time form when starting up from the image. Plug your startup actions into `'cold` (see Section 12.7 [Modifying the Startup Sequence], page 335).

If you insert the option `--application` in front of the image file name, you will get an image that uses the `--appl-image` option instead of the `--image-file` option (see Section 2.1 [Invoking Gforth], page 4). When you execute such an image on Unix (by typing the image name as command), the Gforth engine will pass all options to the image instead of trying to interpret them as engine options.

If you type `gforthmi` with no arguments, it prints some usage instructions.

There are a few wrinkles: After processing the passed *options*, the words `savesystem` and `bye` must be visible. A special doubly indirect threaded version of the `gforth` executable is used for creating the non-relocatable images; you can pass the exact filename of this executable through the environment variable `GFORTHD` (default: `gforth-ditc`). The normal `gforth` executable is used for comparing the non-relocatable images and creating the relocatable image; you can pass the exact filename of this executable through the environment variable `GFORTH`.

12.3.2 `cross.fs`

You can also use `cross`, a batch compiler that accepts a Forth-like programming language (see Chapter 15 [Cross Compiler], page 355).

`cross` allows you to create image files for machines with different data sizes and data formats than the one used for generating the image file. You can also use it to create an application image that does not contain a Forth compiler. These features are bought with restrictions and inconveniences in programming. E.g., addresses have to be stored in memory with special words (`A!`, `A,`, etc.) in order to make the code relocatable.

12.4 Non-Relocatable Image Files

Non-relocatable images have many disadvantages and there is no good reason to use them. They are documented here mainly for completeness and historical interest (originally Gforth did not support relocatable images).

Non-relocatable image files are simple memory dumps of the dictionary. They are specific to the executable (i.e., `gforth` file) they were created with. What's worse, they are specific to the place on which the dictionary resided when the image was created. Now, there is no guarantee that the dictionary will reside at the same place the next time you start Gforth, so there's no guarantee that a non-relocatable image will work the next time (Gforth will complain instead of crashing, though).

Indeed, on OSs with (enabled) address-space layout randomization (ASLR) non-relocatable images are unlikely to work. In Linux you can run a binary without ASLR by prefixing the invocation with `setarch -R`. You need to do this both when creating the image and when using it. This Linux-specific usage is shown below, and you may have to leave it away or replace it with something else on other OSs (if the OS supports disabling ASLR at all).

In addition, the image contains code addresses of the native code for the primitives; with dynamic native-code generation these code addresses point to code that is not saved in the

image and is not regenerated from the image, either, so you need to disable dynamic native code generation with `--no-dynamic` on image creation if you want to use non-relocatable images. The image can then be loaded into an engine with enabled dynamic native code generation, and the result will be that the code in the image will not use dynamic native code, but code compiled after loading the image will (unless something else disables dynamic native-code generation).

Moreover, the code addresses in a non-relocatable image are engine-specific, so you need to use the image with the same engine (`gforth`, `gforth-fast`, `gforth-itc`) that generated it.

You can create a non-relocatable image file with `savesystem`, e.g., on Linux:

```
#create non-relocatable image app.fi:
setarch -R gforth --no-dynamic app.fs -e "savesystem app.fi bye"
#start Gforth with image app.fi:
setarch -R gforth -i app.fi
```

`savesystem` ("*image*" –) gforth-0.2

12.5 Stack and Dictionary Sizes

If you invoke Gforth with a command line flag for the size (see Section 2.1 [Invoking Gforth], page 4), the size you specify is stored in the dictionary. If you save the dictionary with `savesystem` or create an image with `gforthmi`, this size will become the default for the resulting image file. E.g., the following will create a fully relocatable version of `gforth.fi` with a 64MB dictionary:

```
gforthmi gforth.fi -m 64M
```

In other words, if you want to set the default size for the dictionary and the stacks of an image, just invoke `gforthmi` with the appropriate options when creating the image.

Note: For cache-friendly behaviour (i.e., good performance), you should make the sizes of the stacks modulo, say, 2K, somewhat different. E.g., the default stack sizes are: data: 16k (mod 2k=0); fp: 15.5k (mod 2k=1.5k); return: 15k(mod 2k=1k); locals: 14.5k (mod 2k=0.5k).

12.6 Running Image Files

You can invoke Gforth with an image file *image* instead of the default `gforth.fi` with the `-i` flag (see Section 2.1 [Invoking Gforth], page 4):

```
gforth -i image
```

If your operating system supports starting scripts with a line of the form `#! ...`, you just have to type the image file name to start Gforth with this image file (note that the file extension `.fi` is just a convention). I.e., to run Gforth with the image file *image*, you can just type *image* instead of `gforth -i image`. This works because every `.fi` file starts with a line of this format:

```
#! /usr/local/bin/gforth-0.4.0 -i
```

The file and pathname for the Gforth engine specified on this line is the specific Gforth executable that it was built against; i.e. the value of the environment variable `GFORTH` at the time that `gforthmi` was executed.

You can make use of the same shell capability to make a Forth source file into an executable. For example, if you place this text in a file:

```
#! /usr/local/bin/gforth

." Hello, world" CR
bye
```

and then make the file executable (chmod +x in Unix), you can run it directly from the command line. The sequence `#!` is used in two ways; firstly, it is recognised as a "magic sequence" by the operating system[2] secondly it is treated as a comment character by Gforth. Because of the second usage, a space is required between `#!` and the path to the executable (moreover, some Unixes require the sequence `#! /`).

Most Unix systems (including Linux) support exactly one option after the binary name. If that is not enough, you can use the following trick:

```
#! /bin/sh
: ## ; 0 [if]
exec gforth -m 10M -d 1M $0 "$@"
[then]
." Hello, world" cr
bye \ caution: this prevents (further) processing of "$@"
```

First this script is interpreted as shell script, which treats the first two lines as (mostly) comments, then performs the third line, which invokes gforth with this script (`$0`) as parameter and its parameters as additional parameters (`"$@"`). Then this script is interpreted as Forth script, which first defines a colon definition `##`, then ignores everything up to `[then]` and finally processes the following Forth code. You can also use

```
#0 [if]
```

in the second line, but this works only in Gforth-0.7.0 and later.

The `gforthmi` approach is the fastest one, the shell-based one is slowest (needs to start an additional shell). An additional advantage of the shell approach is that it is unnecessary to know where the Gforth binary resides, as long as it is in the `$PATH`.

`#!` (–) gforth-0.2 "hash-bang"

An alias for `\`

12.7 Modifying the Startup Sequence

You can add your own initialization to the startup sequence of an image through the deferred word `'cold`. `'cold` is invoked just before the image-specific command line processing (i.e., loading files and evaluating (`-e`) strings) starts.

A sequence for adding your initialization usually looks like this:

```
:noname
    Defers 'cold \ do other initialization stuff (e.g., rehashing wordlists)
```

[2] The Unix kernel actually recognises two types of files: executable files and files of data, where the data is processed by an interpreter that is specified on the "interpreter line" – the first line of the file, starting with the sequence #!. There may be a small limit (e.g., 32) on the number of characters that may be specified on the interpreter line.

```
    ... \ your stuff
; IS 'cold
```

After `'cold`, Gforth processes the image options (see Section 2.1 [Invoking Gforth], page 4), and then it performs `bootmessage`, another deferred word. This normally prints Gforth's startup message and does nothing else.

So, if you want to make a turnkey image (i.e., an image for an application instead of an extended Forth system), you can do this in several ways:

- If you want to do your interpretation of the OS command-line arguments, hook into `'cold`. In that case you probably also want to build the image with `gforthmi --application` (see Section 12.3.1 [gforthmi], page 331) to keep the engine from processing OS command line options. You can then do your own command-line processing with `next-arg`
- If you want to have the normal Gforth processing of OS command-line arguments, but specify your own command-line options, hook into `process-option`.
- If you want to have more options in addition to the ones that come with Gforth, define words into the `options` vocabulary.
- If you want to display your own boot message, hook into `bootmessage`.

In either case, you probably do not want the word that you execute in these hooks to exit normally, but use `bye` or `throw`. Otherwise the Gforth startup process would continue and eventually present the Forth command line to the user.

`'cold` (–) gforth-0.2 "tick-cold"

Hook (deferred word) for things to do right before interpreting the OS command-line arguments. Normally does some initializations that you also want to perform.

`bootmessage` (–) gforth-0.4

Hook (deferred word) executed right after interpreting the OS command-line arguments. Normally prints the Gforth startup message.

`process-option` (*addr u – ... xt* | *0*) gforth-0.7

Recognizer that processes an option, returns an execute-only xt to process the option

13 Optimizations

This chapter describes the threaded-code basis of `gforth-fast` (essentially as Gforth was in 2001) and the optimizations we have applied since then; the examples show disassembled AMD64 machine code. It also describes what `gforth` and `gforth-itc` do differently from `gforth-fast`.

13.1 Threaded code

Direct-threaded code is a fast interpretation technique, where primitives are represented as code addresses in the interpreted code, and each code address points to the code that implements the primitive.

We will use

```
: squared dup * ;
```

as example. The threaded code for this source code looks as follows:

```
address       content
$7F6208AA7FE0 $5640443C831C \ at $5640443C831C the code of DUP starts
$7F6208AA7FE8 $5640443C74DF \ at $5640443C74DF the code of *   starts
$7F6208AA7FF0 $5640443C6BE6 \ at $5640443C6BE6 the code of ;s  starts
```

The exact numbers vary between builds and between runs of the same build. Looking at these primitives:

```
see dup
Code dup
   $5640443C831C: add    $0x8,%rbx
   $5640443C8320: mov    %r13,(%r10)
   $5640443C8323: sub    $0x8,%r10
   $5640443C8327: mov    (%rbx),%rax
   $5640443C832A: jmp    *%rax
end-code
 ok
see *
Code *
   $5640443C74DF: add    $0x8,%rbx
   $5640443C74E3: imul   0x8(%r10),%r13
   $5640443C74E8: add    $0x8,%r10
   $5640443C74EC: mov    (%rbx),%rax
   $5640443C74EF: jmp    *%rax
end-code
 ok
see ;s
Code ;s
   $5640443C6BE6: mov    (%r14),%rbx
   $5640443C6BE9: add    $0x8,%r14
   $5640443C6BED: mov    (%rbx),%rax
   $5640443C6BF0: jmp    *%rax
end-code
```

We see that they start at the addresses given in the threaded code. We also see the following instructions (not consecutive) in `dup` and `*`, and the last two of these instructions in `;s` (the primitive compiled by `;`).

```
add    $0x8,%rbx   \ update threaded-code instruction pointer (IP)
mov    (%rbx),%rax \ get code address from IP
jmp    *%rax       \ jump to code address
```

This sequence ensures that the execution continues with the next primitive and is called NEXT. `%rbx` is the threaded-code instruction pointer (IP), `%rax` is just used as a temporary register (the actual registers can be different in different builds, even on the same architecture). The NEXT that entered the word `squared` left `%rbx` at $7F6208AA7FE0 and jumped to $5640443C831C (`dup`). The NEXT in `dup` advances `%rbx` to the next threaded-code address, then loads the code address into `%rax` and finally jumps there.

In this way one primitive after another is executed, and for threaded-code control flow, `%rbx` is set differently (as in `;s`, which returns to the caller of `squared`).

The remaining instructions in `dup` and `*` perform the work we are actually interested in; you find the top-of-data-stack (TOS) in `%r13`, and the data stack pointer (SP) in `%r10`. The data stack grows towards smaller addresses.

Gforth compiles most non-primitives into a primitive followed by an inline argument (primitive-centric code). E.g., a variable ist compiled to the primitive `lit` with the body address of the variable as inline argument. A word *X* defined with `create`... (*xt*) `set-does>` is compiled as `lit` with the body address of *X* as inline argument, followed by `compile,`ing *xt*.

That's all nice and well for code coming out of `compile,`, but `execute` expects an xt that fits in one cell. How do we transport the code address and the data address (for a variable) in one cell? Here's (a slightly simplified) `execute`:

```
add    $0x8,%r10         \ adjust SP
mov    -0x10(%r13),%rax \ load code address from code field of xt
mov    %r13,%rdx         \ store xt in W (for use by the executed code)
mov    (%r10),%r13       \ load the new TOS from SP
jmp    *%rax             \ jump to the code address
```

The code addresses that `execute` jumps to are not normal primitives, they are doers: routines like that at `dovar:` (all these routines have names starting with "do", that's why they are called "doers"). The routine at `dovar:` looks as follows:

```
add    $0x8,%rbx   \ NEXT (adjust IP)
mov    %r13,(%r10) \ store TOS (make room for new TOS)
sub    $0x8,%r10   \ adjust SP
mov    %rdx,%r13   \ move W to TOS
mov    (%rbx),%rax \ NEXT
jmp    *%rax       \ NEXT
```

Note that the indirection through a code field as `execute` is doing is what indirect-threaded code does, so we actually have a hybrid of direct-threaded and indirect-threaded code: direct-threaded code for `compile,`d code, indirect-threaded code for `execute` (deferred words also use this).

Is there any chance of doing direct-threaded dispatch when indirect-threaded dispatch is called for, or vice versa? No, because IP always points to direct-threaded code, while every dispatch through an xt is indirect-threaded.

For a word *X* defined with *xt1* `set-does>`, executing *X* through its xt (*xt2*) has another twist: The code address involved is `dodoes:`, and it gets *xt2* in W (`rdx`). It loads the *xt1* from the `>hmextra` slot of the header methods of *X* (see Section 6.34.2 [Header methods], page 300). Here's the code of dodoes (slightly simplified):

```
mov    %r13,(%r10)      \ store TOS (make room for new TOS)
mov    -0x8(%rdx),%rax  \ load header methods of X, using W
mov    %rdx,%r13        \ move W to TOS, i.e. push X's body address
sub    $0x8,%r10        \ update SP
mov    0x18(%rax),%rdx  \ load xt1 from >hmextra into W
mov    -0x10(%rdx),%rax \ load code address from code field of xt
jmp    *%rax            \ jump to the code address
```

Note that dodoes is not just jumped to by a dispatch through an xt (*xt2*), it performs a dispatch through an xt (*xt1*) itself, so it also loads from a code field and sets up W.

Primitive-centric hybrid direct/indirect threaded code is described in: M. Anton Ertl, *Threaded code variations and optimizations (extended version)* (`https://www.complang.tuwien.ac.at/papers/ertl02.ps.gz`), Forth-Tagung '02.

You can invoke `gforth-fast` to execute just threaded code without additional optimizations with

```
gforth-fast --no-dynamic --ss-number=0
```

13.2 Dynamic Superinstructions

The native code of the primitives in our `squared` example can be copied, with the copies being concatenated. Because the NEXTs of the primitives `dup` and `*` then just jump to the addresses right behind, the last two NEXT instructions of each of these primitives can be left away, resulting in:

```
$7F2BFA49D350: add    $0x8,%rbx      \ dup NEXT IP update
$7F2BFA49D354: mov    %r13,(%r10)    \ dup
$7F2BFA49D357: sub    $0x8,%r10      \ dup
$7F2BFA49D35B: add    $0x8,%rbx      \ * NEXT IP update
$7F2BFA49D35F: imul   0x8(%r10),%r13 \ *
$7F2BFA49D364: add    $0x8,%r10      \ *
$7F2BFA49D368: mov    (%r14),%rbx    \ ;s load IP from return stack
$7F2BFA49D36B: add    $0x8,%r14      \ ;s update return stack pointer
$7F2BFA49D36F: mov    (%rbx),%rax    \ ;s NEXT
$7F2BFA49D372: jmp    *%rax          \ ;s NEXT
```

The NEXT instructions of `;s` cannot be left away, because `;s` does not fall through to the next primitive, but performs control flow and needs to jump to the native code corresponding to the target threaded code; and it gets the code address of that native code through a regular direct-threaded code dispatch.

The threaded code for `squared` now points to the start of the respective primitives in the copied code, not to the original instances of this code; e.g., the first cell in `squared`

now points to $7F2BFA49D350, the second to $7F2BFA49D35B, and the third to $7F2BFA49D368.

`See-code` displays the threaded code interleaved with the native code:

```
see-code squared
<squared>     dup    1->1
   $7F2BFA49D350: add    $0x8,%rbx
   $7F2BFA49D354: mov    %r13,(%r10)
   $7F2BFA49D357: sub    $0x8,%r10
<squared+$8>  *    1->1
   $7F2BFA49D35B: add    $0x8,%rbx
   $7F2BFA49D35F: imul   0x8(%r10),%r13
   $7F2BFA49D364: add    $0x8,%r10
<squared+$10> ;s    1->1
   $7F2BFA49D368: mov    (%r14),%rbx
   $7F2BFA49D36B: add    $0x8,%r14
   $7F2BFA49D36F: mov    (%rbx),%rax
   $7F2BFA49D372: jmp    *%rax
```

But note that the threaded code and native code are not interleaved in memory; `<squared>` is $7F2BFAAA7FE0, more than 6MB away from the native code.

An example with more control flow is

```
: mymin 2dup < if drop else nip then ;  ok
see-code mymin
<mymin>       2dup    1->1
   $7F2BFA49D374: add    $0x8,%rbx
   $7F2BFA49D378: mov    0x8(%r10),%rax
   $7F2BFA49D37C: sub    $0x10,%r10
   $7F2BFA49D380: mov    %r13,0x10(%r10)
   $7F2BFA49D384: mov    %rax,0x8(%r10)
<mymin+$8>    <    1->1
   $7F2BFA49D388: add    $0x8,%rbx
   $7F2BFA49D38C: add    $0x8,%r10
   $7F2BFA49D390: cmp    %r13,(%r10)
   $7F2BFA49D393: setl   %r13b
   $7F2BFA49D397: movzbl %r13b,%r13d
   $7F2BFA49D39B: neg    %r13
<mymin+$10>   ?branch    1->1
<mymin+$18>   <mymin+$38>
   $7F2BFA49D39E: add    $0x10,%rbx
   $7F2BFA49D3A2: add    $0x8,%r10
   $7F2BFA49D3A6: test   %r13,%r13
   $7F2BFA49D3A9: mov    -0x8(%rbx),%rsi
   $7F2BFA49D3AD: mov    (%r10),%r13
   $7F2BFA49D3B0: jne    0x7f2bfa49d3ba
   $7F2BFA49D3B2: mov    (%rsi),%rax
   $7F2BFA49D3B5: mov    %rsi,%rbx
   $7F2BFA49D3B8: jmp    *%rax
```

```
<mymin+$20>   drop    1->1
   $7F2BFA49D3BA: add    $0x8,%rbx
   $7F2BFA49D3BE: mov    0x8(%r10),%r13
   $7F2BFA49D3C2: add    $0x8,%r10
<mymin+$28>   branch    1->1
<mymin+$30>   <mymin+$40>
   $7F2BFA49D3C6: add    $0x10,%rbx
   $7F2BFA49D3CA: mov    -0x8(%rbx),%rbx
   $7F2BFA49D3CE: mov    (%rbx),%rax
   $7F2BFA49D3D1: jmp    *%rax
<mymin+$38>   nip    1->1
   $7F2BFA49D3D3: add    $0x8,%rbx
   $7F2BFA49D3D7: add    $0x8,%r10
<mymin+$40>   ;s    1->1
   $7F2BFA49D3DB: mov    (%r14),%rbx
   $7F2BFA49D3DE: add    $0x8,%r14
   $7F2BFA49D3E2: mov    (%rbx),%rax
   $7F2BFA49D3E5: jmp    *%rax
```

`If` compiles a `?branch` with the target address as immediate argument, and `else` compiles a `branch` with the target address as immediate argument, while both `else` provides the target address for the `?branch` and `then` provides the target addres for the `branch`. Both branch primitives are control flow words, load the code address from the target address and jump there if the branch is taken. If the `?branch` is not taken, its native code branches around this control-flow code and falls through to the following primitive.

The name "superinstruction" means that a sequence of primitives (or VM instructions) is combined into one VM instruction, meaning it performs one threaded-code NEXT. "Dynamic superinstruction" means that the superinstructions are not formed at Gforth build time, but at Gforth run time. In the `mymin` example, we have superinstructions for the sequence `2dup < ?branch`, for `drop branch` and for `nip ;s`. But note that the `?branch` does not always perform a NEXT, so one might also see `2dup < ?branch drop branch` as a superinstruction. Moreover, in some cases only the `;s` part of the `nip ;s` superinstruction is performed.

The code copying involved in dynamic superinstructions cannot be performed with all primitives, only with those that are relocatable. If a primitive is not relocatable, the primitive before it has to end with the NEXT instructions, and the threaded-code for that primitive points to the original location of the primitive, like in ordinary threaded code. I.e., this approach can always fall back to threaded code if something is amiss. Fortunately, the most frequently used primitives are usually relocatable.

The code above was generated by using `gforth-fast --ss-states=1 --ss-number=0 --opt-ip-updates=0`; this disables the optimizations discussed in the next sections.

You can disable dynamic superinstructions with `--no-dynamic`, but that also disables several other optimizations.

The general approach of copying fragments of native code that have been generated by a C compiler is called *code-copying compilaton*.

You can read more about the code-copying used in Gforth in: M. Anton Ertl, Bernd Paysan, *Code-Copying Compilation in Production — An Experience Report (*`http://www.euroforth.org/ef25/papers/ertl.pdf`*)*, EuroForth 2025 proceedings.

13.3 Static Superinstructions

Static superinstructions combine a sequence of primitives and are added to the `gforth-fast` engines at Gforth build time. They are used even if dynamic native-code generation is disabled with `--no-dynamic`.

As an example, Gforth has a static superinstruction for the sequence `< ?branch` that occurs in `mymin`. Enabling static superinstructions by leaving away the `--ss-number=0` from the command line above results in the right column below:

```
    gforth-fast --ss-states=1 --opt-ip-updates=0
with --ss-number=0              without --ss-number=0
<mymin+$8>    <    1->1         <mymin+$8>    < ?branch     1->1
 add    $0x8,%rbx               <mymin+$10>   ?branch    1->1
 add    $0x8,%r10               <mymin+$18>   <mymin+$38>
 cmp    %r13,(%r10)              add    $0x18,%rbx
 setl   %r13b                    mov    0x8(%r10),%rax
 movzbl %r13b,%r13d              add    $0x10,%r10
 neg    %r13                     mov    -0x8(%rbx),%rsi
<mymin+$10>   ?branch    1->1    cmp    %r13,%rax
<mymin+$18>   <mymin+$38>        mov    (%r10),%r13
 add    $0x10,%rbx               jl     0x7f51790998b4
 add    $0x8,%r10                mov    (%rsi),%rax
 test   %r13,%r13                mov    %rsi,%rbx
 mov    -0x8(%rbx),%rsi          jmp    *%rax
 mov    (%r10),%r13
 jne    0x7f8c2a49d396
 mov    (%rsi),%rax
 mov    %rsi,%rbx
 jmp    *%rax
```

The benefit of the static superinstruction is that the comparison result does not need to be converted into a cell and that cell then checked the make the branch decision.

The threaded-code slot for `<` contains the address of the static superinstruction, the code slot for the `?branch` still exists, but is not used.

There are only 63 (at the time of this writing) static superinstructions in `gforth-fast`, because the combination of stack caching and the IP update optimization provides the same benefits in many cases. Moreover, currently static superinstructions can only start and end in the canonical stack state, eliminating some potentially beneficial uses.

You can determine how many of the static superinstructions you want to use with `--ss-number=`*n*. The first *n* superinstructions will then be used, no finer-granularity control is provided. Static superinstructions will only be used if they result in code that is at least as short as can be achieved with the other optimizations. You can see the static superinstructions in `gforth-fast` with (on Unix):

```
gforth-fast --print-prims -e bye |& cut -b -18|awk 'NF>1'
```

You can read a little more about static superinstructions in M. Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan, *vmgen — A Generator of Efficient Virtual Machine Interpreters (*`https://www.complang.tuwien.ac.at/papers/ertl+02.ps.gz`*)*, Software: Practice and Experience 32(3), p. 265–294, 2002.

13.4 Stack caching

In the examples up to now the top-of-stack has been kept in a register, which already provides a good benefit compared to keeping all stack items in memory (which is used by the `gforth` engine). Here's the payload (i.e., without NEXT) of `*` for `gforth-fast` and `gforth`:

```
gforth-fast                 gforth
SP=%r10, TOS=%r13           SP=%r14
                            mov    0x8(%r14),%rax
imul   0x8(%r10),%r13       imul   (%r14),%rax
add    $0x8,%r10            add    $0x8,%r14
                            mov    %rax,(%r14)
```

However, by using multiple stack representations, this can be improved even more. In particular, `gforth-fast` uses one stack representation (called `0`) with no stack item in memory, one (`1`) with one stack item in memory, and so on.

Gforth has variants of the most common primitives for the most useful transitions between stack representations. It uses a shortest-path algorithm to select a combination of primitive variants that fits together (i.e., where each primitive variant starts with the representation in which the previous primitive variant finished) and that results in the shortest native-code size (by default).

The transitions are shown as *`s->t`*), meaning that the primitive variant starts with representation *s* and finishes in representation *t*. One can also consider these as transitions in a state machine, so we also talk about a stack representations as a *stack state*.

For the `squared` example that results in:

```
<squared>     dup    1->2
   $7F04266893D7: add    $0x8,%rbx
   $7F04266893DB: mov    %r13,%r15
<squared+$8>  *    2->1
   $7F04266893DE: add    $0x8,%rbx
   $7F04266893E2: imul   %r15,%r13
<squared+$10> ;s    1->1
   $7F04266893E6: mov    (%r14),%rbx
   $7F04266893E9: add    $0x8,%r14
   $7F04266893ED: mov    (%rbx),%rax
   $7F04266893F0: jmp    *%rax
```

Here `%13` contains the first register above the memory part of the stack, `%15` the second one. Note that it depends on the representation which of the two is the top-of-stack: In `0` all stack items are in memory, in `1` the TOS is in `%r13`, and in `2` the TOS is in `%r15` and the stack item below is in `%r13`.

Multi-representation stack-caching reduces memory accesses as well as stack pointer updates.

`gforth-fast` falls back to the canonical representation `1` on control flow (including calls and `exit`s), when dealing with non-relocatable primitives, or when a static superinstruction is selected.

Multi-state stack caching is controlled with `--ss-states=n`, where n gives the number of states: For n=1, only the canonical state `1` is used, for n=2 also state `0`, and the next states enabled are `2` (for n>=3) and `3` (for n>=4). On a number of platforms (in particular, AMD64, ARM A64, RISC-V), `gforth-fast` supports 4 stack representations (`0`–`3`), on some others only 2 (`0`, `1`); the canonical representation is always `1`.

The code above was generated by using `gforth-fast --opt-ip-updates=0`.

You can read more about static superinstructions in M. Anton Ertl, David Gregg, *Stack Caching in Forth (*`http://www.euroforth.org/ef05/ertl-gregg05.pdf`*)*, EuroForth 2005 proceedings.

13.5 IP-update optimization

Many primitives do not need an up-to-date IP. The IP-update optimization delays updating the IP until an up-to-date IP is needed. For the `squared` example, the result is:

```
<squared>     dup    1->2
   $7F9F4AC739F7: mov    %r13,%r15
<squared+$8>  *    2->1
   $7F9F4AC739FA: imul   %r15,%r13
<squared+$10> ;s    1->1
   $7F9F4AC739FE: mov    (%r14),%rbx
   $7F9F4AC73A01: add    $0x8,%r14
   $7F9F4AC73A05: mov    (%rbx),%rax
   $7F9F4AC73A08: jmp    *%rax
```

In this case none of the primitives needs an up-to-date IP; the first two don't access IP at all, while `;s` writes to IP itself and therefore does not need an up-to-date IP, either.

