SBCL User Manual

SBCL version 2.6.10-lp161.276.3-suse
2026-11

This manual is part of the SBCL software system. See the `README` file for more information.
This manual is largely derived from the manual for the CMUCL system, which was produced at Carnegie Mellon University and later released into the public domain. This manual is in the public domain and is provided with absolutely no warranty. See the `COPYING` and `CREDITS` files for more information.

Short Contents

1 Introduction 1
2 Getting Started 3
3 Starting and Stopping 9
4 Compiler 17
5 Debugging 29
6 Efficiency 42
7 Beyond the ANSI Standard 59
8 External Formats 88
9 Foreign Function Interface 91
10 Pathnames 103
11 Streams 105
12 Package Locks 112
13 Threading 116
14 Timers 128
15 Networking 129
16 Profiling 133
17 Contributed Modules 138
18 Deprecation 154
19 Further Resources 160
A Concept Index 164
B Function and Macro Index 167
C Variable and Constant Index 172
D Type Index 173
E Declaration Index 174
Colophon 175

Table of Contents

1 Introduction ... 1
1.1 Beyond this Manual ... 1
1.2 Highlights ... 1

2 Getting Started ... 3
2.1 Supported Platforms ... 3
2.2 Obtaining SBCL ... 3
2.2.1 Binary Distribution ... 3
2.2.2 Source Distribution ... 3
2.2.3 SBCL Repository ... 3
2.3 Building SBCL ... 4
2.3.1 Customizing the Build ... 4
2.3.2 Build Options ... 5
2.3.3 Troubleshooting the Build ... 7
2.3.4 Build Prerequisites ... 7
2.4 Installing SBCL ... 8

3 Starting and Stopping ... 9
3.1 Starting SBCL ... 9
3.1.1 Running without Installation ... 9
3.1.2 Running from Emacs ... 9
3.1.3 Shebang Scripts ... 9
3.1.4 Parts of SBCL ... 9
3.1.5 Finding Parts of SBCL ... 10
3.2 Stopping SBCL ... 10
3.2.1 Exit ... 11
3.2.2 End of File ... 11
3.2.3 Exit on Errors ... 11
3.3 Command-line Options ... 11
3.3.1 Runtime Options ... 11
3.3.2 Toplevel Options ... 12
3.4 Initialization Files ... 13
3.5 Initialization and Exit Hooks ... 14
3.6 Core Images ... 14
3.6.1 Saving a Core Image ... 14
3.6.2 Default Toplevel ... 16

4 Compiler ... 17
4.1 Calling the Compiler ... 17
4.2 Diagnostic Messages ... 19
4.2.1 Diagnostic Severity ... 19
4.2.2 Understanding Compiler Diagnostics ... 19
4.2.2.1 Parts of a Compiler Diagnostic ... 19
4.2.2.2 Original and Actual Source ... 21
4.2.2.3 Processing Paths ... 21
4.2.3 Controlling Verbosity ... 22
4.3 Handling of Types ... 23
4.3.1 Type Declaration Checking ... 23
4.3.2 Precise Type Checking ... 24
4.3.3 Getting Existing Programs to Run ... 24
4.3.4 Implementation Limitations ... 25
4.4 Compiler Policy ... 25

4.5 Compiler Errors . . . 26
4.5.1 Type Errors at Compile Time . . . 26
4.5.2 Errors During Macroexpansion . . . 27
4.5.3 Read Errors . . . 27
4.6 Open Coding and Inline Expansion . . . 27
4.7 Interpreter . . . 28
4.8 Advanced Compiler Use and Efficiency Hints . . . 28

5 Debugging . . . 29
5.1 Debugger . . . 29
5.1.1 Debugger Entry . . . 29
5.1.1.1 Debugger Banner . . . 29
5.1.1.2 Debugger Invocation . . . 29
5.1.2 Debugger Command Loop . . . 30
5.1.3 Stack Frames . . . 30
5.1.3.1 Stack Motion . . . 30
5.1.3.2 How Arguments are Printed . . . 31
5.1.3.3 Function Names . . . 31
5.1.3.4 Entry Point Details . . . 31
5.1.3.5 Debugging Tail Calls . . . 31
5.1.3.6 Unknown Locations and Interrupts . . . 32
5.1.4 Variable Access . . . 32
5.1.4.1 Variable Value Availability . . . 33
5.1.4.2 Note On Lexical Variable Access . . . 33
5.1.5 Source Location Printing . . . 34
5.1.5.1 How the Source is Found . . . 34
5.1.5.2 Source Location Availability . . . 35
5.1.6 Debug Information . . . 35
5.1.7 Exiting the Debugger . . . 36
5.1.8 Information Commands . . . 36
5.1.9 Breakpoint Commands . . . 37
5.1.9.1 Breakpoint Example . . . 37
5.1.10 Single Stepping . . . 38
5.2 LDB . . . 39
5.3 Enabling and Disabling the Debugger . . . 39
5.4 Function Tracing . . . 39

6 Efficiency . . . 42
6.1 Source Optimization . . . 42
6.1.1 Let Optimization . . . 42
6.1.2 Constant Folding . . . 43
6.1.3 Unused Expression Elimination . . . 43
6.1.4 Control Optimization . . . 43
6.1.5 Dead Code Elimination . . . 44
6.1.6 Multiple Values Optimization . . . 45
6.1.7 Style Recommendations . . . 45
6.2 Tail Call . . . 46
6.2.1 Tail Call Optimization . . . 46
6.2.2 Tail Position . . . 46
6.2.3 Tail Recursion . . . 46
6.3 Local Call . . . 47
6.3.1 Local Self-recursive Call . . . 47
6.3.2 Let Call . . . 47
6.3.3 Local Calls to Closures . . . 48
6.3.4 Local Tail Call . . . 48
6.3.5 Local Call Return Values . . . 49

6.4 Block Compilation ... 49
6.4.1 Block Compilation Semantics ... 49
6.4.2 Block Compilation Declarations ... 50
6.4.3 Block Compilation Difficulties ... 50
6.5 Inline Expansion ... 50
6.5.1 Inline Expansion Recording ... 51
6.5.2 Semi-inline Expansion ... 52
6.6 Slot Access ... 53
6.6.1 Structure Object Slot Access ... 53
6.6.2 Standard Object Slot Access ... 53
6.7 Stack Allocation ... 53
6.8 Modular Arithmetic ... 56
6.8.1 Signed Modular Arithmetic ... 56
6.9 Recognized Idioms ... 56
6.9.1 Count Trailing Zeros ... 56
6.10 Global and Always-bound Variables ... 56
6.11 Miscellaneous Efficiency Issues ... 57

7 Beyond the ANSI Standard ... 59
7.1 Idiosyncrasies ... 59
7.1.1 Declarations as Assertions ... 59
7.1.2 Fasl Format ... 59
7.1.3 Compiler-only Implementation ... 59
7.1.4 Defining Constants ... 59
7.1.5 Style Warnings ... 60
7.2 Reader Extensions ... 60
7.2.1 Extended Package Prefix Syntax ... 60
7.2.2 Symbol Name Normalization ... 60
7.2.3 Decimal Syntax for Rationals ... 60
7.3 Package-local Nicknames ... 61
7.4 Package Variance ... 62
7.5 Garbage Collection ... 62
7.5.1 Finalization ... 62
7.5.2 Weak Pointers ... 63
7.5.3 Introspection and Tuning ... 63
7.5.4 Tracing Live Objects Back to Roots ... 64
7.6 Generic Function Dispatch ... 66
7.7 Extended Slot Access ... 66
7.8 Metaobject Protocol ... 66
7.8.1 AMOP Compatibility of Metaobject Protocol ... 66
7.8.2 Metaobject Protocol Extensions ... 68
7.9 Extensible Sequences ... 68
7.9.1 Iterator Protocol ... 71
7.9.2 Simple Iterator Protocol ... 72
7.10 Unix Support ... 73
7.10.1 Running external programs ... 73
7.11 Unicode Support ... 76
7.11.1 Unicode property access ... 76
7.11.2 String operations ... 78
7.11.3 Breaking strings ... 79
7.12 Customization Hooks for Users ... 79
7.13 Tools to Help Developers ... 80
7.14 Resolution of Name Conflicts ... 80
7.15 Hash Table Extensions ... 80
7.16 Random Number Generation ... 82
7.17 Timeouts and Deadlines ... 83
7.17.1 Timeout Parameters ... 83

7.17.2 Synchronous Timeouts ... 84
7.17.3 Asynchronous Timeouts ... 85
7.17.4 Operations Supporting Timeouts and Deadlines ... 85
7.18 Miscellaneous Extensions ... 85
7.19 Stale Extensions ... 86

8 External Formats ... 88
8.1 The Default External Format ... 88
8.2 External Format Designators ... 88
8.3 Character Coding Conditions ... 89
8.4 Converting between Strings and Octet Vectors ... 89
8.5 Supported External Formats ... 89

9 Foreign Function Interface ... 91
9.1 Introduction to the Foreign Function Interface ... 91
9.2 Foreign Types ... 91
9.2.1 Defining Foreign Types ... 91
9.2.2 Foreign Types and Lisp Types ... 92
9.2.3 Foreign Type Specifiers ... 92
9.3 Operations On Foreign Values ... 93
9.3.1 Accessing Foreign Values ... 94
9.3.1.1 Untyped memory ... 94
9.3.2 Coercing Foreign Values ... 94
9.3.3 Foreign Dynamic Allocation ... 94
9.4 Foreign Variables ... 95
9.4.1 Local Foreign Variables ... 95
9.4.2 External Foreign Variables ... 96
9.5 Foreign Data Structure Examples ... 96
9.6 Loading Shared Object Files ... 97
9.7 Foreign Function Calls ... 98
9.8 Calling Lisp From C ... 99
9.8.1 Lisp as a Shared Library ... 100
9.9 Step-by-step Example of the Foreign Function Interface ... 100

10 Pathnames ... 103
10.1 Lisp Pathnames ... 103
10.1.1 Home Directory Specifiers ... 103
10.1.2 The SYS Logical Pathname Host ... 103
10.2 Native Filenames ... 103

11 Streams ... 105
11.1 Stream External Formats ... 105
11.2 Bivalent Streams ... 105
11.3 Gray Streams ... 105
11.3.1 Gray Streams classes ... 105
11.3.2 Methods common to all streams ... 106
11.3.3 Input stream methods ... 106
11.3.4 Character input stream methods ... 106
11.3.5 Output stream methods ... 107
11.3.6 Character output stream methods ... 107
11.3.7 Binary stream methods ... 108
11.3.8 Gray Streams Examples ... 108
11.3.8.1 Character Counting Input Stream ... 108
11.3.8.2 Output Prefixing Character Stream ... 110
11.4 Simple Streams ... 111

12 Package Locks ... 112
12.1 Package Lock Concepts ... 112
12.1.1 Implementation Packages ... 112
12.1.2 Package Lock Violations ... 112
12.1.2.1 Lexical Bindings and Declarations ... 112
12.1.2.2 Other Operations ... 112
12.1.3 Package Locks in Compiled Code ... 113
12.1.4 Operations Violating Package Locks ... 113
12.1.4.1 Operations on Packages ... 113
12.1.4.2 Operations on Symbols ... 113
12.2 Package Lock Dictionary ... 114

13 Threading ... 116
13.1 Threading Basics ... 116
13.1.1 Thread Objects ... 116
13.1.2 Running Threads ... 116
13.1.3 Asynchronous Operations ... 117
13.1.4 Miscellaneous Operations ... 118
13.1.5 Error Conditions ... 118
13.2 Special Variables ... 119
13.3 Atomic Operations ... 119
13.4 Mutex Support ... 122
13.5 Semaphores ... 123
13.6 Waitqueue/condition variables ... 124
13.7 Barriers ... 126
13.8 Sessions/Debugging ... 126
13.9 Foreign threads ... 127
13.10 Implementation on Linux x86oids ... 127

14 Timers ... 128

15 Networking ... 129
15.1 Sockets Overview ... 129
15.2 General Sockets ... 129
15.3 Socket Options ... 130
15.4 INET Domain Sockets ... 131
15.5 Local Domain Sockets ... 132
15.6 Name Service ... 132

16 Profiling ... 133
16.1 Deterministic Profiler ... 133
16.2 Statistical Profiler ... 133

17 Contributed Modules ... 138
17.1 sb-aclrepl ... 138
17.1.1 Usage ... 138
17.1.2 Customization ... 138
17.1.3 Example Initialization ... 138
17.2 sb-concurrency ... 139
17.2.1 Queue ... 139
17.2.2 Mailbox (lock-free) ... 139
17.2.3 Gates ... 140
17.2.4 Frlocks, aka Fast Read Locks ... 141
17.3 sb-cover ... 142

17.4 sb-grovel ... 143
17.4.1 Using sb-grovel in Your Own ASDF System ... 143
17.4.2 Contents of a grovel-constants-file ... 144
17.4.3 Programming with sb-grovel's Structure Types ... 145
17.4.4 Traps and Pitfalls ... 145
17.5 sb-introspect ... 145
17.5.1 Finding Definitions ... 145
17.5.2 Special Variables ... 147
17.5.3 Functions ... 147
17.5.4 Types and Classes ... 147
17.5.5 Allocation ... 148
17.6 sb-manual ... 149
17.6.1 Using PAX ... 149
17.6.2 Browsing Live with PAX ... 149
17.6.3 Fancy Documentation with PAX ... 149
17.7 sb-md5 ... 150
17.8 sb-posix ... 150
17.8.1 Lisp Names for C Names ... 150
17.8.2 Types ... 151
17.8.2.1 File-descriptors ... 151
17.8.2.2 Filenames ... 151
17.8.3 Function Parameters ... 151
17.8.4 Function Return Values ... 152
17.8.5 Lisp Objects and C structures ... 152
17.8.6 Functions with Idiosyncratic Bindings ... 152
17.8.7 Extensions to POSIX ... 153
17.9 sb-queue ... 153
17.10 sb-rotate-byte ... 153

18 Deprecation ... 154
18.1 Why Deprecate? ... 154
18.2 The Deprecation Pipeline ... 154
18.3 Deprecation Conditions ... 155
18.4 Introspecting Deprecation Information ... 155
18.5 Deprecation Declaration ... 156
18.6 Deprecation Examples ... 156
18.7 Deprecated Interfaces in SBCL ... 156
18.7.1 List of Deprecated Interfaces ... 157
18.7.1.1 Early Deprecation ... 157
18.7.1.2 Late Deprecation ... 157
18.7.1.3 Final Deprecation ... 158
18.7.2 Historical Interfaces ... 158

19 Further Resources ... 160
19.1 The SBCL Project ... 160
19.1.1 Volunteer Support ... 160
19.1.2 Commercial Support ... 160
19.1.3 Reporting Bugs ... 160
19.1.3.1 How to Report Bugs Effectively ... 160
19.1.3.2 Getting Backtraces for Low-level Errors ... 161
19.1.4 Editor Integration ... 161
19.1.5 Internals Documentation ... 161
19.1.6 History and Implementation of SBCL ... 161
19.2 Common Lisp Resources ... 162
19.2.1 Language Reference ... 162
19.2.2 Internet Community ... 162

19.2.3 Third-party Libraries 163
19.2.4 Common Lisp Books 163

Appendix A Concept Index 164

Appendix B Function and Macro Index 167

Appendix C Variable and Constant Index 172

Appendix D Type Index 173

Appendix E Declaration Index 174

Colophon 175

1 Introduction

Steel Bank Common Lisp (SBCL) is a free and open-source ANSI Common Lisp implementation with a high-performance native Chapter 4 [compiler], page 17. In addition, it provides Section 7.11 [unicode support], page 76, Chapter 13 [threading], page 116, an interactive environment including a Section 5.1 [debugger], page 29, a Section 16.2 [statistical profiler], page 133, a code coverage tool, and many other extensions.

- Homepage: `http://sbcl.org`
- Section 19.1.3 [reporting bugs], page 160: `https://bugs.launchpad.net/sbcl`
- Mailing lists: `https://sourceforge.net/p/sbcl/mailman/`
- Repository: `https://sourceforge.net/p/sbcl/sbcl/`
- GitHub mirror: `https://github.com/sbcl/sbcl`
- Supported platforms: `https://www.sbcl.org/platform-table.html`

This manual focuses on behavior specific to SBCL, not on behavior common to all implementations of ANSI Common Lisp. See Chapter 19 [further resources], page 160, for information about Section 19.1 [the sbcl project], page 160, and Common Lisp.

1.1 Beyond this Manual

Documentation for non-ANSI extensions for various commands is available interactively. Tools that have their own command prompts (e.g. the Section 5.1 [debugger], page 29, and `inspect`) provide a `help` command. Extensions without a command prompt (such as `trace`) are described in their `documentation` strings, unless your SBCL was compiled with the `--without-sb-doc` option to omit them (see Section 2.3.1 [customizing the build], page 4), in which case they are only readable in the source code.

This manual itself is available interactively via the Section 17.6 [sb manual], page 149, module.

Besides this user manual, the following files are included in Section 2.2.2 [source distribution], page 3s, Section 2.2.1 [binary distribution], page 3s and the Section 2.2.3 [sbcl repository], page 3. When SBCL is installed, they may be found in e.g. `/usr/local/share/doc/sbcl/` or `/usr/share/doc/sbcl/`.

- `COPYING`: License and copyright summary
- `CREDITS`: Authorship information on various parts of SBCL
- `NEWS`: Summaries of changes between various SBCL versions

And, of course, don't forget the Section 19.2.1 [language reference], page 162.

1.2 Highlights

SBCL comes with numerous extensions, some in core and some in modules loadable with `require`. We only list some of the highlights here. Unfortunately, not all of these extensions have proper documentation yet.

- **Foreign Function Interface:** The `sb-alien` package allows interfacing with C-code, loading shared object files, etc. See Chapter 9 [foreign function interface], page 91.

  Section 17.4 [sb grovel], page 143, can be used to partially automate generation of foreign function interface definitions.
- **Native Threads:** SBCL has native threads on numerous platforms, capable of taking advantage of SMP on multiprocessor machines. See Chapter 13 [threading], page 116.
- **Network Interface:** The `sb-bsd-sockets` module is a low-level networking interface, providing both TCP and UDP sockets. See Chapter 15 [networking], page 129.
- **Operating System Interface:** The `sb-ext` package contains a number of functions for running external processes, accessing environment variables, etc.

  The Section 17.8 [sb posix], page 150, module provides a lispy interface to standard POSIX facilities.
- **Profiling:** The `sb-profile` package provides an exact, per-function Section 16.1 [deterministic profiler], page 133, while the `sb-sprof` module is SBCL's Section 16.2 [statistical profiler], page 133, capable of call-graph generation and instruction level profiling, which also supports allocation profiling.
- **Executable Delivery:** The `:executable` argument to `sb-ext:save-lisp-and-die` can produce a stand-alone executable containing both an image of the current Lisp session and an SBCL runtime.

- **Timeouts and Deadlines:** SBCL allows restricting the execution time of individual operations or parts of a computation using `:timeout` arguments to certain blocking operations, synchronous timeouts and asynchronous timeouts. The latter two affect operations without explicit timeout support (such as standard functions and macros). See Section 7.17 [timeouts and deadlines], page 83.
- **Metaobject Protocol:** The `sb-mop` package provides an implementation of the metaobject protocol for the Common Lisp Object System as described in *The Art of the Metaobject Protocol* by Kiczales et al.
- **Extensible Sequences:** SBCL allows users to define subclasses of the `sequence` class. See Section 7.9 [extensible sequences], page 68.
- **Extensible Streams:** The package `sb-gray` provides an implementation of Section 11.3 [gray streams], page 105.

  The Section 11.4 [sb simple streams], page 111, module is an implementation of the Simple Streams API proposed by Franz Inc.
- **CLTL2 Compatibility Layer:** The `sb-cltl2` module provides `sb-cltl2:compiler-let` and environment access functionality described in *Common Lisp The Language, 2nd Edition* which were removed from the language during the ANSI standardization process.
- **Recursive Event Loop:** SBCL provides a recursive event loop (`serve-event`) for doing non-blocking IO on multiple streams without using threads.
- **System Definition Tool:** ASDF (`https://www.sbcl.org/manual/asdf.html`) is a flexible and popular protocol-oriented system definition tool by Daniel Barlow. It is bundled with SBCL. Load it with `(require :asdf)`.

2 Getting Started

2.1 Supported Platforms

See `https://sbcl.org/platform-table.html` for the list of supported operating systems and architectures.

Some operating systems are more equal than others: most of the development and testing is done on x86-64 Linux and x86-64/arm64 Mac OS X.

If an underprivileged platform is important to you, you can help by e.g. testing during the monthly freeze periods and, most importantly, by Section 19.1.3 [reporting bugs], page 160.

2.2 Obtaining SBCL

Your package manager may provide prepackaged binaries. On Debian, for example, you can install a stable, though possibly older, version of SBCL with `apt install sbcl`. The exact details vary depending on your system.

2.2.1 Binary Distribution

The latest official binary distributions (tar and zip archives of pre-compiled SBCL and its documentation) for selected platforms can be downloaded from

- `https://sbcl.org/platform-table.html`

Some older binaries are available at

- `https://sourceforge.net/projects/sbcl/files/sbcl/`

Note that many platforms do not have recent binaries released. In that case, get a Section 2.2.2 [source distribution], page 3, or check out the Section 2.2.3 [sbcl repository], page 3, and build SBCL.

If you were lucky enough to have new enough binary distributions, unpack the archive with

```
bzip2 -cd sbcl-2.6.9-x86-linux-binary.tar.bz2 | tar xvf -
```

replacing `sbcl-2.6.9-x86-linux-binary.tar.bz2` with the name of the tarball you downloaded. Now, you can immediately try Section 3.1.1 [running without installation], page 9.

Then, enter the directory which was unpacked and skip ahead to Section 2.4 [installing sbcl], page 8, to put SBCL in its proper place.

On Windows, you may want to use the MSI installer to unpack and install in one step:

```
msiexec /i sbcl-2.6.9-x86-64-windows.msi
```

2.2.2 Source Distribution

If there is no suitable Section 2.2.1 [binary distribution], page 3, for your platform, your next choice is to download a source distribution (a tar or zip file with the SBCL sources) from

- `https://sourceforge.net/projects/sbcl/files/sbcl/` or
- `https://github.com/sbcl/sbcl/releases`.

See Section 2.3 [building sbcl], page 4, for the next steps.

2.2.3 SBCL Repository

If you want to be on the bleeding edge, you can check out the sources of the latest development version (or any previous development version, for that matter) by using anonymous GIT to SourceForge:

```
$ git clone git://git.code.sf.net/p/sbcl/sbcl sbcl
```

See `https://sourceforge.net/p/sbcl/sbcl/ci/master/tree/` for instructions.

Alternatively, check out the official GitHub mirror at `https://github.com/sbcl/sbcl`:

```
$ git clone https://github.com/sbcl/sbcl.git
```

See Section 2.3 [building sbcl], page 4, for the next steps.

> *Note*: Tracking the repository is not recommended if you're just using SBCL as a tool for other work, but if you're interested in working on SBCL itself, it's a good idea.

2.3 Building SBCL

To build SBCL, you need a Section 2.2.2 [source distribution], page 3, or the Section 2.2.3 [sbcl repository], page 3, checked out. In any case, the instructions are the same.

You also need a working C toolchain and a Common Lisp system (see Section 2.3.4 [build prerequisites], page 7). You also need approximately 128 MB of free RAM+swap.

To build SBCL using another, already installed SBCL version:

```
$ sh make.sh
```

To configure SBCL to install to a non-standard location, you can use the `--prefix` option:

```
$ sh make.sh --prefix=/opt/mysbcl
```

To configure SBCL with a non-standard default dynamic-space size, use the `--dynamic-space-size` option:

```
$ sh make.sh --dynamic-space-size=4gb
$ sh make.sh --dynamic-space-size=800mb
```

If mega- or gigabytes are not specified, the number is taken to be in megabytes. The standard default is 512Mb for 32-bit systems, and 1GB for 64-bit systems (with the exception of OpenBSD where 444MB are used to fit under default ulimits.)

If you don't already have an SBCL binary installed as `sbcl` on your system, you'll need to tell `make.sh` what Lisp to use as the cross-compilation host (cf. `make.sh` for detailed instructions on how to cross compile). For example, to use CMUCL (assuming it has been installed under its default name `lisp`) as the cross-compilation host:

```
$ sh make.sh --xc-host='lisp -batch -noinit'
```

The build may take a long time, especially on older hardware. A successful build ends with a message beginning: "The build seems to have finished successfully...".

To run the regression tests:

```
$ cd ./tests && sh run-tests.sh
```

To build documentation:

```
$ cd ./doc/manual && make
```

This builds the Info, HTML and PDF documentation from the Section 17.6 [sb manual], page 149, contrib. The manual includes documentation strings from the built SBCL. If SBCL itself has not been built yet, but an installed one is found, documentation strings from the installed version are used.

Now you should have the same `src/runtime/sbcl` and `obj/sbcl-home/sbcl.core` files that come with the binary distribution, the next thing is Section 2.4 [installing sbcl], page 8.

2.3.1 Customizing the Build

You can tweak the build options, enabling or disabling things like documentation strings, thread support, or extra debugging code.

The preferred way to do this is using command-line arguments to `make.sh`:

```
--fancy                    Enables a curated set of features.
--with-<feature>           Enables a specific feature.
--without-<feature>        Disables a specific feature.
```

When tracking the Section 2.2.3 [sbcl repository], page 3, it may be annoying having to supply the same arguments to `make.sh` over and over again. You can avoid this by creating a file named `customize-target-features.lisp` in the base directory of the SBCL tree and putting a single lambda expression like this in it:

```
(lambda (list)
  (flet ((enable (x) (pushnew x list))
         (disable (x) (setf list (remove x list))))
    #+nil (enable :sb-show)
    (enable :sb-after-xc-core)
    #+nil (disable :sb-doc)
    list))
```

This function gets a list of features from `make.sh` and returns the transformed list to be used for the build.

> *Note*: There is no direct correspondance between Section 2.3.2 [build options], page 5, and `*features*` in the compiled SBCL. Some build options do make it to `*features*`, while others are dropped, and there are features such as `:ansi-cl` and `:sbcl` with no corresponding build option.

2.3.2 Build Options

- `:sb-doc`

  Retain documentation strings found in the system sources in the target SBCL executable. Traditional Common Lisp folk will want this option set.

- `:sb-show`

  Make more debugging information available (for debugging SBCL itself). If you aren't hacking or troubleshooting SBCL itself, you probably don't want this set.

  At least two varieties of debugging information are enabled by this option:

  - SBCL is compiled with a higher level of `optimize debug`, so that the debugger can tell more about the state of the system.
  - Various code to print debugging messages, and similar debugging code, is compiled only when this feature is present.

  Note that the extra information recorded by the compiler at this higher level of `optimize debug` includes the source location forms. In order for the debugger to use this information, it has to re-`read` the source file. In an ordinary installation of SBCL, this re-`read`ing may not work very well, for either of two reasons:

  - The sources aren't present on the system in the same location that they were on the system where SBCL was compiled.
  - SBCL is using the standard readtable, without the added hackage which allows it to handle things like target features.

  If you want to be able to use the extra debugging information, therefore, be sure to keep the sources around, and run with the readtable configured so that the system sources can be read.

- `:sb-ldb`

  Enable the Section 5.2 [ldb], page 39, the low-level debugger. In the ideal world you would not need this unless you are messing with SBCL at a very low level (e.g. trying to diagnose GC problems, or trying to debug assembly code for a port to a new CPU). However, experience shows that sooner or later everyone `lose()`s, in which case `sb-ldb` can at least provide an informative backtrace.

- `:sb-after-xc-core`

  This isn't really a target Lisp feature at all but controls whether the build process produces an `after-xc.core` file. This can be useful for shortening the edit/compile/debug cycle when you modify SBCL's own source code, as in `slam.sh`. Otherwise you don't need it.

- `:sb-show-assem`

  Enable extra debugging output in the `assem.lisp` assembler / scheduler code.

- `:sb-dyncount`

  Enable code for collecting statistics on usage of various operations, useful for performance tuning of the SBCL system itself. This code is probably pretty stale (having not been tested since the fork from base CMUCL) but might nonetheless be a useful starting point for anyone who wants to collect such statistics in the future.

- `:sb-simd-pack`

  Enabled automatically by `make-config.sh` for platforms which implement short vector SIMD intrinsics.

- `:cycle-counter`

  Enabled automatically by `make-config.sh` for platforms which implement the `%read-cycle-counter` VOP. Can be disabled manually: affects `time`.

- `:immobile-space`

  Build with support for an additional dynamic heap differing from the main dynamic heap in two ways:

  1. It is guaranteed to reside below 4GB so that all pointers into it fit in 32 bits. (Only an issue for >32 bit address space)

  2. All objects therein are immovable, and space is reclaimed by a mark-and-sweep collector.

  That combination of aspects potentially allows various efficiencies in code generation, especially for the x86-64 backend. The extra space has a fixed size which can only be changed by a rebuild, and

out-of-space conditions are not easily preventable, so the space is sized rather generously to sidestep the issue. Additionally, it is assumed that for all objects in the immobile heap, speed of allocation of those objects is relatively unimportant. If unexpected performance regressions are observed, consider disabling this feature and reporting a bug.

- `:immobile-code`

  Allocate most functions in the immobile space. Enabled by default if supported. The down-side of this feature is that the allocator is significantly slower than the allocator for movable code. If a particular application is performance-constrained by speed of creation of compiled functions (not including closures), the feature can be disabled.

- `:compact-instance-header`

  Combine the layout pointer, instance-length, and widetag of an `instance` into a single machine word. This represents a space savings of anywhere from 4% to 8% in typical applications. (Your mileage may vary.)

- `:sb-thread`

  Enable low-level thread primitives support. See Chapter 13 [threading], page 116.

  As of SBCL 2.6.9, threads are part of the default build on x86oid Linux, Darwin, and Windows. In fact, threads are mandatory on Windows. Other platforms that support them include FreeBSD, Solaris.

- `:sb-futex`

  Use `futex`es as our locking primitive. Only available on Linux.

- `:sb-package-locks`

  Enable detection of unportable code (when applied to the `common-lisp` package or SBCL-internal packages) or bad-neighbourly code (when applied to user-level packages), relating to material alteration to packages or to bindings in symbols in packages. See Chapter 12 [package locks], page 112.

- `:sb-unicode`

  Support for the entirety of the 21-bit character space defined by the Unicode consortium, rather than the classical 8-bit ISO-8859-1 character set. See Section 7.11 [unicode support], page 76.

- `:sb-eval`

  Support for a full evaluator that can execute all the CL special forms, as opposed to the traditional SBCL evaluator, which called `compile` for everything complicated. See Section 4.7 [interpreter], page 28.

- `:sb-fasteval`

  Support for a different evaluator (interpreter) with improved performance. You can have only one of `:sb-eval and :sb-fasteval`. See Section 4.7 [interpreter], page 28.

- `:sb-source-locations`

  Record source location information for variables, classes, conditions, packages, etc. Gives much better information on `M-.` in Slime but increases core size by about 100kB.

- `:sb-xref-for-internals`

  Record xref data for SBCL internals. This can be rather useful for people who want to develop on SBCL itself because it will make `M-?` (`slime-edit-uses`) work which lists call/expansion/etc. sites. It will increase the core size by major 5-6MB, though.

- `:ansi-compliant-load-truename`

  `clhs` says that `*load-truename*` has to be bound whenever `*load-pathname*` is because obviously the spec wanted to mandate that one system call turn into an unlimited number. At least you're correct in saying "it's not my code, it's the OS that's slow". Remove this feature if you want to ignore the technical requirement that `*load-truename*` be a variable (as opposed to a symbol-macro).

- `:largefile`

  Some platforms don't use a 32-bit `off_t` by default, and thus can't handle files larger than 2GB. This feature will control whether we'll try to use platform-specific compilation options to enable a 64-bit `off_t`. The intent is for this feature to be automatically enabled by `make-config.sh` on platforms where it's needed and known to work, you shouldn't be enabling it manually. You might however want to disable it, if you need to pass file descriptors to foreign code that uses a 32-bit `off_t`.

- `:relocatable-static-space`

  Allow static space to be relocatable on start-up.

- `:aslr`

  Allow Lisp spaces to be randomly allocated anywhere in virtual address space, i.e. the behavior of passing `mmap` an address of zero. Requires all spaces to be relocatable on start-up.

- `:sb-core-compression`

  Support zstd-based compressed core files. Enable this feature to compile it in. Obviously, doing so adds a dependency on zstd.

- `:sb-safepoint`

  On certain thread-enabled platforms, synchronization between threads for the purpose of stopping and starting the world around GC can be performed using safepoints instead of signals. Enable this feature to compile with safepoints and to use them for GC. This is always enabled on `:win32`.

- `:sb-cover-for-internals`

  Build the SBCL system itself with code coverage data marks (and sequences in the compiled code to set those marks). Inevitably, this makes the system as a whole slower, but allows coverage of tests to be measured.

2.3.3 Troubleshooting the Build

- "GNU Make not found"

  If the GNU make command is not available under the names `make`, `gmake`, or `gnumake`, then define the environment variable `GNUMAKE` to a name where it can be found.

- Segfaults on Linux with ASLR

  If SBCL segfaults immediately upon startup due to Address Space Layout Randomization (ASLR), you can bypass it by executing the host compiler binary via `setarch`:

  ```
  setarch x86_64 -R sbcl
  ```

  See also `:aslr` in Section 2.3.2 [build options], page 5.

- Build crashes mysteriously, machine becomes unstable, etc

  You may be running out of memory. Try increasing swap, or building SBCL with fewer other programs running simultaneously.

- Other
  - Check that the host lisp you're building with is known to work as an SBCL build host, and that your operating system is supported.
  - Try to do a build without loading any initialization files for the cross-compilation host (for example `sh make.sh 'sbcl --userinit /dev/null --sysinit /dev/null'`).
  - Some GCC versions are known to have bugs that affect SBCL compilation: if the error you're encountering seems related to files under `src/runtime`, down- or upgrading GCC may help.
  - Ask for help on the mailing lists referenced in Section 19.1.1 [volunteer support], page 160.

2.3.4 Build Prerequisites

All of the following platforms are supported in the sense of "should work", but some things like loading foreign object files may lag behind on less-used operating systems.

Supported toolchains:

- GNU toolchain
- Sun toolchain with GCC

SBCL can be compiled from source code using (as the host compiler) another ANSI-compliant Common Lisp implementation such as

- SBCL
- CMUCL
- CCL (formerly known as OpenMCL)
- CLISP (only some versions: 2.44.1 is OK, 2.47 is not)
- ECL
- XCL

- ABCL (recent versions only, very slow)

Often, it is most convenient to install an SBCL with your system's package manager to use as the host.

Note that not every release is tested with every possible host compiler. You're most likely to get a clean build with SBCL itself as host, otherwise CCL on a PPC and CMUCL elsewhere.

2.4 Installing SBCL

The installation instructions are the same whether you are have downloaded a Section 2.2.1 [binary distribution], page 3, or built SBCL yourself.

The following command installs SBCL and related documentation under the `/usr/local` directory (typically run as root):

```
# INSTALL_ROOT=/usr/local sh install.sh
```

You can also install SBCL as a user, under your home directory:

```
$ INSTALL_ROOT=/home/me sh install.sh
```

That is, `install.sh` installs SBCL under the directory named by the environment variable `INSTALL_ROOT`.

If `INSTALL_ROOT` is not specified, SBCL is installed into location configured when Section 2.3 [building sbcl], page 4, with the `--prefix` argument of `make.sh`: for official binary distributions, this is the `/usr/local` directory.

The standard installation procedure, outlined above, is to run `install.sh`, which copies all the files to right places, including documentation and contrib modules that have passed their tests. If you need to install by hand, see `install.sh` for details.

Documentation consists of a man page, the SBCL Manual (in info, pdf and html formats), and a few additional text files.

3 Starting and Stopping

3.1 Starting SBCL

If SBCL is installed properly and it is in your `path`, you can simply type `sbcl` at the command line. You should end up in the toplevel *REPL* (read-eval-print loop), where you can interact with SBCL by typing expressions.

```
$ sbcl
This is SBCL 0.8.13.60, an implementation of ANSI Common Lisp.
More information about SBCL is available at <http://www.sbcl.org/>.

SBCL is free software, provided as is, with absolutely no warranty.
It is mostly in the public domain; some portions are provided under
BSD-style licenses.  See the CREDITS and COPYING files in the
distribution for more information.
* (+ 2 2)
4
* (exit)
$
```

See also Section 3.3 [command line options], page 11, and Section 3.2 [stopping sbcl], page 10.

3.1.1 Running without Installation

You can quickly run SBCL without installing it by invoking the `run-sbcl.sh` script. This works both with an unpacked Section 2.2.1 [binary distribution], page 3, and after Section 2.3 [building sbcl], page 4, yourself:

```
$ sh run-sbcl.sh
```

3.1.2 Running from Emacs

To run SBCL as an `inferior-lisp` from Emacs, add something like this to your `.emacs`:

```
;;; The SBCL binary and command-line arguments
(setq inferior-lisp-program "/usr/local/bin/sbcl --noinform")
```

For more information on using SBCL with Emacs, see Section 19.1.4 [editor integration], page 161.

3.1.3 Shebang Scripts

Standard Unix tools that are interpreters follow a common command-line protocol that is necessary to work with "shebang scripts". SBCL supports this via the `--script` command-line option (see Section 3.3 [command line options], page 11).

Example file (`hello.lisp`):

```
#!/usr/local/bin/sbcl --script
(write-line "Hello, World!")
```

Usage from the command line:

```
$ ./hello.lisp
Hello, World!
```

Note that SBCL skips the shebang line when it reads the file:

```
$ sbcl --script hello.lisp
Hello, World!
```

3.1.4 Parts of SBCL

If you only ever use SBCL packaged for your operating system, run it from an installation created with `install.sh`, run it with `run-sbcl.sh`, there is no need to worry about the files necessary at run-time. If you, for example, want to install SBCL manually, package it in a way that deviates from the file hierarchy created by `install.sh`, or run SBCL with your own core file saved by `sb-ext:save-lisp-and-die`, then read on.

The following files play a part in running SBCL.

- *The* `sbcl` *binary*

  This is a standard executable built by compiling and linking an ordinary C program. It provides the runtime environment for the running Lisp image, but it doesn't know much about high-level Lisp stuff (like symbols and printing and objects), so it's pretty useless by itself.

  The binary is `src/runtime/sbcl` in Section 2.2.1 [binary distribution], page 3s and Section 2.2.2 [source distribution], page 3s; and it is installed to `/usr/local/bin/sbcl` by default.

- *The* `sbcl.core` *file*

  This is an image dump written in a special SBCL format that only SBCL understands, and it contains all the high-level Lisp stuff.

  The core file is `obj/sbcl-home/sbcl.core` in distributions; installed to `/usr/local/lib/sbcl/sbcl.core` by default. In addition, there are a number of Chapter 17 [contributed modules], page 138, that extend the basic SBCL functionality.

- *The* `contrib/` *directory*

  This holds the compiled fasl files for the Chapter 17 [contributed modules], page 138, to be loaded on demand.

  The contrib fasl files are in `obj/sbcl-home/contrib/` in distributions; installed to `/usr/local/lib/sbcl/contrib/` by default.

  Note: Only the binary and the core file are absolutely necessary to start SBCL; the contrib fasls are optional.

3.1.5 Finding Parts of SBCL

When you invoke the SBCL binary, it needs to be able to find the other Section 3.1.4 [parts of sbcl], page 9, namely the `sbcl.core` file and the Chapter 17 [contributed modules], page 138, in the `contrib/` directory.

- Finding `sbcl.core`

  If the `--core` runtime option is specified, then its value is used. Example:

  ```
  $ sbcl --core /foo/bar/sbcl.core
  ```

  Else, the runtime looks for `sbcl.core` in the following directories until the first match.

  - The directory denoted by the `SBCL_HOME` environment variable if it is set
  - The current directory
  - The directory of the `sbcl` binary
  - The `../lib/sbcl/` directory relative to the `sbcl` binary

  If no core file was found by the above rules, then the runtime exits with an error message.

- Finding `contrib/`

  Once the runtime loads the core file, the Lisp looks for `contrib/` in the following directories until the first match.

  - The directory denoted by the `SBCL_HOME` environment variable if it is set
  - *The directory of* `sbcl.core`
  - *The directory* `../lib/sbcl` *relative to* `sbcl.core`
  - The current directory
  - The directory of the `sbcl` binary
  - The directory `../lib/sbcl` relative to the `sbcl` binary

  The list of directories is the same as for finding the core file except here we also try directories relative to `sbcl.core`.

  If the `contrib/` directory is not found, then no error is signalled immediately, but loading modules will fail.

3.2 Stopping SBCL

3.2.1 Exit

SBCL can be stopped at any time by calling `sb-ext:exit`, optionally returning a specified numeric value to the calling process. See Chapter 13 [threading], page 116, for information about terminating individual threads.

3.2.2 End of File

By default, SBCL also exits upon reaching the end of input. This can be caused either by the user pressing `Control-D` on an attached terminal, or by the input stream ending when SBCL is used as part of a shell pipeline.

3.2.3 Exit on Errors

SBCL can also be configured to exit if an unhandled error occurs, which is mainly useful for acting as part of a shell pipeline; doing so under most other circumstances would mean giving up large parts of the flexibility and robustness of Common Lisp. See Section 5.1.1 [debugger entry], page 29, and the command-line option `--disable-debugger` in Section 3.3.2 [toplevel options], page 12.

3.3 Command-line Options

Command-line options can be considered an advanced topic; for ordinary interactive use, no command-line arguments should be necessary.

In order to understand the command-line argument syntax for SBCL, it is helpful to understand that the SBCL system is implemented as two components, a low-level runtime environment written in C and a higher-level system written in Common Lisp. Arguments are processed during the respective initializations of the runtime and Lisp system, with any remainder available to user code via `sb-ext:*posix-argv*`.

The full, unambiguous syntax for invoking SBCL at the command line is:

```
sbcl <runtime-option>* --end-runtime-options \
     <toplevel-option>* --end-toplevel-options \
     <user-option>*
```

The `--end-runtime-options` and `--end-toplevel-options` can be omitted, provided that no ambiguities are possible with the option values used. Omitting these elements is probably a bad idea for any batch file where any of the options are under user control because it makes it impossible for SBCL to detect erroneous command-line arguments, which will then be passed on to the user program even if they were intended for the runtime system or the Lisp system.

3.3.1 Runtime Options

- `--core <corefilename>`

  Run the specified Lisp core file instead of the default. Note that if the Lisp core file is a user-created core file, it may run a nonstandard toplevel that does not recognize the standard toplevel options.

- `--dynamic-space-size <size>`

  Size of the dynamic space reserved on startup. The default value is platform-dependent. The units supported are kB (or KiB), MB (or MiB), GB (or GiB), and TB (or TiB). The names are compared case-insensitively, and the default unit is megabytes. Thus, `8`, `8mb`, and `8mib` all mean 8 * 2^20 bytes.

- `--control-stack-size <size>`

  Size of control stack reserved for each thread. The `<size>` argument has the same format as in `--dynamic-space-size`. The default is 2MB.

- `--tls-limit <positive integer>`

  Maximum number of thread-local Section 13.2 [special variables], page 119, in threaded builds. Default value is 4096.

- `--noinform`

  Suppress the printing of any banner or other informational message at startup. This makes it easier to write Lisp programs which work cleanly in Unix pipelines. See also the `--noprint` and `--disable-debugger` options.

- `--disable-ldb`

  Disable the Section 5.2 [ldb], page 39, the low-level debugger. Only effective if SBCL is compiled with LDB. When LDB is not available or is disabled, SBCL exits with a non-zero exit code whenever it would enter LDB.

- `--lose-on-corruption`

  There are some dangerous low-level errors (for instance, control stack exhausted, memory fault) that (or whose handlers) can corrupt the image. By default, SBCL prints a warning, then tries to continue and handle the error in Lisp. However, this will not always work, and SBCL may subsequently malfunction or even hang. With this option, upon encountering such an error, SBCL will exit instead of signalling a Lisp error.

- `--script <filename>`

  As a *runtime* option, this is equivalent to `--noinform --disable-ldb --lose-on-corruption --end-runtime-options --script <filename>`. See the description of `--script` as a *toplevel* option below. If there are no other command-line arguments following `--script`, the filename argument can be omitted.

- `--merge-core-pages`

  When platform support is present, provide hints to the operating system that identical pages may be shared between processes until they are written to. This can be useful to reduce the memory usage on systems with multiple SBCL processes started from similar but differently-named core files, or from compressed cores. Without platform support, do nothing. By default only compressed cores trigger hinting.

- `--no-merge-core-pages`

  Ensures that no sharing hint is provided to the operating system.

- `--help`

  Print some basic information about SBCL, then exit.

- `--version`

  Print SBCL's version information, then exit.

- `--end-runtime-options`

  End processing of runtime options and pass the rest of the command-line arguments to the toplevel function.

In the future, runtime options may be added to control behaviour such as lazy allocation of memory.

3.3.2 Toplevel Options

The following options are supported by the Section 3.6.2 [default toplevel], page 16. If the toplevel function was overriden with `sb-ext:save-lisp-and-die`, the set of available toplevel options is likely very different.

- `--sysinit <filename>`

  Load `<filename>` instead of the default system initialization file (see Section 3.4 [initialization files], page 13).

- `--no-sysinit`

  Don't load a system-wide initialization file. If this option is given, the `--sysinit` option is ignored.

- `--userinit <filename>`

  Load `<filename>` instead of the default user initialization file (see Section 3.4 [initialization files], page 13.)

- `--no-userinit`

  Don't load a user initialization file. If this option is given, the `--userinit` option is ignored.

- `--eval <command>`

  After executing any initialization file, but before starting the read-eval-print loop on standard input, read and evaluate `<command>`. More than one `--eval` option can be used, and all will be read and executed, in the order they appear on the command line.

- `--load <filename>`

  This is equivalent to `--eval '(load "<filename>")'`. The special syntax is intended to reduce quoting headaches when invoking SBCL from shell scripts.

- `--noprint`

  When ordinarily the toplevel "read-eval-print loop" would be executed, execute a "read-eval loop" instead, i.e. don't print a prompt and don't echo results. Combined with the `--noinform` runtime option, this makes it easier to write Lisp scripts that work cleanly in Unix pipelines.

- `--disable-debugger`

  By default, when SBCL encounters an error, it enters the builtin debugger, allowing interactive diagnosis and possible intercession. This option disables the debugger, causing errors to print a backtrace and exit with status 1 instead. When given, this option takes effect before loading initialization files or processing `--eval` and `--load` options. See `sb-ext:disable-debugger` and Section 5.1.1 [debugger entry], page 29.

- `--script <filename>`

  Implies `--no-userinit --no-sysinit --disable-debugger --end-toplevel-options`.

  Causes the system to load the specified file instead of entering the read-eval-print-loop, and exit afterwards. If the file begins with a shebang line, it is ignored.

  If there are no other command-line arguments following, the filename can be omitted: this causes the script to be loaded from standard input instead. In this case, shebang lines are currently *not* ignored.

  In either case, if there is an unhandled error (such as an end of file condition or a broken pipe) on standard input, standard output, or standard error, the script silently exits with code 0. This allows piping output from SBCL to another command without unexpected errors.

  Additionally, the option sets `*compile-verbose*` and `*load-verbose*` to `nil` while loading the file to avoid potentially verbose diagnostic messages printed on the standard output.

- `--end-toplevel-options`

  End processing of toplevel options, leaving the rest of the command-line arguments in `sb-ext:*posix-argv*`.

3.4 Initialization Files

The Section 3.6.2 [default toplevel], page 16, supports customization via the system and user initialization files. These files with `read` and `eval`, not `load`; hence initialization files can be used to set startup `*package*` and `*readtable*`, and for proclaiming a global optimization policy.

- **System Initialization File:** Defaults to `sbclrc` in the directory where `contrib/` is found (see Section 3.1.5 [finding parts of sbcl], page 10), or if that doesn't exist to `/etc/sbclrc`. Can be overridden with the command-line option `--sysinit` or `--no-sysinit` (see Section 3.3.2 [toplevel options], page 12). Set `sb-ext:*sysinit-pathname-function*` to override the above defaulting behavior.

  The system initialization file is intended for system administrators and software packagers to configure locations of installed third party modules, etc.

- **User Initialization File:** Defaults to `$HOME/.sbclrc`. Can be overridden with the command-line option `--userinit` or `--no-userinit` (see Section 3.3.2 [toplevel options], page 12). Set `sb-ext:*userinit-pathname-function*` to override the above defaulting behavior.

  The user initialization file is intended for personal customizations, such as loading certain modules at startup, defining convenience functions to use in the REPL, handling automatic recompilation of fasls (see Section 7.1.2 [fasl format], page 59), etc.

Neither initialization file is required.

`sb-ext:*sysinit-pathname-function*` [Variable]

Controls the sysinit file defaulting of the Section 3.6.2 [default toplevel], page 16. This is `nil` or a designator for a function of zero arguments.

- If `nil`, the default behavior described in Section 3.4 [initialization files], page 13, is in effect.
- If a function, it is called by the default toplevel to obtain a pathname designator for the default sysinit file or `nil`. If the function returns `nil`, no sysinit file is used unless one has been specified on the command line.

`sb-ext:*userinit-pathname-function*` [Variable]

Controls the userinit file defaulting of the Section 3.6.2 [default toplevel], page 16. This is `nil` or a designator for a function of zero arguments.

- If `nil`, the default behavior described in Section 3.4 [initialization files], page 13, is in effect.
- If a function, it is called by the default toplevel to obtain a pathname designator for the default userinit file or `nil`. If the function returns `nil`, no sysinit file is used unless one has been specified on the command line.

3.5 Initialization and Exit Hooks

SBCL provides hooks into the system initialization and exit.

`sb-ext:*init-hooks*` [Variable]

A list of function designators called in an unspecified order when a saved core image starts up, after the system itself has been initialized, but before non-user threads such as the finalizer thread have been started.

Unused by SBCL itself; reserved for the user, contribs and applications.

`sb-ext:*exit-hooks*` [Variable]

A list of function designators called in an unspecified order when SBCL process exits.

Unused by SBCL itself; reserved for the user, contribs and applications.

Using (`sb-ext:exit :abort t`), or calling `exit(3)` directly circumvents these hooks.

3.6 Core Images

3.6.1 Saving a Core Image

SBCL has the ability to save its state as a file for later execution. This functionality is important for its bootstrapping process and is also provided as an extension to the user.

`sb-ext:save-lisp-and-die` *core-file-name* &key *toplevel executable save-runtime-options callable-exports* (*purify t*) *root-structures* (*environment-name* "*auxiliary*") *compression* [Function]

Save a "core image", i.e. enough information to restart a Lisp process later in the same state, in the file of the specified name. Only global state is preserved: the stack is unwound in the process.

The following `&key` arguments are defined:

- `:toplevel`

  The function to run when the created core file is resumed. The default function handles command line toplevel option processing (see Section 3.3.2 [toplevel options], page 12) and runs the top level read-eval-print loop. This function returning is equivalent to (`sb-ext:exit :code` 0) being called.

  `toplevel` functions should always provide an `abort` restart: otherwise code they call will run without one.

- `:executable`

  If true, arrange to combine the SBCL runtime and the core image to create a standalone executable. If false (the default), the core image will not be executable on its own. Executable images always behave as if they were passed the `--noinform` runtime option. If `:executable` is `:elf-object`, then the resulting core will be wrapped in a .o which requires further linking. (EXPERIMENTAL)

- `:save-runtime-options`

  If true, values of Section 3.3.1 [runtime options], page 11, `--dynamic-space-size` and `--control-stack-size` that were used to start SBCL are stored in the standalone executable, and restored when the executable is run. This also inhibits normal runtime option processing, causing all command line arguments to be passed to the toplevel. If `:accept-runtime-options`, then `--dynamic-space-size` and `--control-stack-size` are still processed by the runtime. Meaningless if `:executable` is `nil`.

- `:callable-exports`

  This should be a list of symbols to be initialized to the appropriate alien callables on startup. All exported symbols should be present as global symbols in the symbol table of the runtime before the saved core is loaded. When this list is non-empty, the `:toplevel` argument cannot be supplied.

- `:purify`

  If true (the default), then some objects in the restarted core will be memory-mapped as read-only. Among those objects are numeric vectors that were determined to be compile-time constants, and any immutable values according to the language specification such as symbol names.

- `:root-structures`

  This should be a list of the main entry points in any newly loaded systems. This need not be supplied, but locality and/or `gc` performance may be better if they are. This has two different but related meanings: If `:purify` is true – and only for CheneyGC – the root structures are those which anchor the set of objects moved into static space. On GENCGC – and only on platforms supporting immobile code – these are the functions and/or function-names which commence a depth-first scan of code when reordering based on the statically observable call chain. The complete set of reachable objects is not affected per se. This argument is meaningless if neither enabling precondition holds.

- `:environment-name`

  This has no purpose; it is accepted only for legacy compatibility.

- `:compression`

  This is only meaningful if the runtime was built with the `:sb-core-compression` feature enabled. If `nil` (the default), saves to uncompressed core files. If `:sb-core-compression` was enabled at build-time, the argument may also be an integer from -7 to 22, corresponding to zstd compression levels, or `t` (which is equivalent to the default compression level, 9).

- `:application-type`

  Present only on Windows and meaningful only with `:executable t`. Specifies the subsystem of the executable, `:console` or `:gui`. The notable difference is that `:gui` doesn't automatically create a console window. The default is `:console`.