If an up-to-date IP is needed, a code fragment is inserted that adds the appropriate number to the IP. There is only a limited amount of such fragments, but that's good enough in most cases; and if it's not enough one can insert several such fragments.

An up-to-date IP is typically needed for primitives that access immediate arguments, and before inserting a NEXT in front of a non-relocatable primitive, and at a control-flow join. The IP-update optimization also reduces the need for an up-to-date IP for immediate arguments by providing variants of the most common primitives with immediate arguments that work with an outdated IP (up to some limit).

One might think that the lower number of instructions is the main benefit of this optimization, and this benefit exists, but is relatively small (speedups factors around 1.1 on high-performance hardware on call-dominated benchmarks). But the major benefit on modern high-performance hardware is that in loop-dominated programs the critical data-dependence chain is shortened, resulting in a speedup by a factor of 2 or more for these programs on recent high-performance cores, when the following suboptimization is also performed.

Without IP update optimization `branch` and friends use a load (3-5 cycles latency) to get the target address. The IP-update optimization changes the IP with an add (at the

beginning of the branch code) to point to the target, so the load is now unnecessary. This can be seen in the `mymin` example:

```
gforth-fast --opt-ip-updates=0   gforth-fast
<mymin>       2dup    1->3       <mymin>       2dup    1->3
 add    $0x8,%rbx
 mov    0x8(%r10),%r15            mov    0x8(%r10),%r15
 mov    %r13,%r9                  mov    %r13,%r9
<mymin+$8>    <    3->2          <mymin+$8>    <    3->2
 add    $0x8,%rbx
 cmp    %r9,%r15                  cmp    %r9,%r15
 setl   %r15b                     setl   %r15b
 movzbl %r15b,%r15d               movzbl %r15b,%r15d
 neg    %r15                      neg    %r15
<mymin+$10>   ?branch    2->1    <mymin+$10>   ?branch    2->1
<mymin+$18>   <mymin+$38>        <mymin+$18>   <mymin+$38>
 add    $0x10,%rbx                add    $0x38,%rbx
 mov    -0x8(%rbx),%rsi           mov    (%rbx),%rax
 test   %r15,%r15                 test   %r15,%r15
 jne    <to the next add>         jne    <to the next mov>
 mov    (%rsi),%rax
 mov    %rsi,%rbx
 jmp    *%rax                     jmp    *%rax
<mymin+$20>   drop    1->1       <mymin+$20>   drop    1->1
 add    $0x8,%rbx
 mov    0x8(%r10),%r13            mov    0x8(%r10),%r13
 add    $0x8,%r10                 add    $0x8,%r10
<mymin+$28>   branch    1->1     <mymin+$28>   branch    1->1
<mymin+$30>   <mymin+$40>        <mymin+$30>   <mymin+$40>
 add    $0x10,%rbx                add    $0x8,%rbx
 mov    -0x8(%rbx),%rbx
 mov    (%rbx),%rax               mov    (%rbx),%rax
 jmp    *%rax                     jmp    *%rax
<mymin+$38>   nip    1->1        <mymin+$38>   nip    1->1
 add    $0x8,%rbx                 add    $0x8,%r10
 add    $0x8,%r10                 add    $0x8,%rbx
<mymin+$40>   ;s    1->1         <mymin+$40>   ;s    1->1
 mov    (%r14),%rbx               mov    (%r14),%rbx
 add    $0x8,%r14                 add    $0x8,%r14
 mov    (%rbx),%rax               mov    (%rbx),%rax
 jmp    *%rax                     jmp    *%rax
```

The IP update in the `nip` code is due to having a control-flow join (`then`) after the `nip`. The IP is dead at that point, but Gforth's code generator does not look ahead far enough to know that and leave it away.

One other thing you may notice is that the static superinstruction for `< ?branch` is not used. That's because stack caching results in a shorter code sequence than the static superinstruction (which with the current limitations would require the `1->1` variant of `2dup`.

The IP-update optimization is controlled with `--opt-ip-updates=n`, where n=0 turns off the optimization completely, other values are described in more detail in the See [option-opt-ip-updates], page 7.

You can read more about the IP-update optimization in: M. Anton Ertl, Bernd Paysan, *The Performance Effects of Virtual-Machine Instruction Pointer Updates (*`https://drops.dagstuhl.de/storage/00lipics/lipics-vol313-ecoop2024/LIPIcs.ECOOP.2024.14/LI`
In 38th European Conference on Object-Oriented Programming (ECOOP 2024).

13.6 Literal-based optimizations

Gforth delays the code generation for literals; instead, it pushes the literal on a literal stack, and the implementation of `compile,` for a word (see Section 6.11.10.7 [User-defined compile-comma], page 134) can inspect the literal stack and possibly generate better code. This optimization happens at the threaded-code level, so one can see its effect already with `see`; in some cases `simple-see` is necessary to see the effect.

Here's an example that shows some of the optimizations based on the literal stack:

```
: foo 15 1 + * 2 pick 3 - ;

see foo \ output below
: foo
  #4 lshift third #-3 + ; ok

simple-see foo \ output below
<foo>         lit    0->0
<foo+$8>      #4
<foo+$10>     lshift    0->0
<foo+$18>     third    0->0
<foo+$20>     lit+    0->0
<foo+$28>     #-3
<foo+$30>     ;s    0->0  ok
```

The optimizer for `+` sees and consumes two values on the literal stack and constant-folds them into the value 16 on the literal stack. The optimizer for `*` sees and consumes 16 on the literal stack, and generates `4 lshift` instead of the more expensive `#16 *`. The optimizer for `pick` sees and consumes 2 from the literal stack and generates `third`. The optimizer for `-` sees only one value on the literal stack, and generates the `lit+` primitive with the negated value as immediate argument; in the `see` output, the `lit+` is decompied as a literal followed by `+`.

This optimization if performed at the Forth level and is not controlled by engine options.

13.7 Differences between the engines

`gforth-fast` performs all optimizations described above.

`gforth` is the debugging engine and should report errors as soon as possible, and where they happen, including a full backtrace. Therefore:

- It does not use static superinstructions: the user wants to know in which of the primitives the error happens.

- It has `0` as canonical stack representation (otherwise some stack underflows would go unnoticed) and it disables multi-representation stack caching.
- It disables the IP-update optimization. It also stores the IP to memory at the start of every primitive, and keeps the return-stack pointer in memory. These measures make it possible to report the error location when the error causes a signal (e.g., SIGSEGV) that is then translated into a Forth exception (e.g., throw code -9 (invalid memory access)).

As a result, `gforth` generates substantially more native code; for our `squared` example:

```
  gforth-fast                  gforth
RP=%r14                      RP=0x58(%r13), saved_ip=0x50(%r13)
<squared>     dup    1->2    <squared>     dup    0->0
                              mov    %r15,0x50(%r13)
                              add    $0x8,%r15
                              mov    (%r14),%rax
                              sub    $0x8,%r14
 mov    %r13,%r15             mov    %rax,(%r14)
<squared+$8>  *    2->1      <squared+$8>  *    0->0
                              mov    %r15,0x50(%r13)
                              add    $0x8,%r15
                              mov    0x8(%r14),%rax
 imul   %r15,%r13             imul   (%r14),%rax
                              add    $0x8,%r14
                              mov    %rax,(%r14)
<squared+$10> ;s    1->1     <squared+$10> ;s    0->0
                              mov    %r15,0x50(%r13)
                              mov    0x58(%r13),%rax
 mov    (%r14),%rbx           mov    (%rax),%r10
 add    $0x8,%r14             add    $0x8,%rax
                              mov    %rax,0x58(%r13)
                              mov    %r10,%r15
 mov    (%rbx),%rax           mov    (%r15),%rcx
 jmp    *%rax                 jmp    *%rcx
```

`gforth-itc` provides pure indirect-threaded code, for running code that relies on its properties. For good measure, it provides all the debugging benefits of the `gforth` engine, and inherits all of `gforth`'s debugging advantages as well as performance disadvantages. Moreover, it disables all other optimizations. It disables dynamic native-code generation, because each primitive is invoked through its code field, so there is no way to have several instances of the primitive with different code addresses. And it uses the classic indirect-threaded code compiler `,` instead of the intelligent `compile,`, so literal-based optimizations do not work, either. Note that the code coming from `gforth.fi` has not been compiled with `,`, but code that you compile after loading the image uses `,`.

13.8 Performance

The performance of `gforth-fast` is comparable to that of SwiftForth 4.0, while VFX Forth is faster by very roughly a factor of 2. For the benchmarks where a C version is avail-

able, `gcc -O0` has a similar performance to `gforth-fast`. See Figure 1 of: M. Anton Ertl, Bernd Paysan, *Code-Copying Compilation in Production — An Experience Report (*`http://www.euroforth.org/ef25/papers/ertl.pdf`*)*, EuroForth 2025 proceedings.

Below you see the performance of various Gforth engines with various options on a Ryzen 8700G (Zen4); the numbers are times in milliseconds, lower is better. First the progression of optimizations for `gforth-fast` shown in this chapter is measured, then `gforth` and `gforth-itc`:

```
sieve bubble matrix    fib    fft
145.5  163.3  114.1  166.4   79.6 gforth-fast --no-dynamic --ss-number=0
 54.0   61.8   32.4   49.2   18.1 gforth-fast --opt-ip-updates=0 --ss-states=1 --ss-nu
 53.7   55.1   32.5   41.7   18.3 gforth-fast --opt-ip-updates=0 --ss-states=1
 53.4   59.5   31.8   42.7   14.7 gforth-fast --opt-ip-updates=0
 20.1   21.4   11.7   27.1   12.8 gforth-fast
 82.4  128.7   49.7  106.6   32.9 gforth
173.5  186.3  144.9  282.4   87.9 gforth-itc
```

`gforth` is slower by roughly a factor of 4 on the small benchmarks (where the IP-update optimization is particularly beneficial, so `gforth`'s performance may be closer to `gforth-fast`'s on application benchmarks). `gforth-itc` is slower than `gforth` by roughly another factor of 2.

You can measure your own machine with

```
taskset -c 1 sh -c '
 gforth-fast onebench.fs|grep sieve
 for i in \
    "gforth-fast --no-dynamic --ss-number=0" \
    "gforth-fast --opt-ip-updates=0 --ss-states=1 --ss-number=0" \
    "gforth-fast --opt-ip-updates=0 --ss-states=1" \
    "gforth-fast --opt-ip-updates=0" \
    "gforth-fast" \
    "gforth" \
    "gforth-itc"
   do echo "`$i onebench.fs|grep -v sieve`" $i; done'
```

The `taskset -c 1` limits the rest of the command to run on CPU (core) 1, so that you measure that core design (recent CPUs tend to have several different core designs) and to make sure that the core is revved up. Use different numbers to measure different cores. Leave the `taskset -c 1` away if your OS does not support that command.

You can find numbers for `gforth-fast onebench.fs` on various machines in `Benchres` (`https://raw.githubusercontent.com/forthy42/gforth/refs/heads/master/Benchres`).

14 Engine Construction

This chapter discusses several aspects of how Gforth's engines are constructed. Reading this chapter is not necessary for programming with Gforth. It may be helpful for finding your way in the Gforth sources.

14.1 Portability and Performance

An important goal of the Gforth Project is availability across a wide range of personal machines, in particular different architectures and OSs. This was particularly important in the 1990s with it's wide variety of RISC architectures (that are still supported by Gforth, but we currently cannot test all of them).

However, even in this century new architectures have appeared, and Gforth's portability allows us to support them early on; it usually runs out of the box (but without engine optimizations) on hardware it does not know about; e.g., in 2017 we built it on a QEMU (virtual machine) implementation of RISC-V, before adding more specific support in 2022.

The more specific support ideally consists of just checking that the engine optimizations work, and enabling them in the configuration script. We also define `BFD_ARCH` and `BFD_MACH` for `disasm2`, the disassembler based on GNU binutils (the generic support uses `disasm-gdb` or `dump` instread). On some architectures we have to define a `FLUSH_ICACHE` and tune, e.g., the number of stack representations.

In any case, the specific support is so easy that we take an hour or so to add it once we have access to hardware. E.g., Gforth supported AMD64 (first hardware release 2003) in 2003, while SwiftForth and VFX took until the 2020s to do so. Gforth has supported ARM A64 (first hardware release 2013) in 2014, while SwiftForth and VFX have not yet (in 2026) released an ARM A64 version. Gforth has had specific support for RISC-V since 2022, while SwiftForth and VFX probably don't see enough interest in RISC-V yet to support it.

Our tool for achieving this portability has been to use GNU C as a substrate, in particular the labels-as-values feature (see Section "Labels as Values" in *GNU C Manual*)) present since gcc-2.0 (released February 1992; we started to work on Gforth in July 1992). This extension is supported by gcc (recommended), clang (with slow compiles in clang) and others.

The labels-as-values feature has allowed us to implement threaded code, as well as native-code copying optimizations such as dynamic superinstructions (see Section 13.2 [Dynamic Superinstructions], page 339) and the optimizations based on them. The use of gcc has been a bumpy ride at times (especially in the 2000s), but we managed to work around some of the bumps and the rest were eventually fixed in gcc. Concerning portability, gcc is available for a wide range of architectures and platforms.

Writing in a portable language has the reputation of producing code that is slower than assembly language. However, already at the beginning Gforth ran faster than threaded-code Forth systems written in assembly language like Win32Forth. Over the course of its development we further improved the performance (see Chapter 13 [Optimizations], page 337), and now it is competetive in performance with native-code systems like SwiftForth.

For portability across OSs we used POSIX interfaces, and this works nicely for many OSs, including Linux, MacOS and Android. For Windows it also works, thanks to Cygwin, but building a Windows package is so much work that the we currently recommend using a Linux package of Gforth on WSL. There is no iOS port, however.

14.2 Primitives

If you `locate +`, you see the following source code from the file `prim`:

```
+       ( n1 n2 -- n )          core    plus
n = n1+n2;
```

This code is written in a variant of the `vmgen` language. The first line provides the primitive Forth name, stack effect (including stack item names), wordset name, and pronounciation (which also serves as the C name of the primitive), while the remaining lines are C code that refer to the stack items by name.

Our `vmgen` variant extracts a lot of information from the stack effect notations: The number of items popped from and pushed on the stack, their type, and by what name they are referred to in the C code. It generates a C code from this, incorporating the C code given explicitly. For more details See *Vmgen*.

When you `see +`, you see the variant of the primitive that's used in threaded code: starting in the canonical state and ending in the canonical state, with NEXT (including IP update):

```
see +
Code +
   $55DE56E88427: add    $0x8,%rbx
   $55DE56E8842B: add    $0x8,%r10
   $55DE56E8842F: add    (%r10),%r13
   $55DE56E88432: mov    (%rbx),%rax
   $55DE56E88435: jmp    *%rax
end-code
```

You can see the source and disassembled code of other primitives in the same way.

14.3 Native code of primitive variants

If you `see-code` a colon definition in `gforth`, for the primitives in the colon definition you see native code that is copied from the code that you `see` for the primitive, without or with the part of the NEXT at the end.

If you `see-code` colon definitions in `gforth-fast`, you often see code coming from other variants of the primitive, often without the IP update, or with a different IP update. This code is copied from other variants of the primitive in the engine. There is no simple way to see these variants, but starting the engine with `--print-prims` prints out information about all primitive variants when ending the session. If you are interested in a particular primitive, you can grep for it in the output, e.g.:

```
$ gforth-fast --print-prims -e bye |& grep '^[+] '
+                 1-1  0   67    0   77 0x5555555bc427 len= 4+  7+ 3 send=0
+                 2-1  0  802    0  189 0x5555555c24f9 len= 4+  3+ 3 send=0
+                 3-2  0  803    0   77 0x5555555c2505 len= 4+  3+ 3 send=0
```

The first column contains the name of the primitive, the second the stack-state transition. Then comes the IP offset (discussed below), next the primitive index (just a number), then (if non-zero) the index of the branch-to-IP variant discussed below, then the number of instances of this primitive variant in the generated native code, then the code address of

the native code followed by its length, and finally an indication of whether the primitive ends a dynamic superinstruction.

The length is split into three parts: first the IP update, then the payload, and finally the load of the code address; the actual jump is not included (that's a workaround bad code generation in some gcc versions).

In the above you see that there are three variants of `+` with different stack-state transitions (see Section 13.4 [Stack caching], page 343), that the `2-1` variant is generated more often than the other two combined, and that the `1-1` variant has longer native code than the others.

If you want to use the address and the length to disassemble the primitive variant, you usually need to invoke Gforth on Linux with `setarch -R gforth-fast` to disable ASLR (or equivalent commands on other OSs), on both the `--print-prims` run and the following disassembling session.

E.g., you can disassemble the payload of `+ 3-2` (index 803) with:

```
0x5555555c2505 4 + 3 discode
   $5555555C2509: add    %r9,%r15
```

Here the 4 is the length of the code before the payload, the 3 is the length of the payload.

Concerning the IP offset, consider the following:

```
$ gforth-fast --print-prims -e bye |& grep '^lit '
lit               1-1  0   66    0    0 0x5555555bc413 len= 4+ 11+ 3 send=0
lit               0-1  0  659    0  351 0x5555555c1a4b len= 4+  4+ 3 send=0
lit               0-1  1  660    0    1 0x5555555c1a58 len= 4+  3+ 3 send=0
lit               0-1  2  661    0 1841 0x5555555c1a64 len= 4+  4+ 3 send=0
lit               0-1  3  662    0  340 0x5555555c1a71 len= 4+  4+ 3 send=0
...
lit               1-2  0  683    0  165 0x5555555c1b97 len= 4+  4+ 3 send=0
lit               1-2  1  684    0    6 0x5555555c1ba4 len= 4+  3+ 3 send=0
lit               1-2  2  685    0 1708 0x5555555c1bb0 len= 4+  4+ 3 send=0
lit               1-2  3  686    0  211 0x5555555c1bbd len= 4+  4+ 3 send=0
...
lit               2-3  0  707    0   36 0x5555555c1ce3 len= 4+  4+ 3 send=0
lit               2-3  1  708    0    0 0x5555555c1cf0 len= 4+  3+ 3 send=0
lit               2-3  2  709    0   53 0x5555555c1cfc len= 4+  4+ 3 send=0
lit               2-3  3  710    0  191 0x5555555c1d09 len= 4+  4+ 3 send=0
...
```

Here you see variants of `lit` with different stack-state transitions and different IP offsets (with the offset given as number of cells). The IP offset avoids the need to update the IP before the code for the primitive (see Section 13.5 [IP-update optimization], page 344); instead, the immediate argument is accessed using the offset. E.g, here we see the payload of the `0-1` variant with offsets of 0 and 3 cells (indexes 659 and 662):

```
0x5555555c1a4b 4 + 4 discode
   $5555555C1A4F: mov    -0x8(%rbx),%r13
 ok
0x5555555c1a71 4 + 4 discode
   $5555555C1A75: mov    0x10(%rbx),%r13
```

For the branch-to-IP variants, let's look at `?branch`:

```
gforth-fast --print-prims -e bye |& grep '^?branch '
?branch           1-1  0   21 1505   78 0x5555555bbc70 len= 4+ 24+ 3 send=0
?branch           2-1  0  757 1506    0 0x5555555c20f4 len= 4+ 17+ 3 send=0
?branch           2-1  1  758    0    0 0x5555555c210e len= 4+ 16+ 3 send=0
...
?branch           2-1 23  780    0    2 0x5555555c235b len= 4+ 20+ 3 send=0
?branch           1-1  0 1505    0  325 0x5555555c50b0 len= 0+ 17+ 3 send=0
?branch           2-1  0 1506    0  677 0x5555555c50c8 len= 0+ 10+ 3 send=0
```

Here we see that the `2-1` variant of `?branch` (index 757) has the branch-to-IP variant 1506. The code for the payload of these two variants is:

```
0x5555555c20f4 4 + 17 discode
   $5555555C20F8: mov    -0x8(%rbx),%rsi
   $5555555C20FC: test   %r15,%r15
   $5555555C20FF: jne    0x5555555c2109
   $5555555C2101: mov    (%rsi),%rax
   $5555555C2104: mov    %rsi,%rbx
   $5555555C2107: jmp    *%rax
 ok
0x5555555c50c8 0 + 10 discode
   $5555555C50C8: mov    (%rbx),%rax
   $5555555C50CB: test   %r15,%r15
   $5555555C50CE: jne    0x5555555c50d2
   $5555555C50D0: jmp    *%rax
```

The second variant is used for optimizing short branches in the IP-update optimization by setting the IP to the branch target before the branch (see Section 13.5 [IP-update optimization], page 344).

The primitive `noop` has variants for several purposes:

```
gforth-fast --print-prims -e bye |& grep '^noop '
noop              1-1  0   10    0    3 0x5555555bbb92 len= 4+  0+ 3 send=0
noop              1-1 -24 1445    0    0 0x5555555c4dee len= 7+  0+ 3 send=0
noop              1-1 -23 1446    0    0 0x5555555c4dfa len= 7+  0+ 3 send=0
...
noop              1-1 22 1491    0    0 0x5555555c4fb1 len= 7+  0+ 3 send=0
noop              1-1 23 1492    0    0 0x5555555c4fbd len= 7+  0+ 3 send=0
noop              1-0  0 1493    0 2957 0x5555555c4fcb len= 0+  7+ 3 send=0
noop              2-0  0 1494    0  982 0x5555555c4fd9 len= 0+ 12+ 3 send=0
noop              3-0  0 1495    0  137 0x5555555c4fec len= 0+ 16+ 3 send=0
noop              0-1  0 1496    0  158 0x5555555c5003 len= 0+  8+ 3 send=0
noop              2-1  0 1497    0   89 0x5555555c5012 len= 0+ 10+ 3 send=0
noop              3-1  0 1498    0   35 0x5555555c5023 len= 0+ 14+ 3 send=0
noop              0-2  0 1499    0   40 0x5555555c5038 len= 0+ 12+ 3 send=0
noop              1-2  0 1500    0  150 0x5555555c504b len= 0+ 10+ 3 send=0
noop              3-2  0 1501    0    0 0x5555555c505c len= 0+ 13+ 3 send=0
noop              0-3  0 1502    0   15 0x5555555c5070 len= 0+ 15+ 3 send=0
```

```
noop              1-3  0 1503    0   48 0x5555555c5086 len= 0+ 15+ 3 send=0
noop              2-3  0 1504    0    5 0x5555555c509c len= 0+ 13+ 3 send=0
```

The variants 1445...1492 contain the IP updates that are inserted in various places when necessary, e.g., for 1445:

```
0x5555555c4dee 7 discode
   $5555555C4DEE: sub    $0xc0,%rbx
```

These uses are not counted, so these variants are shown as having count 0.

The variants 1492...1504 contain stack-state transitions from every state to every other state that do nothing else; they are inserted between primitive variants in order to get to a stack state when there is no primitive variant for that (see Section 13.4 [Stack caching], page 343).

The shortest-path algorithm has the option of inserting any stack-state transition between two primitive variants, and selects the stack states in a way that results in a shortest code. You can see this in the counts for `lit` with IP offset 0: the `1-1` variant (payload length 11) is never selected, because the shortest-path algorithm apparently prefers to select a `1-0` transition (payload length 7) followed by a `lit 0-1` (payload length 4) with the same total length.

We also see static superinstructions in the output of `--print-prims`:

```
$ gforth-fast --print-prims -e bye |& grep "^< [?]branch"
< ?branch         1-1  0  611  633    1 0x5555555c142a len= 4+ 28+ 3 send=0
< ?branch         1-1  0  633    0    1 0x5555555c1762 len= 4+ 21+ 3 send=0
```

The first variant is the regular one that branches to the address given in the immediate argument, the second is the branch-to-IP variant. We can see that both are only selected once for the code in the Gforth image.

14.4 Source code of primitive variants

You can find the code that is responsible for generating all the additional variants of the primitives in `peeprules.vmg`. If you build Gforth yourself, you can experiment with changing that. If you break something, it only breaks the `gforth-fast` engine.

First, you find the static superinstructions there, such as

```
cb7 = < ?branch
```

for having a static superinstruction for `< ?branch`. You will find many commented-out lines in this block; most of them are commented out because they no longer provide a benefit with stack caching and the IP-update optimization implemented. A few are commented out because the resulting code does not work.

For these compare-and-branch superinstructions there are branch-to-IP variants such as `cb7a` which are surrounded with code that generates branch-to-ip variants of the same code.

Next, you see variants related to stack caching and IP-uodate optimization. The simplest variant is something like

```
\E S2 S1 state-prim swap
```

which results in generating the `2-1` variant for `swap`. If the engine does not have stack state `2`, no code is generated. Code like

```
\E prim-states +
```

generates all variants that do not change the stack pointer and that consume and produce all stack items in registers. For `+` that means the variants `2-1` and `3-2` (if the highest stack state is `3`). Using `prim-states` produces good results for most primitives, but for stack manipulation words such as `swap` additional variants are useful and we specify them explicitly with `state-prim`.

```
\E branch-states execute
```

generates variants that take their inputs from a register and that end in state `1`; for `execute` it generates `2-1` and `3-1`.

Similarly, for the variants with IP offsets we have:

```
\E S1 S1 state-offset-prim call
```

This generates variants for `call` `1-1` for a range of IP offsets (0–23 is the current setting in the generator).

```
\E prim-states-offsets lit
```

generates variants for stack-state transitions `0-1`, `1-2` and `2-3` (all with all stack items in registers and without stack-pointer update) for a range of IP offsets, resulting in 3*24=72 variants in this case.

You see comments with numbers in many of these lines; they just indicate the dynamic usage frequency of the primitive in some benchmark programs.

Towards the end you see

```
\E gen-ip-updates noop
```

which generates the IP-update variants of `noop`, and

```
\E gen-transitions noop
```

which generates the stack-state transition variants of `noop`.

Finally, there is some not very polished code for generating the branch-to-IP variants of conditional branch primitives.

You can build Gforth with a non-default number of maximum stack items in registers by configuring it with, e.g.,:

```
./configure STACK_CACHE_REGS=4
```

Note that more registers result in slower compilation in Gforth, but can produce speedups (with diminishing returns); the current limits are a good compromise. Also, the architecture may not have enough registers, and asking for more will result in slowdowns due to putting "registers" in memory.

15 Cross Compiler

The cross compiler is used to bootstrap a Forth kernel. Since Gforth is mostly written in Forth, including crucial parts like the outer interpreter and compiler, it needs compiled Forth code to get started. The cross compiler allows to create new images for other architectures, even running under another Forth system.

15.1 Using the Cross Compiler

The cross compiler uses a language that resembles Forth, but isn't. The main difference is that you can execute Forth code after definition, while you usually can't execute the code compiled by cross, because the code you are compiling is typically for a different computer than the one you are compiling on.

The Makefile is already set up to allow you to create kernels for new architectures with a simple make command. The generic kernels using the GCC compiled virtual machine are created in the normal build process with `make`. To create a embedded Gforth executable for e.g. the 8086 processor (running on a DOS machine), type

```
make kernl-8086.fi
```

This will use the machine description from the `arch/8086` directory to create a new kernel. A machine file may look like that:

```
\ Parameter for target systems                          06oct92py

    4 Constant cell             \ cell size in bytes
    2 Constant cell<<           \ cell shift to bytes
    5 Constant cell>bit         \ cell shift to bits
    8 Constant bits/char        \ bits per character
    8 Constant bits/byte        \ bits per byte [default: 8]
    8 Constant float            \ bytes per float
    8 Constant /maxalign        \ maximum alignment in bytes
false Constant bigendian        \ byte order
( true=big, false=little )

include machpc.fs               \ feature list
```

This part is obligatory for the cross compiler itself, the feature list is used by the kernel to conditionally compile some features in and out, depending on whether the target supports these features.

There are some optional features, if you define your own primitives, have an assembler, or need special, nonstandard preparation to make the boot process work. `asm-include` includes an assembler, `prims-include` includes primitives, and `>boot` prepares for booting.

```
: asm-include    ." Include assembler" cr
  s" arch/8086/asm.fs" included ;

: prims-include  ." Include primitives" cr
  s" arch/8086/prim.fs" included ;
```

```
: >boot          ." Prepare booting" cr
  s" ' boot >body into-forth 1+ !" evaluate ;
```

These words are used as sort of macro during the cross compilation in the file `kernel/main.fs`. Instead of using these macros, it would be possible — but more complicated — to write a new kernel project file, too.

`kernel/main.fs` expects the machine description file name on the stack; the cross compiler itself (`cross.fs`) assumes that either `mach-file` leaves a counted string on the stack, or `machine-file` leaves an address, count pair of the filename on the stack.

The feature list is typically controlled using `SetValue`, generic files that are used by several projects can use `DefaultValue` instead. Both functions work like `Value`, when the value isn't defined, but `SetValue` works like `to` if the value is defined, and `DefaultValue` doesn't set anything, if the value is defined.

```
\ generic mach file for pc gforth                       03sep97jaw

true DefaultValue NIL  \ relocating

>ENVIRON

true DefaultValue file          \ controls the presence of the
                                \ file access wordset
true DefaultValue OS            \ flag to indicate a operating system

true DefaultValue prims         \ true: primitives are c-code

true DefaultValue floating      \ floating point wordset is present

true DefaultValue glocals       \ gforth locals are present
                                \ will be loaded
true DefaultValue dcomps        \ double number comparisons

true DefaultValue hash          \ hashing primitives are loaded/present

true DefaultValue xconds        \ used together with glocals,
                                \ special conditionals supporting gforths'
                                \ local variables
true DefaultValue header        \ save a header information

true DefaultValue backtrace     \ enables backtrace code

false DefaultValue ec
false DefaultValue crlf

cell 2 = [IF] &32 [ELSE] &256 [THEN] KB DefaultValue kernel-size

&16 KB          DefaultValue stack-size
&15 KB &512 +   DefaultValue fstack-size
```

```
&15 KB          DefaultValue rstack-size
&14 KB &512 +   DefaultValue lstack-size
```

15.2 How the Cross Compiler Works

16 MINOS2, a GUI library

16.1 MINOS2 object framework

MINOS2 is a GUI library, written in `mini-oof2.fs`'s object model. It has two main class hierarchies:

doc-actor doc-widget

16.1.1 `actor` methods:

doc-caller-w doc-active-w doc-act-name$ doc-clicked doc-scrolled doc-touchdown doc-touchup doc-ukeyed doc-ekeyed doc-?inside doc-focus doc-defocus doc-entered doc-left doc-show doc-hide doc-get doc-set doc-show-you

16.1.2 `widget` methods:

doc-parent-w doc-act doc-name$ doc-x doc-y doc-w doc-h doc-d doc-gap doc-baseline doc-kerning doc-raise doc-border doc-borderv doc-bordert doc-borderl doc-w-color doc-draw-init doc-draw doc-split doc-lastfit doc-hglue doc-dglue doc-vglue doc-hglue doc-dglue doc-vglue doc-xywh doc-xywhd doc-!resize doc-!size doc-dispose-widget doc-.widget doc-par-split doc-resized

Components are composed using a boxes&glue model similar to LaTeX, including paragraph breaking. For the sake of simplicity and portability, MINOS2 only supports a single window, and uses OpenGL for rendering.

MINOS2 furthermore supports animations with the `animation` class. A color index texture is used for different color schemes, and transition between neighboring schemes can also be animated.

doc->animate

You can create named color indexes and assign them color values for the currently active color scheme.

doc-color: doc-new-color: doc-text-color: doc-text-emoji-color: doc-fade-color: doc-text-emoji-fade-color: doc-re-color doc-re-text-color doc-re-emoji-color doc-re-fade-color doc-re-text-emoji-fade-color

For a number of specific objects, there are early bound methods, that only work on these objects

- Viewport

  doc-vp-top doc-vp-bottom doc-vp-left doc-vp-right doc-vp-reslide doc-vp-needed

16.2 MINOS2 tutorial

Tutorials are small files, each showing a bit of MINOS2. For the common framework, the file `minos2/tutorial/tutorial.fs` needs to be loaded first; all other tutorials in the command line argument are included from within that file. Scroll wheel or previous/next mouse buttons as well as clicking on the left or right edge of the window allow navigation between the different tutorials loaded.

I.e. to load the buttons tutorial, you start Gforth with

```
gforth minos2/tutorial/tutorial.fs buttons.fs
```

Available tutorials:

- `buttons.fs`: Clickable buttons
- `plots.fs`: Plot functions
- `markdown.fs`: Markdown document viewer
- `screenshot.fs`: Screenshot function

Appendix A Bugs

Known bugs are described in the file `BUGS` in the Gforth distribution.

If you find a bug, please submit a bug report through `https://savannah.gnu.org/bugs/?func=addbug&group=gforth`.

- A program (or a sequence of keyboard commands) that reproduces the bug.
- A description of what you think constitutes the buggy behaviour.
- The Gforth version used (it is announced at the start of an interactive Gforth session).
- The machine and operating system (on Unix systems `uname -a` will report this information).
- The installation options (you can find the configure options at the start of `config.status`) and configuration (`configure` output or `config.cache`).
- A complete list of changes (if any) you (or your installer) have made to the Gforth sources.

Appendix B Authors and Ancestors of Gforth

B.1 Authors and Contributors

The Gforth project was started in mid-1992 by Bernd Paysan and Anton Ertl. The third major author was Jens Wilke. Neal Crook contributed a lot to the manual. Assemblers and disassemblers were contributed by Andrew McKewan, Christian Pirker, Bernd Thallner, and Michal Revucky. Lennart Benschop (who was one of Gforth's first users, in mid-1993) and Stuart Ramsden inspired us with their continuous feedback. Lennart Benshop contributed `glosgen.fs`, while Stuart Ramsden has been working on automatic support for calling C libraries. Helpful comments also came from Paul Kleinrubatscher, Christian Pirker, Dirk Zoller, Marcel Hendrix, John Wavrik, Barrie Stott, Marc de Groot, Jorge Acerada, Bruce Hoyt, Robert Epprecht, Dennis Ruffer and David N. Williams. Since the release of Gforth-0.2.1 there were also helpful comments from many others; thank you all, sorry for not listing you here (but digging through my mailbox to extract your names is on my to-do list).

Gforth also owes a lot to the authors of the tools we used (GCC, CVS, git and autoconf, among others), and to the creators of the Internet: Gforth was developed across the Internet, and its authors did not meet physically for the first 4 years of development.

B.2 Pedigree

Gforth descends from bigFORTH (1993) and fig-Forth. Of course, a significant part of the design of Gforth was prescribed by Standard Forth.

Bernd Paysan wrote bigFORTH, a descendent from TurboForth, an unreleased 32 bit native code version of VolksForth for the Atari ST, written mostly by Dietrich Weineck.

VolksForth was written by Klaus Schleisiek, Bernd Pennemann, Georg Rehfeld and Dietrich Weineck for the C64 (called UltraForth there) in the mid-80s and ported to the Atari ST in 1986. It descends from fig-Forth.

A team led by Bill Ragsdale implemented fig-Forth on many processors in 1979. Robert Selzer and Bill Ragsdale developed the original implementation of fig-Forth for the 6502 based on microForth.

The principal architect of microForth was Dean Sanderson. microForth was FORTH, Inc.'s first off-the-shelf product. It was developed in 1976 for the 1802, and subsequently implemented on the 8080, the 6800 and the Z80.

All earlier Forth systems were custom-made, usually by Charles Moore, who discovered (as he puts it) Forth during the late 60s. The first full Forth existed in 1971.

A part of the information in this section comes from *The Evolution of Forth (*`https://www.forth.com/resources/evolution/index.html`*)* by Elizabeth D. Rather, Donald R. Colburn and Charles H. Moore, presented at the HOPL-II conference and preprinted in SIGPLAN Notices 28(3), 1993. You can find more historical and genealogical information about Forth there. For a more general (and graphical) Forth family tree look see `https://www.complang.tuwien.ac.at/forth/family-tree/`, *Forth Family Tree and Timeline*.

Appendix C Other Forth-related information

There is an active news group (comp.lang.forth) discussing Forth (including Gforth) and Forth-related issues. Its FAQs (`https://www.complang.tuwien.ac.at/forth/faq/faq-general-2.html`) (frequently asked questions and their answers) contains a lot of information on Forth. You should read it before posting to comp.lang.forth.

Usenet (including comp.lang.forth) has lost a lot of popularity since its heyday in the 1990s, but the result has been a scattering of people across a variety of places in the web, including stackoverflow and github, without any dominant place, and we have not kept track.

The Forth standard is most usable in its HTML form (`https://forth-standard.org/`).

Appendix D Licenses

D.1 GNU Free Documentation License

Version 1.2, November 2002

Copyright © 2000,2001,2002 Free Software Foundation, Inc.
59 Temple Place, Suite 330, Boston, MA 02111-1307, USA

Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

   It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

   You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

   A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any,

be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their

titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

D.1.1 ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.2
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

D.2 GNU GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Copyright © 2007 Free Software Foundation, Inc. http://fsf.org/

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program—to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

0. Definitions.

   "This License" refers to version 3 of the GNU General Public License.

   "Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

   "The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

   To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

   A "covered work" means either the unmodified Program or a work based on the Program.

   To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes

copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

   The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

   A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

   The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

   The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

   The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

   The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

   All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its

content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

   No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

   When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

   You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

   You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

   You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

   a. The work must carry prominent notices stating that you modified it, and giving a relevant date.
   b. The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".
   c. You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any

applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

d. If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

a. Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

b. Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

c. Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

d. Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

e. Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permis-

sions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

a. Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or

b. Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or

c. Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

d. Limiting the use for publicity purposes of names of licensors or authors of the material; or

e. Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or

f. Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will

automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

   You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

    Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

    An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

    You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

    If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

    Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

    The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

    If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

    Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

    THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFEC-

TIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

    IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

    If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and a brief idea of what it does.
Copyright (C) year name of author

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or (at
your option) any later version.

This program is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see http://www.gnu.org/licenses/.
```

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