The save/load process changes the values of some global variables:

- `*standard-output*`, `*debug-io*`, etc

  Everything related to open streams is necessarily changed, since the OS won't let us preserve a stream across save and load.

- `*default-pathname-defaults*`

  This is reinitialized to reflect the working directory where the saved core is loaded.

`save-lisp-and-die` interacts with `sb-alien:load-shared-object`: see its documentation for details.

On threaded platforms, only a single thread may remain running after `sb-ext:*save-hooks*` have run. Applications using multiple threads can be `save-lisp-and-die` friendly by registering a save-hook that quits any additional threads, and an init-hook that restarts them.

This implementation is not as polished and painless as you might like:

- It corrupts the current Lisp image enough that the current process needs to be killed afterwards. This can be worked around by forking another process that saves the core.
- There is absolutely no binary compatibility of core images between different runtime support programs. Even runtimes built from the same sources at different times are treated as incompatible for this purpose.

This isn't because we like it this way, but just because there don't seem to be good quick fixes for either limitation and no one has been sufficiently motivated to do lengthy fixes.

`sb-ext:*save-hooks*` [Variable]

A list of function designators called in an unspecified order before creating a saved core image.

Unused by SBCL itself; reserved for the user and applications.

To facilitate distribution of SBCL applications using external resources, the filesystem location of the SBCL core file being used is available from Lisp.

`sb-ext:*core-pathname*` [Variable]

The absolute pathname of the running SBCL core.

3.6.2 Default Toplevel

SBCL's default toplevel function processes Section 3.3.2 [toplevel options], page 12, runs Section 3.4 [initialization files], page 13, and provides a REPL.

The toplevel function of Section 3.6 [core images], page 14, can be specified with the `:toplevel` argument of `sb-ext:save-lisp-and-die`.

4 Compiler

This chapter discusses most compiler issues other than Chapter 6 [efficiency], page 42, including compiler messages, type safety in the presence of type declarations, the effects of various optimization policies, and the way that inlining and open coding may cause optimized code to differ from a naive translation.

4.1 Calling the Compiler

`compile` *name* &optional (*definition* (*or* (*and* (*symbolp sb-c::name*) (*macro-function sb-c::name*)) (*fdefinition sb-c::name*))) [Function]

Produce a compiled function from `definition`.

- If `definition` is a lambda-expression, it is coerced to a function.
- If `definition` is an interpreted function, it is compiled.
- If `definition` is already a compiled function, it is returned as-is. (Future versions of SBCL might try to recompile the existing definition, but this is not currently supported.)

If `name` is `nil`, the compiled function is returned as the primary value. Otherwise, the resulting compiled function replaces existing function definition of `name`, and `name` is returned as primary value; if `name` is a symbol that names a macro, its `macro-function` is replaced and `name` is returned as primary value.

Also returns a secondary value which is true if any conditions of type `warning` occur during the compilation, and `nil` otherwise.

Tertiary value is true if any conditions of type `error` or `warning` that are not `style-warning`s occur during compilation, and `nil` otherwise.

`compile-file` *input-file* &key (*output-file* `""`) (*verbose* **compile-verbose**) (*print* **compile-print**) (*external-format* *:default*) (*progress* *sb-ext:*compile-progress**) *trace-file* (*block-compile* *sb-ext:*block-compile-default**) *entry-points* (*emit-cfasl* *sb-c::*emit-cfasl**) [Function]

Compile `input-file`, producing a corresponding fasl file and returning its filename.

- `:output-file`

  The name of the fasl to output. The returned pathname of the output file may differ from the pathname of the `:output-file` parameter, e.g. when the latter is a designator for a directory.

- `:verbose`

  If true, information indicating what file is being compiled is printed to `*standard-output*`.

- `:print`

  If true, each top level form in the file is printed to `*standard-output*`.

- `:external-format`

  The external format to use when opening the source file. The default is `:default` which uses the `sb-ext:*default-source-external-format*`.

- `:block-compile` {`nil` | `:specified` | `t`}

  Determines whether multiple functions are compiled together as a unit, resolving function references at compile time. `nil` means that global function names are never resolved at compilation time. `:specified` means that names are resolved at compile-time when convenient (as in a self-recursive call), but the compiler doesn't combine top-level `defun`s. With `:specified`, an explicit `start-block` declaration will enable block compilation. A value of `t` indicates that all forms in the file(s) should be compiled as a unit. The default is the value of `sb-ext:*block-compile-default*`, which is initially `nil`.

- `:entry-points`

  This specifies a list of function names for functions in the file(s) that must be given global definitions. This only applies to block compilation and is useful mainly when `:block-compile t` is specified on a file that lacks `start-block` declarations. If the value is `nil` (the default), then all functions will be globally defined.

- `:trace-file`

  If given, internal data structures are dumped to the specified file, or if a value of `t` is given, to a file of `.trace` type derived from the input file name. (non-standard)

- `:emit-cfasl`

  (Experimental). If true, outputs the toplevel compile-time effects of this file into a separate `.cfasl` file.

`sb-ext:*block-compile-default*` [Variable]

The default value for the `:block-compile` argument to `compile-file`.

`with-compilation-unit` *options* &body *body* [Macro]

Affects compilations that take place within its dynamic extent. It is intended to be e.g. wrapped around the compilation of all files in the same system.

Following options are defined:

- `:override <boolean-form>`

  One of the effects of this form is to delay undefined warnings until the end of the form, instead of giving them at the end of each compilation. If `override` is `nil` (the default), then the outermost `with-compilation-unit` form grabs the undefined warnings. Specifying `:override` true causes that form to grab any enclosed warnings, even if it is enclosed by another `with-compilation-unit`.

- `:policy <optimize-declaration-form>`

  Provides dynamic scoping for global compiler optimization qualities and restrictions, limiting effects of subsequent `optimize` proclamations and calls to `sb-ext:restrict-compiler-policy` to the dynamic scope of `body`.

  If `:override` is false, the specified `:policy` is merged with current global policy. If `:override` is true, the current global policy, including any restrictions, is discarded in favor of the specified `:policy`.

  Supplying `:policy nil` is equivalent to the option not being supplied at all, i.e. dynamic scoping of policy does not take place. If dynamic scoping without changing the current policy is desired, use `:policy '(optimize)` with `:override nil`.

  This option is an SBCL-specific experimental extension: Interface subject to change.

- `:source-namestring <namestring-form>`

  Attaches the value returned by `<namestring-form>` to the internal debug-source information as the namestring of the source file. Normally the namestring of the `input-file` for `compile-file` is used: this option can be used to provide source-file information for functions compiled using `compile`, or to override the `input-file` argument of `compile-file`.

  Can be accessed using `sb-introspect:definition-source-pathname`.

  If both an outer and an inner `with-compilation-unit` provide a `:source-namestring`, the inner one takes precedence. Unaffected by `:override`.

  This is an SBCL-specific extension.

- `:source-plist <plist-form>`

  Attaches the value returned by `<plist-form>` to internal debug-source information of functions compiled in within the dynamic extent of `body`.

  Primarily for use by development environments, in order to e.g. associate function definitions with editor-buffers. Can be accessed using `sb-introspect:definition-source-plist`.

  If an outer `with-compilation-unit` form also provide a `source-plist`, it is appended to the end of the provided `source-plist`. Unaffected by `:override`.

  This is an SBCL-specific extension.

Examples:

```
;; Prevent proclamations from the file leaking, and restrict
;; SAFETY to 3 -- otherwise uses the current global policy.
(with-compilation-unit (:policy '(optimize))
  (restrict-compiler-policy 'safety 3)
  (load "foo.lisp"))
;; Using default policy instead of the current global one,
;; except for DEBUG 3.
(with-compilation-unit (:policy '(optimize debug)
                        :override t)
```

```
  (load "foo.lisp"))
;; Same as if :POLICY had not been specified at all: SAFETY 3
;; proclamation leaks out from WITH-COMPILATION-UNIT.
(with-compilation-unit (:policy nil)
  (declaim (optimize safety))
  (load "foo.lisp"))
```

4.2 Diagnostic Messages

4.2.1 Diagnostic Severity

There are four levels of compiler diagnostic severity:

- error
- warning
- style warning
- note

The first three levels correspond to condition classes which are defined in the ANSI standard for Common Lisp and which have special significance to the `compile` and `compile-file` functions. These levels of compiler error severity occur when the compiler handles conditions of these classes.

The fourth level of compiler error severity, *note*, corresponds to the `sb-ext:compiler-note` condition and is used for problems too mild for the standard condition classes, typically hints about how efficiency might be improved. `sb-ext:code-deletion-note`, a subtype of `sb-ext:compiler-note`, is signalled when the compiler deletes user-supplied code after proving that the code in question is unreachable.

Future work for SBCL includes expanding this hierarchy of types to allow more fine-grained control over emission of diagnostic messages.

`sb-ext:compiler-note` [Condition]

Root of the hierarchy of conditions representing information discovered by the compiler that the user might wish to know, but which does not merit a `style-warning` (or any more serious condition).

`sb-ext:code-deletion-note` [Condition]

A condition type signalled when the compiler deletes code that the user has written, having proved that it is unreachable.

4.2.2 Understanding Compiler Diagnostics

The messages emitted by the compiler contain a lot of detail in a terse format, which may be confusing at first. The messages will be illustrated using this example program:

```
(defmacro zoq (x)
  `(roq (ploq (+ ,x 3))))

(defun froq (y)
  (declare (symbol y))
  (zoq y))
```

The main problem with this program is that it is trying to add `3` to a symbol. Note also that the functions `roq` and `ploq` aren't defined anywhere.

4.2.2.1 Parts of a Compiler Diagnostic

When processing the above program, the compiler will first output this warning:

```
; file: /tmp/froq.lisp
; in: DEFUN FROQ
;     (ZOQ Y)
; --> ROQ PLOQ
; ==>
;   (+ Y 3)
;
; caught WARNING:
```

```
;   Derived type of Y is
;     (VALUES SYMBOL &OPTIONAL),
;   conflicting with its asserted type
;     NUMBER.
;   See also:
;     The SBCL Manual, Node "Handling of Types"
```

In this example we see each of the six possible parts of a compiler diagnostic:

- `file: /tmp/froq.lisp` is the name of the file that the compiler read the relevant code from. The file name is displayed because it may not be immediately obvious when there is an error during compilation of a large system, especially when `with-compilation-unit` is used to delay undefined warnings.
- `in: DEFUN FROQ` is the definition top level form responsible for the diagnostic. It is obtained by taking the first two elements of the enclosing form whose first element is a symbol beginning with `def`. If there is no such enclosing `def` form, then the outermost form is used. If there are multiple `def` forms, then they are all printed from the outside in, separated by `=>`s. In this example, the problem was in the `defun` for `froq`.
- `(zoq y)` is the *original source* form responsible for the diagnostic. Original source means that the form directly appeared in the original input to the compiler, i.e. in the lambda passed to `compile` or in the top level form read from the source file. In this example, the expansion of the `zoq` macro was responsible for the message.
- `--> roq ploq` This is the *processing path* that the compiler used to produce the code that caused the message to be emitted. The processing path is a representation of the evaluated forms enclosing the actual source that the compiler encountered when processing the original source. The path is the first element of each form, or the form itself if the form is not a list. These forms result from the expansion of macros or source-to-source transformation done by the compiler. In this example, the enclosing evaluated forms are the calls to `roq` and `ploq`. These calls resulted from the expansion of the `zoq` macro.
- `==> (+ y 3)` is the *actual source* responsible for the diagnostic. If the actual source appears in the explanation, then we print the next enclosing evaluated form, instead of printing the actual source twice. (This is the form that would otherwise have been the last form of the processing path.) In this example, the problem is with the evaluation of the reference to the variable `y`.
- The warning about the derived type `symbol` conflicting with the asserted type `number` is the *explanation* of the problem. In this example, the problem is that, while the call to `+` requires that its arguments are all of type `number`, the compiler has derived that Y will evaluate to a `symbol`. Note that `(values symbol &optional)` expresses that `y` evaluates to precisely one value.

Note that each part of the message is distinctively marked:

- `file:` and `in:` mark the file and definition, respectively.
- The original source is an indented form with no prefix.
- Each line of the processing path is prefixed with `-->`.
- The actual source form is indented like the original source, but is marked by a preceding `==>` line.
- The explanation is prefixed with the diagnostic severity, which can be `caught ERROR:`, `caught WARNING:`, `caught STYLE-WARNING:`, or `note:`.

After the type warning, the compiler prints the two undefined function warnings. Each part of the message is more specific than the preceding one. If consecutive messages are for nearby locations, then the front part of the messages would be the same. In this case, the compiler omits as much of the second message as is in common with the first. For example:

```
; in: DEFUN FROQ
;     (ZOQ Y)
; --> ROQ
; ==>
;   (PLOQ (+ Y 3))
;
; caught STYLE-WARNING:
;   undefined function: COMMON-LISP-USER::PLOQ
```

```
; ==>
;   (ROQ (PLOQ (+ Y 3)))
;
; caught STYLE-WARNING:
;   undefined function: COMMON-LISP-USER::ROQ
```

In this example, the file, definition and original source are identical for the two messages, so the compiler omits them in the second message. If consecutive messages are entirely identical, then the compiler prints only the first message, followed by: `[Last message occurs <repeats> times]` where `<repeats>` is the number of times the message was given.

If the source was not from a file, then no file line is printed. If the actual source is the same as the original source, then the processing path and actual source will be omitted. If no forms intervene between the original source and the actual source, then the processing path will also be omitted.

4.2.2.2 Original and Actual Source

The *original source* displayed will almost always be a list. If the actual source for a message is a symbol, the original source will be the immediately enclosing evaluated list form. So even if the offending symbol does appear in the original source, the compiler will print the enclosing list and then print the symbol as the actual source (as though the symbol were introduced by a macro.)

When the *actual source* is displayed (and is not a symbol), it will always be code that resulted from the expansion of a macro or a source-to-source compiler optimization. This is code that did not appear in the original source program; it was introduced by the compiler.

Keep in mind that when the compiler displays a source form in an diagnostic message, it always displays the most specific (innermost) responsible form. For example, compiling this function

```
(defun bar (x)
  (let (a)
    (declare (fixnum a))
    (setq a x)
    a))
```

gives this error message

```
; in: DEFUN BAR
;     (LET (A)
;       (DECLARE (FIXNUM A))
;       (SETQ A X)
;       A)
;
; note: deleting unreachable code
;
; caught WARNING:
;   Constant NIL conflicts with its asserted type FIXNUM.
```

This message is not saying that there is a problem somewhere in this `let` – it is saying that there is a problem with the `let` itself. In this example, the problem is that `a`'s `nil` initial value is not a `fixnum`.

4.2.2.3 Processing Paths

The processing path is mainly useful for debugging macros, so if you don't write macros, you can probably ignore it. Consider this example:

```
(defun frox (n)
  (dotimes (i n *undefined*)))
```

Compiling results in this error message:

```
; in: DEFUN FROX
;     (DOTIMES (I N *UNDEFINED*))
; --> DO BLOCK LET TAGBODY RETURN-FROM
; ==>
;   (PROGN *UNDEFINED*)
;
; caught WARNING:
;   undefined variable: COMMON-LISP-USER::*UNDEFINED*
```

Note that `do` appears in the processing path. This is because `dotimes` expands into:

```
(do ((i 0 (1+ i)) (#:g1 n))
    ((>= i #:g1) *undefined*)
  (declare (type unsigned-byte i)))
```

The rest of the processing path results from the macroexpansion of `do`:

```
(block nil
  (let ((i 0) (#:g1 n))
    (declare (type unsigned-byte i))
    (tagbody (go #:g3)
      #:g2    (psetq i (1+ i))
      #:g3    (unless (>= i #:g1) (go #:g2))
      (return-from nil (progn *undefined*)))))
```

In this example, the compiler descended into the `block`, `let`, `tagbody` and `return-from` to reach the `progn` printed as the actual source. This is a place where the "actual source appears in explanation" rule was applied. The innermost actual source form was the symbol `*undefined*` itself, but that also appeared in the explanation, so the compiler backed out one level.

4.2.3 Controlling Verbosity

The compiler can be quite verbose in its diagnostic reporting – rather more than some users would prefer. However, the amount of noise emitted can be controlled.

To control emission of compiler diagnostics (except those with Section 4.2.1 [diagnostic severity], page 19, error) use the `sb-ext:muffle-conditions` and `sb-ext:unmuffle-conditions` declarations, specifying the type of condition that is to be muffled (the muffling is done using an associated `muffle-warning` restart).

Global control:

```
;;; Muffle compiler-notes globally
(declaim (sb-ext:muffle-conditions sb-ext:compiler-note))
```

Local control:

```
;;; Muffle compiler-notes based on lexical scope
(defun foo (x)
  (declare (optimize speed) (fixnum x)
           (sb-ext:muffle-conditions sb-ext:compiler-note))
  (values (* x 5) ; no compiler note from this
    (locally
      (declare (sb-ext:unmuffle-conditions sb-ext:compiler-note))
      ;; this one gives a compiler note
      (* x -5))))
```

`sb-ext:muffle-conditions` [Declaration]

Syntax: `(sb-ext:muffle-conditions &rest types)`.

Silence the diagnostic messages that would be printed when a condition matching any of `types` is signalled at compile time.

`sb-ext:unmuffle-conditions` [Declaration]

Syntax: `(sb-ext:unmuffle-conditions &rest types)`.

Cancel the effect of a previous `sb-ext:muffle-conditions` declaration.

For information about muffling warnings signaled outside of the compiler, see Section 7.12 [customization hooks for users], page 79.

The following variables and macros control *how* the compiler messages are printed.

`sb-ext:*compiler-print-variable-alist*` [Variable]

An association list describing new bindings for special variables to be used by the compiler for error-reporting, etc. E.g. ((`*print-length*` . 10) (`*print-level*` . 6) (`*print-pretty*` . `nil`)).

The variables in the `car` positions are bound to the values in the `cdr` during the execution of some debug commands. When evaluating arbitrary expressions in the debugger, the normal values of the printer control variables are in effect.

Initially empty, `*compiler-print-variable-alist*` is typically used to specify bindings for printer control variables.

`sb-ext:*enclosing-source-cutoff*` [Variable]
The maximum number of enclosing actual source forms (i.e. from macroexpansion) that we print in full. For additional enclosing forms, we print only the `car`.

`sb-ext:*undefined-warning-limit*` [Variable]
If non-null, then an upper limit on the number of unknown function or type warnings that the compiler will print for any given name in a single compilation. This prevents excessive amounts of output when the real problem is a missing definition (as opposed to a typo in the use.)

`sb-ext:*efficiency-note-limit*` [Variable]
This is the maximum number of possible optimization alternatives will be mentioned in a particular efficiency note. `nil` means no limit.

`sb-ext:*efficiency-note-cost-threshold*` [Variable]
This is the minimum cost difference between the chosen implementation and the next alternative that justifies an efficiency note.

`sb-ext:*compile-progress*` [Variable]
When this is true, the compiler prints to `*standard-output*` progress information about the phases of compilation of each function. (This is useful mainly in large block-compilations.)

`sb-ext:define-source-context` *name lambda-list* &body *body* [Macro]
Define how to abbreviate a source context form whose `car` is `name` when it appears in the compiler input. `lambda-list` is a `defmacro` style lambda-list used to parse the form.

`body` should return a list of subforms that fit on one line when printed with the `"~{~S ~}"` format string. There are predefined extractors for `defstruct` and `function` forms. If there is no definition, then the first two subforms are returned.

4.3 Handling of Types

The most important features of the SBCL compiler (similar to the original CMUCL compiler) are its fairly sophisticated understanding of the Common Lisp type system and its conservative approach to the implementation of type declarations.

These two features reward the use of type declarations throughout development, even when high performance is not a concern. Also, as discussed in the chapter on performance (see Chapter 6 [efficiency], page 42), the use of appropriate type declarations can be very important for performance as well.

The SBCL compiler also has a greater knowledge of the Common Lisp type system than other compilers. Support is incomplete only for types involving the `satisfies` type specifier.

4.3.1 Type Declaration Checking

The SBCL compiler treats type declarations differently from most other Lisp compilers. Under the default Section 4.4 [compiler policy], page 25, the compiler doesn't blindly believe type declarations but considers them assertions about the program that should be checked: all type declarations that have not been proven to always hold are asserted at runtime.

Remaining bugs in the compiler's handling of types unfortunately provide some exceptions to this rule. See Section 4.3.4 [implementation limitations], page 25.

There are three type checking policies available in SBCL, selectable via `optimize` declarations.

- **Safe code** `(>= safety 1)`

  Full type checks. All declarations are considered assertions to be checked at compile or run time, and all type checks are precise. The default compilation policy provides full type checks.

- **Unsafe code** `(= safety 0)`

  No type checks. All declarations are believed without assertions. Also disables argument count and array bounds checking.

  > **Warning**: Any type errors in code where type checks are not performed are liable to corrupt the heap.

There is a third option called Weak Type Checking, where declared types may be simplified into faster to check supertypes: for example, `(or (integer -17 -7) (integer 7 17))` is simplified into `(integer -17 17)`. This option is also unsafe and is no longer available by any combination of the standard optimization qualities.

4.3.2 Precise Type Checking

Precise checking means that the check is done as though `typep` had been called with the exact type specifier that appeared in the declaration.

If a variable is declared to be `(integer 3 17)`, then its value must always be an integer between `3` and `17`. If multiple type declarations apply to a single variable, then all the declarations must be correct; it is as though all the types were intersected producing a single `and` type specifier.

To gain maximum benefit from the compiler's type checking, you should always declare the types of function arguments and structure slots as precisely as possible. This often involves the use of `or`, `member`, and other list-style type specifiers.

4.3.3 Getting Existing Programs to Run

Since SBCL's compiler does much more comprehensive type checking than most Lisp compilers, SBCL may detect type errors in programs that have been debugged using other compilers. These errors are mostly incorrect declarations, although compile-time type errors can find actual bugs if parts of the program have never been tested.

Some incorrect declarations can only be detected by run-time type checking. It is very important to initially compile a program with full type checks (high `safety` optimization) and then test this safe version. After the checking version has been tested, then you can consider weakening or eliminating type checks. *This applies even to previously debugged programs* because the SBCL compiler does much more type inference than other Common Lisp compilers, so an incorrect declaration can do more damage.

The most common problem is with variables whose constant initial value doesn't match the type declaration. Incorrect constant initial values will always be flagged by a compile-time type error, and they are simple to fix once located. Consider this code fragment:

```
(prog (foo)
  (declare (fixnum foo))
  (setq foo ...)
  ...)
```

Here `foo` is given an initial value of `nil` but is declared to be a `fixnum`. Even if it is never read, the initial value of a variable must match the declared type. There are two ways to fix this problem. Change the declaration

```
(prog (foo)
  (declare (type (or fixnum null) foo))
  (setq foo ...)
  ...)
```

or change the initial value

```
(prog ((foo 0))
  (declare (fixnum foo))
  (setq foo ...)
  ...)
```

It is generally preferable to change to a legal initial value rather than to weaken the declaration, but sometimes it is simpler to weaken the declaration than to try to make an initial value of the appropriate type.

Another declaration problem occasionally encountered is incorrect declarations on `defmacro` arguments. This can happen when a function is converted into a macro. Consider this macro:

```
(defmacro my-1+ (x)
  (declare (fixnum x))
  `(the fixnum (1+ ,x)))
```

Although legal and well-defined Common Lisp code, the meaning of this definition is almost certainly not what the writer intended. For example, this call is illegal:

```
(my-1+ (+ 4 5))
```

This call is illegal because the argument to the macro is `(+ 4 5)`, which is a `list`, not a `fixnum`. Because of macro semantics, it is hardly ever useful to declare the types of macro arguments. If you really want to assert something about the type of the result of evaluating a macro argument, then put a `the` in the expansion:

```
(defmacro my-1+ (x)
  `(the fixnum (1+ (the fixnum ,x))))
```

In this case, it would be stylistically preferable to change this macro back to a function and declare it inline.
See Section 4.6 [open coding and inline expansion], page 27.

Some more subtle problems are caused by incorrect declarations that can't be detected at compile time. Consider this code:

```
(do ((pos 0 (position #a string :start (1+ pos))))
  ((null pos))
  (declare (fixnum pos))
  ...)
```

Although `pos` is almost always a `fixnum`, it is `nil` at the end of the loop. If this example is compiled with full type checks (the default), then running it will signal a type error at the end of the loop. If compiled without type checks, the program will go into an infinite loop (or perhaps `position` will complain because `(1+ nil)` isn't a sensible start.) Why? Because if you compile without type checks, the compiler just quietly believes the type declaration. Since the compiler believes that `pos` is always a `fixnum`, it believes that `pos` is never `nil`, so `(null pos)` is never true, and the loop exit test is optimized away. Such errors are sometimes flagged by unreachable code notes, but it is still important to initially compile and test any system with full type checks, even if the system works fine when compiled using other compilers.

In this case, the fix is to weaken the type declaration to `(or fixnum null)`. (Actually, this declaration is unnecessary in SBCL, since it already knows that `position` returns a non-negative `fixnum` or `nil`.)

Note that there is usually little performance penalty for weakening a declaration in this way. Any numeric operations in the body can still assume that the variable is a `fixnum` since `nil` is not a legal numeric argument. Another possible fix would be to say:

```
(do ((pos 0 (position #a string :start (1+ pos))))
    ((null pos))
  (let ((pos pos))
    (declare (fixnum pos))
    ...))
```

This would be preferable in some circumstances since it would allow a non-standard representation to be used for the local `pos` variable in the loop body.

4.3.4 Implementation Limitations

If an `ftype` is placed after the function definition, the function won't perform any type checks and the calls to the function will blindly trust the declared types. (`optimize` (`debug` 3)) will not trust any `ftype` declarations.

CLOS slot types form a notable exception. Types declared using the `:type` slot option in `defclass` are asserted if and only if the class was defined in *safe code* and the slot access location is in *safe code* as well. This laxness does not pose any internal consistency issues, as the CLOS slot types are not available for the type inferencer, nor do CLOS slot types provide any efficiency benefits.

4.4 Compiler Policy

The compiler policy is what tells the compiler *how* to compile a program. This is logically (and often textually) distinct from the program itself. Broad control of policy is provided by the `optimize` declaration; other declarations and variables control more specific aspects of compilation.

The `optimize` declaration recognizes six different *qualities*. The qualities are conceptually independent aspects of program performance. In reality, increasing one quality tends to have adverse effects on other qualities. The compiler compares the relative values of qualities when it needs to make a trade-off; i.e. if `speed` is greater than `safety`, then improve speed at the cost of safety.

All qualities take values of type `(integer 0 3)`. The default for all qualities is `1`. Whenever qualities are equal, ties are broken according to a broad idea of what a good default environment is supposed to be. Generally this downplays `speed`, `compilation-speed` and `space` in favour of `safety` and `debug`. Novice and casual users should stick to the default policy. Advanced users often want to improve speed and memory usage at the cost of safety and debuggability.