```
program Copyright (C) year name of author
This program comes with ABSOLUTELY NO WARRANTY; for details type ‘show w’.
This is free software, and you are welcome to redistribute it
under certain conditions; type ‘show c’ for details.
```

The hypothetical commands ‘`show w`’ and ‘`show c`’ should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an “about box”.

You should also get your employer (if you work as a programmer) or school, if any, to sign a “copyright disclaimer” for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see `http://www.gnu.org/licenses/`.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read `http://www.gnu.org/philosophy/why-not-lgpl.html`.

Word Index

This index is a list of Forth words that have "glossary" entries within this manual. Each word is listed with its stack effect and wordset.

!

! (*w a-addr* --) core 81
!!FIXME!! (--) gforth-1.0 266
!@ (*w1 a-addr* -- *w2*) gforth-experimental . 81
!localn (*w noffset* --) gforth-internal . 225

#

(*ud1* -- *ud2*) core 190
#! (--) gforth-0.2 335
#> (*xd* -- *addr u*) core 190
#>> (--) gforth-0.5 190
#bell (-- *c*) gforth-0.2 93
#bs (-- *c*) gforth-0.2 93
#cr (-- *c*) gforth-0.2 93
#del (-- *c*) gforth-0.2 93
#eof (-- *c*) gforth-0.7 93
#esc (-- *c*) gforth-0.5 93
#ff (-- *c*) gforth-0.2 93
#lf (-- *c*) gforth-0.2 93
#line ("u" "*["file"]*" --) gforth-1.0 ... 170
#loc (*nline nchar* "*file*" --) gforth-1.0. 266
#locals (-- *n*) environment 193
#s (*ud* -- *0 0*) core 190
#tab (-- *c*) gforth-0.2 93
#tib (-- *addr*) gforth-obsolete 168

$

$! (*addr1 u $addr* --) gforth-0.7 96
$!len (*u $addr* --) gforth-0.7 96
$+! (*addr1 u $addr* --) gforth-0.7 96
$+!len (*u $addr* -- *addr*) gforth-1.0 96
$+[]! (*c-addr u $[]addr* --) gforth-1.0 ... 97
$+slurp (*fid $addr* --) gforth-1.0 97
$+slurp-file (*c-addr u $addr* --)
gforth-1.0 97
$. (*$addr* --) gforth-1.0 97
$? (-- *n*) gforth-0.2 305
$[] (*u $[]addr* -- *addr'*) gforth-1.0 97
$[]! (*c-addr u n $[]addr* --) gforth-1.0 .. 97
$[]# (*$[]addr* -- *len*) gforth-1.0 97
$[]+! (*c-addr u n $[]addr* --) gforth-1.0 . 97
$[]. (*$[]addr* --) gforth-1.0 98
$[]@ (*n $[]addr* -- *addr u*) gforth-1.0 97
$[]free (*$[]addr* --) gforth-1.0 98
$[]map (*$[]addr xt* --) gforth-1.0 97
$[]slurp (*fid $[]addr* --) gforth-1.0 97
$[]slurp-file (*addr u $[]addr* --)
gforth-1.0 97
$[]Variable ("*name*" --) gforth-1.0 98
$@ (*$addr* -- *addr2 u*) gforth-0.7 96
$@len (*$addr* -- *u*) gforth-0.7 96
$del (*$addr off u* --) gforth-0.7 96
$exec (*xt $addr* --) gforth-1.0 97
$free (*$addr* --) gforth-1.0 96
$init (*$addr* --) gforth-1.0 96
$ins (*addr1 u $addr off* --) gforth-0.7 ... 96
$iter (*.. $addr char xt* -- *..*)
gforth-0.7 96
$over (*addr u $addr off* --) gforth-1.0 ... 96
$slurp (*fid $addr* --) gforth-1.0 97
$slurp-file (*c-addr u $addr* --)
gforth-1.0 97
$split (*c-addr u char* -- *c-addr u1 c-addr2 u2*
) gforth-0.7 94
$substitute (*addr1 len1* -- *addr2 len2 n/ior*
) gforth-experimental 102
$tmp (*xt* -- *addr u*) gforth-1.0 95
$unescape (*addr1 u1* -- *addr2 u2*)
gforth-experimental 102
$value: (*u1* "*name*" -- *u2*)
gforth-experimental 145
$value[]: (*u1* "*name*" -- *u2*)
gforth-experimental 146
$Variable ("*name*" --) gforth-1.0 98

%

%align (*align size* --) gforth-0.4 148
%alignment (*align size* -- *align*)
gforth-0.4 148
%alloc (*align size* -- *addr*) gforth-0.4 .. 148
%allocate (*align size* -- *addr ior*)
gforth-0.4 148
%allot (*align size* -- *addr*) gforth-0.4 .. 148
%size (*align size* -- *size*) gforth-0.4 ... 149

'

' ("*name*" -- *xt*) core 156
'cold (--) gforth-0.2 336
'image (--) gforth-experimental 332
's (*addr1 task* -- *addr2*)
gforth-experimental 274

(

((*compilation 'ccc<close-paren>'* -- *;*
run-time --) core,file 59
(((*addr u* --) regexp-pattern 254
(local) (*addr u* --) local 227

)

) (--) gforth-0.2 267
)) (-- flag) regexp-pattern 254

*

* (n1 n2 -- n) core 61
**} (sys --) regexp-pattern 256
*/ ((n1 n2 n3 -- n4) core 63
*/f (n1 n2 n3 -- n4) gforth-1.0 63
*/mod (n1 n2 n3 -- n4 n5) core 63
*/modf (n1 n2 n3 -- n4 n5) gforth-1.0 63
*/mods (n1 n2 n3 -- n4 n5) gforth-1.0 63
*/s (n1 n2 n3 -- n4) gforth-1.0 63
*} (addr addr' -- addr') regexp-pattern . 256
*align (n --) gforth-1.0 86
*aligned (addr1 n -- addr2) gforth-1.0 ... 86

+

+ (n1 n2 -- n) core 60
+! (n a-addr --) core 81
+!@ (u1 a-addr -- u2)
gforth-experimental 81
++} (sys --) regexp-pattern 256
+} (addr addr' -- addr') regexp-pattern . 256
+after (x1 x2 stack --)
gforth-experimental 150
+char (char --) regexp-cg 255
+chars (addr u --) regexp-cg 255
+class (class --) regexp-cg 255
+DO (compilation -- do-sys ; run-time n1 n2
-- | loop-sys) gforth-0.2 107
+field (noffset1 nsize "name" -- noffset2)
facility-ext 143
+fmode (fam1 rwxrwxrwx -- fam2)
gforth-1.0 197
+load (i*x n -- j*x) gforth-0.2 205
+LOOP (compilation do-sys -- ; run-time
loop-sys1 n -- | loop-sys2) core 108
+ltrace (--) gforth-1.0 266
+status (--) gforth-1.0 258
+thru (i*x n1 n2 -- j*x) gforth-0.2 205
+TO (value ... "name" --) gforth-1.0 122
+x/string (xc-addr1 u1 -- xc-addr2 u2)
xchar-ext 90

,

, (w --) core 77

-

- (n1 n2 -- n) core 61
-- (hmaddr u wid 0 ... --) local-ext 219
--> (--) gforth-0.2 205
->here (addr --) gforth-1.0 76
-[do (compilation -- do-sys ; run-time n1 n2
-- | loop-sys) gforth-experimental 107
-` ("char" --) regexp-pattern 255
-\d (addr -- addr') regexp-pattern 255
-\s (addr -- addr') regexp-pattern 255
-c? (addr class --) regexp-pattern 255
-char (char --) regexp-cg 255
-class (class --) regexp-cg 255
-DO (compilation -- do-sys ; run-time n1 n2
-- | loop-sys) gforth-0.2 108
-inf (-- r) gforth-1.0 71
-infinity (-- r) gforth-1.0 71
-LOOP (compilation do-sys -- ; run-time
loop-sys1 u -- | loop-sys2) gforth-0.2 . 108
-ltrace (--) gforth-1.0 266
-rot (w1 w2 w3 -- w3 w1 w2) gforth-0.2 73
-stack (x stack --)
gforth-experimental 150
-status (--) gforth-1.0 258
-trailing (c_addr u1 -- c_addr u2)
string 94
-trailing-garbage (xc-addr u1 -- xc-addr u2
) xchar-ext 90

.

. (n --) core 205
." (compilation 'ccc"' -- ; run-time --)
core 208
.((compilation&interpretation
'ccc<close-paren>' --) core-ext 208
.-is-dcell? (-- flag)
gforth-experimental 174
... (x1 .. xn -- x1 .. xn) gforth-1.0 263
..char (start end --) regexp-cg 255
.? (addr -- addr') regexp-pattern 255
.\" (compilation 'ccc"' -- ; run-time --)
gforth-0.6 208
.cover-raw (--) gforth-experimental 270
.coverage (--) gforth-experimental 269
.debugline (nfile nline --) gforth-0.6 . 265
.fpath (--) gforth-0.4 201
.hm (nt --) gforth-1.0 301
.id (nt --) gforth-0.6 159
.included (--) gforth-0.5 197
.locale-csv (--) gforth-experimental ... 101
.path (path-addr --) gforth-0.4 202
.quoted-csv (c-addr u --)
gforth-experimental 215
.r (n1 n2 --) core-ext 205
.s (--) tools 263
.sections (--) gforth-1.0 79

.substitute (addr1 len1 -- n / ior) gforth-experimental 102
.unresolved (--) gforth-1.0 114
.voc (wid --) gforth-0.2 184

/

/ (n1 n2 -- n) core 62
// (--) regexp-pattern 256
//g (ptr addr u -- addr' u') regexp-replace 257
//o (ptr addr u -- addr' u') regexp-replace 257
//s (ptr --) regexp-replace 257
/COUNTED-STRING (-- n) environment 192
/f (n1 n2 -- n) gforth-1.0 62
/f-stage1m (n a-reci --) gforth-1.0 65
/f-stage2m (n1 a-reci -- nquotient) gforth-1.0 65
/HOLD (-- n) environment 193
/l (-- u) gforth-0.7 87
/mod (n1 n2 -- n3 n4) core 62
/modf (n1 n2 -- n3 n4) gforth-1.0 63
/modf-stage2m (n1 a-reci -- umodulus nquotient) gforth-1.0 65
/mods (n1 n2 -- n3 n4) gforth-1.0 62
/PAD (-- n) environment 193
/s (n1 n2 -- n) gforth-1.0 62
/string (c-addr1 u1 n -- c-addr2 u2) string 94
/w (-- u) gforth-0.7 86
/x (-- u) gforth-1.0 87

:

: ("name" -- colon-sys) core 123
:: (class "name" --) mini-oof 243
:} (hmaddr u wid 0 xt1 ... xtn --) gforth-1.0 219
:}d (hmaddr u latest latestnt wid state 0 a-addr1 u1 ... --) gforth-1.0 250
:}h (hmaddr u latest latestnt wid state 0 a-addr1 u1 ... --) gforth-1.0 250
:}h1 (hmaddr u latest latestnt wid state 0 a-addr1 u1 ... --) gforth-1.0 250
:}l (hmaddr u latest latestnt wid state 0 a-addr1 u1 ... --) gforth-1.0 250
:}xt (hmaddr u latest latestnt wid state 0 a-addr1 u1 ... --) gforth-1.0 250
:is ("name" --) gforth-experimental 139
:m ("name" -- xt; run-time: object --) objects 238
:method (class "name" --) gforth-experimental 246
:noname (-- xt colon-sys) core-ext 124

;

; (compilation colon-sys -- ; run-time nest-sys --) core 123
;> (compilation colon-sys -- ; run-time -- addr) gforth-obsolete 254
;] (compile-time: quotation-sys -- ; run-time: -- xt) gforth-1.0 125
;abi-code (--) gforth-1.0 286
;code (compilation. colon-sys1 -- colon-sys2) tools-ext 286
;inline (inline:-sys --) gforth-experimental 123
;m (colon-sys --; run-time: --) objects. 238

<

< (n1 n2 -- f) core 68
<# (--) core 189
<< (run-addr addr u -- run-addr) regexp-replace 257
<<" ("string<">" --) regexp-replace 257
<<# (--) gforth-0.5 190
<= (n1 n2 -- f) gforth-0.2 68
<> (n1 n2 -- f) core-ext 68
<{: (compilation -- colon-sys ; run-time --) gforth-obsolete 254
<bind> (class selector-xt -- xt) objects 236
<to-inst> (w xt --) objects 239

=

= (n1 n2 -- f) core 68
=" (<string>" --) regexp-pattern 256
=mkdir (c-addr u wmode -- wior) gforth-0.7 200

>

> (n1 n2 -- f) core 68
>= (n1 n2 -- f) gforth-0.2 68
>> (addr -- addr) regexp-replace 257
>addr (... xt -- addr) gforth-internal .. 250
>back (x stack --) gforth-experimental . 150
>body (xt -- a-addr) core 130
>code-address (xt -- c_addr) gforth-0.2 303
>definer (xt -- definer) gforth-0.2 304
>does-code (xt1 -- xt2) gforth-0.2 304
>float (c-addr u -- f:... flag) floating 188
>float1 (c-addr u c -- f:... flag) gforth-1.0 189
>in (-- addr) core 167
>l (w --) gforth-internal 225
>name (xt -- nt|0) gforth-0.2 158
>number (ud1 c-addr1 u1 -- ud2 c-addr2 u2) core 188

>o (c-addr -- r:c-old) new 246
>order (wid --) gforth-0.5 183
>pow2 (u1 -- u2) gforth-1.0 67
>r (w -- R:w) core 74
>stack (x stack --)
gforth-experimental 149
>string-execute (... xt -- ... c-addr u)
gforth-1.0 95
>time&date&tz (dtime -- nsec nmin nhour nday
nmonth nyear fdst ndstoff c-addrtz utz)
gforth-1.0 305
>uvalue (xt -- addr) gforth-internal ... 134

?

? (a-addr --) tools 264
??? (--) gforth-0.2 265
?cov+ (flag -- flag)
gforth-experimental 269
?DO (compilation -- do-sys ; run-time w1 w2
-- | loop-sys) core-ext 107
?dup (w -- S:... w) core 73
?dup-IF (compilation -- orig ; run-time n --
n|) gforth-0.2 113
?DUP-0=-IF (compilation -- orig ; run-time n
-- n|) gforth-0.2 113
?errno-throw (f --) gforth-1.0 115
?events (--) gforth-experimental 275
?EXIT (--) gforth-0.2 114
?ior (x --) gforth-1.0 115
?LEAVE (compilation -- ; run-time f | f
loop-sys --) gforth-0.2 109
?of (compilation -- of-sys ; run-time f --)
gforth-1.0 111
?rec-found (translation -- translation)
gforth-experimental 180

[

[(--) core 161
['] (compilation. "name" -- ; run-time. -- xt
) core 156
[+LOOP] (n --) gforth-0.2 170
[: (compile-time: -- quotation-sys flag
colon-sys) gforth-1.0 125
[?DO] (n-limit n-index --) gforth-0.2 .. 170
[{: (compilation -- hmaddr u latest wid 0 ;
instantiation ... -- xt) gforth-1.0 ... 250
[AGAIN] (--) gforth-0.2 170
[BEGIN] (--) gforth-0.2 170
[bind] (compile-time: "class" "selector" --
; run-time: ... object -- ...) objects . 237
[char] (compilation '<spaces>ccc' -- ;
run-time -- c) core,xchar-ext 93
[COMP'] (compilation "name" -- ; run-time --
xt1 xt2) gforth-0.2 160
[compile] (compilation "name" -- ; run-time
? -- ?) core-ext 166
[current] (compile-time: "selector" -- ;
run-time: ... object -- ...) objects ... 237
[d:d (compilation -- colon-sys; run-time: d
-- xt ; xt execution: -- d) gforth-1.0 . 249
[d:h (compilation -- colon-sys; run-time: d
-- xt ; xt execution: -- d) gforth-1.0 . 249
[d:h1 (compilation -- colon-sys; run-time: d
-- xt ; xt execution: -- d) gforth-1.0 . 249
[d:l (compilation -- colon-sys; run-time: d
-- xt ; xt execution: -- d) gforth-1.0 . 249
[defined] ("<spaces>name" -- flag)
tools-ext 170
[DO] (n-limit n-index --) gforth-0.2 ... 170
[ELSE] (--) tools-ext 169
[ENDIF] (--) gforth-0.2 169
[f:d (compilation -- colon-sys; run-time: r
-- xt ; xt execution: -- r) gforth-1.0 . 249
[f:h (compilation -- colon-sys; run-time: r
-- xt ; xt execution: -- r) gforth-1.0 . 249
[f:h1 (compilation -- colon-sys; run-time: r
-- xt ; xt execution: -- r) gforth-1.0 . 249
[f:l (compilation -- colon-sys; run-time: r
-- xt ; xt execution: -- r) gforth-1.0 . 249
[FOR] (n --) gforth-0.2 170
[I] (run-time -- n) gforth-0.2 170
[IF] (flag --) tools-ext 169
[IFDEF] ("<spaces>name" --) gforth-0.2. 170
[IFUNDEF] ("<spaces>name" --)
gforth-0.2 170
[LOOP] (--) gforth-0.2 170
[n:d (compilation -- colon-sys; run-time: n
-- xt ; xt execution: -- n) gforth-1.0 . 249
[n:h (compilation -- colon-sys; run-time: n
-- xt ; xt execution: -- n) gforth-1.0 . 249
[n:h1 (compilation -- colon-sys; run-time: n
-- xt ; xt execution: -- n) gforth-1.0 . 249
[n:l (compilation -- colon-sys; run-time: n
-- xt ; xt execution: -- n) gforth-1.0 . 249
[NEXT] (n --) gforth-0.2 170
[noop] (--) gforth-experimental 157
[parent] (compile-time: "selector" -- ;
run-time: ... object -- ...) objects ... 239
[REPEAT] (--) gforth-0.2 170
[THEN] (--) tools-ext 169
[to-inst] (compile-time: "name" -- ;
run-time: w --) objects 239
[undefined] ("<spaces>name" -- flag)
tools-ext 170
[UNTIL] (flag --) gforth-0.2 170
[WHILE] (flag --) gforth-0.2 170

]

] (--) core 161
]] (--) gforth-0.6 163
]L (compilation: n -- ; run-time: -- n)
gforth-0.5 161
]nocov (--) gforth-1.0 269

`

` ("char" --) regexp-pattern 255
`? ("char" --) regexp-pattern 255

@

@ (a-addr -- w) core 81
@localn (noffset -- w) gforth-internal . 225

\

\ (compilation 'ccc<newline>' -- ; run-time --) core-ext,block-ext 60
\$ (addr -- addr) regexp-pattern 255
\((addr -- addr) regexp-pattern 256
\) (addr -- addr) regexp-pattern 256
\^ (addr -- addr) regexp-pattern 255
\\\ (--) gforth-1.0 197
\0 (-- addr u) regexp-pattern..... 256
\c ("rest-of-line" --) gforth-0.7..... 280
\d (addr -- addr') regexp-pattern..... 255
\G (compilation 'ccc<newline>' -- ; run-time --) gforth-0.2..... 60
\s (addr -- addr') regexp-pattern..... 255

{

{ (-- hmaddr u wid 0) gforth-0.2..... 219
{* (addr -- addr addr) regexp-pattern... 256
{** (addr -- addr addr) regexp-pattern.. 256
{+ (addr -- addr addr) regexp-pattern... 256
{++ (addr -- addr addr) regexp-pattern.. 256
{: (-- hmaddr u wid state 0) local-ext .. 218
{{ (addr -- addr addr) regexp-pattern... 256

}

} (hmaddr u wid state 0 xt1 ... xtn --) gforth-0.2 219
}} (addr addr -- addr) regexp-pattern... 256

|

| (--) local-ext 219
|| (addr addr -- addr addr) regexp-pattern..... 256

~

~~ (--) gforth-0.2 265
~~1bt (--) gforth-1.0 265
~~bt (--) gforth-1.0 265
~~Value (n "name" --) gforth-1.0 266
~~Variable ("name" --) gforth-1.0..... 266

0

0< (n -- f) core..... 68
0<= (n -- f) gforth-0.2..... 68
0<> (n -- f) core-ext..... 68
0= (n -- f) core..... 68
0> (n -- f) core-ext..... 68
0>= (n -- f) gforth-0.2..... 68

1

1+ (n1 -- n2) core 60
1- (n1 -- n2) core 61
1/f (r1 -- r2) gforth-0.2 70

2

2! (w1 w2 a-addr --) core..... 82
2* (n1 -- n2) core 67
2, (w1 w2 --) gforth-0.2 77
2/ (n1 -- n2) core 67
2>r (w1 w2 -- R:w1 R:w2) core-ext..... 74
2@ (a-addr -- w1 w2) core..... 82
2Constant (w1 w2 "name" --) double..... 121
2drop (w1 w2 --) core..... 73
2dup (w1 w2 -- w1 w2 w1 w2) core..... 73
2field: (u1 "name" -- u2) gforth-0.7.... 143
2lit, (x1 x2 --) gforth-1.0..... 161
2Literal (compilation w1 w2 -- ; run-time -- w1 w2) double..... 161
2nip (w1 w2 w3 w4 -- w3 w4) gforth-0.2.... 73
2over (w1 w2 w3 w4 -- w1 w2 w3 w4 w1 w2) core..... 73
2r> (R:w1 R:w2 -- w1 w2) core-ext..... 74
2r@ (R:w1 R:w2 -- R:w1 R:w2 w1 w2) core-ext..... 74
2rdrop (R:w1 R:w2 --) gforth-0.2..... 74
2rot (w1 w2 w3 w4 w5 w6 -- w3 w4 w5 w6 w1 w2) double-ext..... 73
2swap (w1 w2 w3 w4 -- w3 w4 w1 w2) core.... 73
2tuck (w1 w2 w3 w4 -- w3 w4 w1 w2 w3 w4) gforth-0.2..... 73
2Value (w1 w2 "name" --) double-ext..... 121
2value: (u1 "name" -- u2) gforth-experimental..... 145
2Variable ("name" --) double..... 120

A

A, (*addr* --) gforth-0.2 77
abi-code ("*name*" -- *colon-sys*) gforth-1.0 285
abort (*??* -- *??*) core,exception-ext 118
ABORT" (*compilation 'ccc"'* -- ; *run-time ... f* --) core,exception-ext 118
abs (*n* -- *u*) core 61
absolute-file? (*addr u* -- *flag*) gforth-1.0 201
accept (*c-addr +n1* -- *+n2*) core 214
AConstant (*addr* "*name*" --) gforth-0.2 .. 121
action-of (*interpretation* "*name*" ... -- *xt*; *compilation* "*name*" -- ; *run-time* ... -- *xt*) core-ext 139
add-cflags (*c-addr u* --) gforth-1.0 282
add-framework (*c-addr u* --) gforth-1.0 . 282
add-incdir (*c-addr u* --) gforth-1.0 282
add-ldflags (*c-addr u* --) gforth-1.0 ... 282
add-lib (*c-addr u* --) gforth-0.7 282
add-libpath (*c-addr u* --) gforth-0.7 ... 282
addr (*interpretation* "*name*" ... -- *addr*; *compilation* "*name*" -- ; *run-time* ... -- *addr*) gforth-1.0 122
ADDRESS-UNIT-BITS (-- *n*) environment ... 192
addressable: (--) gforth-experimental .. 122
adjust-buffer (*u addr* --) gforth-experimental 81
after-locate (-- *u*) gforth-1.0 259
AGAIN (*compilation dest* -- ; *run-time* --) core-ext 112
AHEAD (*compilation* -- *orig* ; *run-time* --) tools-ext 112
Alias (*xt* "*name*" --) gforth-0.2 141
align (--) core 77
aligned (*c-addr* -- *a-addr*) core 85
ALiteral (*compilation addr* -- ; *run-time* -- *addr*) gforth-0.2 161
allocate (*u* -- *a_addr wior*) memory 80
allot (*n* --) core 76
also (--) search-ext 183
also-path (*c-addr len path-addr* --) gforth-0.4 201
and (*w1 w2* -- *w*) core 66
annotate-cov (--) gforth-experimental .. 269
append (*c-addr1 u1 c-addr2 u2* -- *c-addr u*) gforth-0.7 95
arg (*u* -- *addr count*) gforth-0.2 216
argc (-- *addr*) gforth-0.2 216
argv (-- *addr*) gforth-0.2 216
array>mem (*uelements uelemsize* -- *ubytes uelemsize*) gforth-experimental 108
arshift (*n1 u* -- *n2*) gforth-1.0 66
asptr (*class* --) oof 242
assembler (--) tools-ext 285
assert((--) gforth-0.2 267
assert-level (-- *a-addr*) gforth-0.2 267
assert0((--) gforth-0.2 266
assert1((--) gforth-0.2 267
assert2((--) gforth-0.2 267
assert3((--) gforth-0.2 267
ASSUME-LIVE (*compilation orig* -- *orig*) gforth-0.2 222
at-deltaxy (*dx dy* --) gforth-0.7 209
at-xy (*x y* --) facility 209
at&t-syntax (--) gforth-1.0 289
atomic!@ (*w1 a-addr* -- *w2*) gforth-experimental 274
atomic+!@ (*u1 a-addr* -- *u2*) gforth-experimental 274
atomic?!@ (*unew uold a-addr* -- *uprev*) gforth-experimental 275
AUser ("*name*" --) gforth-0.2 273
authors (--) gforth-1.0 10
AValue (*w* "*name*" --) gforth-0.6 121
AVariable ("*name*" --) gforth-0.2 120

B

b (--) gforth-1.0 259
back> (*stack* -- *x*) gforth-experimental . 150
barrier (--) gforth-experimental 275
base (-- *a-addr*) core 187
base-execute (*i*x xt u* -- *j*x*) gforth-0.7 187
basename (*c-addr1 u1* -- *c-addr2 u2*) gforth-0.7 199
before-line (--) gforth-1.0 181
before-locate (-- *u*) gforth-1.0 259
before-word (--) gforth-0.7 181
begin-structure ("*name*" -- *struct-sys 0*) facility-ext 142
BEGIN (*compilation* -- *dest* ; *run-time* --) core 112
bin (*fam1* -- *fam2*) file 197
bind (... "*class*" "*selector*" -- ...) objects 236
bind' ("*class*" "*selector*" -- *xt*) objects 237
bl (-- *c-char*) core 93
blank (*c-addr u* --) string 87
blk (-- *addr*) block 168
block (*u* -- *addr*) block 204
block-included (*a-addr u* --) gforth-0.2 205
block-offset (-- *addr*) gforth-0.5 204
bootmessage (--) gforth-0.4 336
bounds (*u1 u2* -- *u3 u1*) gforth-0.2 107
break" (*'ccc"'* --) gforth-0.4 268
break: (--) gforth-0.4 268
broken-pipe-error (-- *n*) gforth-0.6 215
browse ("*subname*" --) gforth-1.0 259
bt (--) gforth-1.0 261
buffer (*u* -- *addr*) block 204
buffer% (-- *u1 u2*) gforth-experimental .. 81
buffer: (*u* "*name*" --) core-ext 120

bw (--) gforth-1.0 260
bw-cover (--) gforth-1.0 269
bye (--) tools-ext 9

C

c! (c c-addr --) core 81
c$+! (char $addr --) gforth-1.0 96
c++-library ("name" --) gforth-1.0 281
c++-library-name (c-addr u --)
gforth-1.0 281
c, (c --) core 76
c-callback ("forth-name" "{type}" "---"
"type" --) gforth-1.0 282
c-callback-thread ("forth-name" "{type}"
"---" "type" --) gforth-1.0 282
c-function ("forth-name" "c-name" "{type}"
"---" "type" --) gforth-0.7 280
c-funptr ("forth-name" <{>"c-typecast"<}>
"{type}" "---" "type" --) gforth-1.0 .. 280
c-library ("name" --) gforth-0.7 281
c-library-name (c-addr u --)
gforth-0.7 281
c-value ("forth-name" "c-name" "---" "type"
--) gforth-1.0 280
c-variable ("forth-name" "c-name" --)
gforth-1.0 280
c>s (x -- n) gforth-1.0 84
c? (addr class --) regexp-pattern 255
c@ (c-addr -- c) core 81
C" (compilation "ccc<quote>" -- ; run-time
-- c-addr) core-ext 102
C: (compilation "name" -- a-addr xt;
run-time c --) gforth-0.2 219
C^ (compilation "name" -- a-addr xt;
run-time c --) gforth-0.2 219
call-c (... w -- ...) gforth-0.2 284
callback# (-- n) gforth-experimental ... 283
capscompare (c-addr1 u1 c-addr2 u2 -- n)
gforth-0.7 95
capssearch (c-addr1 u1 c-addr2 u2 -- c-addr3
u3 flag) gforth-1.0 95
capsstring-prefix? (c-addr1 u1 c-addr2 u2 --
f) gforth-1.0 95
case (compilation -- case-sys ; run-time --
) core-ext 111
catch (x1 .. xn xt -- y1 .. ym 0 / z1 .. zn
error) exception 116
catch-nobt (x1 .. xn xt -- y1 .. ym 0 / z1 ..
zn error) gforth-experimental 116
cell (-- u) gforth-0.2 85
cell% (-- align size) gforth-0.4 149
cell+ (a-addr1 -- a-addr2) core 85
cell- (a-addr1 -- a-addr2) core 85
cell/ (n1 -- n2) gforth-1.0 85
cells (n1 -- n2) core 85
cfield: (u1 "name" -- u2) facility-ext .. 143
char ('<spaces>ccc' -- c)
core,xchar-ext 92
char% (-- align size) gforth-0.4 149
char+ (c-addr1 -- c-addr2) core 85
char- (c-addr1 -- c-addr2) gforth-0.7 ... 85
charclass (--) regexp-cg 254
chars (n1 -- n2) core 85
cilk-bye (--) cilk 276
cilk-init (--) cilk 276
cilk-sync (--) cilk 276
class (parent-class -- align offset)
objects 237, 243
class->map (class -- map) objects 237
class-inst-size (class -- addr)
objects 237
class-override! (xt sel-xt class-map --)
objects 237
class-previous (class --) objects 237
class; (--) oof 242
class>order (class --) objects 237
clear-path (path-addr --) gforth-0.5 ... 201
clearstack (... --) gforth-0.2 264
clearstacks (... --) gforth-0.7 264
close-dir (wdirid -- wior) gforth-0.5 .. 200
close-file (wfileid -- wior) file 197
close-pipe (wfileid -- wretval wior)
gforth-0.2 215
cmove (c-from c-to u --) string 87
cmove> (c-from c-to u --) string 87
code ("name" -- colon-sys) tools-ext ... 286
code-address! (c_addr xt --)
gforth-obsolete 303
color-cover (--) gforth-1.0 269
common-list (list1 list2 -- list3)
gforth-internal 227
COMP' ("name" -- xt1 xt2) gforth-0.2 160
compare (c-addr1 u1 c-addr2 u2 -- n)
string 93
compile, (xt --) core-ext 165
compile-color (--) gforth-1.0 210
compile-only (--) gforth-0.2 153
compile-only? (nt -- flag) gforth-1.0 .. 159
compiling (... translation -- ...)
gforth-experimental 180
compsem: (--) gforth-experimental 156
const-does> (run-time: w*uw r*ur uw ur
"name" --) gforth-obsolete 137
Constant (w "name" --) core 121
construct (... object --) objects 237
context (-- addr) gforth-0.2 185
contof (compilation case-sys1 of-sys --
case-sys2 ; run-time --) gforth-1.0 ... 111
convert (ud1 c-addr1 -- ud2 c-addr2)
gforth-obsolete 188
CORE (-- f) environment 193
CORE-EXT (-- f) environment 193
cores (-- u) cilk 276
count (c-addr1 -- c-addr2 u) core 102

cov% (--) gforth-experimental 269
cov+ (--) gforth-experimental 269
cputime (-- *duser dsystem*) gforth-0.5 .. 306
cr (--) core 208
Create ("*name*" --) core 119
create-file (*c-addr u wfam* -- *wfileid wior*) file 197
create-from (*nt* "*name*" --) gforth-1.0 .. 136
critical-section (*xt mutex* --) gforth-experimental 274
cs-vocabulary ("*name*" --) gforth-1.0 ... 183
cs-wordlist (-- *wid*) gforth-1.0 183
CS-DROP (*dest/orig* --) gforth-1.0 112
CS-PICK (*dest0/orig0 dest1/orig1 ... destu/origu u* -- *... dest0/orig0*) tools-ext 112
CS-ROLL (*destu/origu .. dest0/orig0 u* -- *.. dest0/orig0 destu/origu*) tools-ext 112
cstring>sstring (*c-addr* -- *c-addr u*) gforth-0.2 95
csv-quote (-- *c*) gforth-experimental ... 215
csv-separator (-- *c*) gforth-experimental 215
ctz (*x* -- *u*) gforth-1.0 67
current (-- *addr*) gforth-0.2 185
current' ("*selector*" -- *xt*) objects 237
current-interface (-- *addr*) objects 237
cvalue: (*u1* "*name*" -- *u2*) gforth-experimental 144

D

d+ (*ud1 ud2* -- *ud*) double 61
d- (*d1 d2* -- *d*) double 61
d. (*d* --) double 206
d.r (*d n* --) double 206
d< (*d1 d2* -- *f*) double 68
d<= (*d1 d2* -- *f*) gforth-0.2 68
d<> (*d1 d2* -- *f*) gforth-0.2 68
d= (*d1 d2* -- *f*) double 68
d> (*d1 d2* -- *f*) gforth-0.2 68
d>= (*d1 d2* -- *f*) gforth-0.2 68
d>f (*d* -- *r*) floating 69
d>s (*d* -- *n*) double 61
d0< (*d* -- *f*) double 68
d0<= (*d* -- *f*) gforth-0.2 68
d0<> (*d* -- *f*) gforth-0.2 68
d0= (*d* -- *f*) double 68
d0> (*d* -- *f*) gforth-0.2 69
d0>= (*d* -- *f*) gforth-0.2 69
d2* (*d1* -- *d2*) double 67
d2/ (*d1* -- *d2*) double 67
D: (*compilation* "*name*" -- *a-addr xt; run-time x1 x2* --) gforth-0.2 219
D^ (*compilation* "*name*" -- *a-addr xt; run-time x1 x2* --) gforth-0.2 219
dabs (*d* -- *ud*) double 61
dark-mode (--) gforth-1.0 211
darshift (*d1 u* -- *d2*) gforth-1.0 67
dbg ("*name*" --) gforth-0.2 268
debug-fid (-- *file-id*) gforth-1.0 265
dec. (*n* --) gforth-0.2 205
dec.r (*u n* --) gforth-0.5 206
decimal (--) core 187
default (--) gforth-experimental 101
default-color (--) gforth-1.0 210
default-input (--) gforth-1.0 211
defer (--) oof 242
Defer ("*name*" --) core-ext 138
defer! (*xt xt-deferred* --) core-ext 140
defer: (*u1* "*name*" -- *u2*) gforth-experimental 146
defer@ (*... xt-deferred* -- *xt*) core-ext . 140
defers (*compilation* "*name*" -- *; run-time ... -- ...*) gforth-0.2 139
definer! (*definer xt* --) gforth-obsolete 304
defines (*xt class* "*name*" --) mini-oof ... 243
definitions (--) search 183
delete (*c-addr u u1* --) gforth-0.7 94
delete-file (*c-addr u* -- *wior*) file 197
delta-i (*r:ulimit r:u* -- *r:ulimit r:u u2*) gforth-1.0 109
depth (-- *+n*) core 264
df! (*r df-addr* --) floating-ext 82
df@ (*df-addr* -- *r*) floating-ext 82
dfalign (--) floating-ext 77
dfaligned (*c-addr* -- *df-addr*) floating-ext 86
dffield: (*u1* "*name*" -- *u2*) floating-ext . 143
dfloat% (-- *align size*) gforth-0.4 149
dfloat+ (*df-addr1* -- *df-addr2*) floating-ext 86
dfloat/ (*n1* -- *n2*) gforth-1.0 86
dfloats (*n1* -- *n2*) floating-ext 85
dfvalue: (*u1* "*name*" -- *u2*) gforth-experimental 145
dict-new (*... class* -- *object*) objects .. 237
dirname (*c-addr1 u1* -- *c-addr1 u2*) gforth-0.7 199
disasm (*c-addr u* --) gforth-0.5 289
disasm-gdb (*addr u* --) gforth-0.7 289
disasm2 (*c-addr u* --) gforth-1.0 289
discode (*addr u* --) gforth-0.2 288
dlshift (*ud1 u* -- *ud2*) gforth-1.0 66
dmax (*d1 d2* -- *d*) double 61
dmin (*d1 d2* -- *d*) double 61
dnegate (*d1* -- *d2*) double 61
DO (*compilation* -- *do-sys ; run-time w1 w2* -- *loop-sys*) core 108
doabicode: (-- *addr*) gforth-1.0 304
docol: (-- *addr*) gforth-0.2 303
docon: (-- *addr*) gforth-0.2 303
dodefer: (-- *addr*) gforth-0.2 304
dodoes: (-- *addr*) gforth-0.6 304
does-code! (*xt2 xt1* --) gforth-0.2 304

DOES> (*compilation colon-sys1 -- colon-sys2*) core 129
dofield: (-- *addr*) gforth-0.2 304
DONE (*compilation do-sys -- ; run-time* --) gforth-0.2 109
double% (-- *align size*) gforth-0.4 149
douser: (-- *addr*) gforth-0.2 304
dovalue: (-- *addr*) gforth-experimental . 304
dovar: (-- *addr*) gforth-0.2 304
dpl (-- *a-addr*) gforth-0.2 188
drol (*ud1 u -- ud2*) gforth-1.0 67
drop (*w* --) core 72
dror (*ud1 u -- ud2*) gforth-1.0 68
drshift (*ud1 u -- ud2*) gforth-1.0 66
du/mod (*d u -- n u1*) gforth-1.0 63
du< (*ud1 ud2 -- f*) double-ext 69
du<= (*ud1 ud2 -- f*) gforth-0.2 69
du> (*ud1 ud2 -- f*) gforth-0.2 69
du>= (*ud1 ud2 -- f*) gforth-0.2 69
dump (*addr u* --) tools 264
dup (*w -- w w*) core 72

E

e. (*f: r* --) gforth-1.0 206
e.exact (*r* --) gforth-experimental 206
e.p (*r u* --) gforth-experimental 206
early (--) oof 242
edit ("*name*" --) gforth-1.0 259
edit-line (*c-addr n1 n2 -- n3*) gforth-0.6 214
ekey (-- *u*) facility-ext 212
ekey>char (*u -- u false* | *c true*) facility-ext 212
ekey>fkey (*u1 -- u2 f*) facility-ext 212
ekey>xchar (*u -- u false* | *xc true*) xchar-ext 212
ekey? (-- *flag*) facility-ext 212
ELSE (*compilation orig1 -- orig2 ; run-time* --) core 112
emit (*c* --) core 209
emit-file (*c wfileid -- wior*) gforth-0.2 198
empty-buffers (--) block-ext 204
end-c-library (--) gforth-0.7 281
end-class (*align offset* "*name*" --) objects 237, 243
end-class-noname (*align offset -- class*) objects 237
end-code (*colon-sys* --) gforth-0.2 286
end-interface ("*name*" --) objects 237
end-interface-noname (-- *interface*) objects 238
end-methods (--) objects 238
end-struct (*align size* "*name*" --) gforth-0.2 149
end-structure (*struct-sys +n* --) facility-ext 142
endcase (*compilation case-sys -- ; run-time x* --) core-ext 111
ENDIF (*compilation orig -- ; run-time* --) gforth-0.2 113
endof (*compilation case-sys1 of-sys -- case-sys2 ; run-time* --) core-ext 111
endscope (*compilation scope -- ; run-time* --) gforth-0.2 220
endtry (*compilation -- ; run-time R:sys1* --) gforth-0.5 117
endtry-iferror (*compilation orig1 -- orig2 ; run-time R:sys1* --) gforth-0.7 118
environment (--) gforth-0.6 195
environment-wordlist (-- *wid*) gforth-0.2 195
environment? (*c-addr u -- false / ... true*) core 192
erase (*addr u* --) core-ext 87
error-color (--) gforth-1.0 210
error-hl-inv (--) gforth-1.0 210
error-hl-ul (--) gforth-1.0 210
evaluate (*... addr u -- ...*) core,block . 168
event-loop (--) gforth-experimental 275
exception (*addr u -- n*) gforth-0.2 115
exceptions (*xt n1 -- n2*) gforth-1.0 115
execute (*xt* --) core 157
execute-exit (*compilation -- ; run-time xt nest-sys* --) gforth-1.0 157
execute-parsing (*... addr u xt -- ...*) gforth-0.6 182
execute-parsing-file (*i*x fileid xt -- j*x*) gforth-0.6 182
execute-task (*xt -- task*) gforth-experimental 272
EXIT (*compilation -- ; run-time nest-sys* --) core 114
exitm (--) objects 238
expand-where (--) gforth-1.0 260
expect (*c-addr +n* --) gforth-obsolete .. 214
extend-mem (*addr1 u1 u -- addr addr2 u2*) gforth-experimental 80
extend-structure (*n* "*name*" -- *struct-sys n*) gforth-1.0 147
extra-section (*usize* "*name*" --) gforth-1.0 79

F

f! (*r f-addr* --) floating 82
f* (*r1 r2 -- r3*) floating 69
f** (*r1 r2 -- r3*) floating-ext 70
f*+ (*r1 r2 r3 -- r*) gforth-1.0 69
f+ (*r1 r2 -- r3*) floating 69
f, (*f* --) gforth-0.2 76
f- (*r1 r2 -- r3*) floating 69
f-rot (*r1 r2 r3 -- r3 r1 r2*) gforth-1.0 ... 73
f. (*r* --) floating-ext 206
f.rdp (*rf +nr +nd +np* --) gforth-0.6 207

f.s (--) gforth-0.2 263
f.s-precision (-- u) gforth-1.0 263
f/ (r1 r2 -- r3) floating 70
f< (r1 r2 -- f) floating 72
f<= (r1 r2 -- f) gforth-0.2 72
f<> (r1 r2 -- f) gforth-0.2 72
f= (r1 r2 -- f) gforth-0.2 72
f> (r1 r2 -- f) gforth-0.2 72
f>= (r1 r2 -- f) gforth-0.2 72
f>buf-rdp (rf c-addr +nr +nd +np --)
gforth-0.6 192
f>d (r -- d) floating 69
f>l (r --) gforth-internal 226
f>r (r --) gforth-experimental 74
f>s (r -- n) floating-ext 69
f>str-rdp (rf +nr +nd +np -- c-addr nr)
gforth-0.6 192
f@ (f-addr -- r) floating 82
f@localn (noffset -- r)
gforth-internal 225
f~ (r1 r2 r3 -- flag) floating-ext 72
f~abs (r1 r2 r3 -- flag) gforth-0.5 72
f~rel (r1 r2 r3 -- flag) gforth-0.5 72
f0< (r -- f) floating 72
f0<= (r -- f) gforth-0.2 72
f0<> (r -- f) gforth-0.2 72
f0= (r -- f) floating 72
f0> (r -- f) gforth-0.2 72
f0>= (r -- f) gforth-0.2 72
f2* (r1 -- r2) gforth-0.2 70
f2/ (r1 -- r2) gforth-0.2 70
F: (compilation "name" -- a-addr xt;
run-time r --) gforth-0.2 220
F^ (compilation "name" -- a-addr xt;
run-time r --) gforth-0.2 220
fabs (r1 -- r2) floating-ext 70
facos (r1 -- r2) floating-ext 71
facosh (r1 -- r2) floating-ext 71
falign (--) floating 77
faligned (c-addr -- f-addr) floating 85
falog (r1 -- r2) floating-ext 70
false (-- f) core-ext 60
fancy-after-l (c-addr1 u1 lineno1 -- c-addr2
u2 lineno2) gforth-internal 258
fasin (r1 -- r2) floating-ext 71
fasinh (r1 -- r2) floating-ext 71
fast-throw (... nerror -- ... nerror)
gforth-experimental 114
fatan (r1 -- r2) floating-ext 71
fatan2 (r1 r2 -- r3) floating-ext 71
fatanh (r1 -- r2) floating-ext 71
faxpy (ra f-x nstridex f-y nstridey ucount --
) gforth-0.5 71
fclearstack (r0 .. rn --) gforth-1.0 264
fconstant (r "name" --) floating 121
fcopysign (r1 r2 -- r3) gforth-1.0 70
fcos (r1 -- r2) floating-ext 71
fcosh (r1 -- r2) floating-ext 71
fdepth (-- +n) floating 264
fdrop (r --) floating 73
fdup (r -- r r) floating 73
fe. (r --) floating-ext 206
fexp (r1 -- r2) floating-ext 70
fexpm1 (r1 -- r2) floating-ext 70
ffield: (u1 "name" -- u2) floating-ext .. 143
ffourth (r1 r2 r3 r4 -- r1 r2 r3 r4 r1)
gforth-1.0 73
field (align1 offset1 align size "name" --
align2 offset2) gforth-0.2 149
field: (u1 "name" -- u2) facility-ext ... 143
file-eof? (wfileid -- flag) gforth-0.6 . 198
file-position (wfileid -- ud wior) file . 198
file-size (wfileid -- ud wior) file 198
file-status (c-addr u -- wfam wior)
file-ext 198
file>fpath (addr1 u1 -- addr2 u2)
gforth-1.0 201
file>path (c-addr1 u1 path-addr -- c-addr2
u2) gforth-1.0 201
filename-match (c-addr1 u1 c-addr2 u2 --
flag) gforth-0.5 200
fill (c-addr u c --) core 87
find (c-addr -- xt +-1 | c-addr 0)
core,search 184
find-name (c-addr u -- nt | 0)
gforth-0.2 158
find-name-in (c-addr u wid -- nt | 0)
gforth-1.0 158
fkey. (u --) gforth-1.0 214
flit, (r --) gforth-1.0 162
FLiteral (compilation r -- ; run-time -- r)
floating 161
fln (r1 -- r2) floating-ext 70
flnp1 (r1 -- r2) floating-ext 70
float (-- u) gforth-0.3 85
float% (-- align size) gforth-0.4 149
float+ (f-addr1 -- f-addr2) floating 85
float/ (n1 -- n2) gforth-1.0 85
floating-stack (-- n) environment 193
floats (n1 -- n2) floating 85
flog (r1 -- r2) floating-ext 70
floor (r1 -- r2) floating 70
FLOORED (-- f) environment 193
flush (--) block 204
flush-file (wfileid -- wior) file-ext .. 198
flush-icache (c-addr u --) gforth-0.2 .. 286
fm/mod (d1 n1 -- n2 n3) core 63
fmax (r1 r2 -- r3) floating 70
fmin (r1 r2 -- r3) floating 70
fnegate (r1 -- r2) floating 70
fnip (r1 r2 -- r2) gforth-0.2 73
FOR (compilation -- do-sys ; run-time u --
loop-sys) gforth-0.2 108
FORK (compilation -- orig ; run-time f --)
gforth-0.7 257
form (-- nlines ncols) gforth-0.2 210

Forth (--) search-ext 184
forth-wordlist (-- wid) search 183
forward ("name" --) gforth-1.0 114
fourth (w1 w2 w3 w4 -- w1 w2 w3 w4 w1)
gforth-1.0 72
fover (r1 r2 -- r1 r2 r1) floating 73
fp! (f-addr -- f:...) gforth-0.2 75
fp. (r --) floating-ext 206
fp@ (f:... -- f-addr) gforth-0.2 75
fp0 (-- a-addr) gforth-0.4 75
fpath (-- path-addr) gforth-0.4 201
fpick (f:... u -- f:... r) gforth-0.4 73
fr> (-- r) gforth-experimental 74
fr@ (-- r) gforth-experimental 74
free (a_addr -- wior) memory 80
free-closure (xt --) gforth-1.0 249
free-mem-var (addr --)
gforth-experimental 80
frot (r1 r2 r3 -- r2 r3 r1) floating 73
fround (r1 -- r2) floating 70
fs. (r --) floating-ext 206
fsin (r1 -- r2) floating-ext 71
fsincos (r1 -- r2 r3) floating-ext 71
fsinh (r1 -- r2) floating-ext 71
fsqrt (r1 -- r2) floating-ext 70
fswap (r1 r2 -- r2 r1) floating 73
ftan (r1 -- r2) floating-ext 71
ftanh (r1 -- r2) floating-ext 71
fthird (r1 r2 r3 -- r1 r2 r3 r1)
gforth-1.0 73
ftrunc (r1 -- r2) floating-ext 70
ftuck (r1 r2 -- r2 r1 r2) gforth-0.2 73
fvalue (r "name" --) floating-ext 122
fvalue: (u1 "name" -- u2)
gforth-experimental 145
fvariable ("name" --) floating 120

G

g (--) gforth-1.0 259
get-block-fid (-- wfileid) gforth-0.2 .. 204
get-current (-- wid) search 183
get-dir (c-addr1 u1 -- c-addr2 u2)
gforth-0.7 200
get-order (-- widn .. wid1 n) search 183
get-recs (recs-xt -- xtu .. xt1 u)
gforth-experimental 176
get-stack (stack -- x1 .. xn n)
gforth-experimental 150
getenv (c-addr1 u1 -- c-addr2 u2)
gforth-0.2 305
gforth (-- c-addr u)
gforth-environment 194
gg (--) gforth-1.0 260

H

h. (u --) gforth-1.0 205
halt (task --) gforth-experimental 273
heap-new (... class -- object) objects .. 238
help ("rest-of-line" --) gforth-1.0 9
here (-- addr) core 76
hex (--) core-ext 187
hex. (u --) gforth-0.2 205
hold (char --) core 190
holds (addr u --) core-ext 190
how: (--) oof 242

I

i (R:n -- R:n n) core 108
i' (R:w R:w2 -- R:w R:w2 w) gforth-0.2 .. 108
id. (nt --) gforth-0.6 159
IF (compilation -- orig ; run-time f --)
core 112
iferror (compilation orig1 -- orig2 ;
run-time --) gforth-0.7 117
immediate (--) core 152
immediate? (nt -- flag) gforth-1.0 302
implementation (interface --) objects .. 238
in ("voc" "defining-word" --)
gforth-experimental 183
in-colon-def? (-- flag)
gforth-experimental 166
in-wordlist (wordlist "defining-word" --)
gforth-experimental 183
include (... "file" -- ...) file-ext 196
include-file (i*x wfileid -- j*x) file .. 196
included (i*x c-addr u -- j*x) file 196
included? (c-addr u -- f) gforth-0.2 196
inf (-- r) gforth-1.0 71
infile-execute (... xt file-id -- ...)
gforth-0.7 199
infile-id (-- file-id) gforth-0.4 199
infinity (-- r) gforth-1.0 71
info-color (--) gforth-1.0 210
init-asm (--) gforth-0.2 285
init-buffer (addr --)
gforth-experimental 81
init-object (... class object --)
objects 238
initiate (xt task --)
gforth-experimental 272
inline: ("name" -- inline:-sys)
gforth-experimental 123
input-color (--) gforth-1.0 210
insert (c-addr1 u1 c-addr2 u2 --)
gforth-0.7 94
inst-value (align1 offset1 "name" -- align2
offset2) objects 238
inst-var (align1 offset1 align size "name"
-- align2 offset2) objects 238
INT-[I] (-- n) gforth-1.0 170
intel-syntax (--) gforth-1.0 289

interface (--) objects 238
interpret/compile: (int-xt comp-xt "name" --) gforth-0.2 154
interpreting (... translation -- ...) gforth-experimental 180
intsem: (--) gforth-experimental 156
invalid-char (-- xc) gforth-experimental 91
invert (w1 -- w2) core 66
IS (xt ... "name" --) core-ext 138

J

j (R:n R:w1 R:w2 -- n R:n R:w1 R:w2) core 108
JOIN (orig --) gforth-0.7 257

K

k (R:n R:w1 R:w2 R:w3 R:w4 -- n R:n R:w1 R:w2 R:w3 R:w4) gforth-0.3 108
k-alt-mask (-- u) facility-ext 213
k-backspace (-- u) gforth-1.0 213
k-ctrl-mask (-- u) facility-ext 213
k-delete (-- u) facility-ext 213
k-down (-- u) facility-ext 212
k-end (-- u) facility-ext 212
k-enter (-- u) gforth-1.0 213
k-eof (-- u) gforth-1.0 213
k-f1 (-- u) facility-ext 213
k-f10 (-- u) facility-ext 213
k-f11 (-- u) facility-ext 213
k-f12 (-- u) facility-ext 213
k-f2 (-- u) facility-ext 213
k-f3 (-- u) facility-ext 213
k-f4 (-- u) facility-ext 213
k-f5 (-- u) facility-ext 213
k-f6 (-- u) facility-ext 213
k-f7 (-- u) facility-ext 213
k-f8 (-- u) facility-ext 213
k-f9 (-- u) facility-ext 213
k-home (-- u) facility-ext 212
k-insert (-- u) facility-ext 213
k-left (-- u) facility-ext 212
k-mute (-- u) gforth-1.0 213
k-next (-- u) facility-ext 213
k-pause (-- u) gforth-1.0 213
k-prior (-- u) facility-ext 212
k-right (-- u) facility-ext 212
k-sel (-- u) gforth-1.0 213
k-shift-mask (-- u) facility-ext 213
k-tab (-- u) gforth-1.0 213
k-up (-- u) facility-ext 212
k-voldown (-- u) gforth-1.0 213
k-volup (-- u) gforth-1.0 213
k-winch (-- u) gforth-1.0 213
key (-- c) core 211
key-file (fd -- key) gforth-0.4 198
key-ior (-- c|ior) gforth-1.0 211
key? (-- flag) facility 211
key?-file (wfileid -- f) gforth-0.4 198
kill (task --) gforth-experimental 272
kill-task (--) gforth-experimental 272

L

l (--) gforth-1.0 259
l! (w c-addr --) gforth-0.7 83
l, (l --) gforth-1.0 77
l>s (x -- n) gforth-1.0 84
l@ (c-addr -- u) gforth-0.7 83
L" (Interpretation "string<">" -- lsid; Compilation "string<">" --) gforth-experimental 100
lalign (--) gforth-1.0 86
laligned (addr -- addr') gforth-1.0 86
latest (-- nt|0) gforth-0.6 158
latestnt (-- nt|0) gforth-1.0 158
latestxt (-- xt) gforth-0.6 124
lbe (u1 -- u2) gforth-1.0 83
LEAVE (compilation -- ; run-time loop-sys --) core 109
lfield: (u1 "name" -- u2) gforth-1.0 143
lib-error (-- c-addr u) gforth-0.7 283
lib-sym (c-addr1 u1 c-addr2 -- c-addr3) gforth-0.4 283
license (--) gforth-0.2 10
light-mode (--) gforth-1.0 211
line-end-hook (--) gforth-0.7 181
list (u --) block-ext 204
list-size (list -- u) gforth-internal .. 227
lit, (x --) gforth-1.0 161
Literal (compilation n -- ; run-time -- n) core 161
ll (--) gforth-1.0 260
lle (u1 -- u2) gforth-1.0 83
load (i*x u -- j*x) block 204
load-cov (--) gforth-experimental 269
locale! (addr u lsid --) gforth-experimental 101
locale-csv ("name" --) gforth-experimental 101
locale-csv-out ("name" --) gforth-experimental 101
Locale: ("name" --) gforth-experimental 101
locale@ (lsid -- c-addr u) gforth-experimental 100
locales (--) gforth-experimental 100
locals| (... "name ..." --) local-ext ... 219
locate ("name" --) gforth-1.0 258
lock (mutex --) gforth-experimental 274
log2 (u -- n) gforth-1.0 67
LOOP (compilation do-sys -- ; run-time loop-sys1 -- | loop-sys2) core 108
lp! (c-addr --) gforth-internal 75

lp+! (*noffset* --) gforth-internal 225
lp+n (*noffset* -- *c-addr*)
gforth-internal 225
lp@ (-- *c-addr*) gforth-internal 75
lp0 (-- *a-addr*) gforth-0.4 75
lrol (*u1 u* -- *u2*) gforth-1.0 67
lror (*u1 u* -- *u2*) gforth-1.0 67
lshift (*u1 u* -- *u2*) core 66
lvalue: (*u1* "*name*" -- *u2*)
gforth-experimental 145

M

m* (*n1 n2* -- *d*) core 61
m*/ (*d1 n2 u3* -- *dquot*) double 64
m+ (*d1 n* -- *d2*) double 61
m: (-- *xt colon-sys; run-time: object* --)
objects 238
macros-wordlist (-- *wid*)
gforth-experimental 101
magenta-input (--) gforth-1.0 211
make-latest (*nt* --) gforth-1.0 137
map-vocs (... *xt* -- ...) gforth-1.0 185
marker ("<*spaces*> *name*" --) core-ext ... 264
max (*n1 n2* -- *n*) core 61
max-float (-- *r*) environment 193
MAX-CHAR (-- *u*) environment 192
MAX-D (-- *d*) environment 193
MAX-N (-- *n*) environment 193
MAX-U (-- *u*) environment 193
MAX-UD (-- *ud*) environment 193
MAX-XCHAR (-- *xchar*) environment 193
maxalign (--) gforth-0.2 77
maxaligned (*addr1* -- *addr2*) gforth-0.2 .. 86
maxdepth-.s (-- *addr*) gforth-0.2 263
mem+do (*compilation* -- *w xt do-sys; run-time*
addr ubytes +nstride --)
gforth-experimental 108
mem, (*addr u* --) gforth-0.6 77
mem-do (*compilation* -- *w xt do-sys; run-time*
addr ubytes +nstride --)
gforth-experimental 108
method (*xt* "*name*" --) objects .. 238, 242, 243
methods (*class* --) objects 238
min (*n1 n2* -- *n*) core 61
mkdir-parents (*c-addr u mode* -- *ior*)
gforth-0.7 200
mod (*n1 n2* -- *n*) core 62
modf (*n1 n2* -- *n*) gforth-1.0 62
modf-stage2m (*n1 a-reci* -- *umodulus*)
gforth-1.0 65
mods (*n1 n2* -- *n*) gforth-1.0 62
move (*c-from c-to ucount* --) core 87
ms (*u* --) facility-ext 305
mutex ("*name*" --) gforth-experimental .. 274
mux (*u1 u2 u3* -- *u*) gforth-1.0 66
mwords (*["pattern"]* --) gforth-1.0 184

N

n (--) gforth-1.0 259
n/a (--) gforth-experimental 134
n>r (*x1 .. xn n* -- *R:xn..R:x1 R:n*)
tools-ext 74
name (-- *c-addr u*) gforth-obsolete 181
name>compile (*nt* -- *xt1 xt2*) tools-ext .. 158
name>interpret (*nt* -- *xt*) tools-ext 158
name>link (*nt1* -- *nt2 / 0*) gforth-1.0 ... 159
name>string (*nt* -- *addr u*) tools-ext 158
NaN (-- *r*) gforth-1.0 71
native@ (*lsid* -- *c-addr u*)
gforth-experimental 100
needs (... "*name*" -- ...) gforth-0.2 197
negate (*n1* -- *n2*) core 61
new (*class* -- *o*) mini-oof 243
newline (-- *c-addr u*) gforth-0.5 93
newtask (*stacksize* -- *task*)
gforth-experimental 272
newtask4 (*u-data u-return u-fp u-locals* --
task) gforth-experimental 272
next-arg (-- *addr u*) gforth-0.7 216
next-case (*compilation case-sys* -- ;
run-time --) gforth-1.0 111
next-section (--) gforth-1.0 79
NEXT (*compilation do-sys* -- ; *run-time*
loop-sys1 -- | *loop-sys2*) gforth-0.2 .. 108
nextname (*c-addr u* --) gforth-0.2 125
nip (*w1 w2* -- *w2*) core-ext 72
nocov[(--) gforth-1.0 269
noname (--) gforth-0.2 124
noname-from (*xt* --) gforth-1.0 136
noop (--) gforth-0.2 157
nosplit? (*addr1 u1 addr2 u2* -- *addr1 u1 addr2*
u2 flag) gforth-experimental 94
nothrow (--) gforth-0.7 116
nr> (*R:xn..R:x1 R:n* -- *x1 .. xn n*)
tools-ext 74
ns (*ud* --) gforth-1.0 305
nt (--) gforth-1.0 261
ntime (-- *dtime*) gforth-1.0 306
nw (--) gforth-1.0 260

O

o> (*r:c-addr* --) new 246
object (-- *class*) objects 238, 243
object-' ("*name*" -- *xt*) oof 242
object-: ("*name*" --) oof 241
object-:: ("*name*" --) oof 241
object-[] (*n* "*name*" --) oof 241
object-asptr (*o* "*name*" --) oof 241
object-bind (*o* "*name*" --) oof 241
object-bound (*class addr* "*name*" --) oof . 241
object-class ("*name*" --) oof 241
object-class? (*o* -- *flag*) oof 241
object-definitions (--) oof 241
object-dispose (--) oof 241

object-endwith (--) oof ... 242
object-init (... --) oof ... 241
object-is (xt "name" --) oof ... 241
object-link ("name" -- class addr) oof .. 241
object-new (-- o) oof ... 241
object-new[] (n -- o) oof ... 241
object-postpone ("name" --) oof ... 242
object-ptr ("name" --) oof ... 241
object-self (-- o) oof ... 241
object-super ("name" --) oof ... 241
object-with (o --) oof ... 242
obsolete? (nt -- flag) gforth-1.0 ... 159
of (compilation -- of-sys ; run-time x1 x2 -- |x1) core-ext ... 111
off (a-addr --) gforth-0.2 ... 60
on (a-addr --) gforth-0.2 ... 60
once (--) gforth-1.0 ... 265
Only (--) search-ext ... 184
open-blocks (c-addr u --) gforth-0.2 ... 204
open-dir (c-addr u -- wdirid wior) gforth-0.5 ... 200
open-file (c-addr u wfam -- wfileid wior) file ... 197
open-lib (c-addr1 u1 -- c-addr2) gforth-0.4 ... 283
open-path-file (addr1 u1 path-addr -- wfileid addr2 u2 0 | ior) gforth-0.2 ... 201
open-pipe (c-addr u wfam -- wfileid wior) gforth-0.2 ... 215
opt: (compilation -- colon-sys2 ; run-time -- nest-sys) gforth-1.0 ... 134
or (w1 w2 -- w) core ... 66
order (--) search-ext ... 184
os-class (-- c-addr u) gforth-environment ... 194
os-type (-- c-addr u) gforth-environment ... 194
out (-- addr) gforth-1.0 ... 208
outfile-execute (... xt file-id -- ...) gforth-0.7 ... 199
outfile-id (-- file-id) gforth-0.2 ... 199
over (w1 w2 -- w1 w2 w1) core ... 72
overrides (xt "selector" --) objects ... 238

P

pad (-- c-addr) core-ext ... 87
page (--) facility ... 210
parse (xchar "ccc<xchar>" -- c-addr u) core-ext,xchar-ext ... 181
parse-name ("name" -- c-addr u) core-ext ... 181
parse-word (-- c-addr u) gforth-obsolete ... 181
path+ (path-addr "dir" --) gforth-0.4 .. 202
path= (path-addr "dir1|dir2|dir3" --) gforth-0.4 ... 202
pause (--) gforth-experimental ... 273
perform (a-addr --) gforth-0.2 ... 157
pi (-- r) gforth-0.2 ... 71
pick (S:... u -- S:... w) core-ext ... 73
place (c-addr1 u c-addr2 --) gforth-experimental ... 102
postpone ("name" --) core ... 162
postpone, (xt1 xt2 --) gforth-0.2 ... 160
postpone-color (--) gforth-1.0 ... 210
postponing (... translation --) gforth-experimental ... 180
pow2? (u -- f) gforth-1.0 ... 67
precision (-- u) floating-ext ... 207
prepend-where (--) gforth-1.0 ... 260
preserve (compilation "name" -- ; run-time --) gforth-1.0 ... 140
previous (--) search-ext ... 183
previous-section (--) gforth-1.0 ... 79
print (object --) objects ... 239
printdebugdata (--) gforth-0.2 ... 265
process-option (addr u -- ... xt | 0) gforth-0.7 ... 336
program (--) gforth-experimental ... 100
protected (--) objects ... 239
ptr (--) oof ... 242
public (--) objects ... 239

Q

query (--) gforth-obsolete ... 168
quit (?? -- ??) core ... 306

R

r'@ (r:w r:w2 -- r:w r:w2 w) gforth-1.0 .. 74
r/o (-- fam) file ... 197
r/w (-- fam) file ... 197
r> (R:w -- w) core ... 74
r@ (R:w -- R:w w) core ... 74
random (u1 -- u2) gforth-1.0 ... 69
rdrop (R:w --) gforth-0.2 ... 74
read-csv (c-addr u xt --) gforth-experimental ... 215
read-dir (c-addr u1 wdirid -- u2 flag wior) gforth-0.5 ... 200
read-file (c-addr u1 wfileid -- u2 wior) file ... 197
read-line (c_addr u1 wfileid -- u2 flag wior) file ... 198
rec-body (addr u -- translation) gforth-experimental ... 175
rec-complex (c-addr u -- translation) gforth-experimental ... 174
rec-dtick (c-addr u -- translation) gforth-experimental ... 175
rec-env (c-addr u -- translation) gforth-1.0 ... 175
rec-filter (c-addr u xt: filter xt: rec -- translation) gforth-experimental ... 178

rec-float (c-addr u -- translation)
gforth-experimental 174
rec-forth (c-addr u -- translation)
gforth-experimental 175
rec-forth-nt? (c-addr u -- nt | 0)
gforth-experimental 178
rec-local (c-addr u -- translation)
gforth-experimental 174
rec-meta (addr u -- xt translate-to | 0)
gforth-1.0 175
rec-moof2 (c-addr u -- nt translate-moof2 |
translate-none) mini-oof2 246
rec-name (c-addr u -- translation)
gforth-experimental 173
rec-none (c-addr u -- translate-none)
gforth-experimental 176
rec-number (c-addr u -- translation)
gforth-experimental 174
rec-scope (c-addr u -- translation)
gforth-experimental 174
rec-sequence: (xtu .. xt1 u "name" --)
gforth-experimental 175
rec-string (c-addr u -- translation)
gforth-experimental 174
rec-tick (c-addr u -- translation)
gforth-experimental 175
rec-to (c-addr u -- translation)
gforth-experimental 174
recs (--) gforth-experimental 173
recurse (... -- ...) core 113
recursive (compilation -- ; run-time --)
gforth-0.2 113
refill (-- flag)
core-ext,block-ext,file-ext 182
rename-file (c-addr1 u1 c-addr2 u2 -- wior)
file-ext 197
REPEAT (compilation orig dest -- ; run-time
--) core 113
replace-word (xt1 xt2 --) gforth-1.0 ... 266
replacer: ("name" --)
gforth-experimental 102
replaces (addr1 len1 addr2 len2 --)
string-ext 101
reposition-file (ud wfileid -- wior)
file 198
represent (r c-addr u -- n f1 f2)
floating 192
require (... "file" -- ...) file-ext 196
required (i*x addr u -- i*x) file-ext ... 196
resize (a_addr1 u -- a_addr2 wior)
memory 80
resize-file (ud wfileid -- wior) file ... 198
restart (task --) gforth-experimental .. 273
restore (compilation orig1 -- ; run-time --
) gforth-0.7 118
restore-input (x1 .. xn n -- flag)
core-ext 168