If the value for a quality is `0` or `3`, then it may have a special interpretation. A value of `0` means totally unimportant, and a 3 means ultimately important. These extreme optimization values enable "heroic" compilation strategies that are not always desirable and sometimes self-defeating. Specifying more than one quality as `3` is not desirable, since it doesn't tell the compiler which quality is most important.

These are the optimization qualities:

- `speed`: How fast the program should run. `speed 3` enables some optimizations that may hurt debuggability. Ordinarily, when the `speed` quality is high, the compiler emits notes to notify the programmer about its inability to apply various optimizations. For selective muffling of these notes, see Section 4.2.3 [controlling verbosity], page 22.
- `compilation-speed`: How fast the compiler should run. If this is high, then some expensive optimizations may be disabled.
- `space`: How much space the compiled code should take up. A high value makes `inline` functions semi-inlined, and `0` enables semi-inlining of `sb-ext:maybe-inline` functions (see Section 6.5.2 [semi inline expansion], page 52).
- `debug`: How debuggable the program should be. The quality is treated differently from the other qualities: each value indicates a particular level of debugger information; it is not compared with the other qualities. See Section 5.1.6 [debug information], page 35, for the details.
- `safety`: How much error checking should be done. If `speed`, `space` or `compilation-speed` is more important than `safety`, then Section 4.3.1 [type declaration checking], page 23, is weakened. `safety 0` also suppresses argument count checking, unbound-symbol checking, array bounds checks, and checking of tag existence in `return-from` and `go`.
- `sb-ext:inhibit-warnings`: This is a deprecated extension that determines how little (or how much) diagnostic output should be printed during compilation. This quality is compared to other qualities to determine whether to print style notes and warnings concerning those qualities. If `speed` is greater than `inhibit-warnings`, then notes about how to improve speed will be printed, etc. The default value is `1`, so raising the value for any standard quality above its default enables notes for that quality. If it is `3`, then all notes and most non-serious warnings are inhibited. This is useful with `declare` to suppress warnings about unavoidable problems.

See also the `:policy` argument of `with-compilation-unit`.

In the following, we discuss extensions related to compiler policies.

`sb-ext:describe-compiler-policy` &optional *spec* [Function]
Print all global optimization settings, augmented by `spec`.

`sb-ext:restrict-compiler-policy` &optional *quality* (*min 0*) (*max 3*) [Function]
Restrict the range of values the optimization `quality` can take. The effect is that `optimize` declarations with `quality` are clamped to the restricted range. `quality` must be a valid optimization quality or `nil`. `min` is the minimum allowed value, and `max` is the maximum; both of type (`integer` 0 3).

Returns two alists describing the current minimum and maximum policy restrictions on all optimization qualities.

If `quality` is `nil` or not given, nothing is done. Use this to query the current restrictions.

If `quality` is non-`nil`, `min` is 0 and `max` is 3, any existing restrictions on `quality` are removed. In general, restrictions can be loosened.

See also `:policy` option in `with-compilation-unit`, which can impose dynamic scoping on restrictions.

4.5 Compiler Errors

4.5.1 Type Errors at Compile Time

If the compiler can prove at compile time that some portion of the program cannot be executed without a type error, then it will give a `style-warning` at compile time. This sort of type warning happens moderately often in the expansion of complex macros and in inline functions.

For example, consider this code fragment:

```
(defun raz (foo)
  (let ((x (if foo 13 "str")))
    (declare (fixnum x))
    (print x)))
```

Compilation produces this warning:

```
; in: DEFUN RAZ
;     (LET ((X (IF FOO 13 "str")))
;       (DECLARE (FIXNUM X))
```

```
;       (PRINT X))
;
; caught STYLE-WARNING:
;   Constant "str" conflicts with its asserted type FIXNUM.
```

The warning means that if `foo` is `nil`, then `x` will be initialized to `"str"`, which the `fixnum` declaration makes illegal.

This warning doesn't always indicate an incorrect program because

- it is possible that the offending code would never actually be executed at run-time due to some higher level consistency constraint unknown to the compiler; and
- in rare cases, the type error is actually intended and part of the normal control flow.

  Note: In the common situation where the value triggering the type error is `nil`, this warning is suppressed.

4.5.2 Errors During Macroexpansion

The compiler handles errors that happen during macroexpansion, turning them into compiler errors. If you want to debug the error (to debug a macro), you can set `*break-on-signals*` to `error`. For example, this definition:

```
(defun foo (e l)
  (do ((current l (cdr current))
       ((atom current) nil))
      (when (eq (car current) e) (return current))))
```

gives this error:

```
; in: DEFUN FOO
;     (DO ((CURRENT L (CDR CURRENT))
;          ((ATOM CURRENT) NIL))
;         (WHEN (EQ (CAR CURRENT) E) (RETURN CURRENT)))
;
; caught ERROR:
;   (in macroexpansion of (DO # #))
;   (hint: For more precise location, try *BREAK-ON-SIGNALS*.)
;   DO step variable is not a symbol: (ATOM CURRENT)
```

4.5.3 Read Errors

SBCL's compiler does not attempt to recover from read errors when reading a source file but instead just reports the offending character position and gives up on the entire source file.

4.6 Open Coding and Inline Expansion

Since Common Lisp forbids the redefinition of standard functions, the compiler can have special knowledge of these standard functions embedded in it. This special knowledge is used in various ways (open coding, Section 6.5 [inline expansion], page 50, source transformation), but the implications to the user are basically the same:

- Attempts to redefine standard functions may be frustrated since the function may never be called. Although it is technically illegal to redefine standard functions, users sometimes want to implicitly redefine these functions when they are debugging using the `trace` macro. Special-casing of standard functions can be inhibited using the `notinline` declaration, but even then some phases of analysis such as type inferencing are applied by the compiler.
- The compiler can have multiple alternate implementations of standard functions that implement different trade-offs of speed, space and safety. This selection is based on the Section 4.4 [compiler policy], page 25.

When a function call is *open coded*, inline code whose effect is equivalent to the function call is substituted for that function call. When a function call is *closed coded*, it is usually left as is, although it might be turned into a call to a different function with different arguments. As an example, if `nthcdr` were to be open coded, then

```
(nthcdr 4 foobar)
```

might turn into

```
(cdr (cdr (cdr (cdr foobar))))
```

or even

```
(do ((i 0 (1+ i))
  (list foobar (cdr foobar)))
  ((= i 4) list))
```

If `nth` is closed coded, then

```
(nth x l)
```

might stay the same, or turn into something like

```
(car (nthcdr x l))
```

In general, open coding sacrifices space for speed, but some functions (such as `car`) are so simple that they are always open-coded. Even when not open-coded, a call to a standard function may be transformed into a different function call (as in the last example) or compiled as *static call*. Static function calls use a more efficient calling convention that forbids redefinition.

4.7 Interpreter

By default, SBCL implements `eval` by calling the native-code compiler. There are two, mutually exclusive interpreter implementations: sb-eval (the default) and sb-fasteval. These are intended for when using the compiler is undesirable, for example due to compilation overhead.

An interpreterless SBCL can be compiled by passing `--without-sb-eval` to `make.sh`, while sb-fasteval can be enabled with `--without-sb-eval --with-sb-fasteval` (see Section 2.3.1 [customizing the build], page 4).

Unlike in some other Lisp implementations, in SBCL, interpreted code is not safer or more debuggable than compiled code.

`sb-ext:*evaluator-mode*` [Variable]

Toggle between different evaluator implementations. If set to `:compile`, an implementation of `eval` that calls the compiler will be used. If set to `:interpret`, an interpreter will be used. `:interpret` is a valid value only if SBCL has been compiled with either the `sb-eval` or the `sb-fasteval` feature.

4.8 Advanced Compiler Use and Efficiency Hints

For more advanced usages of the compiler, please see the chapter of the same name in the CMUCL manual. Many aspects of the compiler have stayed exactly the same, and there is a much more detailed explanation of the compiler's behavior and how to maximally optimize code in their manual. In particular, while SBCL no longer supports byte-code compilation, it does support CMUCL's block compilation facility allowing whole program optimization and increased use of the local call convention.

Unlike CMUCL, SBCL is able to open-code forward-referenced type tests while block compiling. This helps with mutually referential `defstruct`s in particular.

5 Debugging

This chapter documents the debugging facilities of SBCL, including the debugger, single-stepper and `trace`, and the effect of `(optimize debug)` declarations.

5.1 Debugger

5.1.1 Debugger Entry

5.1.1.1 Debugger Banner

When you enter the debugger, it looks something like this:

```
* (symbol-value 'x)

debugger invoked on a UNBOUND-VARIABLE @B80027A0C1 in thread
#<THREAD tid=893088 "main thread" RUNNING {1200038003}>:
  The variable X is unbound.

Type HELP for debugger help, or (SB-EXT:EXIT) to exit from SBCL.

restarts (invokable by number or by possibly-abbreviated name):
  0: [CONTINUE   ] Retry using X.
  1: [USE-VALUE  ] Use specified value.
  2: [STORE-VALUE] Set specified value and use it.
  3: [ABORT      ] Exit debugger, returning to top level.

(SYMBOL-VALUE X)
0]
```

The first group of lines describe what the error was that put us in the debugger. This is followed by the "beginner help line", which appears only if `sb-debug:*debug-beginner-help-p*` is true (default).

Next comes a listing of the active restart names, along with their descriptions – the ways we can restart execution after this error. Restarts can be selected by entering the corresponding number or name. If the name coincides with the name of a debugger command or that of another restart, use its number.

The current frame appears right underneath the restarts, immediately followed by the debugger prompt, where `0` is the index of the current frame.

`sb-debug:*debug-beginner-help-p*` [Variable]

Should entering the debugger print a beginner-oriented help message?

5.1.1.2 Debugger Invocation

The debugger is invoked when

- `error` is called and the condition it signals is not handled, or
- `break` is called, or
- `signal` is called with a condition that matches the current `*break-on-signals*`, or
- `invoke-debugger` is called.

Unless the debugger is to be entered by `break`, ANSI mandates that the value of `*debugger-hook*`, if non-`nil`, be called with two arguments: the condition that caused the debugger to be invoked and the previous value of `*debugger-hook*`. When this happens, `*debugger-hook*` is bound to `nil` to prevent recursive errors. However, ANSI also mandates that `*debugger-hook*` not be invoked when the debugger is to be entered by the `break` function. For users who wish to provide an alternate debugger interface (and thus catch `break` entries into the debugger), SBCL provides `sb-ext:*invoke-debugger-hook*`, which is invoked during any entry into the debugger.

`sb-ext:*invoke-debugger-hook*` [Variable]

This is either `nil` or a designator for a function of two arguments, to be run when the debugger is about to be entered. The function is run with `*invoke-debugger-hook*` bound to `nil` to minimize recursive

errors, and receives as arguments the condition that triggered debugger entry and the previous value of `*invoke-debugger-hook*`.

This mechanism is an SBCL extension similar to the standard `*debugger-hook*`. In contrast to `*debugger-hook*`, it is observed by `invoke-debugger` even when called by `break`.

5.1.2 Debugger Command Loop

The debugger is an interactive read-eval-print loop much like the normal top level, but some symbols are interpreted as debugger commands instead of being evaluated. A debugger command starts with the symbol name of the command, possibly followed by some arguments on the same line. Some commands prompt for additional input. Debugger commands can be abbreviated by any unambiguous prefix: `help` can be typed as `h`, `he`, etc.

The package is not significant in debugger commands; any symbol with the name of a debugger command will work. If you want to show the value of a variable that happens also to be the name of a debugger command you can wrap the variable in a `progn` to hide it from the command loop.

The debugger prompt is `<frame>]`, where `<frame>` is the number of the current frame. Frames are numbered starting from zero at the top (most recent call), increasing down to the bottom. The current frame is the frame that commands refer to.

The debugger can be invoked again within a debugger session following the normal rules (see Section 5.1.1.2 [debugger invocation], page 29). In such nested debugger sessions, the prompt is `<frame>[<nesting-level>]`.

It is possible to override the normal printing behaviour in the debugger by using the `sb-ext:*debug-print-variable-alist*`.

`sb-ext:*debug-print-variable-alist*` [Variable]

An association list describing new bindings for special variables to be used within the debugger. For example,

((`*print-length*` . 10) (`*print-level*` . 6) (`*print-pretty*` . `nil`))

The variables in the `car` positions are bound to the values in the `cdr` during the execution of some debug commands. When the same variable has multiple bindings specified, the earliest one takes precendence.

Entering the debugger establishes these bindings, as does the `backtrace` debugger command (see Section 5.1.8 [information commands], page 36) among other things.

Initially empty, `*debug-print-variable-alist*` is typically used to provide bindings for printer control variables.

5.1.3 Stack Frames

A *stack frame* is the run-time representation of a call to a function; the frame stores the state that a function needs to remember what it is doing. Frames have:

- *Variables* (see Section 5.1.4 [variable access], page 32), which are the values being operated on.
- *Arguments* to the call (which are really just particularly interesting variables).
- A current source location (Section 5.1.5 [source location printing], page 34), which is the place in the program where the function was running when it stopped to call another function, or because of an interrupt or error.

5.1.3.1 Stack Motion

These commands move to a new stack frame and print the name of the function and the values of its arguments in the style of a Lisp function call:

- `up`: Move up to the next higher frame. More recent function calls are considered to be higher on the stack.
- `down`: Move down to the next lower frame.
- `bottom`: Move to the lowest frame.
- `frame [<n>]`: Move to the frame with the specified number. Prompts for the number if not supplied. The frame with number 0 is the frame where the debugger was entered.

5.1.3.2 How Arguments are Printed

A frame is printed to look like a function call but with the actual argument values in the argument positions. So the frame for this call in the source:

```
(myfun (+ 3 4) 'a)
```

would look like this:

```
(MYFUN 7 A)
```

All keyword and optional arguments are displayed with their actual values; if the corresponding argument was not supplied, the value will be the default. So this call:

```
(subseq "foo" 1)
```

would look like this:

```
(SUBSEQ "foo" 1 3)
```

And this call:

```
(string-upcase "test case")
```

would look like this:

```
(STRING-UPCASE "test case" :START 0 :END NIL)
```

The arguments to a function call are displayed by accessing the argument variables. Although those variables are initialized to the actual argument values, they can be set inside the function; in this case the new value will be displayed.

`&rest` arguments are handled somewhat differently. The value of the rest argument variable is displayed as the spread-out arguments to the call, so:

```
(format t "~A is a ~A." "This" 'test)
```

would look like this:

```
(FORMAT T "~A is a ~A." "This" 'TEST)
```

Rest arguments cause an exception to the normal display of keyword arguments in functions that have both `&rest` and `&key` arguments. In this case, the keyword argument variables are not displayed at all; the rest arg is displayed instead. So for these functions, only the keywords actually supplied will be shown, and the values displayed will be the argument values, not values of the (possibly modified) variables.

If the variable for an argument is never referenced by the function, it will be deleted. The variable value is then unavailable, so the debugger prints `#<unused-arg>` instead of the value. Similarly, if for any of a number of reasons the value of the variable is unavailable or not known to be available (Section 5.1.4 [variable access], page 32), then `#<unavailable-arg>` will be printed instead of the argument value.

Note that Section 4.6 [open coding and inline expansion], page 27, affect what frames are present in the debugger, see Section 5.1.6 [debug information], page 35.

5.1.3.3 Function Names

If a function is defined by `defun` it will appear in backtrace by that name. Functions defined by `labels` and `flet` will appear as `(flet <name>)` and `(labels <name>)` respectively. Anonymous lambdas will appear as `(lambda <lambda-list>)`.

5.1.3.4 Entry Point Details

Sometimes the compiler introduces new functions that are used to implement a user function but are not directly specified in the source. This is mostly done for argument type and count checking.

With recursive or block compiled functions, an additional `external` frame may appear before the frame representing the first call to the recursive function or entry to the compiled block. This is a consequence of the way the compiler works: there is nothing odd with your program. You may also see `cleanup` frames during the execution of `unwind-protect` cleanup code, and `optional` for variable argument entry points.

5.1.3.5 Debugging Tail Calls

Section 6.2.1 [tail call optimization], page 46, can eliminate frames. Consider this backtrace:

```
(BAR ...)
(FOO ...)
```

Because of tail calls, it is not necessarily the case that `foo` directly called `bar`. It may be that `foo` called some other function `foo2`, which then tail-called `bar`, as in this example:

```
(defun foo ()
```

```
  ...
  (foo2 ...)
  ...)

(defun foo2 (...)
  ...
  (bar ...))

(defun bar (...)
  ...)
```

If there is any doubt about how one function called another, it can usually be eliminated by finding the source location in the calling frame. See Section 5.1.5 [source location printing], page 34.

5.1.3.6 Unknown Locations and Interrupts

The debugger operates using special debugging information attached to the compiled code. This debug information tells the debugger what it needs to know about the locations in the code where the debugger can be invoked. If the debugger somehow encounters a location not described in the debug information, then it is said to be *unknown*. If the code location for a frame is unknown, then some variables may be inaccessible, and the source location cannot be precisely displayed.

There are three reasons why a code location could be unknown:

- There is inadequate Section 5.1.6 [debug information], page 35, due to the value of the `debug` optimization quality.
- The debugger was entered because of an interrupt such as `C-c`.
- A hardware error such as a bus error occurred in code that was compiled unsafely due to the value of the `safety` optimization quality. See Section 4.4 [compiler policy], page 25.

In the last two cases, the values of argument variables are accessible, but may be incorrect. For more details on when variable values are accessible, see Section 5.1.4.1 [variable value availability], page 33.

It is possible for an interrupt to happen when a function call or return is in progress. The debugger may then flame out with some obscure error or insist that the bottom of the stack has been reached, when the real problem is that the current stack frame can't be located. If this happens, return from the interrupt and try again.

5.1.4 Variable Access

There are two ways to access the current frame's local variables in the debugger: `list-locals` and `sb-debug:var`.

The debugger doesn't really understand lexical scoping; it has just one namespace for all the variables in the current stack frame. If a symbol is the name of multiple variables in the same function, then the reference appears ambiguous, even though lexical scoping specifies which value is visible at any given source location. If the scopes of the two variables are not nested, then the debugger can resolve the ambiguity by observing that only one variable is accessible.

When there are ambiguous variables, the evaluator assigns each one a small integer identifier. The `sb-debug:var` function uses this identifier to distinguish between ambiguous variables. The `list-locals` command prints the identifier. In the following example, there are two variables named `x`. The first one has identifier 0 (which is not printed), the second one has identifier 1.

```
X  =  1
X#1  =  2
```

- `list-locals [<prefix>]`: This command prints the name and value of all variables in the current frame whose name has the specified `<prefix>`, which may be a string or a symbol. If no `<prefix>` is given, then all available variables are printed. If a variable has a potentially ambiguous name, then the name is printed with a `#<identifier>` suffix, where `<identifier>` is the small integer used to make the name unique.

`sb-debug:var` *name* &optional (*id 0*) [Function]

Return a variable's value if possible. `name` is a simple-string or symbol. If it is a simple-string, it is an initial substring of the variable's name. If name is a symbol, it has the same name and package as the

variable whose value this function returns. If the symbol is uninterned, then the variable has the same name as the symbol, but it has no package.

If name is the initial substring of variables with different names, then this returns no values after displaying the ambiguous names. If name determines multiple variables with the same name, then you must use the optional `id` argument to specify which one you want. If you left `id` unspecified, then this returns no values after displaying the distinguishing id values.

The result of this function is limited to the availability of variable information. This is `setf`able.

5.1.4.1 Variable Value Availability

The value of a variable may be unavailable to the debugger in portions of the program where Lisp says that the variable is defined. If a variable value is not available, the debugger will not let you read or write that variable. With one exception, the debugger will never display an incorrect value for a variable. Rather than displaying incorrect values, the debugger tells you the value is unavailable.

The one exception is this: if you interrupt (e.g. with `C-c`) or if there is an unexpected hardware error such as a bus error (which should only happen in unsafe code; see Section 4.3.1 [type declaration checking], page 23), then the values displayed for arguments to the interrupted frame might be incorrect. This exception applies only to the interrupted frame: any frame farther down the stack will be fine.

> *Note*: Since the location of an interrupt or hardware error will always be an unknown location, non-argument variable values will never be available in the interrupted frame. See Section 5.1.3.6 [unknown locations and interrupts], page 32.)

If the necessary Section 5.1.6 [debug information], page 35, was not recorded, the value of a variable is unavailable. This can happen for the following reasons:

- The value of the `debug` optimization quality may have omitted debug information needed to determine whether the variable is available. Unless a variable is an argument, its value will only be available when `debug` is at least 2.
- The compiler did lifetime analysis and determined that the value was no longer needed, even though its scope had not been exited. Lifetime analysis is inhibited when the `debug` optimization quality is 3.
- The variable's name is an uninterned symbol (gensym). To save space, the compiler only dumps debug information about uninterned variables when the `debug` optimization quality is 3.

The value of a variables may be unavailable also for other reasons:

- The frame's location is unknown (see Section 5.1.3.6 [unknown locations and interrupts], page 32) because the debugger was entered due to an interrupt or unexpected hardware error. Under these conditions the values of arguments will be available, but might be incorrect. This is the exception mentioned above.
- The variable (or the code referencing it) was optimized out of existence. Variables with no reads are always optimized away. The degree to which the compiler deletes variables will depend on the value of the `compilation-speed` optimization quality, but most source-level optimizations are done under all compilation policies.
- The variable is never set and its definition looks like

  ```
  (let ((var1 var2))
    ...)
  ```

  In this case, `var1` is substituted with `var2`.
- The variable is never set and is referenced exactly once. In this case, the reference is substituted with the variable initial value.

To summarize, since it is especially useful to be able to get the arguments to a function, values of argument variables are available (even at unknown locations) when the `speed` optimization quality is less than 3 and the `debug` quality is at least 1. For non-argument variables, `debug` must be at least 2 for values to be available, and even then, values are only available at known locations.

5.1.4.2 Note On Lexical Variable Access

When the debugger command loop establishes variable bindings for available variables, these variable bindings have lexical scope and dynamic extent. You can close over them, but such closures can't be used as upward function arguments.

> *Note*: The variable bindings are actually created using the Lisp `symbol-macrolet` special form.

You can also set local variables using `setq`, but if the variable was closed over in the original source and never set, then setting the variable in the debugger may not change the value in all the functions the variable is defined in. Another risk of setting variables is that you may assign a value of a type that the compiler proved the variable could never take on. This may result in bad things happening.

5.1.5 Source Location Printing

One of the debugger's capabilities is source level debugging of compiled code. These commands display the source location for the current frame:

- `source [<context>]`: This command displays the file that the current frame's function was defined from (if it was defined from a file), and then the source form responsible for generating the code that the current frame was executing. If `<context>` is specified, then it is an integer specifying the number of enclosing levels of list structure to print.

The source form for a location in the code is the innermost list present in the original source that encloses the form responsible for generating that code. If the actual source form is not a list, then some enclosing list will be printed. For example, if the source form was a reference to the variable `*some-random-special*`, then the innermost enclosing evaluated form will be printed. Here are some possible enclosing forms:

```
(let ((a *some-random-special*))
  ...)

(+ *some-random-special* ...)
```

If the code at a location was generated from the expansion of a macro or a source-level compiler optimization, then the form in the original source that expanded into that code will be printed. Suppose the file `/usr/me/mystuff.lisp` looked like this:

```
(defmacro mymac ()
  '(myfun))

(defun foo ()
  (mymac)
  ...)
```

If `foo` has called `myfun`, and is waiting for it to return, then the `source` command would print:

```
; File: /usr/me/mystuff.lisp

(MYMAC)
```

Note that the macro use was printed, not the actual function call form, `(myfun)`.

If enclosing source is printed by giving an argument to `source` or `vsource`, then the actual source form is marked by wrapping it in a list whose first element is `#:***here***`. In the previous example, `source 1` would print:

```
; File: /usr/me/mystuff.lisp

(DEFUN FOO ()
  (#:***HERE***
   (MYMAC))
  ...)
```

5.1.5.1 How the Source is Found

If the code was defined from Lisp by `compile` or `eval`, then the source can always be reliably located. If the code was defined from a fasl file created by `compile-file`, then the debugger gets the source forms it prints by reading them from the original source file. This is a potential problem, since the source file might have moved or changed since the time it was compiled.

The source file is opened using the `truename` of the source file pathname originally given to the compiler. This is an absolute pathname with all logical names and symbolic links expanded. If the file can't be located using this name, then the debugger gives up and signals an error.

If the source file can be found, but has been modified since the time it was compiled, the debugger prints this warning:

```
; File has been modified since compilation:
```

```
;   <filename>
; Using form offset instead of character position.
```

where `<filename>` is the name of the source file. It then proceeds using a robust but not foolproof heuristic for locating the source. This heuristic works if:

- No top-level forms before the top-level form containing the source have been added or deleted, and
- the top-level form containing the source has not been modified much. (More precisely, none of the list forms beginning before the source form have been added or deleted.)

If the heuristic doesn't work, the displayed source will be wrong, but will probably be near the actual source. If the "shape" of the top-level form in the source file is too different from the original form, then an error will be signaled. When the heuristic is used, the source location commands are noticeably slowed.

Source location printing can also be confused if (after the source was compiled) a read-macro you used in the code was redefined to expand into something different, or if a read-macro ever returns the same `eq` list twice. If you don't define read macros and don't use `##` in perverted ways, you don't need to worry about this.

5.1.5.2 Source Location Availability

Source location information is only available when the `debug` optimization quality is at least 2. If source location information is unavailable, the source commands will give an error message.

If source location information is available, but the source location is unknown because of an interrupt or unexpected hardware error (see Section 5.1.3.6 [unknown locations and interrupts], page 32), then the command will print

```
Unknown location: using block start.
```

and then proceed to print the source location for the start of the *basic block* enclosing the code location. It's a bit complicated to explain exactly what a basic block is, but here are some properties of the block start location:

- The block start location may be the same as the true location.
- The block start location will never be later in the program's flow of control than the true location.
- No conditional control structures (such as `if`, `cond`, `or`) will intervene between the block start and the true location (but note that some conditionals present in the original source could be optimized away.) Function calls *do not* end basic blocks.
- The head of a loop will be the start of a block.
- The programming language concept of block structure and the Lisp `block` special form are totally unrelated to the compiler's basic block.

In other words, the true location lies between the printed location and the next conditional (but watch out because the compiler may have changed the program on you.)

5.1.6 Debug Information

The Section 4.4 [compiler policy], page 25, specified by `optimize` declarations affects the behavior seen in the debugger. The `debug` quality directly affects the debugger by controlling the amount of debugger information dumped. Other optimization qualities have indirect but observable effects due to changes in the way compilation is done.