restrict (--) gforth-0.2 154
return-stack-cells (-- n) environment .. 193
reveal (--) gforth-0.2 136
reveal! (xt wid --) core-ext 136
rnd (-- x) gforth-1.0 69
rol (u1 u -- u2) gforth-1.0 67
roll (x0 x1 .. xn n -- x1 .. xn x0)
core-ext 73
Root (--) gforth-0.2 185
ror (u1 u -- u2) gforth-1.0 67
rot (w1 w2 w3 -- w2 w3 w1) core 73
rp! (a-addr --) gforth-0.2 75
rp@ (-- a-addr) gforth-0.2 75
rp0 (-- a-addr) gforth-0.4 75
rpick (R:wu ... R:w0 u -- R:wu ... R:w0 wu)
gforth-1.0 74
rshift (u1 u -- u2) core 66

S

s+ (c-addr1 u1 c-addr2 u2 -- c-addr u)
gforth-0.7 95
s// (addr u -- ptr) regexp-replace 257
s>> (addr -- addr) regexp-replace 257
s>d (n -- d) core 61
s>f (n -- r) floating-ext 69
s\" (Interpretation 'ccc"' -- c-addr u)
core-ext,file-ext 92
S" (Interpretation 'ccc"' -- c-addr u)
core,file 92
safe/string (c-addr1 u1 n -- c-addr2 u2)
gforth-1.0 94
save-buffers (--) block 204
save-cov (--) gforth-experimental 269
save-input (-- x1 .. xn n) core-ext 168
save-mem (addr1 u -- addr2 u) gforth-0.2 . 80
save-mem-dict (addr1 u -- addr2 u)
gforth-0.7 77
savesystem ("image" --) gforth-0.2 334
scan (c-addr1 u1 c -- c-addr2 u2)
gforth-0.2 94
scan-back (c-addr u1 c -- c-addr u2)
gforth-0.7 94
scan-translate-string (c-addr1 u1 'ccc"' --
translation) gforth-experimental 178
scope (compilation -- scope ; run-time --)
gforth-0.2 220
scr (-- a-addr) block-ext 204
scvalue: (u1 "name" -- u2)
gforth-experimental 145
seal (--) gforth-0.2 185
search (c-addr1 u1 c-addr2 u2 -- c-addr3 u3
flag) string 94
search-wordlist (c-addr count wid -- 0 | xt
+-1) search 184
see ("<spaces>name" --) tools 261
see-code ("name" --) gforth-0.7 261
see-code-range (addr1 addr2 --)
gforth-0.7 262

seed! (x --) gforth-1.0 ... 69
select (u1 u2 f -- u) gforth-1.0 ... 60
selector ("name" --) objects ... 239
send-event (xt task --)
gforth-experimental ... 275
set->comp (xt --) gforth-1.0 ... 302
set->int (xt --) gforth-1.0 ... 302
set-compsem (xt --)
gforth-experimental ... 156
set-current (wid --) search ... 183
set-dir (c-addr u -- wior) gforth-0.7 ... 200
set-does> (xt --) gforth-1.0 ... 129
set-execute (ca --) gforth-1.0 ... 301
set-name>link (xt --) gforth-1.0 ... 303
set-name>string (xt --) gforth-1.0 ... 303
set-optimizer (xt --) gforth-1.0 ... 134
set-order (widn .. wid1 n --) search ... 183
set-precision (u --) floating-ext ... 207
set-recs (xtu .. xt1 u recs-xt --)
gforth-experimental ... 176
set-stack (x1 .. xn n stack --)
gforth-experimental ... 150
set-to (to-xt --) gforth-1.0 ... 134
sf! (r sf-addr --) floating-ext ... 82
sf@ (sf-addr -- r) floating-ext ... 82
sfalign (--) floating-ext ... 77
sfaligned (c-addr -- sf-addr)
floating-ext ... 85
sffield: (u1 "name" -- u2) floating-ext . 143
sfloat% (-- align size) gforth-0.4 ... 149
sfloat+ (sf-addr1 -- sf-addr2)
floating-ext ... 85
sfloat/ (n1 -- n2) gforth-1.0 ... 85
sfloats (n1 -- n2) floating-ext ... 85
sfvalue: (u1 "name" -- u2)
gforth-experimental ... 145
sh ("..." --) gforth-0.2 ... 305
sh-get (c-addr u -- c-addr2 u2)
gforth-1.0 ... 305
shift-args (--) gforth-0.7 ... 216
short-where (--) gforth-1.0 ... 260
sign (n --) core ... 190
simple-fkey-string (u1 -- c-addr u)
gforth-1.0 ... 214
simple-see ("name" --) gforth-0.6 ... 261
simple-see-range (addr1 addr2 --)
gforth-0.6 ... 261
skip (c-addr1 u1 c -- c-addr2 u2)
gforth-0.2 ... 94
sleep (task --) gforth-experimental ... 273
slit, (c-addr1 u --) gforth-1.0 ... 162
SLiteral (Compilation c-addr1 u -- ;
run-time -- c-addr2 u) string ... 162
slurp-fid (fid -- c-addr u) gforth-0.6 .. 198
slurp-file (c-addr1 u1 -- c-addr2 u2)
gforth-0.6 ... 198
slvalue: (u1 "name" -- u2)
gforth-experimental ... 145
sm/rem (d1 n1 -- n2 n3) core ... 63
source (-- c-addr u) core ... 167
source-id (-- 0 | -1 | fileid)
core-ext,file ... 168
sourcefilename (-- c-addr u)
gforth-0.2 ... 197
sourceline# (-- u) gforth-0.2 ... 197
sp! (a-addr -- S:...) gforth-0.2 ... 75
sp@ (S:... -- a-addr) gforth-0.2 ... 75
sp0 (-- a-addr) gforth-0.4 ... 75
space (--) core ... 208
spaces (u --) core ... 208
span (-- addr) gforth-obsolete ... 214
spawn (xt --) cilk ... 276
spawn1 (x xt --) cilk ... 276
spawn2 (x1 x2 xt --) cilk ... 276
stack (n -- stack) gforth-experimental . 149
stack-cells (-- n) environment ... 193
stack: (n "name" --)
gforth-experimental ... 149
stack> (stack -- x)
gforth-experimental ... 149
stacksize (-- u) gforth-experimental ... 272
stacksize4 (-- u-data u-return u-fp u-locals
) gforth-experimental ... 272
staged/-divisor (addr1 -- addr2)
gforth-1.0 ... 66
staged/-size (-- u) gforth-1.0 ... 65
state (-- a-addr) core,tools-ext ... 169
static (--) oof ... 242
status-color (--) gforth-1.0 ... 210
stderr (-- wfileid) gforth-0.2 ... 199
stdin (-- wfileid) gforth-0.4 ... 198
stdout (-- wfileid) gforth-0.2 ... 198
stop (--) gforth-experimental ... 272
stop-dns (dtimeout --)
gforth-experimental ... 272
stop-ns (timeout --)
gforth-experimental ... 272
str< (c-addr1 u1 c-addr2 u2 -- f)
gforth-0.6 ... 94
str= (c-addr1 u1 c-addr2 u2 -- f)
gforth-0.6 ... 93
str=? (addr1 addr u -- addr2)
regexp-pattern ... 256
string, (c-addr u --) gforth-0.2 ... 102
string-parse (c-addr1 u1 "ccc<string>" --
c-addr2 u2) gforth-1.0 ... 181
string-prefix? (c-addr1 u1 c-addr2 u2 -- f)
gforth-0.6 ... 94
string-suffix? (c-addr1 u1 c-addr2 u2 -- f)
gforth-1.0 ... 94
struct (-- align size) gforth-0.2 ... 149
sub-list? (list1 list2 -- f)
gforth-internal ... 227
substitute (addr1 len1 addr2 len2 -- addr2
len3 n/ior) string-ext ... 102
success-color (--) gforth-1.0 ... 210

swap (*w1 w2 -- w2 w1*) core 73
swvalue: (*u1* "*name*" *-- u2*) gforth-experimental 145
Synonym ("*name*" "*oldname*" *--*) tools-ext 141
system (*c-addr u --*) gforth-0.2 305

T

table (*-- wid*) gforth-0.2 183
task (*ustacksize* "*name*" *--*) gforth-experimental 272
th (*a-addr1 n -- a-addr2*) gforth-0.7 86
th! (*u a-addr n --*) gforth-1.0 86
th@ (*a-addr n -- u*) gforth-1.0 86
THEN (*compilation orig -- ; run-time --*) core 112
third (*w1 w2 w3 -- w1 w2 w3 w1*) gforth-1.0 72
this (*-- object*) objects 239
thread-deadline (*d --*) gforth-experimental 273
threading-method (*-- n*) gforth-0.2 303
throw (*y1 .. ym nerror -- y1 .. ym / z1 .. zn nerror*) exception 114
thru (*i*x n1 n2 -- j*x*) block-ext 204
tib (*-- addr*) gforth-obsolete 167
time&date (*-- nsec nmin nhour nday nmonth nyear*) facility-ext 305
to-class: (*xt table* "*name*" *--*) gforth-experimental 134
to-table: ("*name*" "*to-word*" "*+to-word*" "*addr-word*" "*action-of-word*" "*is-word*" *--*) gforth-experimental 133
to-this (*object --*) objects 239
TO (*value ...* "*name*" *--*) core-ext 122
toupper (*xc1 -- xc2*) gforth-0.2 91
translate-cell (*x -- translation*) gforth-experimental 177
translate-complex (*r1 r2 -- translation*) gforth-experimental 177
translate-dcell (*xd -- translation*) gforth-experimental 177
translate-env (*c-addr1 u1 -- translation*) gforth-experimental 178
translate-float (*r -- translation*) gforth-experimental 177
translate-name (*nt -- translation*) gforth-experimental 177
translate-string (*c-addr1 u1 -- translation*) gforth-experimental 178
translate-to (*n xt -- translation*) gforth-experimental 178
translate: (*int-xt comp-xt post-xt* "*name*" *--*) gforth-experimental 179
traverse-wordlist (*... xt wid -- ...*) tools-ext 158
true (*-- f*) core-ext 60
try (*compilation -- orig ; run-time -- R:sys1*) gforth-0.5 116
tt (*u --*) gforth-1.0 260
tuck (*w1 w2 -- w2 w1 w2*) core-ext 73
type (*c-addr u --*) core 209
typewhite (*c-addr u --*) gforth-0.2 209

U

u*/ (*u1 u2 u3 -- u4*) gforth-1.0 63
u*/mod (*u1 u2 u3 -- u4 u5*) gforth-1.0 63
u-[do (*compilation -- do-sys ; run-time u1 u2 -- | loop-sys*) gforth-experimental 107
u. (*u --*) core 205
u.r (*u n --*) core-ext 205
u/ (*u1 u2 -- u*) gforth-1.0 62
u/-stage1m (*u a-reci --*) gforth-1.0 65
u/-stage2m (*u1 a-reci -- uquotient*) gforth-1.0 65
u/mod (*u1 u2 -- u3 u4*) gforth-1.0 63
u/mod-stage2m (*u1 a-reci -- umodulus uquotient*) gforth-1.0 65
u< (*u1 u2 -- f*) core 68
u<= (*u1 u2 -- f*) gforth-0.2 68
u> (*u1 u2 -- f*) core-ext 68
u>= (*u1 u2 -- f*) gforth-0.2 68
U+DO (*compilation -- do-sys ; run-time u1 u2 -- | loop-sys*) gforth-0.2 107
U-DO (*compilation -- do-sys ; run-time u1 u2 -- | loop-sys*) gforth-0.2 108
uallot (*n1 -- n2*) gforth-0.3 273
ud. (*ud --*) gforth-0.2 206
ud.r (*ud n --*) gforth-0.2 206
ud/mod (*ud1 u2 -- urem udquot*) gforth-0.2 63
UDefer ("*name*" *--*) gforth-1.0 273
um* (*u1 u2 -- ud*) core 61
um/mod (*ud u1 -- u2 u3*) core 63
umax (*u1 u2 -- u*) gforth-1.0 61
umin (*u1 u2 -- u*) gforth-0.5 61
umod (*u1 u2 -- u*) gforth-1.0 62
umod-stage2m (*u1 a-reci -- umodulus*) gforth-1.0 65
uncolored-mode (*--*) gforth-1.0 211
under+ (*n1 n2 n3 -- n n2*) gforth-0.3 61
unescape (*addr1 u1 dest -- dest u2*) string-ext 102
unlock (*mutex --*) gforth-experimental 274
unloop (*R:w1 R:w2 --*) core 109
UNREACHABLE (*--*) gforth-0.2 221
UNTIL (*compilation dest -- ; run-time f --*) core 112
unused (*-- u*) core-ext 76
unused-words (*--*) gforth-1.0 260
up@ (*-- a-addr*) new 274
update (*--*) block 204
updated? (*u -- f*) gforth-0.2 204
use ("*file*" *--*) gforth-0.2 204

User ("name" --) gforth-0.2 273
user' ("name" -- u)
gforth-experimental 274
utime (-- dtime) gforth-0.5 305
UValue ("name" --) gforth-1.0 273

V

v* (f-addr1 nstride1 f-addr2 nstride2 ucount
-- r) gforth-0.5 70
Value (w "name" --) core-ext 121
value: (u1 "name" -- u2)
gforth-experimental 144
value[]: (u1 "name" -- u2)
gforth-experimental 146
var (size --) oof 242, 243
Variable ("name" --) core 120
vlist (--) gforth-0.2 184
Vocabulary ("name" --) gforth-0.2 185
vocs (--) gforth-0.2 185

W

w! (w c-addr --) gforth-0.7 82
w, (x --) gforth-1.0 77
w/o (-- fam) file 197
w>s (x -- n) gforth-1.0 84
w@ (c-addr -- u) gforth-0.5 82
W: (compilation "name" -- a-addr xt;
run-time x --) gforth-0.2 219
W^ (compilation "name" -- a-addr xt;
run-time x --) gforth-0.2 219
wake (task --) gforth-experimental 273
walign (--) gforth-1.0 86
waligned (addr -- addr') gforth-1.0 86
warning-color (--) gforth-1.0 210
WARNING" (compilation 'ccc"' -- ; run-time f
--) gforth-1.0 118
warnings (-- addr) gforth-0.2 118
wbe (u1 -- u2) gforth-1.0 83
wfield: (u1 "name" -- u2) gforth-1.0 143
where ("name" --) gforth-1.0 259
whereg ("name" --) gforth-1.0 260
WHILE (compilation dest -- orig dest ;
run-time f --) core 113
within (u1 u2 u3 -- f) core-ext 68
wle (u1 -- u2) gforth-1.0 83
word (char "<chars>ccc<char>-- c-addr)
core 181
wordlist (-- wid) search 183
wordlist-words (wid --) gforth-0.6 184
wordlists (-- n) environment 193
words (--) tools 184
wrap-xt (... xt1 xt2 xt3 -- ...)
gforth-1.0 140
write-file (c-addr u1 wfileid -- wior)
file 198
write-line (c-addr u wfileid -- ior)
file 198
wrol (u1 u -- u2) gforth-1.0 67
wror (u1 u -- u2) gforth-1.0 67
WTF?? (--) gforth-1.0 265
wvalue: (u1 "name" -- u2)
gforth-experimental 144
ww (u --) gforth-1.0 259

X

x! (w c-addr --) gforth-1.0 83
x, (x --) gforth-1.0 77
x-size (xc-addr u1 -- u2) xchar 90
x-width (xc-addr u -- n) xchar-ext 91
x>s (x -- n) gforth-1.0 84
x@ (c-addr -- u) gforth-1.0 83
x\string- (xc-addr u1 -- xc-addr u2)
xchar-ext 90
xalign (--) gforth-1.0 86
xaligned (addr -- addr') gforth-1.0 86
xbe (u1 -- u2) gforth-1.0 83
xc!+ (xc xc-addr1 -- xc-addr2) xchar 90
xc!+? (xc xc-addr1 u1 -- xc-addr2 u2 f)
xchar 90
xc, (xc --) xchar 91
xc-size (xc -- u) xchar 90
xc-width (xc -- n) xchar-ext 91
xc@ (xc-addr -- xc) xchar-ext 90
xc@+ (xc-addr1 -- xc-addr2 xc) xchar 90
xc@+? (xc-addr1 u1 -- xc-addr2 u2 xc)
gforth-experimental 90
xchar+ (xc-addr1 -- xc-addr2) xchar 90
xchar- (xc-addr1 -- xc-addr2) xchar-ext . 90
XCHAR-ENCODING (-- addr u) environment . 193
XCHAR-MAXMEM (-- u) environment 194
xd! (ud c-addr --) gforth-1.0 83
xd, (xd --) gforth-1.0 77
xd>s (xd -- d) gforth-1.0 84
xd@ (c-addr -- ud) gforth-1.0 83
xdbe (ud1 -- ud2) gforth-1.0 83
xdle (ud1 -- ud2) gforth-1.0 83
xemit (xc --) xchar 209
xfield: (u1 "name" -- u2) gforth-1.0 143
xhold (xc --) xchar-ext 91
xkey (-- xc) xchar 211
xkey? (-- flag) xchar 211
xle (u1 -- u2) gforth-1.0 83
xor (w1 w2 -- w) core 66
xt-locate (nt/xt --) gforth-1.0 259
xt-new (... class xt -- object) objects . 239
xt-see (xt --) gforth-0.2 261
xt-see-code (xt --) gforth-1.0 261
xt-simple-see (xt --) gforth-1.0 261
xt>name (xt -- nt) gforth-1.0 158
XT: (compilation "name" -- a-addr xt;
run-time xt1 --) gforth-1.0 220

Z

Z: (*compilation* "*name*" -- *a-addr xt;*
run-time r1 r2 --) gforth-1.0 220
Z^ (*compilation* "*name*" -- *a-addr xt;*
run-time r1 r2 --) gforth-1.0 220
zvalue: (*u1* "*name*" -- *u2*)
gforth-experimental 145

Concept and Word Index

Not all entries listed in this index are present verbatim in the text. This index also duplicates, in abbreviated form, all of the words listed in the Word Index (only the names are listed for the words here).

!

! 81
!!FIXME!! 266
!@ 81
!localn 225

"

", stack item type 59

#

#-prefix for decimal numbers 54
..... 190
#! 335
#> 190
#>> 190
#bell 93
#bs 93
#cr 93
#del 93
#eof 93
#esc 93
#ff 93
#lf 93
#line 170
#loc 266
#locals 193
#s 190
#tab 93
#tib 168

$

$! 96
$!len 96
$+! 96
$+!len 96
$+[]! 97
$+slurp 97
$+slurp-file 97
$-prefix for hexadecimal numbers 54
$. 97
$? 305
$[] 97
$[]! 97
$[]# 97
$[]+! 97
$[] 98
$[]@ 97
$[]free 98
$[]map 97
$[]slurp 97
$[]slurp-file 97
$[]Variable 98
$@ 96
$@len 96
$del 96
$exec 97
$free 96
$init 96
$ins 96
$iter 96
$over 96
$slurp 97
$slurp-file 97
$split 94
$substitute 102
$tmp 95
$unescape 102
$value: 145
$value[]: 146
$Variable 98

%

%-prefix for binary numbers 54
%align 148
%alignment 148
%alloc 148
%allocate 148
%allot 148
%size 149

&

&-prefix for decimal numbers 54

'

' 156
', stack item type 59
'-prefix for characters/code points 54
'cold 336
'image 332
's 274

(

(... 59
((... 254
(local) ... 227

)

) ... 267
)) ... 254

*

* ... 61
**} ... 256
*/ ... 63
*/f ... 63
*/mod ... 63
*/modf ... 63
*/mods ... 63
*/s ... 63
*} ... 256
*align ... 86
*aligned ... 86

+

+ ... 60
+! ... 81
+!@ ... 81
++} ... 256
+} ... 256
+after ... 150
+char ... 255
+chars ... 255
+class ... 255
+DO ... 107
+field ... 143
+fmode ... 197
+load ... 205
+LOOP ... 108
+ltrace ... 266
+status ... 258
+thru ... 205
+to *name* semantics, changing them ... 132
+TO ... 122
+x/string ... 90

,

, ... 77

–

- ... 61
-- ... 219
–, tutorial ... 18
--> ... 205
–appl-image, command-line option ... 4
–application, gforthmi option ... 333
–clear-dictionary, command-line option ... 6
–code-block-size, command-line option ... 7
–data-stack-size, command-line option ... 5
–debug, command-line option ... 5
–debug-mcheck, command-line option ... 5
–diag, command-line option ... 5
–dictionary-size, command-line option ... 4
–die-on-signal, command-line-option ... 6
–dynamic, command-line option ... 6
–fp-stack-size, command-line option ... 5
–help, command-line option ... 5
–ignore-async-signals, command-line-option ... 6
–image file, invoke image file ... 334
–image-file, command-line option ... 4
–locals-stack-size, command-line option ... 5
–map_32bit, command-line option ... 5
–no-0rc, command-line option ... 4
–no-dynamic command-line option ... 341
–no-dynamic, command-line option ... 6
–no-dynamic-image, command-line option ... 6
–no-offset-im, command-line option ... 6
–no-super, command-line option ... 6
–offset-image, command-line option ... 5
–opt-ip-updates, command-line option ... 7
–path, command-line option ... 4
–print-metrics, command-line option ... 7
–print-nonreloc, command-line option ... 8
–print-prims, command-line option ... 7
–print-sequences, command-line option ... 8
–return-stack-size, command-line option ... 5
–ss-greedy, command-line option ... 7
–ss-min-..., command-line options ... 6
–ss-number, command-line option ... 6
–tpa-noautomaton, command-line option ... 8
–tpa-noequiv, command-line option ... 8
–tpa-trace, command-line option ... 8
–version, command-line option ... 5
–vm-commit, command-line option ... 5
->here ... 76
-[do ... 107
-` ... 255
-\d ... 255
-\s ... 255
-c? ... 255
-char ... 255
-class ... 255
-d, command-line option ... 5
-D, command-line option ... 5
-DO ... 108
-f, command-line option ... 5
-h, command-line option ... 5

-i, command-line option ... 4
-i, invoke image file ... 334
-inf ... 71
-infinity ... 71
-l, command-line option ... 5
-LOOP ... 108
-ltrace ... 266
-m, command-line option ... 4
-p, command-line option ... 4
-r, command-line option ... 5
-rot ... 73
-stack ... 150
-status ... 258
-trailing ... 94
-trailing-garbage ... 90
-v, command-line option ... 5
-W, command-line option ... 9
-Wall, command-line option ... 9
-Werror, command-line option ... 9
-Won, command-line option ... 9
-Wpedantic, command-line option ... 9

.

. ... 205
." ... 208
.", how it works ... 51
.(... 208
.-is-dcell? ... 174
... 263
..char ... 255
.? ... 255
.\" ... 208
.cover-raw ... 270
.coverage ... 269
.debugline ... 265
.emacs ... 327
.fi files ... 330
.fpath ... 201
.gforth-history ... 10
.hm ... 301
.id ... 159
.included ... 197
.locale-csv ... 101
.path ... 202
.quoted-csv ... 215
.r ... 205
.s ... 263
.sections ... 79
.substitute ... 102
.unresolved ... 114
.voc ... 184

/

/ ... 62
// ... 256
//g ... 257
//o ... 257
//s ... 257
/COUNTED-STRING ... 192
/f ... 62
/f-stage1m ... 65
/f-stage2m ... 65
/HOLD ... 193
/l ... 87
/mod ... 62
/modf ... 63
/modf-stage2m ... 65
/mods ... 62
/PAD ... 193
/s ... 62
/string ... 94
/w ... 86
/x ... 87

:

: ... 123
:, passing data across ... 162
:: ... 243
:} ... 219
:}d ... 250
:}h ... 250
:}h1 ... 250
:}l ... 250
:}xt ... 250
:is ... 139
:m ... 238
:method ... 246
:noname ... 124

;

; ... 123
;> ... 254
;] ... 125
;abi-code ... 286
;code ... 286
;CODE ending sequence ... 322
;CODE, *name* not defined via CREATE ... 322
;CODE, processing input ... 322
;inline ... 123
;m ... 238
;m usage ... 232

<

< ... 68
<# ... 189
<< ... 257
<<" ... 257
<<# ... 190
<= ... 68
<> ... 68
<{: ... 254
<bind> ... 236
<to-inst> ... 239

=

= ... 68
=" ... 256
=mkdir ... 200

>

> ... 68
>= ... 68
>> ... 257
>addr ... 250
>back ... 150
>body ... 130
>BODY of non-CREATEd words ... 315
>code-address ... 303
>definer ... 304
>does-code ... 304
>float ... 188
>float1 ... 189
>in ... 167
>IN greater than input buffer ... 314
>l ... 225
>name ... 158
>number ... 188
>o ... 246
>order ... 183
>pow2 ... 67
>r ... 74
>stack ... 149
>string-execute ... 95
>time&date&tz ... 305
>uvalue ... 134

?

? ... 264
??? ... 265
?cov+ ... 269
?DO ... 107
?dup ... 73
?dup-IF ... 113
?DUP-0=-IF ... 113
?errno-throw ... 115
?events ... 275
?EXIT ... 114
?ior ... 115
?LEAVE ... 109
?of ... 111
?rec-found ... 180

[

[... 161
['] ... 156
[+LOOP] ... 170
[: ... 125
[?DO] ... 170
[{: ... 250
[AGAIN] ... 170
[BEGIN] ... 170
[bind] ... 237
[bind] usage ... 231
[char] ... 93
[COMP'] ... 160
[compile] ... 166
[current] ... 237
[d:d ... 249
[d:h ... 249
[d:h1 ... 249
[d:l ... 249
[defined] ... 170
[DO] ... 170
[ELSE] ... 169
[ENDIF] ... 169
[f:d ... 249
[f:h ... 249
[f:h1 ... 249
[f:l ... 249
[FOR] ... 170
[I] ... 170
[IF] ... 169
[IF] and POSTPONE ... 322
[IF], end of the input source before matching [ELSE] or [THEN] ... 322
[IFDEF] ... 170
[IFUNDEF] ... 170
[LOOP] ... 170
[n:d ... 249
[n:h ... 249
[n:h1 ... 249
[n:l ... 249
[NEXT] ... 170
[noop] ... 157
[parent] ... 239
[parent] usage ... 232
[REPEAT] ... 170
[THEN] ... 169
[to-inst] ... 239
[undefined] ... 170
[UNTIL] ... 170
[WHILE] ... 170

]

] 161
]] 163
]L 161
]nocov 269

‘

` 255
` prefix 156
` prefix of word 56
`? 255
`` prefix of word 56

@

@ 81
@localn 225

\

\, editing with Emacs 327
\, line length in blocks 316
\ 60
\$ 255
\(..... 256
\) 256
\^ 255
\\\ 197
\0 256
\c 280
\d 255
\G 60
\s 255

{

{ 219
{* 256
{** 256
{+ 256
{++ 256
{: 218
{{ 256

}

} 219
}} 256

|

| 219
|| 256

~

~~ 265
~~, removal with Emacs 327
~~1bt 265
~~bt 265
~~Value 266
~~Variable 266

0

0< 68
0<= 68
0<> 68
0= 68
0> 68
0>= 68
0x-prefix for hexadecimal numbers 54

1

1+ 60
1- 61
1/f 70

2

2! 82
2* 67
2, ... 77
2/ 67
2>r 74
2@ 82
2Constant 121
2drop 73
2dup 73
2field: 143
2lit, ... 161
2Literal 161
2nip 73
2over 73
2r> 74
2r@ 74
2rdrop 74
2rot 73
2swap 73
2tuck 73
2Value 121
2value: 145
2Variable 120

A

a_, stack item type ... 59
A, ... 77
abi-code ... 285
abort ... 118
ABORT" ... 118
ABORT", exception abort sequence ... 311
abs ... 61
absolute-file? ... 201
abstract class ... 230, 240
accept ... 214
ACCEPT, display after end of input ... 310
ACCEPT, editing ... 310
AConstant ... 121
action-of ... 139
action-of *name* semantics, changing them ... 132
add-cflags ... 282
add-framework ... 282
add-incdir ... 282
add-ldflags ... 282
add-lib ... 282
add-libpath ... 282
addr ... 122
addr *name* semantics, changing them ... 132
address alignment exception ... 315
address alignment exception, stack overflow ... 313
address arithmetic words ... 84
address unit ... 84
address unit, size in bits ... 311
ADDRESS-UNIT-BITS ... 192
addressable: ... 122
adjust-buffer ... 81
after-locate ... 259
AGAIN ... 112
AHEAD ... 112
Alias ... 141
aliases ... 141
align ... 77
aligned ... 85
aligned addresses ... 310
alignment faults ... 315
alignment of addresses for types ... 84
alignment tutorial ... 30
ALiteral ... 161
allocate ... 80
allot ... 76
also ... 183
also, too many word lists in search order ... 323
also-path ... 201
ambiguous conditions, block words ... 316
ambiguous conditions, core words ... 312
ambiguous conditions, double words ... 317
ambiguous conditions, facility words ... 318
ambiguous conditions, file words ... 319
ambiguous conditions, floating-point words ... 320
ambiguous conditions, locals words ... 321
ambiguous conditions, programming-tools words ... 322
ambiguous conditions, search-order words ... 323
and ... 66
angles in trigonometric operations ... 71
annotate-cov ... 269
ans-report.fs ... 270
append ... 95
arg ... 216
argc ... 216
argument input source different than current input source for RESTORE-INPUT ... 314
argument type mismatch ... 313
argument type mismatch, RESTORE-INPUT ... 314
arguments, OS command line ... 216
argv ... 216
arithmetic words ... 60
arithmetics tutorial ... 15
array, iterating over ... 106
array>mem ... 108
arrays ... 119
arrays tutorial ... 38
arshift ... 66
ASCII and UTF-8 ... 88
asptr ... 242
assembler ... 285
ASSEMBLER, search order capability ... 322
assert(... 267
assert-level ... 267
assert0(... 266
assert1(... 267
assert2(... 267
assert3(... 267
assertions ... 266
assignment conversion ... 251
ASSUME-LIVE ... 222
at-deltaxy ... 209
at-xy ... 209
at&t-syntax ... 289
AT-XY can't be performed on user output device ... 318
atomic operations ... 274
atomic!@ ... 274
atomic+!@ ... 274
atomic?!@ ... 275
Attempt to use zero-length string as a name ... 314
au (address unit) ... 84
AUser ... 273
authors ... 10
authors of Gforth ... 361
auto-indentation of Forth code in Emacs ... 328
AValue ... 121
AVariable ... 120

B

b ... 259
back> ... 150
backtrace ... 307
backtrace examination ... 260
backtraces with gforth-fast ... 308
barrier ... 275
base ... 187
base is not decimal (REPRESENT, F., FE., FS.) ... 320
base-execute ... 187
basename ... 199
basic objects usage ... 230
batch processing with Gforth ... 9
before-line ... 181
before-locate ... 259
before-word ... 181
begin-structure ... 142
BEGIN ... 112
benchmarking Forth systems ... 347
big-endian ... 82
bin ... 197
bind ... 236
bind usage ... 231
bind' ... 237
bitwise operation words ... 66
bl ... 93
blank ... 87
blk ... 168
BLK, altering BLK ... 317
block ... 204
block buffers ... 202
block number invalid ... 316
block read not possible ... 316
block transfer, I/O exception ... 316
block words, ambiguous conditions ... 316
block words, implementation-defined options ... 316
block words, other system documentation ... 317
block words, system documentation ... 316
block-included ... 205
block-offset ... 204
blocks ... 202
blocks file ... 202
blocks files, use with Emacs ... 329
blocks in files ... 319
blocks.fb ... 202
body-relative address input format ... 56
Boolean flags ... 60
bootmessage ... 336
bounds ... 107
break" ... 268
break: ... 268
broken-pipe-error ... 215
browse ... 259
bt ... 261
buffer ... 204
buffer% ... 81
buffer: ... 120
bug reporting ... 360
bw ... 260
bw-cover ... 269
bye ... 9
bye during gforthmi ... 333
byte order ... 82

C

c! ... 81
c$+! ... 96
c++-library ... 281
c++-library-name ... 281
c, ... 76
c, stack item type ... 58
c-callback ... 282
c-callback-thread ... 282
c-function ... 280
c-funptr ... 280
c-library ... 281
c-library-name ... 281
c-value ... 280
c-variable ... 280
c>s ... 84
c? ... 255
c_, stack item type ... 59
c@ ... 81
C function pointers to Forth words ... 282
C function pointers, calling from Forth ... 280
C functions, calls to ... 277
C functions, declarations ... 278
C interface ... 276
C" ... 102
C: ... 219
C^ ... 219
call-c ... 284
Callback functions written in Forth ... 282
callback# ... 283
calling a definition ... 113
calling C functions ... 277
capscompare ... 95
capssearch ... 95
capsstring-prefix? ... 95
case ... 111
case as generalized control structure ... 110
case sensitivity ... 59
case-sensitivity characteristics ... 312
case-sensitivity for name lookup ... 310
CASE control structure ... 103
catch ... 116
catch and backtraces ... 307
catch and this ... 235
catch in m: ... ;m ... 232
catch-nobt ... 116
cell ... 85
cell size ... 312
cell% ... 149
cell+ ... 85
cell- ... 85

cell-aligned addresses ... 310
cell/ ... 85
cells ... 85
CFA ... 303
cfield: ... 143
changing the compilation word list (during compilation) ... 323
char ... 92
char size ... 312
char% ... 149
char+ ... 85
char- ... 85
character editing of ACCEPT and EXPECT ... 310
character encoding ... 88
character literals ... 91
character set ... 310
character strings - displaying ... 209
character strings - moving and copying ... 87
character strings - representations ... 88
character-aligned address requirements ... 310
character-set extensions and matching of names ... 310
Characters - chars=bytes, extended characters, user-perceived characters ... 88
characters - displaying ... 209
characters tutorial ... 29
charclass ... 254
chars ... 85
child class ... 229
child words ... 127
cilk-bye ... 276
cilk-init ... 276
cilk-sync ... 276
class ... 228, 237, 243
class binding ... 231
class binding as optimization ... 232
class binding, alternative to ... 231
class binding, implementation ... 235
class declaration ... 242
class definition, restrictions ... 230, 240
class implementation and representation ... 235
class scoping implementation ... 235
class usage ... 230, 240
class->map ... 237
class-inst-size ... 237
class-inst-size discussion ... 231
class-override! ... 237
class-previous ... 237
class; ... 242
class; usage ... 240
class>order ... 237
classes and scoping ... 233
clear screen ... 210
clear-path ... 201
clearstack ... 264
clearstacks ... 264
clock tick duration ... 318
close-dir ... 200
close-file ... 197
close-pipe ... 215
closure conversion ... 251
closures ... 248
cmove ... 87
cmove> ... 87
code ... 286
code address ... 303
code coverage ... 268
code field ... 303
code space ... 75
code words ... 285
code-address! ... 303
code-copying native-code compilation ... 339
CODE ending sequence ... 322
CODE, processing input ... 322
colon definitions ... 123, 124
colon definitions, nesting ... 125
colon definitions, tutorial ... 18
colon-sys, passing data across : ... 162
color output to the terminal ... 210
color themes ... 211
color-cover ... 269
combined word ... 154
command line arguments, OS ... 216
command-line editing ... 10
command-line options ... 4
comment editing commands ... 327
comments ... 59
comments tutorial ... 17
common-list ... 227
comp-i.fs ... 331
comp.lang.forth ... 362
COMP' ... 160
compare ... 93
compare and swap ... 275
comparison of object models ... 246
comparison tutorial ... 23
compilation and interpretation semantics, arbitrary combination ... 154
compilation semantics ... 50
compilation semantics tutorial ... 33
compilation semantics, default ... 152
compilation semantics, usage ... 151
compilation token ... 160
compilation tokens, tutorial ... 41
compilation word list ... 182
compilation word list, change before definition ends ... 323
compile state ... 166
compile, ... 165
compile-color ... 210
compile-only ... 153
compile-only warning, for ' etc. ... 313
compile-only words ... 153
compile-only? ... 159
compiled code examination ... 261
compiling ... 180
compiling compilation semantics ... 162

compiling prompt ... 257
compiling words ... 160
complex numbers, input format ... 55
compsem: ... 156
conditional compilation ... 169
conditionals, tutorial ... 22
const-does> ... 137
Constant ... 121
constant folding ... 346
constants ... 120
construct ... 237
construct discussion ... 231
context ... 185
context-sensitive help ... 327
contiguous region ... 76
contiguous regions and heap allocation ... 80
contiguous regions in dictionary allocation ... 76
contiguous regioons and sections ... 78
contof ... 111
contributors to Gforth ... 361
control characters as delimiters ... 310
control structures ... 103
control structures for selection ... 103
control structures, user-defined ... 112
control-flow stack ... 112
control-flow stack items, locals information ... 226
control-flow stack underflow ... 322
control-flow stack, format ... 310
convert ... 188
core words, ambiguous conditions ... 312
core words, implementation-defined options ... 310
core words, other system documentation ... 316
core words, system documentation ... 310
cores ... 276
CORE ... 193
CORE-EXT ... 193
count ... 102
counted loops ... 104
counted loops with negative increment ... 106
counted string, maximum size ... 311
counted strings ... 88
cov% ... 269
cov+ ... 269
cputime ... 306
cr ... 208
Create ... 119
create-file ... 197
create-from ... 136
create...does> tutorial ... 37
CREATE ... DOES> ... 126
CREATE ... DOES>, applications ... 128
CREATE ... DOES>, details ... 129
CREATE ... SET-DOES> ... 127
CREATE and alignment ... 84
creating objects ... 231
critical-section ... 274
cross-compiler ... 333, 355
cross.fs ... 333, 355
cs-vocabulary ... 183
cs-wordlist ... 183
CS-DROP ... 112
CS-PICK ... 112
CS-PICK, fewer than u+1 items on the control flow-stack ... 322
CS-ROLL ... 112
CS-ROLL, fewer than u+1 items on the control flow-stack ... 322
cstring>sstring ... 95
csv-quote ... 215
csv-separator ... 215
ct (compilation token) ... 160
CT, tutorial ... 41
ctz ... 67
current ... 185
current' ... 237
current-interface ... 237
current-interface discussion ... 235
currying ... 128
cursor positioning ... 209
cvalue: ... 144

D

d+ ... 61
d, stack item type ... 59
d- ... 61
d. ... 206
d.r ... 206
d< ... 68
d<= ... 68
d<> ... 68
d= ... 68
d> ... 68
d>= ... 68
d>f ... 69
d>s ... 61
d0< ... 68
d0<= ... 68
d0<> ... 68
d0= ... 68
d0> ... 69
d0>= ... 69
d2* ... 67
d2/ ... 67
D: ... 219
D>F, d cannot be presented precisely as a float ... 321
D>S, d out of range of n ... 317
D^ ... 219
dabs ... 61
dark-mode ... 211
darshift ... 67
data space ... 75
data space - reserving some ... 76
data space available ... 316
data space containing definitions gets de-allocated ... 315

data space pointer not properly aligned, ,, C, . 315
data space read/write with incorrect alignment . 315
data stack . . . 72
data stack manipulation words . . . 72
data structure locals . . . 218
data-space, read-only regions . . . 311
dbg . . . 268
debug tracer editing commands . . . 327
debug-fid . . . 265
debugging . . . 265
debugging output, finding the source location in Emacs . . . 327
debugging Singlestep . . . 267
dec. . . . 205
dec.r . . . 206
decimal . . . 187
declaring C functions . . . 278
decompilation tutorial . . . 18
default . . . 101
default type of locals . . . 218
default-color . . . 210
default-input . . . 211
defer . . . 242
Defer . . . 138
defer! . . . 140
defer: . . . 146
defer@ . . . 140
deferred words . . . 138
defers . . . 139
definer . . . 304
definer! . . . 304
defines . . . 243
defining defining words . . . 125
defining words . . . 119
defining words tutorial . . . 37
defining words without name . . . 124
defining words, name given in a string . . . 125
defining words, simple . . . 119
defining words, user-defined . . . 125
definition . . . 44
definitions . . . 183
definitions, tutorial . . . 18
delete . . . 94
delete-file . . . 197
delta-i . . . 109
depth . . . 264
depth changes during interpretation . . . 271
depth-changes.fs . . . 271
deque . . . 149
design of stack effects, tutorial . . . 21
dest, control-flow stack item . . . 112
df! . . . 82
df_, stack item type . . . 59
df@ . . . 82
df@ or df! used with an address that is not double-float aligned . . . 320
dfalign . . . 77
dfaligned . . . 86
dffield: . . . 143
dfloat% . . . 149
dfloat+ . . . 86
dfloat/ . . . 86
dfloats . . . 85
dfvalue: . . . 145
dict-new . . . 237
dict-new discussion . . . 231
dictionary allocation direction . . . 76
dictionary in persistent form . . . 330
dictionary memory . . . 75
dictionary overflow . . . 313
dictionary size default . . . 334
digits > 35 . . . 310
Directories . . . 199
dirname . . . 199
disasm . . . 289
disasm-gdb . . . 289
disasm2 . . . 289
disassembler, general . . . 288
discode . . . 288
dividing by zero . . . 313
dividing by zero, floating-point . . . 321
Dividing classes . . . 233
dividing integers . . . 62
dividing many integers with the same divisor . . . 64
Division by zero . . . 62, 64
division rounding . . . 312
division with potentially negative operands . . . 60
dlshift . . . 66
dmax . . . 61
dmin . . . 61
dnegate . . . 61
DO loops . . . 104
doabicode: . . . 304
docol: . . . 303
docon: . . . 303
dodefer: . . . 304
dodoes: . . . 304
does-code! . . . 304
does> tutorial . . . 37
does>-code . . . 304
DOES> . . . 129
DOES> in a separate definition . . . 129
DOES> in interpretation state . . . 129
DOES> of non-CREATEd words . . . 315
DOES>, visibility of current definition . . . 312
DOES>-parts, stack effect . . . 127
dofield: . . . 304
DONE . . . 109
double precision arithmetic words . . . 61
double words, ambiguous conditions . . . 317
double words, system documentation . . . 317
double% . . . 149
double-cell numbers, input format . . . 54
double-ended queue . . . 149
doubly indirect threaded code . . . 333
douser: . . . 304

dovalue: ... 304
dovar: ... 304
DO ... 108
dpl ... 188
drol ... 67
drop ... 72
dror ... 68
drshift ... 66
du/mod ... 63
du< ... 69
du<= ... 69
du> ... 69
du>= ... 69
dump ... 264
dup ... 72
duration of a system clock tick ... 318
dynamic allocation of memory ... 80
dynamic superinstructions with replication ... 339
Dynamically linked libraries in C interface ... 281

E

e. ... 206
e.exact ... 206
e.p ... 206
early ... 242
early binding ... 231
edit ... 259
edit-line ... 214
editing in ACCEPT and EXPECT ... 310
ekey ... 212
ekey>char ... 212
ekey>fkey ... 212
ekey>xchar ... 212
ekey? ... 212
EKEY, encoding of keyboard events ... 317
elements of a Forth system ... 52
ELSE ... 112
Emacs and Gforth ... 327
emit ... 209
emit-file ... 198
EMIT and non-graphic characters ... 310
empty-buffers ... 204
end-c-library ... 281
end-class ... 237, 243
end-class usage ... 230
end-class-noname ... 237
end-code ... 286
end-interface ... 237
end-interface usage ... 234
end-interface-noname ... 238
end-methods ... 238
end-struct ... 149
end-struct usage ... 148
end-structure ... 142
endcase ... 111
ENDIF ... 113
endless loop ... 104
endof ... 111
endscope ... 220
endtry ... 117
endtry-iferror ... 118
engine ... 349
engine performance ... 347
engine portability ... 349
environment ... 195
environment variable input format ... 56
environment variables ... 11, 333
environment wordset ... 58
environment-wordlist ... 195
environment? ... 192
ENVIRONMENT? string length, maximum ... 311
environmental queries ... 192
environmental restrictions ... 309
equality of floats ... 72
erase ... 87
error messages ... 307
error output, finding the source location in Emacs ... 327
error-color ... 210
error-hl-inv ... 210
error-hl-ul ... 210
etags.fs ... 327
evaluate ... 168
event-loop ... 275
examining data ... 263
exception ... 115
exception abort sequence of ABORT" ... 311
exception source code ... 260
exception when including source ... 318
exception words, implementation-defined options ... 317
exception words, system documentation ... 317
exceptions ... 114, 115
exceptions tutorial ... 36
executable image file ... 334
execute ... 157
execute-exit ... 157
execute-parsing ... 182
execute-parsing-file ... 182
execute-task ... 272
executing code on startup ... 9
execution frequency ... 268
execution semantics (aka interpretation semantics) ... 150
execution token ... 44, 156
execution token input format ... 56
execution token of last defined word ... 124
execution token of words with undefined execution semantics ... 313
execution tokens tutorial ... 35
exercises ... 53
exit in m: ... ;m ... 232
exitm ... 238
exitm discussion ... 232
EXIT ... 114

`expand-where` ... 260
`expect` ... 214
`EXPECT`, display after end of input ... 310
`EXPECT`, editing ... 310
exponent too big for conversion (`DF!`, `DF@`, `SF!`, `SF@`) ... 321
`extend-mem` ... 80
`extend-structure` ... 147
extended records ... 146
`extra-section` ... 79

F

`f!` ... 82
`f!` used with an address that is not float aligned ... 320
`f*` ... 69
`f**` ... 70
`f*+` ... 69
`f+` ... 69
`f,` ... 76
`f`, stack item type ... 58
`f-` ... 69
`f-rot` ... 73
`f.` ... 206
`f.rdp` ... 207
`f.s` ... 263
`f.s-precision` ... 263
`f/` ... 70
`f<` ... 72
`f<=` ... 72
`f<>` ... 72
`f=` ... 72
`f>` ... 72
`f>=` ... 72
`f>buf-rdp` ... 192
`f>d` ... 69
`f>l` ... 226
`f>r` ... 74
`f>s` ... 69
`f>str-rdp` ... 192
`f_`, stack item type ... 59
`f@` ... 82
`f@` used with an address that is not float aligned ... 320
`f@localn` ... 225
`f~` ... 72
`f~abs` ... 72
`f~rel` ... 72
`f0<` ... 72
`f0<=` ... 72
`f0<>` ... 72
`f0=` ... 72
`f0>` ... 72
`f0>=` ... 72
`f2*` ... 70
`f2/` ... 70
`F:` ... 220
`F>D`, integer part of float cannot be represented by *d* ... 321
`F^` ... 220
`fabs` ... 70
facility words, ambiguous conditions ... 318
facility words, implementation-defined options ... 317
facility words, system documentation ... 317
`facos` ... 71
`FACOS`, `|`*float*`|>1` ... 321
`facosh` ... 71
`FACOSH`, *float*`<1` ... 321
factoring ... 43
factoring similar colon definitions ... 128
factoring tutorial ... 20
`falign` ... 77
`faligned` ... 85
`falog` ... 70
`false` ... 60
fam (file access method) ... 197
`fancy-after-l` ... 258
`fasin` ... 71
`FASIN`, `|`*float*`|>1` ... 321
`fasinh` ... 71
`FASINH`, *float*`<0` ... 321
`fast-throw` ... 114
`fatan` ... 71
`fatan2` ... 71
`FATAN2`, both arguments are equal to zero ... 320
`fatanh` ... 71
`FATANH`, `|`*float*`|>1` ... 321
`faxpy` ... 71
`fclearstack` ... 264
`fconstant` ... 121
`fcopysign` ... 70
`fcos` ... 71
`fcosh` ... 71
`fdepth` ... 264
FDL, GNU Free Documentation License ... 363
`fdrop` ... 73
`fdup` ... 73
`fe.` ... 206
fetch and add ... 274
`fexp` ... 70
`fexpm1` ... 70
`ffield:` ... 143
`ffourth` ... 73
`field` ... 149
`field` usage ... 148
`field` usage in class definition ... 230
`field:` ... 143
file access methods used ... 318
file exceptions ... 318
file input nesting, maximum depth ... 319
file line terminator ... 318
file name format ... 318
file search path ... 200
file words, ambiguous conditions ... 319
file words, implementation-defined options ... 318
file words, system documentation ... 318
`file-eof?` ... 198

file-handling ... 197
file-position ... 198
file-size ... 198
file-status ... 198
file>fpath ... 201
file>path ... 201
FILE-STATUS, returned information ... 318
filename-match ... 200
filenames in ~~ output ... 265
filenames in assertion output ... 267
files ... 196
files containing blocks ... 319
files containing Forth code, tutorial ... 17
files tutorial ... 32
fill ... 87
find ... 184
find-name ... 158
find-name-in ... 158
first definition ... 48
first field optimization ... 143
fkey ... 214
flags on the command line ... 4
flags tutorial ... 23
flat address space ... 76
flat closures ... 248
flavours of locals ... 217
flit, ... 162
FLiteral ... 161
fln ... 70
FLN, *float*<=0 ... 321
flnp1 ... 70
FLNP1, *float*<=-1 ... 321
float ... 85
float% ... 149
float+ ... 85
float/ ... 85
floating point arithmetic words ... 69
floating point numbers, format and range ... 320
floating point tutorial ... 31
floating point unidentified fault, integer division ... 313
floating-point arithmetic, pitfalls ... 69
floating-point comparisons ... 72
floating-point constants ... 71
floating-point dividing by zero ... 321
floating-point numbers, input format ... 54
floating-point numbers, rounding or truncation ... 320
floating-point output ... 206
floating-point result out of range ... 320
floating-point stack ... 72
floating-point stack in the standard ... 72
floating-point stack manipulation words ... 73
floating-point stack size ... 320
floating-point stack width ... 320
Floating-point unidentified fault ... 62
Floating-point unidentified fault (on integer division) ... 64
floating-point unidentified fault, F>D ... 321
floating-point unidentified fault, FACOS, FASIN or FATANH ... 321
floating-point unidentified fault, FACOSH ... 321
floating-point unidentified fault, FASINH or FSQRT ... 321
floating-point unidentified fault, FLN or FLOG ... 321
floating-point unidentified fault, FLNP1 ... 321
floating-point unidentified fault, FP divide-by-zero ... 321
floating-point words, ambiguous conditions ... 320
floating-point words, implementation-defined options ... 320
floating-point words, system documentation ... 319
floating-stack ... 193
floats ... 85
flog ... 70
FLOG, *float*<=0 ... 321
floor ... 70
floored division ... 62
FLOORED ... 193
flush ... 204
flush-file ... 198
flush-icache ... 286
fm/mod ... 63
fmax ... 70
fmin ... 70
fnegate ... 70
fnip ... 73
FOR ... 108
FOR loops ... 107
foreign language interface ... 276
FORGET, deleting the compilation word list ... 322
FORGET, *name* can't be found ... 322
FORGET, removing a needed definition ... 323
forgeting words ... 264
FORK ... 257
form ... 210
format and range of floating point numbers ... 320
format of glossary entries ... 57
formatted numeric output ... 189
Forth ... 184
Forth - an introduction ... 43
Forth mode in Emacs ... 327
Forth source files ... 196
Forth Tutorial ... 14
Forth-related information ... 362
forth-wordlist ... 183
forth.el ... 327
forward ... 114
fourth ... 72
fover ... 73
fp! ... 75
fp ... 206
fp@ ... 75
fp0 ... 75
FP output ... 206
FP tutorial ... 31
fpath ... 201

fpick ... 73
fr> ... 74
fr@ ... 74
free ... 80
free-closure ... 249
free-mem-var ... 80
frequently asked questions ... 362
frot ... 73
fround ... 70
fs. ... 206
fsin ... 71
fsincos ... 71
fsinh ... 71
fsqrt ... 70
FSQRT, *float*<0 ... 321
fswap ... 73
ftan ... 71
FTAN on an argument *r1* where cos(*r1*) is zero ... 320
ftanh ... 71
fthird ... 73
ftrunc ... 70
ftuck ... 73
fully relocatable image files ... 331
functions, tutorial ... 18
fvalue ... 122
fvalue: ... 145
fvariable ... 120

G

g ... 259
gdb disassembler ... 288
general control structures (case) ... 110
general files ... 197
get-block-fid ... 204
get-current ... 183
get-dir ... 200
get-order ... 183
get-recs ... 176
get-stack ... 150
getenv ... 305
gforth ... 194
Gforth - leaving ... 9
Gforth environment ... 4
Gforth extensions ... 324
Gforth files ... 11
Gforth locals ... 217
Gforth performance ... 347
Gforth stability ... 2
gforth-ditc ... 333
gforth-fast and backtraces ... 308
gforth-fast, difference from gforth ... 308
gforth.el ... 327
gforth.el, installation ... 327
gforth.fi, relocatability ... 331
GFORTH – environment variable ... 11, 333
GFORTHD – environment variable ... 11, 333
GFORTHHIST – environment variable ... 11
gforthmi ... 331
GFORTHPATH – environment variable ... 11
GFORTHSYSTEMPREFIX – environment variable ... 11
gg ... 260
giving a name to a library interface ... 280
glossary notation format ... 57
GNU C for the engine ... 349
goals of the Gforth project ... 2

H

h. ... 205
halt ... 273
header fields ... 299
header methods ... 300
header space ... 182
heap allocation ... 80
heap memory ... 75
heap-new ... 238
heap-new discussion ... 231
heap-new usage ... 230
help ... 9
here ... 76
hex ... 187
hex. ... 205
highlighting Forth code in Emacs ... 328
hilighting Forth code in Emacs ... 328
history file ... 10
hold ... 190
holds ... 190
hooks in the text interpreter ... 181
how: ... 242

I

i ... 108
i' ... 108
I/O - blocks ... 202
I/O - file-handling ... 196
I/O - keyboard and display ... 205
I/O exception in block transfer ... 316
id. ... 159
IDE (integrated development environment) ... 258
if, tutorial ... 22
IF control structure ... 103
iferror ... 117
IF ... 112
image file ... 330
image file background ... 330
image file initialization sequence ... 335
image file invocation ... 334
image file loader ... 330
image file, executable ... 334
image file, fully relocatable ... 331
image file, non-relocatable ... 333
image file, stack and dictionary sizes ... 334
image file, turnkey applications ... 336
image license ... 330

`immediate` ... 152
immediate words ... 50, 152
immediate, tutorial ... 33
`immediate?` ... 302
`implementation` ... 238
implementation of locals ... 225
`implementation` usage ... 234
implementation-defined options, block words ... 316
implementation-defined options, core words ... 310
implementation-defined options, exception words ... 317
implementation-defined options, facility words ... 317
implementation-defined options, file words ... 318
implementation-defined options, floating-point words ... 320
implementation-defined options, locals words ... 321
implementation-defined options, memory-allocation words ... 322
implementation-defined options, programming-tools words ... 322
implementation-defined options, search-order words ... 323
`in` ... 183
`in-colon-def?` ... 166
`in-wordlist` ... 183
`include` ... 196
`include` search path ... 200
`include`, placement in files ... 327
`include-file` ... 196
`INCLUDE-FILE`, *file-id* is invalid ... 319
`INCLUDE-FILE`, I/O exception reading or closing *file-id* ... 319
`included` ... 196
`included?` ... 196
`INCLUDED`, I/O exception reading or closing *file-id* ... 319
`INCLUDED`, named file cannot be opened ... 319
including files ... 196
including files, stack effect ... 196
indentation of Forth code in Emacs ... 328
`inf` ... 71
`infile-execute` ... 199
`infile-id` ... 199
`infinity` ... 71
`info-color` ... 210
inheritance ... 229
`init-asm` ... 285
`init-buffer` ... 81
`init-object` ... 238
`init-object` discussion ... 231
initialization of locals ... 218
initialization sequence of image file ... 335
`initiate` ... 272
`inline:` ... 123
inner interpreter and text interpreter ... 166
inner interpreter implementation ... 337
input format for body-relative addresses ... 56
input format for characters/code points ... 54
input format for double-cell numbers ... 54
input format for environment variables ... 56
input format for execution tokens ... 56
input format for floating-point numbers ... 54
input format for name tokens ... 56
input format for single-cell numbers ... 54
input format for strings ... 56
input from pipes ... 12
input line size, maximum ... 319
input line terminator ... 311
Input Redirection ... 199
input sources ... 168
input stream ... 181
input, single-key ... 211
input, string from terminal ... 214
`input-color` ... 210
`insert` ... 94
`inst-value` ... 238
`inst-value` usage ... 233
`inst-value` visibility ... 233
`inst-var` ... 238
`inst-var` implementation ... 235
`inst-var` usage ... 232
`inst-var` visibility ... 233
instance variables ... 228
instruction pointer updates ... 344
insufficient data stack or return stack space ... 313
insufficient space for loop control parameters ... 313
insufficient space in the dictionary ... 313
`INT-[I]` ... 170
integer conversion, `base` ... 187
integer to string conversion ... 189
integer types, ranges ... 311
integrated development environment ... 258
`intel-syntax` ... 289
`interface` ... 238
interface implementation ... 236
interface to C functions ... 276
`interface` usage ... 234
interfaces for objects ... 234
interpret state ... 166
Interpret/Compile states ... 169
`interpret/compile:` ... 154
interpretation and compilation semantics, arbitrary combination ... 154
interpretation semantics ... 50
interpretation semantics (aka execution semantics) ... 150
interpretation semantics tutorial ... 33
interpretation semantics, default ... 152
interpretation semantics, usage ... 150
interpreter - outer ... 166
interpreter directives ... 169
`interpreting` ... 180
Interpreting a compile-only word ... 313
Interpreting a compile-only word, for a local ... 321
interpreting a word with undefined interpretation semantics ... 313

intsem: ... 156
invalid block number ... 316
Invalid memory address ... 313
Invalid memory address, stack overflow ... 313
Invalid name argument, TO ... 315, 321
invalid-char ... 91
invert ... 66
invoking a selector ... 229
invoking Gforth ... 4
invoking image files ... 334
ior type description ... 59
ior values and meaning ... 318, 322
IP updates ... 344
is *name* semantics, changing them ... 132
IS ... 138
items on the stack after interpretation ... 271
iterate over array ... 106

J

j ... 108
JOIN ... 257

K

k ... 108
k-alt-mask ... 213
k-backspace ... 213
k-ctrl-mask ... 213
k-delete ... 213
k-down ... 212
k-end ... 212
k-enter ... 213
k-eof ... 213
k-f1 ... 213
k-f10 ... 213
k-f11 ... 213
k-f12 ... 213
k-f2 ... 213
k-f3 ... 213
k-f4 ... 213
k-f5 ... 213
k-f6 ... 213
k-f7 ... 213
k-f8 ... 213
k-f9 ... 213
k-home ... 212
k-insert ... 213
k-left ... 212
k-mute ... 213