Unlike the other optimization qualities (which are compared in relative value to evaluate tradeoffs), the `debug` optimization quality is directly translated to a level of debug information. This absolute interpretation allows the user to count on a particular amount of debug information being available, even when the values of the other qualities are changed during compilation. We list what debug information is available at different levels of the `debug` quality:

- `0<=`: Only the function name and enough information to allow the stack to be parsed.
- `1<=`: Argument variables. The value will only be available if the argument variable is never set and `speed` is not 3.

  Also, argument documentation (printed argument lists) and derived argument/result type information. This makes `describe` more informative and allows the compiler to do compile-time argument count and type checking for any calls compiled at run-time.
- `2<=`: Interned local variables, source location information, and lifetime information that tells the debugger when arguments are available (even when `speed` is 3 or the argument is set).

- `3`: Uninterned variables.
  - In addition, lifetime analysis is disabled (even when `speed` is 3), ensuring that all variable values are available at any known location within the scope of the binding. This has a speed penalty in addition to the obvious space penalty.
  - Section 6.2.1 [tail call optimization], page 46, is disabled. Thus, backtraces will contain frames for all invoked functions, even those in tail positions.

Inlining of local functions is inhibited so that they may be `trace`d.

- `> (max speed space)`: If `debug` is greater than both `speed` and `space`, the command `return` can be used to continue execution by returning a value from the current stack frame.
- `> (max speed space compilation-speed)`: If `debug` is greater than all of `speed`, `space` and `compilation-speed` the code will be steppable (see Section 5.1.10 [single stepping], page 38).

As you can see, if the `speed` quality is 3, debugger performance is degraded. This effect comes from the elimination of argument variable special-casing (see Section 5.1.4.1 [variable value availability], page 33). Some degree of speed/debuggability tradeoff is unavoidable, but the effect is not too drastic when `debug` is at least 2.

In addition to `inline` and `notinline` declarations, the relative values of the `speed` and `space` qualities also change whether functions are inline expanded. If a function is inline expanded, then there will be no frame to represent the call, and the arguments will be treated like any other local variable. Functions may also be semi inline (see Section 6.5.2 [semi inline expansion], page 52), in which case there is a frame to represent the call, but the call is to an optimized local version of the function, not to the original function. See Section 4.6 [open coding and inline expansion], page 27, for more.

5.1.7 Exiting the Debugger

These commands get you out of the debugger.

- `toplevel`: Throw to top level.
- `restart [<n>]` or `<n>`: Invoke the `<n>`th restart case as displayed by the `error` command. If `<n>` is not specified, the available restart cases are reported. When using the short form (`<n>`), an integer for which there is no corresponding restart is evaluated following normal REPL semantics.
- `continue`: Call `continue` with the condition given that invoked the debugger. If there is no restart case named `continue`, then an error is signaled. This is not a debugger command but the name of a commonly available restart.
- `abort`: Call `abort` on the condition that invoked the debugger. This is useful for popping debug command loop levels or aborting to top level, as the case may be. This is not a debugger command but the name of a commonly available restart.
- `return <value>`: Return `value` from the current stack frame. This command is available when the `debug` optimization quality is greater than both `speed` and `space`. Care must be taken that the value is of the same type as SBCL expects the stack frame to return.
- `restart-frame`: Restart execution of the current stack frame. This command is available when the `debug` optimization quality is greater than both `speed` and `space` and when the frame is for a global function. If the function is redefined in the debugger before the frame is restarted, the new function will be used.

5.1.8 Information Commands

Most of these commands print information about the current frame or function, but a few show general information.

- `help` or `?`: Display a synopsis of debugger commands.
- `describe`: Call `describe` on the current function and display the number of local variables.
- `print`: Display the current function call as it would be displayed by moving to this frame.
- `error`: Print the condition given to `invoke-debugger` and the active proceed cases.
- `backtrace [<n>]`: Display all the frames from the current to the bottom. Only shows `<n>` frames if specified. The printing is controlled by `sb-debug:*debug-print-variable-alist*`.

5.1.9 Breakpoint Commands

SBCL supports setting of breakpoints inside compiled functions and stepping of compiled code. Breakpoints can only be set at known locations (see Section 5.1.3.6 [unknown locations and interrupts], page 32), so these commands are largely useless unless the `debug` optimize quality is at least 2 (see Section 5.1.6 [debug information], page 35). These commands manipulate breakpoints:

- `breakpoint <location> [<option> <value>]*`: Set a breakpoint in some function. `<location>` may be an integer code location number (as displayed by `list-locations`) or a keyword. The keyword can be used to indicate setting a breakpoint at the function start (`:start`, `:s`) or function end (`:end`, `:e`). The `breakpoint` command has `:condition`, `:break`, `:print` and `:function` options which work similarly to the `trace` options.
- `list-locations [<function>]` or `ll [<function>]`: List all the code locations in the current frame's function, or in `<function>` if it is supplied. The display format is the code location number, a colon and then the source form for that location:

  ```
  3: (1- N)
  ```

  If consecutive locations have the same source, then a numeric range like `3-5:` will be printed. For example, a default function call has a known location both immediately before and after the call, which would result in two code locations with the same source. The listed function becomes the new default function for breakpoint setting (via the `breakpoint`) command.
- `list-breakpoints` or `lb`: List all currently active breakpoints with their breakpoint number.
- `delete-breakpoint [<number>]` or `db [<number>]`: Delete a breakpoint specified by its breakpoint number. If no number is specified, delete all breakpoints.
- `step*`: Step to the next possible breakpoint location in the current function. This always steps over function calls, instead of stepping into them.

5.1.9.1 Breakpoint Example

Consider this definition of the factorial function:

```
(defun ! (n)
  (if (zerop n)
      1
      (* n (! (1- n)))))
```

This debugger session demonstrates the use of breakpoints:

```
* (break)  ; invoke debugger

debugger invoked on a SIMPLE-CONDITION in thread 11184: break

restarts (invokable by number or by possibly-abbreviated name):
  0: [CONTINUE] Return from BREAK.
  1: [ABORT   ] Reduce debugger level (leaving debugger, returning to toplevel).
  2: [TOPLEVEL] Restart at toplevel READ/EVAL/PRINT loop.
("varargs entry for top level local call BREAK" "break")
0] ll #'!

0-1: (SB-INT:NAMED-LAMBDA ! (N) (BLOCK ! (IF (ZEROP N) 1 (* N (! #)))))
2: (BLOCK ! (IF (ZEROP N) 1 (* N (! (1- N)))))
3: (ZEROP N)
4: (* N (! (1- N)))
5: (1- N)
6: (! (1- N))
7-8: (* N (! (1- N)))
9-10: (IF (ZEROP N) 1 (* N (! (1- N))))
0] br 4

(* N (! (1- N)))
1: 4 in !
added
```

```
0] toplevel

> (! 10) ; Call the function

*Breakpoint hit*

Restarts:
  0: [CONTINUE] Return from BREAK.
  1: [ABORT   ] Return to Top-Level.

Debug  (type H for help)

(! 10) ; We are now in first call (arg 10) before the multiply
Source: (* N (! (1- N)))
3] step*

*Step*

(! 10) ; We have finished evaluation of (1- n)
Source: (1- N)
3] step*

*Breakpoint hit*

Restarts:
  0: [CONTINUE] Return from BREAK.
  1: [ABORT   ] Return to Top-Level.

Debug  (type H for help)

(! 9) ; We hit the breakpoint in the recursive call
Source: (* N (! (1- N)))
3]
```

Note: The `step*` command differs from the single stepping commands in that it also functions in compiled code which has not been compiled with stepping instrumentation. It simply steps to the next compiled code location. In the future, this form of stepping may be improved enough to subsume the instrumentation-based stepping commands, which have much higher overhead.

5.1.10 Single Stepping

SBCL includes an instrumentation-based single-stepper for compiled code, which can be invoked via the `step` macro or from within the debugger. See Section 5.1.6 [debug information], page 35, for details on enabling stepping for compiled code.

The following debugger commands are used for controlling single stepping.

- `start`: Select the `continue` restart if one exists and starts single stepping. None of the other single stepping commands can be used before stepping has been started either by using `start` or by using the standard `step` macro.
- `step`: Step into the current form. Stepping will be resumed when the next form that has been compiled with stepper instrumentation is evaluated.
- `next`: Step over the current form. Stepping will be disabled until evaluation of the form is complete.
- `out`: Step out of the current frame. Stepping will be disabled until the topmost stack frame that had been stepped into returns.
- `stop`: Stop the single stepper and resumes normal execution.

`step` *form* [Macro]
The form is evaluated with single stepping enabled. Function calls outside the lexical scope of the form can be stepped into only if the functions in question have been compiled with sufficient `debug` policy to be at least partially steppable.

5.2 LDB

LDB, the low-level debugger, is implemented in C and can work even when the Lisp Section 5.1 [debugger], page 29, cannot due to e.g. heap or stack exhaustion. Failed assertions in the low-level runtime (see Section 3.3 [command line options], page 11) trigger entering LDB if it's

- available (if SBCL was not compiled with `--without-sb-ldb`; see Section 2.3.1 [customizing the build], page 4) and
- not disabled (see `--disable-ldb` in Section 3.3.1 [runtime options], page 11, and `sb-ext:disable-debugger`).

Else, a backtrace of at most 100 stack frames is printed, and SBCL exits with a non-zero exit code.
The commands supported can be explored at the LDB prompt by typing `help` followed by an enter.
See the related `--lose-on-corruption` option in Section 3.3.1 [runtime options], page 11.

5.3 Enabling and Disabling the Debugger

In certain contexts (e.g. non-interactive applications), it may be desirable to turn off the SBCL debugger and Section 5.2 [ldb], page 39, (and possibly re-enable them).

`sb-ext:disable-debugger` [Function]
When invoked, this function will turn off both the Section 5.1 [debugger], page 29, and Section 5.2 [ldb], page 39, (the low-level debugger), similar to specifying `--disable-debugger` (see Section 3.3.2 [toplevel options], page 12) and `--disable-ldb` (see Section 3.3.1 [runtime options], page 11) on the command line. See also `enable-debugger`.

`sb-ext:enable-debugger` [Function]
Restore the Section 5.1 [debugger], page 29, and Section 5.2 [ldb], page 39, if they have been turned off by `disable-debugger`, `--disable-debugger` (see Section 3.3.2 [toplevel options], page 12) or `--disable-ldb` (see Section 3.3.1 [runtime options], page 11).

5.4 Function Tracing

The tracer causes selected functions to print their arguments and their results whenever they are called. Options allow conditional printing of the trace information and conditional breakpoints on function entry or exit.
In SBCL, tracing can be done either by temporarily redefining the function name (encapsulation) or using breakpoints. When breakpoints are used, the function object itself is destructively modified to cause the tracing action. The advantage of using breakpoints is that tracing works even when the function is anonymously called via `funcall`, that function object identity is preserved, and that anonymous and local functions can also be traced.

`trace` &rest *specs* [Macro]
`trace {<option> <global-value>}* {<name> {<option> <local-value>}*}*`
`trace` is a debugging tool that provides information when specified functions are called. In its simplest form:

```
(TRACE NAME-1 NAME-2 ...)
```

The `name`s are not evaluated. Each may be one of the following:

- `symbol`, denoting a function or macro.
- `fname`, a valid function name, denoting a function.
- `(method fname qualifiers* (specializers*))` denoting a method.
- `(compiler-macro symbol)` denoting a compiler macro.
- `(labels fname :in outer-name)` or `(flet fname :in outer-name)` denoting a local function where `outer-name` may be any of the previous names for functions, macros, methods or compiler macros. Tracing local functions may require `debug` policy 3 to inhibit inlining.

- `string` denoting all functions fbound to symbols whose home package is the package with the given name.

Options allow modification of the default behavior. Each option is a pair of an option keyword and a value form. Global options are specified before the first name, and affect all functions traced by a given use of `trace`. Options may also be interspersed with function names, in which case they act as local options, only affecting tracing of the immediately preceding function name. Local options override global options.
By default, `trace` causes a printout on `*trace-output*` each time that one of the named functions is entered or returns. (This is the basic, ANSI Common Lisp behavior of `trace`.)
The following options are defined:

- `:report <report-type>`

  If `report-type` is `trace` (the default) then information is reported by printing immediately. If `report-type` is `nil`, then the only effect of the trace is to execute other options (e.g. `print` or `break`). Otherwise, `report-type` is treated as a function designator and, for each trace event, funcalled with 5 arguments: trace depth (a non-negative integer), a function name or a function object, a keyword (`:enter`, `:exit` or `:non-local-exit`), a stack frame, and a list of values (arguments or return values).
- `:condition <form>`
- `:condition-after <form>`
- `:condition-all <form>`

  If `:condition` is specified, then `trace` does nothing unless `form` evaluates to true at the time of the call. `:condition-after` is similar, but suppresses the initial printout, and is tested when the function returns. `:condition-all` tries both before and after.
- `:break <form>`
- `:break-after <form>`
- `:break-all <form>`

  If specified, and `form` evaluates to true, then the debugger is invoked at the start of the function, at the end of the function, or both, according to the respective option.
- `:print <form>`
- `:print-after <form>`
- `:print-all <form>`

  In addition to the usual printout, the result of evaluating `form` is printed at the start of the function, at the end of the function, or both, according to the respective option. Multiple print options cause multiple values to be printed.
- `:wherein <names>`

  If specified, `names` is a function name or list of names. `trace` does nothing unless a call to one of those functions encloses the call to this function (i.e. it would appear in a backtrace.) Anonymous functions have string names like "DEFUN FOO".
- `:encapsulate` {`:default` | `t` | `nil`}

  If `t`, the default, tracing is done via encapsulation (redefining the function name) rather than by modifying the function. `:default` is not the default but means to use encapsulation for interpreted functions and funcallable instances, breakpoints otherwise. When encapsulation is used, forms are *not* evaluated in the function's lexical environment, but `sb-debug:arg` can still be used.
- `:methods` {`t` | `nil`}

  If `t`, any function argument naming a generic function will have its methods traced in addition to the generic function itself.
- `:function <function-form>`

  This is a not really an option but rather another way of specifying what function to trace. The `function-form` is evaluated immediately, and the resulting function is traced.

`:condition`, `:break` and `:print` forms are evaluated in a context which mocks up the lexical environment of the called function, so that `sb-debug:var` and `sb-debug:arg` can be used. The `*-after` and `*-all` forms can use also use `sb-debug:arg`. In forms which are evaluated after the function call, `(sb-debug:arg n)` returns the `n`th value returned by the function.

In the case of functions where the known return convention is used to optimize, encapsulation may be necessary in order to make tracing work at all. The symptom of this occurring is an error stating

```
Error in function FOO: :FUNCTION-END breakpoints are
currently unsupported for the known return convention.
```

in such cases we recommend using `(TRACE FOO :ENCAPSULATE t)`.

`untrace` &rest *specs* [Macro]
Remove tracing from the specified functions. Untraces all functions when called with no arguments.

`sb-debug:*trace-indentation-step*` [Variable]
The increase in trace indentation at each call level.

`sb-debug:*max-trace-indentation*` [Variable]
If the trace indentation exceeds this value, then indentation restarts at 0.

`sb-debug:*trace-encapsulate-default*` [Variable]
The default value for the `:encapsulate` option to `trace`.

`sb-debug:*trace-report-default*` [Variable]
The default value for the `:report` option to `trace`.

6 Efficiency

6.1 Source Optimization

This section describes source-level transformations that the SBCL compiler does on programs in an attempt to make them more efficient. Although source-level optimizations can make existing programs more efficient, the biggest advantage of this sort of optimization is that it makes it easier to write efficient programs. If a clean, straightforward implementation can be transformed into an efficient one, then there is no need for tricky and dangerous hand optimization.

6.1.1 Let Optimization

The primary optimization of let variables is to delete them when they are unnecessary. Whenever the value of a let variable is a constant, a constant variable or a constant (local or non-notinline) function, the variable is deleted, and references to the variable are replaced with references to the constant expression. This is useful primarily in the expansion of macros or inline functions, where argument values are often constant in any given call but are, in general, non-constant expressions that must be bound to preserve order of evaluation. Let variable optimization eliminates the need for macros to carefully avoid spurious bindings and also makes inline functions just as efficient as macros.

A particularly interesting class of constant is a local function. Substituting for lexical variables that are bound to a function can substantially improve the efficiency of functional programming styles. For example,

```
(let ((a #'(lambda (x) (zow x))))
  (funcall a 3))
```

effectively transforms to

```
(zow 3)
```

This transformation is done even when the function is a closure, as in

```
(let ((a (let ((y (zug)))
           #'(lambda (x) (zow x y)))))
  (funcall a 3))
```

becoming

```
(zow 3 (zug))
```

A constant variable is a lexical variable that is never assigned to, always keeping its initial value. Whenever possible, avoid setting lexical variables – instead bind a new variable to the new value. Except for loop variables, it is almost always possible to avoid setting lexical variables. This form:

```
(let ((x (f x)))
  ...)
```

is more efficient than

```
(setq x (f x))
...
```

Setting variables makes the program more difficult to understand, both to the compiler and to the programmer. SBCL compiles assignments very efficiently, but most let optimizations are only done on constant variables.

Constant variables with only a single use are also optimized away, even when the initial value is not constant (modulo side effects). For example, this expansion of `incf`:

```
(let ((#:g3 (+ x 1)))
  (setq x #:g3))
```

becomes

```
(setq x (+ x 1))
```

The type semantics of this transformation are more important than the elimination of the variable itself. Consider what happens when `x` is declared to be a `fixnum`. After the transformation, the compiler can infer that a non-fixnum result would be a type error. This allows it to completely eliminate the code path for `bignum`s, restricting the math exclusively to fast, register-level operations (though an overflow check may still be retained depending on Section 4.3.1 [type declaration checking], page 23).

Another variable optimization deletes any variable that is never read. This causes the initial value and any assigned values to be unused, allowing those expressions to be deleted if they have no side-effects.

Note that a `let` is actually a degenerate case of Section 6.3 [local call], page 47, and that let optimization can be done on calls that weren't created by a `let`. Also, local call allows an applicative style of iteration that is totally assignment-free.

6.1.2 Constant Folding

Constant folding is an optimization that replaces a call of constant arguments with the constant result of that call. Constant folding is done on all standard functions for which it is legal. Inline expansion allows folding of any constant parts of the definition and can be done even on functions that have side-effects.
It is convenient to rely on constant folding when programming, as in this example:

```
(defconstant limit 42)

(defun foo ()
  (... (1- limit) ...))
```

Constant folding is also helpful when writing macros or inline functions, since it usually eliminates the need to write a macro that special-cases constant arguments.

6.1.3 Unused Expression Elimination

If the value of any expression is not used, and the expression has no side-effects, then it is deleted. As with Section 6.1.2 [constant folding], page 43, this optimization applies most often when cleaning up after inline expansion and other optimizations.
Note that the compiler will eliminate parts of unused expressions known to be side-effect free, even if there are other unknown parts. For example, `list` has no side-effects, so

```
(let ((a (list (foo) (bar))))
  (if t
      (zow)
      (raz a)))
```

becomes

```
(progn (foo) (bar))
(zow)
```

6.1.4 Control Optimization

The most important optimization of control is recognizing when an `if` test is known at compile time, then deleting the `if`, the test expression, and the unreachable branch. This can be considered a special case of constant folding, although the test doesn't have to be truly constant as long as it is definitely false or definitely not false. Note also, that type inference propagates the result of an `if` test to the true and false branches.
A related `if` optimization transforming

```
(if (if a b c) x y)
```

into

```
(if a
    (if b x y)
    (if c x y))
```

The opportunity for this sort of optimization usually results from a conditional macro. For example,

```
(if (not a) x y)
```

is actually implemented as

```
(if (if a nil t) x y)
```

which is transformed to

```
(if a
    (if nil x y)
    (if t x y))
```

which is then optimized to

```
(if a y x)
```

Note that due to the compiler's internal representations, the IF-IF situation will be recognized even if other forms are wrapped around the inner `if` as in

```
(if (let ((g ...))
```

```
    (loop
      ...
      (return (not g))
      ...))
  x y)
```

In SBCL, all the Common Lisp macros really are macros written in terms of `if`, `block` and `tagbody`, so user-defined control macros can be just as efficient as the standard ones. The compiler emits basic blocks using a heuristic that minimizes the number of unconditional branches. The code in a `tagbody` will not be emitted in the order it appeared in the source, so there is no point in arranging the code to make control drop through to the target.

6.1.5 Dead Code Elimination

The compiler will delete code whenever it can prove that the code can never be executed. Code becomes unreachable when

- an `if` is optimized away, or
- there is an explicit unconditional control transfer such as `go` or `return-from`, or
- the last reference to a local function is deleted (or there never was any reference).

When code that appeared in the original source is deleted, the compiler prints a note to indicate a possible problem (or at least unnecessary code.) For example:

```
(defun foo ()
  (if t
      (write-line "True.")
      (write-line "False.")))
```

will result in this note:

```
; in: DEFUN FOO
;     (WRITE-LINE "False.")
;
; note: deleting unreachable code
```

It is important to pay attention to unreachable code notes, since they often indicate a subtle type error. For example,

```
(defstruct foo a b)

(defun lose (x)
  (let ((a (foo-a x))
        (b (if x (foo-b x) "never")))
    (values a b)))
```

results in this note:

```
; in: DEFUN LOSE
;     (IF X
;         (FOO-B X)
;         "never")
; ==>
;   "never"
;
; note: deleting unreachable code
```

The `"never"` branch is unreachable because type inference knows that the argument to `foo-a` must be a `foo` and thus can't be `nil`. Presumably the programmer forgot that `x` could be `nil` when writing the binding for `a`.

Here is an example with an incorrect declaration:

```
(defun count-a (string)
  (do ((pos 0 (position #\a string :start (1+ pos)))
       (count 0 (1+ count)))
      ((null pos) count)
    (declare (fixnum pos))))
```

This time our note is:

```
; in: DEFUN COUNT-A
;     (DO ((POS 0 (POSITION #\a STRING :START (1+ POS)))
;          (COUNT 0 (1+ COUNT)))
;         ((NULL POS) COUNT)
;       (DECLARE (FIXNUM POS)))
; --> BLOCK LET TAGBODY RETURN-FROM PROGN
; ==>
;   COUNT
;
; note: deleting unreachable code
```

The problem here is that `pos` can never be null since it is declared a `fixnum`.

It takes some experience with unreachable code notes to be able to tell what they are trying to say. In non-obvious cases, the best thing to do is to call the function in a way that should cause the unreachable code to be executed. Either you will get a type error, or you will find that there truly is no way for the code to be executed.

Not all unreachable code results in a note:

- A note is only given when the unreachable code textually appears in the original source. This prevents spurious notes due to the optimization of macros and inline functions but sometimes also foregoes a note that would have been useful.
- Since accurate source information is not available for non-list forms, there is an element of heuristic in determining whether or not to give a note about an atom. Spurious notes may be given when a macro or inline function defines a variable that is also present in the calling function. Notes about easily introduced constants are never given since it is too easy to confuse them in expanded code with ones in the original source. These constants are `nil`, `t`, keywords and unboxed values such as `fixnum`s, `character`s, and `single-float`s on 64-bit platforms.
- Notes are only given about code unreachable due to control flow. There is no note when an expression is deleted because its value is unused, since this is a common consequence of other optimizations.

6.1.6 Multiple Values Optimization

Within a function, the compiler implements uses of multiple values particularly efficiently. Multiple values can be kept in arbitrary registers, so using multiple values doesn't imply stack manipulation and representation conversion. For example,

```
(let ((a (if x (foo x) u))
      (b (if x (bar x) v)))
  ...)
```

is actually more efficient when written as

```
(multiple-value-bind (a b)
    (if x
        (values (foo x) (bar x))
        (values u v))
  ...)
```

Also, see Section 6.3.5 [local call return values], page 49, for information on how local call provides efficient support for multiple function return values.

6.1.7 Style Recommendations

Source level optimization makes possible a clearer and more relaxed programming style:

- Don't use macros purely to avoid function call. If you want an inline function, write it as a function and declare it `inline`. It's clearer, less error-prone, and works just as well.
- Don't write macros that try to "optimize" their expansion in trivial ways such as avoiding binding variables for simple expressions. The compiler does these optimizations too and is less likely to make a mistake.
- Make use of local functions (i.e. `labels` and `flet`) and Section 6.2 [tail call], page 46s in places where it is clearer. Section 6.3 [local call], page 47, is faster than full call.

- Avoid setting local variables when possible. Binding a new `let` variable is at least as efficient as setting an existing variable and is easier to understand, both for the compiler and the programmer.
- Instead of writing similar code over and over again so that it can be hand customized for each use, define a macro or inline function, and let the compiler do the work.

6.2 Tail Call

A call to a function is a tail call if nothing has to be done after the call returns. That is, when the call returns, the returned values are immediately returned from the calling function. Tail calls can be compiled to much more efficient code than normal code. See `https://en.wikipedia.org/wiki/Tail_call` for more. Note that both Section 6.3 [local call], page 47s and full calls can be tail calls.

6.2.1 Tail Call Optimization

SBCL performs tail call optimization (TCO) although this is not required by the Common Lisp standard.

In Lisp terms, you can think of TCO as converting a function call to a `psetq` that assigns the argument values to the called function's variables, followed by a `go` to the start of the called function. This makes clear an inherent efficiency advantage of tail calls: in addition to not having to allocate a stack frame, there is no need to prepare for the call to return (e.g. by computing a return PC).

Consider the following trivial example:

```
(defun fun1 (x)
  (fun2 x x))
```

FUN1 is compiled to native code that modifies its frame (to have the arguments of `fun2` in the right places) and control is transferred via a single jump instruction to `fun2`.

In general, a call is compiled to a tail call if it is in Section 6.2.2 [tail position], page 46, and the `debug` optimization quality is at most 2. Thus, by default, all calls in tail position are compiled to tail calls. See Section 5.1.3.5 [debugging tail calls], page 31.

6.2.2 Tail Position

A call is said to be in tail position unless it is enclosed by at least one of the following:

- a special binding (e.g. by `let`, `progv`),
- a `catch`,
- an `unwind-protect`,
- a `block` and the block name has been closed over,
- a `tagbody` and a `go` tag has been closed over.

These inhibit Section 6.2.1 [tail call optimization], page 46, because they require cleanups when the stack is unwound.

6.2.3 Tail Recursion

Tail recursion is when a function makes a Section 6.2 [tail call], page 46, to itself. In this example, the recursive call to `myfun` is tail-recursive:

```
(defun myfun (x)
  (if (oddp (random x))
      (isqrt x)
      (myfun (1- x))))
```

Because it is a Section 6.2 [tail call], page 46, tail recursion can be compiled to much more efficient code than general recursion. In fact, tail recursion is equivalent to iteration both semantically and in efficiency.