k-next ... 213
k-pause ... 213
k-prior ... 212
k-right ... 212
k-sel ... 213
k-shift-mask ... 213
k-tab ... 213
k-up ... 212
k-voldown ... 213
k-volup ... 213
k-winch ... 213
kern*.fi, relocatability ... 331
key ... 211
key-file ... 198
key-ior ... 211
key? ... 211
key?-file ... 198
keyboard events, encoding in EKEY ... 317
kill ... 272
kill-task ... 272
Kuehling, David ... 327

L

l ... 259
l! ... 83
l, ... 77
l>s ... 84
l@ ... 83
L" ... 100
lalign ... 86
laligned ... 86
LANG – environment variable ... 11
last word was headerless ... 315
late binding ... 231
latest ... 158
latestnt ... 158
latestxt ... 124
lbe ... 83
LC_ALL – environment variable ... 11
LC_CTYPE – environment variable ... 11
LEAVE ... 109
leaving definitions, tutorial ... 27
leaving Gforth ... 9
leaving loops, tutorial ... 27
length of a line affected by \ ... 316
lfield: ... 143
lib-error ... 283
lib-sym ... 283
Libraries in C interface ... 281
library interface names ... 280
license ... 10
license for images ... 330
lifetime of locals ... 223
light-mode ... 211
line terminator on input ... 311
line-end-hook ... 181
lines and the text interpreter ... 167
list ... 204
list-size ... 227
LIST display format ... 316
lit, ... 161
Literal ... 161
literal stack ... 346
literal tutorial ... 40
Literals ... 160

Literals (in source code) ... 54
literals for characters and strings ... 91
little-endian ... 82
ll ... 260
lle ... 83
load ... 204
load-cov ... 269
loader for image files ... 330
loading files at startup ... 9
loading Forth code, tutorial ... 17
local in interpretation state ... 321
local variables, tutorial ... 21
locale and case-sensitivity ... 310
locale! ... 101
locale-csv ... 101
locale-csv-out ... 101
Locale: ... 101
locale@ ... 100
locales ... 100
locals ... 217
locals and POSTPONE or]] ... 223
locals and return stack ... 73
locals flavours ... 217
locals implementation ... 225
locals information on the control-flow stack ... 226
locals initialization ... 218
locals lifetime ... 223
locals programming style ... 224
locals stack ... 72, 225
locals types ... 217
locals use without TO etc. ... 224
locals visibility ... 220
locals words, ambiguous conditions ... 321
locals words, implementation-defined options ... 321
locals words, system documentation ... 321
locals, default type ... 218
locals, Gforth style ... 217
locals, maximum number in a definition ... 321
locals, Standard Forth style ... 227
locals| ... 219
locate ... 258
lock ... 274
log2 ... 67
loop control parameters not available ... 315
loops without count ... 104
loops, counted ... 104
loops, counted, tutorial ... 25
loops, endless ... 104
loops, indefinite, tutorial ... 24
LOOP ... 108
lp! ... 75
lp+! ... 225
lp+n ... 225
lp@ ... 75
lp0 ... 75
lrol ... 67
lror ... 67
lshift ... 66
LSHIFT, large shift counts ... 315
lvalue: ... 145

M

m* ... 61
m*/ ... 64
m+ ... 61
m: ... 238
m: usage ... 232
Macros ... 162
macros ... 160
macros, advanced tutorial ... 40
macros-wordlist ... 101
magenta-input ... 211
make-latest ... 137
map-vocs ... 185
mapping block ranges to files ... 319
marker ... 264
max ... 61
max-float ... 193
MAX-CHAR ... 192
MAX-D ... 193
MAX-N ... 193
MAX-U ... 193
MAX-UD ... 193
MAX-XCHAR ... 193
maxalign ... 77
maxaligned ... 86
maxdepth-.s ... 263
maximum depth of file input nesting ... 319
maximum number of locals in a definition ... 321
maximum number of word lists in search order ... 323
maximum size of a counted string ... 311
maximum size of a definition name, in characters ... 311
maximum size of a parsed string ... 311
maximum size of input line ... 319
maximum string length for ENVIRONMENT?, in characters ... 311
mem+do ... 108
mem, ... 77
mem-do ... 108
memory access words ... 81
memory access/allocation tutorial ... 28
memory alignment tutorial ... 30
memory barrier ... 275
memory block words ... 87
memory overcommit for dictionary and stacks ... 5
memory words ... 75
memory-allocation word set ... 80
memory-allocation words, implementation-defined options ... 322
memory-allocation words, system documentation ... 322
message send ... 229
meta recognizer ... 56
metacompiler ... 333, 355

method ... 228, 238, 242, 243
method conveniences ... 232
method map ... 235
method selector ... 228
method usage ... 240
methods ... 238
methods...end-methods ... 233
min ... 61
mini-oof ... 242
mini-oof example ... 243
mini-oof usage ... 243
minimum search order ... 323
miscellaneous words ... 306
mixed precision arithmetic words ... 61
mkdir-parents ... 200
mod ... 62
modf ... 62
modf-stage2m ... 65
modifying >IN ... 49
Modifying a word defined earlier ... 137
modifying the contents of the input buffer or a string literal ... 314
mods ... 62
modulus ... 62
most recent definition does not have a name (IMMEDIATE) ... 315
motivation for object-oriented programming ... 228
move ... 87
ms ... 305
MS, repeatability to be expected ... 318
Multiple exits from begin ... 109
multitasker ... 271
Must now be used inside C-LIBRARY, see C interface doc ... 285
mutex ... 274
mux ... 66
mwords ... 184

N

n ... 259
n, stack item type ... 59
n/a ... 134
n>r ... 74
name ... 181
name dictionary ... 44
name field address ... 159
name lookup, case-sensitivity ... 310
name not defined by VALUE or (LOCAL) used by TO ... 321
name not defined by VALUE used by TO ... 315
name not found ... 312
name not found (', POSTPONE, ['], [COMPILE]) ... 315
name space ... 75
name token (nt) ... 158
name, maximum length ... 311
name>compile ... 158
name>interpret ... 158
name>link ... 159
name>string ... 158
names for defined words ... 125
NaN ... 71
native-code compilation ... 339
native@ ... 100
NDCS word ... 154
needs ... 197
negate ... 61
negative increment for counted loops ... 106
Neon model ... 246
nested colon definitions ... 125
new ... 243
newline ... 93
newline character on input ... 311
newtask ... 272
newtask4 ... 272
next-arg ... 216
next-case ... 111
next-section ... 79
nextname ... 125
NEXT ... 108
NFA ... 159
nip ... 72
nocov[... 269
non-graphic characters and EMIT ... 310
non-relocatable image files ... 333
noname ... 124
noname-from ... 136
noop ... 157
nosplit? ... 94
notation of glossary entries ... 57
nothrow ... 116
nr> ... 74
ns ... 305
nt ... 261
nt (name token) ... 158
nt input format ... 56
nt token input format ... 56
nt, stack item type ... 59
ntime ... 306
number conversion - traps for the unwary ... 188
number conversion, base ... 187
number of bits in one address unit ... 311
number representation and arithmetic ... 311
numeric comparison words ... 68
numeric output - formatted ... 189
numeric output - simple/free-format ... 205
numeric output, FP ... 206
nw ... 260

O

o> ... 246
object ... 228, 238, 243
object allocation options ... 231
object class ... 231
object creation ... 231
object interfaces ... 234
object models, comparison ... 246
object-' ... 242
object-: ... 241
object-:: ... 241
object-[] ... 241
object-asptr ... 241
object-bind ... 241
object-bound ... 241
object-class ... 241
object-class? ... 241
object-definitions ... 241
object-dispose ... 241
object-endwith ... 242
object-init ... 241
object-is ... 241
object-link ... 241
object-map discussion ... 235
object-new ... 241
object-new[] ... 241
object-oriented programming ... 229, 239
object-oriented programming motivation ... 228
object-oriented programming style ... 231
object-oriented terminology ... 228
object-postpone ... 242
object-ptr ... 241
object-self ... 241
object-super ... 241
object-with ... 242
objects ... 229
objects, basic usage ... 230
objects.fs ... 229, 239
objects.fs Glossary ... 236
objects.fs implementation ... 235
objects.fs properties ... 229
obsolete? ... 159
of ... 111
off ... 60
ok prompt ... 257
on ... 60
once ... 265
Only ... 184
oof ... 239
oof.fs ... 229, 239
oof.fs base class ... 241
oof.fs properties ... 239
oof.fs usage ... 240
oof.fs, differences to other models ... 247
open-blocks ... 204
open-dir ... 200
open-file ... 197
open-lib ... 283
open-path-file ... 201
open-pipe ... 215
operating system - passing commands ... 305
operator's terminal facilities available ... 316
opt: ... 134
optimizations ... 337
optimizations in different engines ... 346
options on the command line ... 4
or ... 66
order ... 184
orig, control-flow stack item ... 112
os-class ... 194
os-type ... 194
OS command line arguments ... 216
other system documentation, block words ... 317
other system documentation, core words ... 316
out ... 208
outer interpreter ... 43, 45, 166
outfile-execute ... 199
outfile-id ... 199
output in pipes ... 12
Output Redirection ... 199
output to terminal ... 209
over ... 72
overcommit memory for dictionary and stacks ... 5
overflow of the pictured numeric output string ... 314
overrides ... 238
overrides usage ... 230

P

pad ... 87
PAD size ... 312
PAD use by nonstandard words ... 316
page ... 210
parameter stack ... 72
parameters are not of the same type (DO, ?DO, WITHIN) ... 315
parent class ... 229
parent class binding ... 231
parse ... 181
parse-name ... 181
parse-word ... 181
parsed string overflow ... 314
parsed string, maximum size ... 311
parsing words ... 49
path for included ... 200
path+ ... 202
path= ... 202
pause ... 273
pedigree of Gforth ... 361
perform ... 157
performance of some Forth interpreters ... 347
persistent form of dictionary ... 330
pi ... 71
pick ... 73
pictured numeric output ... 189
pictured numeric output buffer, size ... 312

pictured numeric output string, overflow 314
pipes, creating your own 215
pipes, Gforth as part of 12
place 102
postpone 162
postpone tutorial 39
postpone, ... 160
postpone-color 210
POSTPONE applied to [IF] 322
POSTPONE or [COMPILE] applied to TO 315
postponing 180
postponing prompt 257
pow2? 67
precision 207
precompiled Forth code 330
Preface 1
prefix ` 156
prepend-where 260
preserve 140
previous 183
previous, search order empty 323
previous-section 79
primitives 350
print 239
printdebugdata 265
private discussion 233
procedures, tutorial 18
process-option 336
program 100
program data space available 316
programming style, locals 224
programming style, object-oriented 231
programming tools 257
programming-tools words, ambiguous conditions 322
programming-tools words, implementation-defined options 322
programming-tools words, system documentation 322
prompt 257, 312
pronunciation of words 57
protected 239
protected discussion 233
pthread 271
ptr 242
public 239

Q

query 168
quit 306
quotations 125

R

r'@ 74
r, stack item type 59
r/o 197
r/w 197
r> 74
r@ 74
random 69
Random numbers 69
ranges for integer types 311
rdrop 74
read-csv 215
read-dir 200
read-file 197
read-line 198
read-only data space regions 311
reading from file positions not yet written 319
rec-body 175
rec-complex 174
rec-dtick 175
rec-env 175
rec-filter 178
rec-float 174
rec-forth 175
rec-forth-nt? 178
rec-local 174
rec-meta 175
rec-moof2 246
rec-name 173
rec-none 176
rec-number 174
rec-scope 174
rec-sequence: 175
rec-string 174
rec-tick 175
rec-to 174
receiving object 229
reciprocal of integer 64
recognizer, stack effect 176
Recognizers normal usage 171
recongizers 171
records 141
records tutorial 38
recover (old Gforth versions) 117
recs 173
recurse 113
RECURSE appears after DOES> 314
recursion tutorial 26
recursive 113
recursive definitions 113
Redirection 199
refill 182
regexps 254
relocating loader 330
remainder 62
rename-file 197
repeatability to be expected from the execution of MS 318

REPEAT ... 113
replace-word ... 266
replacer: ... 102
replaces ... 101
replication ... 339
report the words used in your program ... 270
reposition-file ... 198
REPOSITION-FILE, outside the file's boundaries ... 319
represent ... 192
REPRESENT, results when *float* is out of range ... 320
require ... 196
require, placement in files ... 327
required ... 196
reserving data space ... 76
resize ... 80
resize-file ... 198
restart ... 273
restore ... 118
restore-input ... 168
RESTORE-INPUT, Argument type mismatch ... 314
restrict ... 154
result out of range ... 314
Result out of range ... 62
Result out of range (on integer division) ... 64
return stack ... 72
return stack and locals ... 73
return stack dump with gforth-fast ... 308
return stack manipulation words ... 73
return stack space available ... 316
return stack tutorial ... 27
return stack underflow ... 314
return-stack-cells ... 193
returning from a definition ... 113
reveal ... 136
reveal! ... 136
rnd ... 69
rol ... 67
roll ... 73
Root ... 185
ror ... 67
rot ... 73
rounding of floating-point numbers ... 320
rp! ... 75
rp@ ... 75
rp0 ... 75
rpick ... 74
rshift ... 66
RSHIFT, large shift counts ... 315
run-time code generation, tutorial ... 40
run-time semantics ... 152
running Gforth ... 4
running image files ... 334
Rydqvist, Goran ... 327

S

s+ ... 95
s// ... 257
s>> ... 257
s>d ... 61
s>f ... 69
s\" ... 92
S" ... 92
S", number of string buffers ... 319
S", size of string buffer ... 319
safe/string ... 94
save-buffers ... 204
save-cov ... 269
save-input ... 168
save-mem ... 80
save-mem-dict ... 77
savesystem ... 334
savesystem during gforthmi ... 333
scan ... 94
scan-back ... 94
scan-translate-string ... 178
scope ... 220
scope of locals ... 220
scoping and classes ... 233
scr ... 204
scripting with Gforth ... 9
scvalue: ... 145
seal ... 185
search ... 94
search order stack ... 182
search order, maximum depth ... 323
search order, minimum ... 323
search order, tutorial ... 41
search path control, source files ... 201
search path for files ... 200
search-order words, ambiguous conditions ... 323
search-order words, implementation-defined options ... 323
search-order words, system documentation ... 323
search-wordlist ... 184
sections and contiguous regions ... 78
see ... 261
see tutorial ... 18
see-code ... 261
see-code-range ... 262
SEE, source and format of output ... 322
seed! ... 69
select ... 60
selection control structures ... 103
selector ... 228, 239
selector implementation, class ... 235
selector invocation ... 229
selector invocation, restrictions ... 230, 240
selector usage ... 230
selectors and stack effects ... 231
selectors common to hardly-related classes ... 234
semantics tutorial ... 33

semantics, arbitrary combination of interpretation and compilation ... 154
semantics, changing the to/+to/action-of/is/addr *name* semantics ... 132
semantics, interpretation and compilation ... 150
semantics, interpretation/execution ... 150
semantics, run-time ... 152
send-event ... 275
set->comp ... 302
set->int ... 302
set-compsem ... 156
set-current ... 183
set-dir ... 200
set-does> ... 129
set-execute ... 301
set-name>link ... 303
set-name>string ... 303
set-optimizer ... 134
set-order ... 183
set-precision ... 207
set-recs ... 176
set-stack ... 150
set-to ... 134
sf! ... 82
sf_, stack item type ... 59
sf@ ... 82
sf@ or sf! used with an address that is not single-float aligned ... 320
sfalign ... 77
sfaligned ... 85
sffield: ... 143
sfloat% ... 149
sfloat+ ... 85
sfloat/ ... 85
sfloats ... 85
sfvalue: ... 145
sh ... 305
sh-get ... 305
Shared libraries in C interface ... 281
shell commands ... 305
shift-args ... 216
short-where ... 260
sign ... 190
sign extension ... 82
silent exiting from Gforth ... 12
simple defining words ... 119
simple loops ... 104
simple-fkey-string ... 214
simple-see ... 261
simple-see-range ... 261
single precision arithmetic words ... 60
single-assignment style for locals ... 224
single-cell numbers, input format ... 54
single-key input ... 211
singlestep Debugger ... 267
size of buffer at WORD ... 311
size of the dictionary and the stacks ... 4
size of the keyboard terminal buffer ... 312
size of the pictured numeric output buffer ... 312
size of the scratch area returned by PAD ... 312
size parameters for command-line options ... 4
skip ... 94
sleep ... 273
slit, ... 162
SLiteral ... 162
slurp-fid ... 198
slurp-file ... 198
slvalue: ... 145
sm/rem ... 63
source ... 167
source code for exception ... 260
source code of a word ... 258
source location of error or debugging output in Emacs ... 327
source-id ... 168
SOURCE-ID, behaviour when BLK is non-zero ... 319
sourcefilename ... 197
sourceline# ... 197
sp! ... 75
sp@ ... 75
sp0 ... 75
space ... 208
space delimiters ... 310
spaces ... 208
span ... 214
spawn ... 276
spawn1 ... 276
spawn2 ... 276
speed, startup ... 12
stability of Gforth ... 2
stability of words ... 58
stack ... 149
stack caching ... 343
stack depth changes during interpretation ... 271
stack effect ... 57
Stack effect design, tutorial ... 21
stack effect of DOES>-parts ... 127
stack effect of included files ... 196
stack effects of selectors ... 231
stack empty ... 314
stack item types ... 58
stack manipulation tutorial ... 16
stack manipulation words ... 72
stack manipulation words, floating-point stack ... 73
stack manipulation words, return stack ... 73
stack manipulations words, data stack ... 72
stack overflow ... 313
stack pointer manipulation words ... 75
stack representation ... 343
stack size default ... 334
stack size, cache-friendly ... 334
stack space available ... 316
stack state ... 343
stack tutorial ... 15
stack underflow ... 314
stack, user-defined ... 149

stack-cells ... 193
stack-effect comments, tutorial ... 18
stack: ... 149
stack> ... 149
stacksize ... 272
stacksize4 ... 272
staged/-divisor ... 66
staged/-size ... 65
Standard conformance of Gforth ... 309
starting Gforth tutorial ... 14
startup sequence for image file ... 335
Startup speed ... 12
state ... 169
state - effect on the text interpreter ... 49
state-smart words (are a bad idea) ... 155
STATE values ... 312
static ... 242
status bar ... 257
status-color ... 210
stderr ... 199
stderr and pipes ... 12
stdin ... 198
stdout ... 198
stop ... 272
stop-dns ... 272
stop-ns ... 272
str< ... 94
str= ... 93
str=? ... 256
String input format ... 56
string input from terminal ... 214
string larger than pictured numeric output area (f., fe., fs.) ... 321
string literals ... 91
string longer than a counted string returned by WORD ... 315
String to number conversion ... 188
string words with $ ... 96
string, ... 102
string-parse ... 181
string-prefix? ... 94
string-suffix? ... 94
strings - see character strings ... 88
strings tutorial ... 29
struct ... 149
struct usage ... 148
structs tutorial ... 38
structure extension ... 146
structure of Forth programs ... 51
structures ... 141
Structures in Forth200x ... 142
sub-list? ... 227
substitute ... 102
success-color ... 210
superclass binding ... 231
superinstructions ... 339
superinstructions, static ... 342
swap ... 73
swvalue: ... 145
symmetric division ... 62
Synonym ... 141
synonyms ... 141
syntax tutorial ... 14
system ... 305
system dictionary space required, in address units ... 316
system documentation ... 309
system documentation, block words ... 316
system documentation, core words ... 310
system documentation, double words ... 317
system documentation, exception words ... 317
system documentation, facility words ... 317
system documentation, file words ... 318
system documentation, floating-point words ... 319
system documentation, locals words ... 321
system documentation, memory-allocation words ... 322
system documentation, programming-tools words ... 322
system documentation, search-order words ... 323
system prompt ... 312

T

table ... 183
TAGS file ... 327
target compiler ... 333, 355
task ... 272
task-local data ... 273
terminal buffer, size ... 312
terminal output ... 209
terminal size ... 210
terminology for object-oriented programming ... 228
text interpreter ... 43, 45, 166
text interpreter - effect of state ... 49
text interpreter - input sources ... 168
th ... 86
th! ... 86
th@ ... 86
THEN ... 112
third ... 72
this ... 239
this and catch ... 235
this implementation ... 235
this usage ... 232
thread-deadline ... 273
threaded code implementation ... 337
threading words ... 303
threading-method ... 303
throw ... 114
THROW-codes used in the system ... 317
thru ... 204
tib ... 167
tick (') ... 156
time-related words ... 305
time&date ... 305

TMP, TEMP - environment variable ... 11
to *name* semantics, changing them ... 132
to-class: ... 134
to-table: ... 133
to-this ... 239
TO on non-VALUEs ... 315
TO on non-VALUEs and non-locals ... 321
tokens for words ... 156
toupper ... 91
TO ... 122
TOS definition ... 45
translate-cell ... 177
translate-complex ... 177
translate-dcell ... 177
translate-env ... 178
translate-float ... 177
translate-name ... 177
translate-string ... 178
translate-to ... 178
translate: ... 179
translation ... 176
translation token ... 176
translation-token words, stack effect ... 177
traverse-wordlist ... 158
trigonometric operations ... 71
true ... 60
truncation of floating-point numbers ... 320
try ... 116
tt ... 260
tuck ... 73
turnkey image files ... 336
Tutorial ... 14
type ... 209
types of locals ... 217
types of stack items ... 58
types tutorial ... 20
typewhite ... 209

U

u*/ ... 63
u*/mod ... 63
u, stack item type ... 59
u-[do ... 107
u. ... 205
u.r ... 205
u/ ... 62
u/-stage1m ... 65
u/-stage2m ... 65
u/mod ... 63
u/mod-stage2m ... 65
u< ... 68
u<= ... 68
u> ... 68
u>= ... 68
U+DO ... 107
U-DO ... 108
uallot ... 273
ud, stack item type ... 59
ud. ... 206
ud.r ... 206
ud/mod ... 63
UDefer ... 273
um* ... 61
um/mod ... 63
umax ... 61
umin ... 61
umod ... 62
umod-stage2m ... 65
unaligned memory access ... 82
uncolored-mode ... 211
undefined word ... 312
undefined word, ', POSTPONE, ['], [COMPILE] ... 315
under+ ... 61
unescape ... 102
unexpected end of the input buffer ... 314
unlock ... 274
unloop ... 109
unmapped block numbers ... 319
UNREACHABLE ... 221
UNTIL ... 112
UNTIL loop ... 104
unused ... 76
unused-words ... 260
unwind-protect ... 117
up@ ... 274
update ... 204
UPDATE, no current block buffer ... 317
updated? ... 204
upper and lower case ... 59
use ... 204
User ... 273
user input device, method of selecting ... 311
user output device, method of selecting ... 311
user space ... 273
user variables ... 273
user' ... 274
user-defined defining words ... 125
Uses of a word ... 259
UTF-8 and ASCII ... 88
utime ... 305
UValue ... 273

V

v* ... 70
Value ... 121
value-flavoured locals ... 217
value-flavoured words ... 122
value: ... 144
value[]: ... 146
values ... 121
var ... 242, 243
Variable ... 120
variable-flavoured locals ... 217
variable-flavoured words ... 122

variables ... 120
variadic C functions ... 279
versions, invoking other versions of Gforth ... 9
`view` (called `locate` in Gforth) ... 258
viewing the documentation of a word in Emacs ... 327
viewing the source of a word in Emacs ... 327
virtual function ... 228
virtual function table ... 235
virtual machine ... 349
virtual machine instructions ... 350
visibility of locals ... 220
`vlist` ... 184
vocabularies vs. wordlists ... 186
Vocabularies, usage ... 186
`Vocabulary` ... 185
`vocs` ... 185
vocstack empty, `previous` ... 323
vocstack full, `also` ... 323

W

`w!` ... 82
`w,` ... 77
`w`, stack item type ... 59
`w/o` ... 197
`w>s` ... 84
`w@` ... 82
`W:` ... 219
`W^` ... 219
`wake` ... 273
`walign` ... 86
`waligned` ... 86
`warning-color` ... 210
`WARNING"` ... 118
`warnings` ... 118
`wbe` ... 83
`wfield:` ... 143
`where` ... 259
where to go next ... 52
`whereg` ... 260
`WHILE` ... 113
`WHILE` loop ... 104
wid ... 182
`wid`, stack item type ... 59
wior type description ... 59
wior values and meaning ... 318
`within` ... 68
`wle` ... 83
word ... 44, 181
word glossary entry format ... 57
word list for defining locals ... 226
word lists ... 182
word lists - why use them? ... 185
word name too long ... 312
`WORD` buffer size ... 311
`WORD`, string overflow ... 315
`wordlist` ... 183
wordlist usage ... 186
`wordlist-words` ... 184
`wordlists` ... 193
wordlists tutorial ... 41
words ... 57, 184
words used in your program ... 270
words, forgetting ... 264
wordset ... 58
`wrap-xt` ... 140
`write-file` ... 198
`write-line` ... 198
`wrol` ... 67
`wror` ... 67
`WTF??` ... 265
`wvalue:` ... 144
`ww` ... 259

X

`x!` ... 83
`x,` ... 77
`x`, stack item type ... 59
`x-size` ... 90
`x-width` ... 91
`x>s` ... 84
`x@` ... 83
`x\string-` ... 90
`xalign` ... 86
`xaligned` ... 86
`xbe` ... 83
`xc!+` ... 90
`xc!+?` ... 90
`xc,` ... 91
`xc-size` ... 90
`xc-width` ... 91
`xc@` ... 90
`xc@+` ... 90
`xc@+?` ... 90
`xchar+` ... 90
`xchar-` ... 90
`XCHAR-ENCODING` ... 193
`XCHAR-MAXMEM` ... 194
`xd!` ... 83
`xd,` ... 77
`xd>s` ... 84
`xd@` ... 83
`xdbe` ... 83
`xdle` ... 83
`xemit` ... 209
`xfield:` ... 143
`xhold` ... 91
`xkey` ... 211
`xkey?` ... 211
`xle` ... 83
`xor` ... 66
xt ... 44, 156
xt input format ... 56
`xt`, stack item type ... 59
`xt-locate` ... 259

`xt-new` ... 239
`xt-see` ... 261
`xt-see-code` ... 261
`xt-simple-see` ... 261
`xt>name` ... 158
XT tutorial ... 35
`XT:` ... 220

Z

`Z:` ... 220
`Z^` ... 220
zero-length string as a name ... 314
Zsoter's object-oriented model ... 247
`zvalue:` ... 145