Why would you want to write a program recursively when you can write it using a loop? Well, the main answer is that recursion is a more general mechanism, so it can express some solutions simply that are awkward to write as a loop. Some programmers also feel that recursion is a stylistically preferable way to write loops because it avoids assigning variables. For example, instead of writing

```
(defun fun1 (x)
  (something x))

(defun fun2 (y)
```

```
  (other-thing y))

(do ((x 7 (fun2 (fun1 x))))
    (nil))
```

... you can write

```
(defun fun1 (x)
  (fun2 (something x)))

(defun fun2 (y)
  (fun1 (other-thing y)))

(fun1 7)
```

The tail-recursive definition is actually more efficient, in addition to being (arguably) clearer. As the number of functions and the complexity of their call graph increases, the simplicity of using recursion becomes compelling. Consider the advantages of writing a large finite-state machine with separate tail-recursive functions instead of using a single huge `prog`.

6.3 Local Call

The compiler supports two kinds of function call: full call and local call. Full call is the standard calling convention; its late binding and generality make Lisp what it is but create unavoidable overheads. When the compiler can compile the calling function and the called function simultaneously, it can use local call to avoid some of this overhead. Local call is really a collection of compilation strategies that omit some aspect of full call. In some cases, local call can be totally free.

Local call makes the use of the lexical function binding forms `flet` and `labels` much more efficient. A local call is always faster than a full call, and in many cases it is much faster. Similarly, local call is a natural approach to block compilation (see `compile-file`), a compilation technique that resolves function references at compile time. Block compilation speeds up function calls but increases compilation times and prevents function redefinition.

6.3.1 Local Self-recursive Call

Section 6.3 [local call], page 47, is used when a function defined by `defun` calls itself and block compilation is used (see `compile-file`). For example:

```
(defun fact (n)
  (if (zerop n)
      1
      (* n (fact (1- n)))))
```

This use of local call speeds recursion but can also complicate debugging since `trace` will only show the first call to FACT and not the recursive calls. This is because the recursive calls directly jump to the start of the function and don't indirect through `symbol-function`. This is similar to Section 5.1.3.5 [debugging tail calls], page 31.

6.3.2 Let Call

Because Section 6.3 [local call], page 47, avoids unnecessary call overheads, the compiler internally uses local call to implement some macros and special forms that are not normally thought of as involving a function call. For example, this `let`:

```
(let ((a (foo))
      (b (bar)))
  ...)
```

is internally represented as though it was macroexpanded into:

```
(funcall #'(lambda (a b)
             ...)
         (foo)
         (bar))
```

which results in just as good code as compiling `let` directly, but it also benefits local calls that are semantically the same as `let`. For example, these definitions are all the same as far as the compiler is concerned:

```
(defun foo ()
```

```
  ...some other stuff...
  (let ((a something))
    ...some stuff...))

(defun foo ()
  (flet ((localfun (a)
           ...some stuff...))
    ...some other stuff...
    (localfun something)))

(defun foo ()
  (let ((funvar #'(lambda (a)
                    ...some stuff...)))
    ...some other stuff...
    (funcall funvar something)))
```

Although local call is most efficient when the function is called only once, a call doesn't have to be equivalent to a `let` to be more efficient than full call. All local calls avoid the overhead of argument count checking and keyword argument parsing, and there are a number of other advantages that apply in many common situations. See Section 6.1.1 [let optimization], page 42, for a discussion of the optimizations done on let calls.

6.3.3 Local Calls to Closures

Section 6.3 [local call], page 47, allows for much more efficient use of closures, since the closure environment doesn't need to be allocated on the heap or even stored in memory at all. In this example, there is no penalty for `localfun` referencing `a` and `b`:

```
(defun foo (a b)
  (flet ((localfun (x)
           (1+ (* a b x))))
    (if (= a b)
        (localfun (- a))
        (localfun a))))
```

In local call, the compiler effectively passes closed-over values as extra arguments, so there is no need for you to "optimize" `localfun` by explicitly passing in lexically visible values. Closures may also be subject to Section 6.1.1 [let optimization], page 42.

> *Note*: indirect value cells are currently always allocated on the heap when a variable is both assigned to (with `setq` or `setf`) and closed over, regardless of whether the closure is a local function or not. This is another reason to avoid setting variables when you don't have to.

6.3.4 Local Tail Call

Local tail calls are particularly efficient because they combine the benefits of Section 6.3 [local call], page 47s (simplified calling convention) and Section 6.2 [tail call], page 46s (reusing the stack frame). Scheme programmers write loops with tail-recursive local calls instead of using the imperative `go` and `setq`. This has not caught on in the Common Lisp community for historical reasons and because the standard does not require tail calls. In SBCL, users can choose to write loops such as

```
(defun recurse-fact (n)
  (labels ((loop* (n total)
             (if (zerop n)
                 total
                 (loop* (1- n) (* n total)))))
    (loop* n 1)))
```

which is compiled to just as efficient code as

```
(defun loop-fact (n)
  (let ((r 1))
    (loop for i downfrom n above 0
          do (setq r (* i r)))
    r))
```

6.3.5 Local Call Return Values

One of the more subtle costs of full call comes from allowing arbitrary numbers of return values. This overhead can be avoided in Section 6.3 [local call], page 47s to functions that always return the same number of values. For efficiency reasons (as well as stylistic ones), you should write functions so that they always return the same number of values. This may require passing extra `nil` arguments to `values` in some cases, but the result is more efficient, not less so.

6.4 Block Compilation

Block compilation allows calls to global functions defined by `defun` to be compiled as Section 6.3 [local call], page 47s. The function call can be in a different top-level form than the `defun` or even in a different file.

In addition, block compilation allows the declaration of the *entry points* to the block-compiled portion. An entry point is a function that may be called from outside the block-compiled unit. If a function is not an entry point, then it can be compiled more efficiently since all calls are known at compile time. In particular, if a function is only called in one place, then it will be let converted. This effectively inline expands the function but without the code duplication that results from defining the function normally and then declaring it inline.

The main advantage of block compilation is that it preserves efficiency in programs even when (for readability and syntactic convenience) they are broken up into many small functions. There is absolutely no overhead for calling a non-entry point function that is defined purely for modularity (i.e. called only in one place).

Block compilation also allows for more unboxed and non-descriptor arguments and return values in non-trivial programs.

6.4.1 Block Compilation Semantics

The effect of block compilation can be envisioned as the compiler turning all the `defun`s in the block-compiled unit into a single `labels` form. For example,

```
(declaim (sb-ext:start-block fun1 fun3))

(defun fun1 ()
  ...)

(defun fun2 ()
  ...
  (fun1)
  ...)

(defun fun3 (x)
  (if x
      (fun1)
      (fun2)))

(declaim (sb-ext:end-block))
```

becomes

```
(labels ((fun1 ()
           ...)
         (fun2 ()
           ...
           (fun1)
           ...)
         (fun3 (x)
           (if x
               (fun1)
               (fun2))))
  (setf (fdefinition 'fun1) #'fun1)
  (setf (fdefinition 'fun3) #'fun3))
```

Calls between the block-compiled functions are Section 6.3 [local call], page 47s, so changing the global definition of `fun1` will have no effect on what `fun2` does; `fun2` will keep calling the old `fun1`.

The entry points `fun1` and `fun3` are still installed in `symbol-function` as the global definitions of the functions, so a full call to an entry point works just as before. However, `fun2` is not an entry point, so it is not globally defined. In addition, `fun2` is only called in one place, so it will be let converted.

6.4.2 Block Compilation Declarations

The following declarations allow fine-grained control of block compilation. They are only legal as global declarations (`declaim` or `proclaim`).

`sb-ext:start-block` [Declaration]

Syntax: `(sb-ext:start-block &rest entry-point-names)`

Only valid as a global proclamation.

When processed by the compiler, this declaration marks the start of a Section 6.4 [block compilation], page 49, unit and specifies the entry points to that block. If no entry points are specified, then all functions are made into entry points. If already block compiling, then the compiler ends the current block and starts a new one.

`sb-ext:end-block` [Declaration]

Syntax: `(sb-ext:end-block &rest symbols)`

Only valid as a global proclamation.

This declaration ends a Section 6.4 [block compilation], page 49, unit without starting a new one. This is useful mainly when only a portion of a file is worth block compiling.

6.4.3 Block Compilation Difficulties

The main problem with block compilation is that the compiler uses large amounts of memory when it is block compiling. This places an upper limit on the amount of code that can be block compiled as a unit. To make best use of block compilation, it is necessary to locate the parts of the program containing many internal calls, and then add the appropriate `sb-ext:start-block` declarations. When writing new code, it is a good idea to put in block compilation declarations from the very beginning, since writing block declarations correctly requires accurate knowledge of the program's function call structure. If you want to initially develop code with full incremental redefinition, you can compile with `sb-ext:*block-compile-default*` set to `nil` (the default).

Note that if a `defun` appears in a non-null lexical environment, then calls to it cannot be block compiled.

Unless files are very small, it is probably impractical to block compile multiple files as a unit by specifying a list of files to `compile-file`. Section 6.5.2 [semi inline expansion], page 52, provides another way to extend block compilation across file boundaries.

6.5 Inline Expansion

SBCL can expand almost any function inline, including functions with keyword arguments. The only restrictions are that keyword argument keywords in the call must be constant, and that global function definitions (`defun`) must be done in a null lexical environment (not nested in a `let` or other binding form.) Local functions (e.g. `flet`) can be inline expanded in any environment. Combined with source-level optimization, inline expansion can be used for things that would otherwise require macros for efficient implementation. In SBCL, macros don't have any efficiency advantage, so they need only be used where a macro's syntactic flexibility is required.

Inline expansion is a compiler optimization technique that reduces the overhead of a function call by simply not doing the call: instead, the compiler effectively rewrites the program to appear as though the definition of the called function was inserted at each call site. In Lisp, this is straightforwardly expressed by inserting the `lambda` corresponding to the original definition. Consider

```
(declaim (inline my-1+))
(defun my-1+ (x)
  (+ x 1))
```

With this definition, `(my-1+ someval)` is equivalent to `((lambda (x) (+ x 1)) someval)`.

`sb-ext:*inline-expansion-limit*` [Variable]

An upper limit on the number of inline function calls that will be expanded in any given component (single function or block compilation unit).

Regardless of this limit, expanding a large function inline can increase the size of a program considerably. If the program becomes too large, inline expansion hurts speed rather than helping it, since hardware resources such as physical memory and cache will be exhausted. Inline expansion is called for:

- When profiling has shown that a relatively simple function is called so often that a large amount of time is being wasted in the calling of that function (as opposed to running in that function). If a function is complex, it will take a long time to run relative the time spent in call, so the speed advantage of inline expansion is diminished at the same time the space cost of inline expansion is increased. Of course, if a function is rarely called, then the overhead of calling it is also insignificant.
- With functions so simple that they take less space to inline expand than would be taken to call the function (such as `my-1+` above). It would require intimate knowledge of the compiler to be certain when inline expansion would reduce space, but it is generally safe to inline expand functions whose definition is a single function call, or a few calls to simple functions.

In addition to this speed/space tradeoff from inline expansion's avoidance of the call, inline expansion can also reveal opportunities for optimization. SBCL's extensive source-level optimization can make use of context information from the caller to tremendously simplify the code resulting from the inline expansion of a function.

The main form of caller context is local information about the actual argument values: what the argument types are and whether the arguments are constant. Knowledge about argument types can eliminate run-time type tests (e.g. for generic arithmetic). Constant arguments in a call provide opportunities for constant folding optimization after inline expansion.

A hidden way that constant arguments are often supplied to functions is through the defaulting of unsupplied optional or keyword arguments. There can be a huge efficiency advantage to inline expanding functions that have complex keyword-based interfaces, such as this definition of the `member` function:

```
(declaim (inline member))
(defun member (item list &key
               (key #'identity)
               (test #'eql testp)
               (test-not nil notp))
  (do ((list list (cdr list)))
      ((null list) nil)
    (let ((car (car list)))
      (if (cond (testp
                 (funcall test item (funcall key car)))
                (notp
                 (not (funcall test-not item (funcall key car))))
                (t
                 (funcall test item (funcall key car))))
          (return list)))))
```

After inline expansion,

```
(member a l :key #'foo-a :test #'char=)
```

is simplified to the obvious code:

```
(do ((list list (cdr list)))
    ((null list) nil)
  (let ((car (car list)))
    (if (char= item (foo-a car))
        (return list))))
```

In this example, there could easily be more than an order of magnitude improvement in speed. In addition to eliminating the original call to `member`, inline expansion also allows the calls to `char=` and `foo-a` to be open-coded. We go from a loop with three tests and two calls to a loop with one test and no calls.

See also Section 6.1 [source optimization], page 42.

6.5.1 Inline Expansion Recording

Inline expansion requires that the source for the inline expanded function to be available when calls to the function are compiled. The compiler doesn't remember the inline expansion of every function, since that would take an excessive amount of space. Instead, the programmer must tell the compiler to record the inline expansion before the definition of the inline expanded function is compiled. This is done by globally

declaring the function inline before the function is defined, by using the `inline` and `sb-ext:maybe-inline` declarations.

In addition to recording the inline expansion of inline functions at the time the function is compiled, `compile-file` also puts the inline expansion in the output file. When the output file is loaded, the inline expansion is made available for subsequent compilations; there is no need to compile the definition again to record the inline expansion.

If a function is declared inline, but no expansion is recorded, then the compiler will give style warning like

```
; caught STYLE-WARNING:
;   Call to MY-1+ could not be inlined because its source code
;   was not saved. A global INLINE or SB-EXT:MAYBE-INLINE proclamation must be
;   in effect to save function definitions for inlining.
```

If you get this, check that the `inline` declaration appears before the `defun`, which in turn is before the calls that are to be inline expanded. Alternatively, the style warning can be silenced by declaring the function `notinline` at the call site.

This style warning will also be given if the inline expansion could not be recorded because the `defun` was in a non-null lexical environment.

6.5.2 Semi-inline Expansion

Semi-inline expansion shares a single copy of a function across all the calls in a component by converting the inline expansion into a Section 6.3 [local call], page 47. This takes up less space when there are multiple calls but also provides less opportunity for context-dependent optimization. When there is only one call, the result is identical to normal Section 6.5 [inline expansion], page 50.

The inlining decision takes into account the inlining declaration of the function to be called and the optimization qualities in effect, where both are taken at the call site.

- `sb-ext:maybe-inline`: Semi-inlined if `space` is `0`; else, not inlined at all.
- `inline`: Semi-inlined if (`< speed space`) or (`< speed compilation-speed`); else, inlined normally.

`sb-ext:maybe-inline` [Declaration]

Syntax: `(sb-ext:maybe-inline &rest function-names)`

The `maybe-inline` declaration is similar to `inline` but indicates that inline expansion may sometimes be desirable, rather than saying that inline expansion should almost always be done. When used in a global declaration, it causes the expansions for the named functions to be recorded by `defun`.

When its expansion is recorded, a call to a function that is `maybe-inline` is semi-inlined if `space` is `0`, where both the `maybe-inline` declaration and the value of the `space` optimization quality is at the call site. If `space` is non-zero, then the call is not inlined.

Note that the use of the `sb-ext:maybe-inline` declaration followed by the `defun` is preferable to the standard idiom of

```
(declaim (inline myfun))
(defun myfun () ...)
(declaim (notinline myfun))

;; Calls to MYFUN here are not inline expanded.

(defun somefun ()
  (declare (inline myfun))
  ;; Calls to MYFUN here are inline expanded.
  ...)
```

The problem with using `notinline` in this way is that it does more than just suppress inline expansion: it also forbids the compiler to use any knowledge of `myfun` until a later `inline` declaration overrides the `notinline`. This prevents compiler warnings about incorrect calls to the function and also prevents Section 6.4 [block compilation], page 49.

When the goal is merely to control whether inline expansion is done by default, it is preferable to use `sb-ext:maybe-inline` rather than `notinline`. The `notinline` declaration should be reserved for those special occasions when a function may be redefined at run-time, so the compiler must be told that the obvious definition of a function is not necessarily the one that will be in effect at the time of the call.

6.6 Slot Access

6.6.1 Structure Object Slot Access

Structure slot accessors are efficient only if the compiler is able to open code them: compiling a call to a structure slot accessor before the structure is defined, declaring one `notinline`, or passing it as a functional argument to another function causes severe performance degradation.

6.6.2 Standard Object Slot Access

The most efficient way to access a slot of a `standard-object` is by using `slot-value` with a constant slot name argument inside a `defmethod` body, where the variable holding the instance is a specializer parameter of the method and is never assigned to. The cost is roughly 1.6 times that of an open coded structure slot accessor.

Second most efficient way is to use a CLOS slot accessor or `slot-value` with a constant slot name argument but in circumstances other than specified above. This may be up to 3 times as slow as the method described above.

Example:

```
(defclass foo () ((bar)))

;; Fast: specializer and never assigned to
(defmethod quux ((foo foo) new)
  (let ((old (slot-value foo 'bar)))
    (setf (slot-value foo 'bar) new)
    old))

;; Slow: not a specializer
(defmethod quux ((foo foo) new)
  (let* ((temp foo)
         (old (slot-value temp 'bar)))
    (setf (slot-value temp 'bar) new)
    old))

;; Slow: assignment to FOO
(defmethod quux ((foo foo) new)
  (let ((old (slot-value foo 'bar)))
    (setf (slot-value foo 'bar) new)
    (setf foo new)
    old))
```

Note that when profiling code such as this, the first few calls to the generic function are not representative, as the dispatch mechanism is lazily set up during those calls.

6.7 Stack Allocation

SBCL has fairly extensive support for performing allocations on the stack when a variable or function is declared `dynamic-extent`. The `dynamic-extent` declarations are not verified but are simply trusted as long as `sb-ext:*stack-allocate-dynamic-extent*` is true.

`sb-ext:*stack-allocate-dynamic-extent*` [Variable]
: If true (the default), the compiler believes `dynamic-extent` declarations and stack allocates otherwise inaccessible parts of the object whenever possible.

SBCL recognizes any value which a variable declared `dynamic-extent` can take on as having dynamic extent. This means that, in addition to the value a variable is bound to initially, a value assigned to a variable by `setq` is also recognized as having dynamic extent when the variable is declared `dynamic-extent`. Users can thus build complex structures on the stack using iteration and `setq`.

At present, SBCL implements stack allocation for the following kinds of values when they are recognized as having dynamic extent:

- `&rest` lists;

- the results of `cons`, `list`, `list*`, and `vector`;
- the result of simple forms of `make-array`: stack allocation is possible only if the resulting array is known to be both simple and one-dimensional, and it has a constant `:element-type`;

  Warning: Stack space is limited, so allocation of a large vector may cause stack overflow. Stack overflow checks are done except in 0 `safety` policies.
- closures defined with `flet` or `labels` with a bound `dynamic-extent` declaration;
- anonymous closures defined with `lambda`;
- user-defined structures when the structure constructor defined using `defstruct` has been declared `inline`;

  Note: Structures with *raw* slots can currently be stack-allocated only on x86 and x86-64. A raw slot is one whose declared type is a subtype of exactly one of: `double-float`, `single-float`, `(complex double-float)`, `(complex single-float)`, or `sb-ext:word`; but as an exception to the preceding, any subtype of `fixnum` is not stored as raw despite also being a subtype of `sb-ext:word`.
- otherwise-inaccessible parts of objects recognized to be dynamic extent. The support for detecting when this applies is very sophisticated. The compiler can do this detection when any value form for a variable contains conditional allocations, function calls, inlined functions, anonymous closures, or even other variables. This allows stack allocation of complex structures.

Examples:

```
;;; Declaiming a structure constructor inline before definition makes
;;; stack allocation possible.
(declaim (inline make-thing))
(defstruct thing obj next)

;;; Stack allocation of various objects bound to DYNAMIC-EXTENT
;;; variables.
(let* ((list (list 1 2 3))
       (nested (cons (list 1 2) (list* 3 4 (list 5))))
       (vector (make-array 3 :element-type 'single-float))
       (thing (make-thing :obj list
                          :next (make-thing :obj (make-array 3))))
       (closure (let ((y ...)) (lambda () y))))
  (declare (dynamic-extent list nested vector thing closure))
  ...)

;;; Stack allocation of objects assigned to DYNAMIC-EXTENT variables.
(let ((x nil))
  (declare (dynamic-extent x))
  (setq x (list 1 2 3))
  (dotimes (i 10)
    (setq x (cons i x)))
  ...)

;;; Stack allocation of arguments to a local function is equivalent
;;; to stack allocation of local variable values.
(flet ((f (x)
         (declare (dynamic-extent x))
         ...))
  ...
  (f (list 1 2 3))
  (f (cons (cons 1 2) (cons 3 4)))
  ...)

;;; Stack allocation of &REST lists
(defun foo (&rest args)
  (declare (dynamic-extent args))
```

```
  ...)
```

As a notable exception to recognizing otherwise inaccessible parts of other recognized dynamic extent values, SBCL does not as of 1.0.48.21 propagate dynamic-extentness through `&rest` arguments – but another conforming implementation might, so portable code should not rely on this.

```
(declaim (inline foo))
(defun foo (fun &rest arguments)
  (declare (dynamic-extent arguments))
  (apply fun arguments))

(defun bar (a)
  ;; SBCL will heap allocate the result of (LIST A), and stack
  ;; allocate only the spine of the &rest list -- so this is
  ;; safe but unportable.
  ;;
  ;; Another implementation, including earlier versions of SBCL
  ;; might consider (LIST A) to be otherwise inaccessible and
  ;; stack-allocate it as well!
  (foo #'car (list a)))
```

If dynamic extent constraints specified in the Common Lisp standard are violated, the best that can happen is for the program to have garbage in variables and return values; more commonly, the system will crash. In particular, it is important to realize that this can interact in suprising ways with the otherwise inaccessible parts criterion:

```
(let* ((a (list 1 2 3))
       (b (cons a a)))
   (declare (dynamic-extent b))
   ;; Unless A is accessed elsewhere as well, SBCL will consider
   ;; it to be otherwise inaccessible -- it can only be accessed
   ;; through B, after all -- and stack allocate it as well.
   ;;
   ;; Hence returning (CAR B) here is unsafe.
   ...)
```

SBCL also performs sophisticated escape analysis to enable automatic stack allocation of local functions without any bound dynamic extent declarations in many situations where the compiler can prove that no uses escape (traditional Lisp terminology names this situation "all uses are downward funargs"). For example, in the following function, the local function `#'predicatep` is stack allocated, because the compiler understands that the built-in function `position-if` only uses its first argument as a downward funarg:

```
(let ((acc 0))
  (flet ((predicatep (num) (plusp (+ num off))))
    (dotimes (i 10)
      (incf acc (position-if #'predicatep array)))
    (if (plusp off)
        (incf acc (if (positivep acc) 10 3))
        (incf acc (position-if #'predicatep array))))
  acc)
```

Users can also declare that their own functions take downward funargs by adding bound dynamic extent declarations on the function arguments.

```
(defun trivial-hof (fun arg)
  (declare (dynamic-extent fun))
  (funcall fun 3 arg))
```

Currently, such dynamic extent declarations only cause stack allocation of downward funargs at call sites on sufficiently unsafe policy. This is partly because the compiler is currently not able to detect incorrect usage of dynamic extent declarations.

```
(defun autodxclosure1 (&optional (x 4))
  ;; Calling a higher-order function will only implicitly
  ;; stack-allocate a funarg if the callee is trusted (a CL:
  ;; function) or the caller is unsafe.
  (declare (optimize speed (safety 0) (debug 0)))
```

```
(trivial-hof (lambda (a b) (+ a b x)) 92))
```

6.8 Modular Arithmetic

Some numeric functions have a property: n lower bits of the result depend only on n lower bits of (all or some) arguments. If the compiler sees an expression of form `(logand <expr> <mask>)`, where `<expr>` is a tree of such *good* functions and `<mask>` is known to be of type `(unsigned-byte <w>)`, where `<w>` is a *good* width, all intermediate results will be cut to `<w>` bits (but it is not done for variables and constants!). This often results in an ability to use simple machine instructions for the functions.

Consider this example:

```
(defun i (x y)
  (declare (type (unsigned-byte 32) x y))
  (ldb (byte 32 0) (logxor x (lognot y))))
```

The result of `(lognot y)` will be negative and of type `(signed-byte 33)`, so a naive implementation on a 32-bit platform is unable to use 32-bit arithmetic here. But modular arithmetic optimizer is able to do it: because the result is cut down to 32 bits, the compiler will replace `logxor` and `lognot` with versions cutting results to 32 bits, and because terminals (here, expressions `x` and `y`) are also of type `(unsigned-byte 32)`, 32-bit machine arithmetic can be used.

As of SBCL 0.8.5 good functions are `+`, `-`, `logand`, `logior`, `logxor`, `lognot` and their combinations; and `ash` with the positive second argument. Good widths are 32 on 32-bit CPUs and 64 on 64-bit CPUs. While it is possible to support smaller widths as well, currently this is not implemented.

6.8.1 Signed Modular Arithmetic

Sign-extending the result in the following way will be translated into signed modular arithmetic:

```
(defun add (a b)
  (declare (type (signed-byte 64) a b))
  (let ((u (ldb (byte 64 0) (+ a b))))
    (logior u (- (mask-field (byte 1 63) u)))))
```

6.9 Recognized Idioms

Common Lisp doesn't directly expose all features present in modern hardware. Some code patterns are recognized and turned into more efficient hardware instructions without requiring the use of internal features.

6.9.1 Count Trailing Zeros

```
(defun ctz (n)
  (declare (type (unsigned-byte 64) n))
  (integer-length (ldb (byte 64 0) (lognor n (- n)))))
```

is turned into hardware instructions on arm64 and x86-64. It returns 64 when `n` is 0. `n` can also be `(signed-byte 64)` or `fixnum`.

6.10 Global and Always-bound Variables

`sb-ext:defglobal` *name value* &optional *doc* [Macro]

Defines `name` as a global variable that is always bound. `value` is evaluated and assigned to `name` both at compile- and load-time but only if `name` is not already bound.

Global variables share their values between all threads. They cannot be locally bound, declared special, defined as constants, or defined as symbol macros.

See also the declarations `sb-ext:global` and `sb-ext:always-bound`.

`sb-ext:global` [Declaration]

Syntax: `(sb-ext:global &rest symbols)`

Only valid as a global proclamation.

Specifies that the named symbols cannot be proclaimed or locally declared `special`. Proclaiming an already special or constant variable name as `sb-ext:global` signals an error. Allows more efficient value lookup in threaded environments in addition to expressing programmer intention.

`sb-ext:always-bound` [Declaration]

Syntax: `(sb-ext:always-bound &rest symbols)`

Only valid as a global proclamation.

Specifies that the named symbols are always bound. Inhibits `makunbound` of the named symbols. Proclaiming an unbound symbol as `sb-ext:always-bound` signals an error. Allows the compiler to elide boundness checks from value lookups.

6.11 Miscellaneous Efficiency Issues

The `sb-ext:purify` function (available with conservative GC) causes SBCL first to collect all garbage, then to mark all uncollected objects as permanent, never again attempting to collect them as garbage. This can cause a large increase in efficiency when using a primitive garbage collector, or a more moderate increase in efficiency when using a more sophisticated garbage collector which is well suited to the program's memory usage pattern. It also allows permanent code to be frozen at fixed addresses, a precondition for using copy-on-write to share code between multiple Lisp processes. This is less important with modern generational GC, but not all SBCL platforms use such a garbage collector.

The `sb-ext:truly-the` special form declares the type of the result of the operations, producing its argument; the declaration is not checked. In short: don't use it.

The `sb-ext:freeze-type` declaration declares that a type will never change, which can make type testing (e.g. with `typep`) more efficient for structure types.

FIXME: The remaining material in the CMUCL manual about getting good performance from the compiler should be reviewed, and added to this section. In the meantime, the original CMUCL manual is still 95+% correct for the SBCL's compiler. See the sections

- Advanced Compiler Use and Efficiency Hints
- Advanced Compiler Introduction
- More About Types in Python
- Type Inference
- Object Representation
- Numbers
- General Efficiency Hints
- Efficiency Notes

Besides this information from the CMUCL manual, there are a few other points to keep in mind.

- The CMUCL manual doesn't seem to state it explicitly, but Python (the CMUCL compiler, not the language) has a mental block about type inference when assignment is involved. Python is very aggressive and clever about inferring the types of values bound with `let`, `let*`, inline function call, and so forth. However, it's much more passive and dumb about inferring the types of values assigned with `setq`, `setf`, and friends. It would be nice to fix this, but in the meantime don't expect that just because it's very smart about types in most respects it will be smart about types involved in assignments. (This doesn't affect its ability to benefit from explicit type declarations involving the assigned variables, only its ability to get by without explicit type declarations.)
- Since the time the CMUCL manual was written, CMUCL (and thus SBCL) has gotten a generational GC. This means that there are some efficiency implications of various patterns of memory usage which aren't discussed in the CMUCL manual. (Some new material should be written about this.)
- SBCL has some important known efficiency problems. Perhaps the most important are
  - The garbage collector is not particularly efficient, at least on platforms without the generational collector (as of SBCL 0.8.9, all except x86).
  - Various aspects of the PCL implementation of CLOS are more inefficient than necessary.

Finally, note that Common Lisp defines many constructs which, in the infamous phrase, "could be compiled efficiently by a sufficiently smart compiler". The phrase is infamous because making a compiler which actually is sufficiently smart to find all these optimizations systematically is well beyond the state of the art of current compiler technology. Instead, they're optimized on a case-by-case basis by hand-written code, or not optimized at all if the appropriate case hasn't been hand-coded. Some cases where no such hand-coding has been done as of SBCL version 0.6.3 include

- `(reduce #'f x)` where the type of `x` is known at compile time,
- various bit vector operations, e.g. `(position 0 some-bit-vector)`,
- specialized sequence idioms, e.g. `(remove item list :count 1)`,
- cases where local compilation policy does not require excessive type checking, e.g. `(locally (declare (safety 1)) (assoc item list))` (which currently performs safe `endp` checking internal to `assoc`).

If your system's performance is suffering because of some construct which could in principle be compiled efficiently, but which the SBCL compiler can't in practice compile efficiently, consider writing a patch to the compiler and submitting it for inclusion in the main sources. Such code is often reasonably straightforward to write; search the sources for the string `deftransform` to find many examples (some straightforward, some less so).

7 Beyond the ANSI Standard

SBCL is a mostly-conforming implementation of the ANSI Common Lisp standard. Essentially every type of non-conformance is considered a bug (see Section 19.1.3 [reporting bugs], page 160). The exceptions involve the following internal inconsistencies in the standard:

- `prog2` returns the primary value of its second form, as specified in the *Arguments and Values* section of the specification for that operator, not that of its first form, as specified in the *Description*.
- The `string` type is considered to be the union of all types `(array c (size))` for all non-`nil` subtypes `c` of `character`, excluding arrays specialized to the empty type.
- The `:order` long form option in `define-method-combination` method group specifiers accepts the value `nil` as well as `:most-specific-first` and `:most-specific-last`, in order to allow programmers to declare that the order of methods playing that role in the method combination does not matter.

This section covers smaller extensions to standard Common Lisp functionality and how SBCL operates in the white patches of undefined behavior.

7.1 Idiosyncrasies

The information in this section describes some of the ways that SBCL deals with choices the ANSI standard leaves to the implementation.

7.1.1 Declarations as Assertions

Declarations are generally treated as assertions. This general principle, its implications, and the bugs which still keep the compiler from quite satisfying this principle, are discussed in Section 4.3.1 [type declaration checking], page 23.

7.1.2 Fasl Format

SBCL fasl-format is binary compatible only with the exact SBCL version it was generated with. While this is obviously suboptimal, it has proven more robust than trying to maintain fasl compatibility across versions: accidentally breaking things is far too easy and can lead to hard to diagnose bugs.

The following snippet handles fasl recompilation automatically for ASDF-based systems and makes a good candidate for inclusion in the user or system initialization file (see Section 3.4 [initialization files], page 13).

```
(require :asdf)

;;; If a fasl was stale, try to recompile and load (once).
(defmethod asdf:perform :around ((o asdf:load-op)
                                 (c asdf:cl-source-file))
   (handler-case (call-next-method o c)
      ;; If a fasl was stale, try to recompile and load (once).
      (sb-ext:invalid-fasl ()
         (asdf:perform (make-instance 'asdf:compile-op) c)
         (call-next-method))))
```

7.1.3 Compiler-only Implementation

By default, SBCL is a compiler-only implementation of Common Lisp. That is, for all but a few special cases, `eval` creates a lambda expression, calls `compile` on the lambda expression to create a compiled function, and then calls `funcall` on the resulting function object. A more traditional Section 4.7 [interpreter], page 28, is also available on default builds; it is usually only called internally. This is explicitly allowed by the ANSI standard but leads to some oddities; e.g. at default settings, `functionp` and `compiled-function-p` are equivalent, and they collapse into the same function when SBCL is built without the interpreter.

7.1.4 Defining Constants

SBCL is quite strict about ANSI's definition of `defconstant`. ANSI says that doing `defconstant` of the same symbol more than once is undefined unless the new value is `eql` to the old value. Conforming to this specification is a nuisance when the "constant" value is only constant under some weaker test like `string=` or `equal`.

It's especially annoying because, in SBCL, `defconstant` takes effect not only at load time but also at compile time, so that just compiling and loading reasonable code like

```
(defconstant +foobyte+ '(1 4))
```

runs into this undefined behavior. Many implementations of Common Lisp try to help the programmer around this annoyance by silently accepting the undefined code and trying to do what the programmer probably meant.

SBCL instead treats the undefined behavior as an error. Often such code can be rewritten in portable ANSI Common Lisp which has the desired behavior. The code above can be given an exactly defined meaning by replacing `defconstant` either with `defparameter` or with a customized macro which does the right thing, e.g.

```
(defmacro define-constant (name value &optional doc)
  `(defconstant ,name (if (boundp ',name) (symbol-value ',name) ,value)
                ,@(when doc (list doc))))
```

or possibly along the lines of the `sb-int:defconstant-eqx` macro used internally in the implementation of SBCL itself. In circumstances where this is not appropriate, the programmer can handle the condition type `sb-ext:defconstant-uneql` and choose either the `continue` restart or `abort` restart as appropriate.

7.1.5 Style Warnings

SBCL gives style warnings about various kinds of perfectly legal code, e.g.

- multiple `defun`s of the same symbol in different units;
- special variables not named in the conventional `*foo*` style, and lexical variables unconventionally named in the `*foo*` style.

This causes friction with people who point out that other ways of organizing code (especially avoiding the use of `defgeneric`) are just as aesthetically stylish. However, these warnings should be read not as *warning, bad aesthetics detected, you have no style* but as *warning, this style keeps the compiler from understanding the code as well as you might like*. That is, unless the compiler warns about such conditions, there's no way for the compiler to warn about some programming errors which would otherwise be easy to overlook.

7.2 Reader Extensions

7.2.1 Extended Package Prefix Syntax

SBCL supports extended package prefix syntax, which allows specifying an alternate package instead of `*package*` for the reader to use as the default package for interning symbols:

```
<package-name>::<form-with-interning-into-package>
```

Example:

```
'foo::(bar quux zot) == '(foo::bar foo::quux foo::zot)
```

`*package*` is not rebound during the course of reading a form with extended package prefix syntax; if `foo::bar` would cause a read-time package lock violation, so does `foo::(bar)`.

7.2.2 Symbol Name Normalization

SBCL also extends the reader to normalize all symbols to *Normalization Form KC* in builds with Unicode enabled. Whether symbols are normalized is controlled by

`sb-ext:readtable-normalization` *readtable* [Function]

Returns `t` if `readtable` normalizes symbols to NFKC, and `nil` otherwise. The `readtable-normalization` of the standard readtable is `t`.

Symbols created by `intern` and similar functions are not affected by this setting. If `sb-ext:readtable-normalization` is `t`, symbols that are not normalized are escaped during printing.

7.2.3 Decimal Syntax for Rationals

SBCL supports a decimal syntax for rationals, modelled after the standard syntax for floating-point numbers. If a number with floating-point syntax has an exponent marker of `r` or `r` (rather than one of the standard exponent markers), it is read as the rational with the exact value of the decimal number expressed as a float.

In addition, setting or binding the value of `*read-default-float-format*` to `rational` around a call to `read` or `read-from-string` has the effect that floating-point numbers without exponent markers are read as rational numbers, as if there had been an explicit `r` or `r` marker.

Floating point numbers of all types are printed with an exponent marker while the value of `*read-default-float-format*` is `rational`; however, rational numbers are printed in their standard syntax, irrespective of the value of `*read-default-float-format*`.

7.3 Package-local Nicknames

SBCL allows giving packages local nicknames: they allow short and easy-to-use names to be used without fear of name conflict associated with normal nicknames.

A local nickname is valid only when inside the package for which it has been specified. Different packages can use same local nickname for different global names, or different local nicknames for the same global name.

The symbol `:package-local-nicknames` in `*features*` denotes the support for this feature.

`defpackage` options are extended to include

```
:local-nicknames (<local-nickname> <actual-package-name>)*
```

with the semantics of adding the package package-local nicknames `<local-nickname>`s for the corresponding `<actual-package-name>`s.

In the following example, the local nickname `"BAR"` refers to the package `"FOO"`, and the local nickname `"B"` is added to be able to refer to the thus hidden `"BAR"` package:

```
(defpackage :bar (:intern "X"))
(defpackage :foo (:intern "X"))
(defpackage :quux (:use :cl) (:local-nicknames (:bar :foo) (:b :bar)))
(find-symbol "X" :foo) ; => FOO::X
(find-symbol "X" :bar) ; => BAR::X
(let ((*package* (find-package :quux)))
  (find-symbol "X" :bar))               ; => FOO::X
(let ((*package* (find-package :quux)))
  (find-symbol "X" :b))                 ; => BAR::X
```

`sb-ext:package-local-nicknames` *package-designator* [Function]

Returns an alist of `(local-nickname . actual-package)` describing the nicknames local to the designated package.

When in the designated package, calls to `find-package` with any of the local-nicknames will return the corresponding actual-package instead. This also affects all implied calls to `find-package`, including those performed by the reader.

When printing a package prefix for a symbol with a package local nickname, the local nickname is used instead of the real name in order to preserve print-read consistency.

Experimental: interface subject to change.

`sb-ext:package-locally-nicknamed-by-list` *package-designator* [Function]

Returns a list of packages which have a local nickname for the designated package.

Experimental: interface subject to change.

`sb-ext:add-package-local-nickname` *local-nickname actual-package* &optional (*package-designator* (*sb-int:sane-package*)) [Function]

Adds `local-nickname` for `actual-package` in the designated package, defaulting to current package. `local-nickname` must be a string designator, and `actual-package` must be a package designator.

Returns the designated package.

Signals a continuable error if `local-nickname` is already a package local nickname for a different package, or if `local-nickname` is one of `"CL"`, `"COMMON-LISP"`, `"KEYWORD"`, or if `local-nickname` is a global name or nickname for the package to which the nickname would be added.

When in the designated package, calls to `find-package` with the `local-nickname` will return the package the designated `actual-package` instead. This also affects all implied calls to `find-package`, including those performed by the reader.

When printing a package prefix for a symbol with a package local nickname, local nickname is used instead of the real name in order to preserve print-read consistency.

Experimental: interface subject to change.

`sb-ext:remove-package-local-nickname` *old-nickname* &optional (*package-designator* (*sb-int:sane-package*)) [Function]

If the designated package had `old-nickname` as a local nickname for another package, it is removed. Returns true if the nickname existed and was removed, and `nil` otherwise.

Experimental: interface subject to change.

7.4 Package Variance

`defpackage clhs` specifies that *if the new definition is at variance with the current state of that package, the consequences are undefined*. SBCL by default signals a full warning and retains as much of the package state as possible. This can be adjusted with the following variable.

`sb-ext:*on-package-variance*` [Variable]

Specifies behavior when redefining a package using `defpackage` and the definition is in variance with the current state of the package.

The value should be of the form:

```
(:warn [t | packages-names] :error [t | package-names])
```

specifying which packages get which behaviour – with `t` signifying the default unless otherwise specified. If default is not specified, `:warn` is used.

- `:warn` keeps as much state as possible and causes SBCL to signal a full warning.
- `:error` causes SBCL to signal an error when the variant `defpackage` form is executed, with restarts provided for user to specify what action should be taken.

Example:

```
(setf *on-package-variance* '(:warn (:swank :swank-backend) :error t))
```

specifies to signal a warning if SWANK package is in variance, and an error otherwise.

7.5 Garbage Collection

SBCL provides additional garbage collection functionality not specified by ANSI.

`sb-ext:gc` &key *full* (*gen 0*) &allow-other-keys [Function]

Initiate a garbage collection.

The default is to initiate a nursery collection, which may in turn trigger a collection of one or more older generations as well. If FULL is true, all generations are collected. If GEN is provided, it can be used to specify the oldest generation guaranteed to be collected.

`sb-ext:*after-gc-hooks*` [Variable]

Called after each garbage collection, except for garbage collections triggered during thread exits. In a multithreaded environment these hooks may run in any thread.

7.5.1 Finalization

Finalization allows code to be executed after an object has been garbage collected. This is useful for example for releasing foreign memory associated with a Lisp object.

`sb-ext:finalize` *object function* &key *dont-save* [Function]

Arrange for the designated `function` to be called when there are no more references to `object`, including references in `function` itself.

If `dont-save` is true, the finalizer will be cancelled when `save-lisp-and-die` is called: this is useful for finalizers deallocating system memory, which might otherwise be called with addresses from the old image.

In a multithreaded environment `function` may be called in any thread. In both single and multithreaded environments `function` may be called in any dynamic scope: consequences are unspecified if `function` is not fully re-entrant.

Errors from `function` are handled and cause a `warning` to be signalled in whichever thread the `function` was called in.

Examples:

```
;;; GOOD, assuming RELEASE-HANDLE is re-entrant.
(let* ((handle (get-handle))
       (object (make-object handle)))
 (finalize object (lambda () (release-handle handle)))
 object)
;;; BAD, finalizer refers to object being finalized, causing
;;; it to be retained indefinitely!
(let* ((handle (get-handle))
       (object (make-object handle)))
  (finalize object
            (lambda ()
              (release-handle (object-handle object)))))
;;; BAD, not re-entrant!
(defvar *rec* nil)

(defun oops ()
 (when *rec*
   (error "recursive OOPS"))
 (let ((*rec* t))
   (gc))) ; or just cons enough to cause one
(progn
  (finalize "oops" #'oops)
  (oops)) ; GC causes re-entry to #'oops due to the finalizer
          ; -> ERROR, caught, WARNING signalled
```

`sb-ext:cancel-finalization` *object* [Function]
Cancel all finalizations for `object`, returning `t` if it had a finalizer.

7.5.2 Weak Pointers

Weak pointers allow references to objects to be maintained without keeping them from being garbage collected: useful for building caches among other things.

Hash tables can also have weak keys and values. See Section 7.15 [hash table extensions], page 80.

`sb-ext:make-weak-pointer` *object* [Function]
Allocate and return a weak pointer which points to `object`.

`sb-ext:weak-pointer-value` *weak-pointer* [Function]
If `weak-pointer` is valid, return the value of `weak-pointer` and `t`. If the referent of `weak-pointer` has been garbage collected, returns the values `nil` and `nil`.

7.5.3 Introspection and Tuning

`sb-ext:*gc-run-time*` [Variable]
Total CPU time spent doing garbage collection (as reported by `get-internal-run-time`.) Initialized to zero on startup. It is safe to bind this to zero in order to measure `gc` time inside a certain section of code, but doing so may interfere with results reported by e.g. `time`.

`sb-ext:*gc-real-time*` [Variable]
Total real time spent doing garbage collection (as reported by `get-internal-real-time`.) Initialized to zero on startup.

`sb-ext:bytes-consed-between-gcs` [Function]
The amount of memory that will be allocated before the next garbage collection is initiated. This can be set with `setf`.

On GENCGC platforms this is the nursery size, and defaults to 5% of dynamic space size.

Note that currently, changes to this value are lost when saving core.

`sb-ext:dynamic-space-size` [Function]
Size of the dynamic space in bytes.

`sb-ext:get-bytes-consed` [Function]
Return the number of bytes consed since the program began. Typically this result will be a consed bignum, so if you have an application (e.g. profiling) which can't tolerate the overhead of consing bignums, you'll probably want either to hack in at a lower level (as the code in the `sb-profile` package does), or to design a more microefficient interface and submit it as a patch.

`sb-ext:gc-logfile` [Function]
Return the pathname used to log garbage collections. Can be `setf`. Default is `nil`, meaning collections are not logged. If non-null, the designated file is opened before and after each collection, and generation statistics are appended to it.

`sb-ext:generation-average-age` *generation* [Function]
Average age of memory allocated to `generation`: average number of times objects allocated to the generation have seen younger objects promoted to it. Available on GENCGC platforms only.
Experimental: interface subject to change.

`sb-ext:generation-bytes-allocated` *generation* [Function]
Number of bytes allocated to `generation` currently. Available on GENCGC platforms only.
Experimental: interface subject to change.

`sb-ext:generation-bytes-consed-between-gcs` *generation* [Function]
Number of bytes that can be allocated to `generation` before that generation is considered for garbage collection. This value is meaningless for generation 0 (the nursery): see `bytes-consed-between-gcs` instead. Default is 5% of the dynamic space size divided by the number of non-nursery generations. Can be assigned to using `setf`. Available on GENCGC platforms only.
Experimental: interface subject to change.

`sb-ext:generation-minimum-age-before-gc` *generation* [Function]
Minimum average age of objects allocated to `generation` before that generation is may be garbage collected. Default is 0.75. See also `generation-average-age`. Can be assigned to using `setf`. Available on GENCGC platforms only.
Experimental: interface subject to change.

`sb-ext:generation-number-of-gcs` *generation* [Function]
Number of times garbage collection has been done on `generation` without promotion. Available on GENCGC platforms only.
Experimental: interface subject to change.

`sb-ext:generation-number-of-gcs-before-promotion` *generation* [Function]
Number of times garbage collection is done on `generation` before automatic promotion to the next generation is triggered. Default is 1. Can be assigned to using `setf`. Available on GENCGC platforms only.
Experimental: interface subject to change.

7.5.4 Tracing Live Objects Back to Roots

This feature is intended to help expert users diagnose rare low-level issues and should not be needed during normal usage. On top of that, the interface and implementation are experimental and may change at any time without further notice.
It is sometimes important to understand why a given object is retained in the Lisp image instead of being garbage collected. To help with this problem, SBCL provides a mechanism that searches through the different memory spaces, builds a path of references from a root to the object in question and finally reports this paths:

`sb-ext:search-roots` *weak-pointers* &key (*criterion :oldest*) (*ignore '(* ** *** / // ///)*) (*print t*) [Function]
Find roots keeping the targets of `weak-pointers` alive.
`weak-pointers` must be a single `sb-ext:weak-pointer` or a list of those, pointing to objects for which roots should be searched.
`criterion` determines just how rooty (how deep) a root must be in order to be considered. Possible values are:

- `:oldest`

  This says we can stop upon seeing an object in the oldest gen to `gc`, or older. This is the easiest test to satisfy.

- `:pseudo-static`

  This is usually the same as `:oldest`, unless the oldest gen to `gc` has been decreased.

- `:static`

  To find a root of an image-backed object, you want to stop only at a truly `:static` object.

`ignore` is a list of objects to treat as if nonexistent in the heap. It can often be useful for finding a path to an interned symbol other than through its package by specifying the package as an ignored object.

`print` controls whether discovered paths should be returned or printed. Possible values are

- `:verbose`

  Return no values. Print discovered paths using a verbose format with each node of each path on a separate line.

- true (other than `:verbose`)

  Return no values. Print discovered paths using a compact format with all nodes of each path on a single line.

- `nil`

  Do not print any output. Instead return the discovered paths as a list of lists. Each list has the form

  ```
  (TARGET . (ROOT NODE*))
  ```

  where `target` is one of the target of one of the `weak-pointers`.

  `root` is a description of the root at which the path starts and has one of the following forms:

  - `:static`

    If the root of the path is a non-collectible heap object.

  - `:pinned`

    If an unknown thread stack pins the root of the path.

  - `((thread-name | thread-object) symbol currentp)`

    If the path begins at a special binding of `symbol` in a thread. `currentp` is a `boolean` indicating whether the value is current or shadowed by another binding.

  - `((thread-name | thread-object) guessed-pc)`

    If the path begins at a lexical variable in the function whose code contains `guessed-pc`.

  Each `node` in the remainder of the path is a cons (`object . slot`) indicating that the slot at index `slot` in `object` references the next path node.

Experimental: subject to change without prior notice.

An example of using this could look like this:

```
* (defvar *my-string* (list 1 2 "my string"))
*MY-STRING*

* (sb-ext:search-roots (sb-ext:make-weak-pointer (third *my-string*)))
-> ((SIMPLE-VECTOR 3)) #x10004E9EAF[2] -> (SYMBOL) #x5044100F[1] -> (CONS) #x100181FAE7[1] -> (C
```

The single line of output on `*standard-output*` shows the path from a root to `"my string"`: the path starts with SBCL's internal package system data structures followed by the symbol (`cl-user:*my-string*`) followed the three cons cells of the list.

The `:print :verbose` argument produces similar behavior but describes the path elements in more detail:

```
* (sb-ext:search-roots (sb-ext:make-weak-pointer (third *my-string*))
                       :print :verbose)
Path to "my string":
 6       10004E9EAF [   2] a (simple-vector 3)
 0         5044100F [   1] COMMON-LISP-USER::*MY-STRING*
 0       100181FAE7 [   1] a cons
```

```
0       100181FAF7 [   1] a cons
0       100181FB07 [   0] a cons
```

The `:print nil` argument is a bit different:

```
* (sb-ext:search-roots (sb-ext:make-weak-pointer (third *my-string*))
                       :print nil)
(("my string" :STATIC (#(*MY-STRING* 0 0) . 2) (*MY-STRING* . 1)
  ((1 2 "my string") . 1) ((2 "my string") . 1) (("my string") . 0)))
```

There is no output on `*standard-output*`, and the return value is a single path for the target object `"my string"`. As before, the path shows the symbol and the three cons cells.

7.6 Generic Function Dispatch

If a generic function with standard or short method combination is called, and the set of applicable methods does not include any primary methods, then the generic function `sb-pcl:no-primary-method` will be invoked with the arguments being the invoked generic function and its arguments, similar to the standard function `no-applicable-method`. As with `no-applicable-method`, the default method on `sb-pcl:no-primary-method` signals an error; programmers may define methods on it.

7.7 Extended Slot Access

The slot access functions `slot-value`, `(setf slot-value)`, `slot-boundp` and `slot-makunbound` are defined to function as expected on conditions (of metaclass `sb-pcl::condition-class`) and, with some limitations, on structures (of metaclass `structure-class`).

For structures:

- The name of a slot for the purposes of the slot access functions is the symbol used as the slot-name in the slot-description in the `defstruct` form;
- `slot-value` and `slot-boundp` function as expected, including (for `slot-value`) calling and respecting the return value of `slot-unbound` if the slot is unbound;
- `(setf slot-value)` functions as expected, including performing type checks to verify that the new value is of an appropriate type for the slot;
- `slot-makunbound` makes the slot unbound only when the slot corresponds to an `&aux` argument with no default in a by-order-of-arguments (BOA) constructor. In all other cases calling `slot-makunbound` on a structure signals an error.
- If any of the slot access functions is called with a structure instance which does not have a slot of the given name, `slot-missing` is called and the return value of the effective method, if any, is respected.

7.8 Metaobject Protocol

7.8.1 AMOP Compatibility of Metaobject Protocol

SBCL supports a metaobject protocol which is intended to be compatible with AMOP; present exceptions to this (as distinct from current bugs) are:

- `sb-mop:compute-effective-method` only returns one value, not two. There is no record of what the second return value was meant to indicate, and apparently no clients for it.
- The direct superclasses of `sb-mop:funcallable-standard-object` are (`function standard-object`) instead of the correct (`standard-object function`).

  This is to ensure that the `standard-object` class is the last of the standardized classes before class `t` appearing in the precedence list of `generic-function` and `standard-generic-function`, as required by `clhs 1.4.4.5`.
- The arguments `:declare` and `:declarations` are both accepted by `ensure-generic-function`, with the leftmost argument defining the declarations to be stored and returned by `sb-mop:generic-function-declarations`.

  Where AMOP specifies `:declarations` as the keyword argument to `ensure-generic-function`, the Common Lisp standard specifies `:declare`. Portable code should use `:declare`.

- Although SBCL obeys the requirement in AMOP that `sb-mop:validate-superclass` should treat `standard-class` and `sb-mop:funcallable-standard-class` as compatible metaclasses, we impose an additional requirement at class finalization time: a class of metaclass `sb-mop:funcallable-standard-class` must have `function` in its superclasses, and a class of metaclass `standard-class` must not.

  After a class has been finalized, it is associated with a class prototype which is accessible by a standard MOP function `sb-mop:class-prototype`. The user can then ask whether this object is a `function` or not in several different ways: whether it is a function according to `typep`; whether its `class-of` is `subtypep function`, or whether `function` appears in the superclasses of the class. The additional consistency requirement comes from the desire to make all of these answers the same.

  The following class definitions are bad, and will lead to errors either immediately or if an instance is created:

  ```
  (defclass bad-object (funcallable-standard-object)
    ()
    (:metaclass standard-class))
  (defclass bad-funcallable-object (standard-object)
    ()
    (:metaclass funcallable-standard-class))
  ```

  The following definition is acceptable:

  ```
  (defclass mixin ()
    ((slot :initarg slot)))
  (defclass funcallable-object (funcallable-standard-object mixin)
    ()
    (:metaclass funcallable-standard-class))
  ```

  and leads to a class whose instances are funcallable and have one slot.

  Note that this requirement also applies to the class `sb-mop:funcallable-standard-object`, which has metaclass `sb-mop:funcallable-standard-class` rather than `standard-class` as AMOP specifies.

- The requirement that *no portable class may inherit, by virtue of being a direct or indirect subclass of a specified class, any slot for which the name is a symbol accessible in the* `common-lisp-user` *package or exported by any package defined in the ANSI Common Lisp standard.* is interpreted to mean that the standardized classes themselves should not have slots named by external symbols of public packages.

  The rationale behind the restriction is likely to be similar to the ANSI Common Lisp restriction on defining functions, variables and types named by symbols in the Common Lisp package: preventing two independent pieces of software from colliding with each other.

- Specializations of the `new-value` argument to (`setf sb-mop:slot-value-using-class`) are not allowed: all user-defined methods must have a specializer of the class `t`.

  This prohibition is motivated by a separation of layers: the `sb-mop:slot-value-using-class` family of functions is intended for use in implementing different and new slot allocation strategies, rather than in performing application-level dispatching. Additionally, with this requirement, there is a one-to-one mapping between metaclass, class and slot-definition-class tuples and effective methods of (`setf sb-mop:slot-value-using-class`), which permits optimization of (`setf sb-mop:slot-value-using-class`)'s discriminating function in the same manner as for `sb-mop:slot-value-using-class` and `sb-mop:slot-boundp-using-class`.

  Note that application code may specialize on the `new-value` argument of slot accessors.

- The class named by the `name` argument to `sb-mop:ensure-class`, if any, is only redefined if it is the proper name of that class; otherwise, a new class is created.

  This is consistent with the description `sb-mop:ensure-class` in AMOP as the functional version of `defclass`, which has this behaviour; however, it is not consistent with the weaker requirement in AMOP, which states that any class found by `find-class`, no matter what its `class-name`, is redefined.

- An error is not signaled in the case of the `:name` initialization argument for `sb-mop:slot-definition` objects being a constant, when the slot definition is of type `sb-pcl::structure-slot-definition` (i.e. it is associated with a class of type `structure-class`).

  This allows code which uses constant names for structure slots to continue working as specified in ANSI, while enforcing the constraint for all other types of slot.

- The class `t` is not an instance of the `built-in-class` metaclass.

  AMOP specifies, in the _Inheritance Structure of Metaobject Classes_ section, that the class `t` should be an instance of `built-in-class`. However, it also specifies that `sb-mop:validate-superclass` should return true (indicating that a direct superclass relationship is permissible) if the second argument is the class `t`. Also, ANSI specifies that classes with metaclass `built-in-class` may not be subclassed using `defclass`, and also that the class `t` is the universal superclass, inconsistent with it being a `built-in-class`.
- Uses of `change-class` and redefinitions of classes with `defclass` (or the functional interfaces `sb-mop:ensure-class` or `sb-mop:ensure-class-using-class`) must ensure that for each slot with allocation `:instance` or `:class`, the set of applicable methods on the `sb-mop:slot-value-using-class` family of generic functions is the same before and after the change.

  This is required for correct operation of the protocol to update instances for the new or redefined class, and can be seen as part of the contract of the `:instance` or `:class` allocations.
- Metaobject protocol users may wish to override `sb-mop:compute-discriminating-function` for their own generic function classes. Overriding implementations of `sb-mop:compute-discriminating-function` must, in order to participate in the `no-applicable-method` and `sb-pcl:no-primary-method` protocols, perform appropriate checks on the return value of `compute-applicable-methods` before processing the effective method; the standard effective method contains error-invoking forms, but those forms have no access to the generic function invocation's arguments.

7.8.2 Metaobject Protocol Extensions

In addition, SBCL supports extensions to the Metaobject protocol from AMOP; at present, they are:

- Compile-time support for generating specializer metaobjects from specializer names in `defmethod` forms is provided by the `sb-pcl:make-method-specializers-form` function, which returns a form which, when evaluated in the lexical environment of the `defmethod`, returns a list of specializer metaobjects. This operator suffers from similar restrictions to those affecting `sb-mop:make-method-lambda`, namely that the generic function must be defined when the `defmethod` form is expanded, so that the correct method of `sb-pcl:make-method-specializers-form` is invoked. The system-provided method on `sb-pcl:make-method-specializers-form` generates a call to `find-class` for each symbol specializer name, and a call to `sb-mop:intern-eql-specializer` for each `(EQL <x>)` specializer name.
- Run-time support for converting between specializer names and specializer metaobjects, mostly for the purposes of `find-method`, is provided by `sb-pcl:parse-specializer-using-class` and `sb-pcl:unparse-specializer-using-class`, which dispatch on their first argument, the generic function associated with a method with the given specializer. The system-provided methods on those methods convert between classes and proper names and between lists of the form `(EQL <x>)` and interned eql specializer objects.
- Distinguishing unbound instance allocated slots from bound ones when using `sb-mop:standard-instance-access` and `sb-mop:funcallable-standard-instance-access` is possible by comparison to the symbol-macro `sb-pcl:+slot-unbound+`.

7.9 Extensible Sequences

ANSI Common Lisp has a class `sequence` with subclasses `list` and `vector`, on which the sequence functions like `find`, `subseq`, etc. operate. As an extension to the ANSI specification, SBCL allows additional subclasses of `sequence` to be defined.

> A motivation, rationale and additional examples for the design of this extension can be found in the paper *Rhodes, Christophe (2007): User-extensible sequences in Common Lisp* available for download at `http://www.doc.gold.ac.uk/~mas01cr/papers/ilc2007/sequences-20070301.pdf`.

Users of this extension just make instances of `sequence` subclasses and transparently operate on them using sequence functions:

```
(coerce (subseq (make-instance 'my-sequence) 5 10) 'list)
```

From this perspective, no distinction between builtin and user-defined `sequence` subclasses should be necessary.

Providers of the extension, that is of user-defined `sequence` subclasses, have to adhere to a *sequence protocol* which consists of a set of generic functions in the `sequence` package.

A minimal `sequence` subclass has to specify `standard-object` and `sequence` as its superclasses and has to be the specializer of the `sequence` parameter of methods on at least the following generic functions:

`sb-sequence:length` *sequence* [Generic function]
Returns the length of `sequence` or signals a `sequence:protocol-unimplemented` error if the sequence protocol is not implemented for the class of `sequence`.

`sb-sequence:elt` *sequence index* [Generic function]
Returns the element at position `index` of `sequence` or signals a `sequence:protocol-unimplemented` error if the sequence protocol is not implemented for the class of `sequence`.

`sb-sequence:elt` *sequence index* [Setf generic function]
Returns the element at position `index` of `sequence` or signals a `sequence:protocol-unimplemented` error if the sequence protocol is not implemented for the class of `sequence`.

`sb-sequence:adjust-sequence` *sequence length* &key *initial-element initial-contents* [Generic function]
Returns destructively modified `sequence` or a freshly allocated sequence of the same class as `sequence` of length `length`. Elements of the returned sequence are initialized to `initial-element`, if supplied, initialized to `initial-contents` if supplied, or identical to the elements of `sequence` if neither is supplied. Signals a `sequence:protocol-unimplemented` error if the sequence protocol is not implemented for the class of `sequence`.

`sb-sequence:make-sequence-like` *sequence length* &key *initial-element initial-contents* [Generic function]
Returns a freshly allocated sequence of length `length` and of the same class as `sequence`. Elements of the new sequence are initialized to `initial-element`, if supplied, initialized to `initial-contents` if supplied, or undefined if neither is supplied. Signals a `sequence:protocol-unimplemented` error if the sequence protocol is not implemented for the class of `sequence`.

`make-sequence-like` is needed for functions returning freshly-allocated sequences such as `subseq` or `copy-seq`. `adjust-sequence` is needed for functions which destructively modify their arguments such as `delete`. In fact, all other sequence functions can be implemented in terms of the above functions and actually are, if no additional methods are defined. However, relying on these generic implementations, in particular not implementing the Section 7.9.1 [exseq iterator protocol], page 71, can incur a high performance penalty.

When the sequence protocol is only partially implemented for a given `sequence` subclass, an attempt to apply one of the missing operations to instances of that class signals the following condition:

`sb-sequence:protocol-unimplemented` [Condition]
This error is signaled if a sequence operation is applied to an instance of a sequence class that does not support the operation.

In addition to the mandatory functions above, methods on the sequence functions listed below can be defined. There are some noteworthy irregularities:

- The function `sb-sequence:emptyp` does not have a counterpart in the `cl` package. It is intended to be used instead of `sb-sequence:length` when working with lazy or infinite sequences.
- `sb-sequence:dosequence` does not have a direct counterpart either. It is like `dolist` in spirit but traverses generic sequences.
- The functions `map`, `concatenate` and `merge` receive a type designator specifying the type of the constructed sequence as their first argument. However, the corresponding generic functions `sb-sequence:map`, `sb-sequence:concatenate` and `sb-sequence:merge` receive a prototype instance of the requested `sequence` subclass instead.
- `cl:map-into` has no generic sequence counterpart, as its lambda list does not provide reasonable specialization opportunities, but it supports extensible sequences directly.

`sb-sequence:emptyp` *sequence* [Generic function]
Returns `t` if `sequence` is an empty sequence and `nil` otherwise. Signals an error if `sequence` is not a sequence.

`sb-sequence:dosequence` (*element sequence* &optional *return*) &body *body* [Macro]
Executes `body` with `element` subsequently bound to each element of `sequence`, then returns `return`.

The remaining list parallels the *Sequence Dictionary*, see 17.3 in the ANSI spec.

`sb-sequence:copy-seq` *sequence* [Generic function]

`sb-sequence:fill` *sequence item* &key *start end* [Generic function]

`sb-sequence:subseq` *sequence start* &optional *end* [Generic function]

`sb-sequence:map` *result-prototype function sequence* &rest *sequences* [Function]
Implements `cl:map` for extended sequences.

`result-prototype` corresponds to the `result-type` of `cl:map` but receives a prototype instance of an extended sequence class instead of a type specifier. By dispatching on `result-prototype`, methods on this generic function specify how extended sequence classes act when they are specified as the result type in a `cl:map` call. `result-prototype` may not be fully initialized and thus should only be used for dispatch and to determine its class.

Another difference to `cl:map` is that `function` is a function, not a function designator.

`sb-sequence:reduce` *function sequence* &key *from-end start end initial-value key* [Generic function]

`sb-sequence:search` *sequence1 sequence2* &key *from-end start1 end1 start2 end2 test test-not key* [Generic function]

`sb-sequence:mismatch` *sequence1 sequence2* &key *from-end start1 end1 start2 end2 test test-not key* [Generic function]

`sb-sequence:replace` *sequence1 sequence2* &key *start1 end1 start2 end2* [Generic function]

`sb-sequence:concatenate` *result-prototype* &rest *sequences* [Function]
Implements `cl:concatenate` for extended sequences.

`result-prototype` corresponds to the `result-type` of `cl:concatenate` but receives a prototype instance of an extended sequence class instead of a type specifier. By dispatching on `result-prototype`, methods on this generic function specify how extended sequence classes act when they are specified as the result type in a `cl:concatenate` call. `result-prototype` may not be fully initialized and thus should only be used for dispatch and to determine its class.

`sb-sequence:merge` *result-prototype sequence1 sequence2 predicate* &key *key* [Function]
Implements `cl:merge` for extended sequences.

`result-prototype` corresponds to the `result-type` of `cl:merge` but receives a prototype instance of an extended sequence class instead of a type specifier. By dispatching on `result-prototype`, methods on this generic function specify how extended sequence classes act when they are specified as the result type in a `cl:merge` call. `result-prototype` may not be fully initialized and thus should only be used for dispatch and to determine its class.

Another difference to `cl:merge` is that `predicate` is a function, not a function designator.

Counting:

`sb-sequence:count` *item sequence* &key *from-end start end test test-not key* [Generic function]

`sb-sequence:count-if` *pred sequence* &key *from-end start end key* [Generic function]

`sb-sequence:count-if-not` *pred sequence* &key *from-end start end key* [Generic function]

Reversing:

`sb-sequence:reverse` *sequence* [Generic function]

`sb-sequence:nreverse` *sequence* [Generic function]

Sorting:

`sb-sequence:sort` *sequence predicate* &key *key* [Generic function]

`sb-sequence:stable-sort` *sequence predicate* &key *key* [Generic function]

Finding an element:

`sb-sequence:find` *item sequence* &key *from-end start end test test-not key* [Generic function]

`sb-sequence:find-if` *pred sequence* &key *from-end start end key* [Generic function]

`sb-sequence:find-if-not` *pred sequence* &key *from-end start end key* [Generic function]

Finding a position:

`sb-sequence:position` *item sequence* &key *from-end start end test test-not key* [Generic function]

`sb-sequence:position-if` *pred sequence* &key *from-end start end key* [Generic function]

`sb-sequence:position-if-not` *pred sequence* &key *from-end start end key* [Generic function]

Substituting elements:

`sb-sequence:substitute` *new old sequence* &key *start end from-end test test-not count key* [Generic function]

`sb-sequence:substitute-if` *new predicate sequence* &key *start end from-end count key* [Generic function]

`sb-sequence:substitute-if-not` *new predicate sequence* &key *start end from-end count key* [Generic function]

`sb-sequence:nsubstitute` *new old sequence* &key *start end from-end test test-not count key* [Generic function]

`sb-sequence:nsubstitute-if` *new predicate sequence* &key *start end from-end count key* [Generic function]

`sb-sequence:nsubstitute-if-not` *new predicate sequence* &key *start end from-end count key* [Generic function]

Removing elements:

`sb-sequence:remove` *item sequence* &key *from-end test test-not start end count key* [Generic function]

`sb-sequence:remove-if` *predicate sequence* &key *from-end start end count key* [Generic function]

`sb-sequence:remove-if-not` *predicate sequence* &key *from-end start end count key* [Generic function]

`sb-sequence:delete` *item sequence* &key *from-end test test-not start end count key* [Generic function]

`sb-sequence:delete-if` *predicate sequence* &key *from-end start end count key* [Generic function]

`sb-sequence:delete-if-not` *predicate sequence* &key *from-end start end count key* [Generic function]

Removing duplicates:

`sb-sequence:remove-duplicates` *sequence* &key *from-end test test-not start end key* [Generic function]

`sb-sequence:delete-duplicates` *sequence* &key *from-end test test-not start end key* [Generic function]

7.9.1 Iterator Protocol

The general iterator protocol allows subsequently accessing some or all elements of a sequence in forward or reverse direction. Users first call `sb-sequence:make-sequence-iterator` to create an iteration state and receive functions to query and mutate it. These functions allow, among other things, moving to, retrieving or modifying elements of the sequence. The iteration state consists of a state object, a limit object, a from-end indicator and six functions to query or mutate this state.

An iterator is created by calling:

`sb-sequence:make-sequence-iterator` *sequence* &key *from-end start end* [Function]
Returns a sequence iterator for `sequence` or, if `start` and/or `end` are supplied, the subsequence bounded by `start` and `end` as nine values:

1. iterator state 2. limit 3. from-end 4. step function 5. endp function 6. element function 7. setf element function 8. index function 9. copy state function

If `from-end` is `nil`, the constructed iterator visits the specified elements in the order in which they appear in `sequence`. Otherwise, the elements are visited in the opposite order.

The six functions (items 4-9 in the list) have the same contract as the generic functions described in Section 7.9.2 [exseq simple iterator protocol], page 72. In fact, when there is no specialized method for a particular `sequence` subclass, `sb-sequence:make-sequence-iterator` calls `sb-sequence:make-simple-sequence-iterator` and returns those six generic functions.

The following convenience macros simplify traversing sequences using iterators:

`sb-sequence:with-sequence-iterator` (&optional *iterator limit from-end-p step endp element set-element index copy*) (*sequence* &key *from-end* (*start 0*) *end*) &body *body* [Macro]
Executes `body` with the elements of `vars` bound to the iteration state returned by `sequence:make-sequence-iterator` for `sequence` and `args`. Elements of `vars` may be `nil` in which case the corresponding value returned by `sequence:make-sequence-iterator` is ignored.

`sb-sequence:with-sequence-iterator-functions` (&optional *step endp elt setf index copy*) (*sequence* &rest *args* &key *from-end start end*) &body *body* [Macro]
Executes `body` with the names `step`, `endp`, `elt`, `setf`, `index` and `copy` bound to local functions which execute the iteration state query and mutation functions returned by `sequence:make-sequence-iterator` for `sequence` and `args`. When some names are not supplied or `nil` is supplied for a given name, no local functions are established for those names. The functions established for `step`, `endp`, `elt`, `setf`, `index` and `copy` have dynamic extent.

7.9.2 Simple Iterator Protocol

For cases in which the full flexibility and performance of the general sequence iterator protocol is not required, there is a simplified sequence iterator protocol consisting of a few generic functions which can be specialized for iterator classes:

`sb-sequence:iterator-step` *sequence iterator from-end* [Generic function]
Moves `iterator` one position forward or backward in `sequence` depending on the iteration direction encoded in `from-end`.

`sb-sequence:iterator-endp` *sequence iterator limit from-end* [Generic function]
Returns non-`nil` when `iterator` has reached `limit` (which may correspond to the end of `sequence`) with respect to the iteration direction encoded in `from-end`.

`sb-sequence:iterator-element` *sequence iterator* [Generic function]
Returns the element of `sequence` associated to the position of `iterator`.

`sb-sequence:iterator-element` *sequence iterator* [Setf generic function]
Returns the element of `sequence` associated to the position of `iterator`.

`sb-sequence:iterator-index` *sequence iterator* [Generic function]
Returns the position of `iterator` in `sequence`.

`sb-sequence:iterator-copy` *sequence iterator* [Generic function]
Returns a copy of `iterator` which also traverses `sequence` but can be mutated independently of `iterator`.

Iterator objects implementing the above simple iteration protocol are created by calling the following generic function:

`sb-sequence:make-simple-sequence-iterator` *sequence* &key *from-end start end* [Generic function]
Returns a sequence iterator for `sequence`, `start`, `end` and `from-end` as three values:

1. iterator state 2. limit 3. from-end

The returned iterator can be used with the generic iterator functions described in Section 7.9.2 [exseq simple iterator protocol], page 72.

7.10 Unix Support

`sb-ext:*posix-argv*` [Variable]
A list of strings related to the UNIX command line (`argv` in C).
Section 3.3.1 [runtime options], page 11, are processed and removed by the runtime. The default toplevel also removes Section 3.3.2 [toplevel options], page 12, that it processes.

`sb-ext:posix-getenv` *name* [Function]
Return the `value` part of the environment string `name=value` which corresponds to `name`, or `nil` if there is none. See `getenv(3)`.

`sb-ext:posix-environ` [Function]
Return the Unix environment as a list of `simple-string`s. See `man environ`.

`sb-ext:exit` &key *code abort* (*timeout sb-ext:*exit-timeout**) [Function]
Terminates the process, causing SBCL to exit with `code`. `code` defaults to 0 when `abort` is false, and 1 when it is true.
When `abort` is false (the default), current thread is first unwound, `*exit-hooks*` are run, other threads are terminated, and standard output streams are flushed before SBCL calls `exit(3)` – at which point `atexit(3)` functions will run. If multiple threads call `exit` with `abort` being false, the first one to call it will complete the protocol.
When `abort` is true, SBCL exits immediately by calling `_exit(2)` without unwinding stack, or calling exit hooks. Note that `_exit(2)` does not call `atexit(3)` functions unlike `exit(3)`.
Recursive calls to `exit` cause `exit` to behave as if `abort` was true.
`timeout` controls waiting for other threads to terminate when `abort` is `nil`. Once current thread has been unwound and `*exit-hooks*` have been run, spawning new threads is prevented and all other threads are terminated by calling `sb-thread:terminate-thread` on them. The system then waits for them to finish using `sb-thread:join-thread`, waiting at most a total `timeout` seconds for all threads to join. Those threads that do not finish in time are simply ignored while the exit protocol continues. `timeout` defaults to `*exit-timeout*`, which in turn defaults to 60. `timeout nil` means to wait indefinitely.
Note that `timeout` applies only to `sb-thread:join-thread`, not `*exit-hooks*`. Since `sb-thread:terminate-thread` is asynchronous, getting multithreaded application termination with complex cleanups right using it can be tricky. To perform an orderly synchronous shutdown use an exit hook instead of relying on implicit thread termination.
Consequences are unspecified if serious conditions occur during `exit` excepting errors from `*exit-hooks*`, which cause warnings and stop execution of the hook that signaled, but otherwise allow the exit process to continue normally.

7.10.1 Running external programs

External programs can be run with `sb-ext:run-program`.

> *Note*: In SBCL versions prior to 1.0.13, `sb-ext:run-program` searched for executables in a manner somewhat incompatible with other languages. As of this version, SBCL uses the system library routine `execvp(3)`, and no longer contains the function `find-executable-in-search-path`, which implemented the old search. Users who need this function may find it in `run-program.lisp` versions 1.67 and earlier in SBCL's CVS repository here `http://sbcl.cvs.sourceforge.net/sbcl/sbcl/src/code/run-program.lisp?view=log`. However, we caution such users that this search routine finds executables that system library routines do not.

`sb-ext:run-program` *program args* &key *env environment* (*wait t*) *search pty input* *if-input-does-not-exist output* (*if-output-exists :error*) (*error :output*) (*if-error-exists :error*) *status-hook* (*external-format :default*) *directory preserve-fds use-posix-spawn* [Function]
`run-program` creates a new process specified by `program`. `args` is a list of strings to be passed literally to the new program. In POSIX environments, this list becomes the array supplied as the second parameter to the execv() or execvp() system call, each list element becoming one array element. The strings should not contain shell escaping, as there is no shell involvement. Further note that while conventionally the process receives its own pathname in argv[0], that is automatic, and the 0th string should not be present in `args`.
The program arguments and the environment are encoded using the default external format for streams.

`run-program` will return a `process` structure. See the CMU Common Lisp Users Manual for details about the `process` structure.

Notes about Unix environments (as in the `:environment` and `:env` args):

- The SBCL implementation of `run-program`, like Perl and many other programs, but unlike the original CMUCL implementation, copies the Unix environment by default.
- Running Unix programs from a setuid process, or in any other situation where the Unix environment is under the control of someone else, is a mother lode of security problems. If you are contemplating doing this, read about it first. (The Perl community has a lot of good documentation about this and other security issues in script-like programs.)

The `&key` arguments have the following meanings:

- `:environment`

  A list of `string`s describing the new Unix environment (as in "man environ"). The default is to copy the environment of the current process.
- `:env`

  An old, lossy representation of the new Unix environment, for compatibility with CMUCL. If given, `:environment` defaults to this (converted to the new format). It is an error to specify both `:env` and `:environment`.
- `:search`

  Look for `program` in each of the directories in the child's `$PATH` environment variable. Otherwise an absolute pathname is required.
- `:wait`

  If non-`nil` (default), wait until the created process finishes. If `nil`, continue running Lisp until the program finishes.
- `:pty` (not supported on win32)

  Either `t`, `nil`, or a stream. Unless `nil`, the subprocess is established under a `pty`. If `:pty` is a stream, all output to this pty is sent to this stream, otherwise the `process-pty` slot is filled in with a stream connected to pty that can read output and write input.
- `:input`

  Either `t`, `nil` (the default), a pathname, a stream, or `:stream`.
  - `t`: the standard input for the current process is inherited.
  - `nil`: `/dev/null` (nul on win32) is used.
  - Pathname: the specified file is used.
  - Stream: all the input is read from that stream and sent to the subprocess.
  - `:stream`: the `process-input` slot is filled in with a stream that sends its output to the process.
- `:if-input-does-not-exist` (when `:input` is the name of a file)

  It is one of:
  - `:error` to generate an error
  - `:create` to create an empty file
  - `nil` (the default) to return `nil` from `run-program`
- `:output`

  Either `t`, `nil` (the default), a pathname, a stream, or `:stream`.
  - `t`: the standard output for the current process is inherited.
  - `nil`: `/dev/null` (nul on win32) is used.
  - Pathname: the specified file is used.
  - Stream: all the output from the process is written to this stream.
  - `:stream`: the `process-output` slot is filled in with a stream that can be read to get the output.
- `:error`

  Same as `:output`, additionally accepts `:output`, making all error output routed to the same place as normal output. Defaults to `:output`.

- `:if-output-exists` (when `:output` is the name of a file)

  It is one of:

  - `:error` (the default) to generate an error
  - `:supersede` to supersede the file with output from the program
  - `:append` to append output from the program to the file
  - `nil` to return `nil` from `run-program`, without doing anything
- `:if-error-exists`

  Same as `:if-output-exists`, controlling `:error` output to files. Ignored when `:error :output`. Defaults to `:error`.
- `:status-hook`

  This is a function the system calls whenever the status of the process changes. The function takes the process as an argument.
- `:external-format`

  The external-format to use for `:input`, `:output`, and `:error :stream`s.
- `:directory`

  Specifies the directory in which the program should be run. `nil` (the default) means the directory is unchanged.
- `:preserve-fds`

  A sequence of file descriptors which should remain open in the child process.

Windows specific options:

- `:escape-arguments` (default `t`)

  Controls escaping of the arguments passed to CreateProcess.
- `:window` (default `nil`)

  When `nil`, the subprocess decides how it will display its window. The following options control how the subprocess window should be displayed: `:hide`, `:show-normal`, `:show-maximized`, `:show-minimized`, `:show-no-activate`, `:show-min-no-active`, `:show-na`.

  > *Note*: console application subprocesses may or may not display a console window depending on whether the SBCL runtime is itself a console or GUI application. Invoke `cmd /c start` to consistently display a console window or use the `:window :hide` option to consistently hide the console window.

When `sb-ext:run-program` is called with `:wait nil`, an process object is returned. The following functions are available for use with processes:

`sb-ext:process-p` *object* [Function]
`t` if `object` is a `process`, `nil` otherwise.

`sb-ext:process-input` *instance* [Function]
The input stream of the process or `nil`.

`sb-ext:process-output` *instance* [Function]
The output stream of the process or `nil`.

`sb-ext:process-error` *instance* [Function]
The error stream of the process or `nil`.

`sb-ext:process-alive-p` *process* [Function]
Return `t` if `process` is still alive, `nil` otherwise. Can return a false positive on a closed process.

`sb-ext:process-status` *process* [Function]
Return the current status of `process`. The result is one of `:running`, `:stopped`, `:exited`, `:signaled`.

`sb-ext:process-wait` *process* &optional *check-for-stopped* [Function]
Wait for `process` to quit running for some reason. When `check-for-stopped` is `t`, also returns when `process` is stopped. Returns `process`.

`sb-ext:process-exit-code` *process* [Function]
The exit code or the signal of a stopped process.

`sb-ext:process-core-dumped` *instance* [Function]
`t` if a core image was dumped by the process.

`sb-ext:process-close` *process* [Function]
Close all streams connected to `process`, stop maintaining the status slot. After `process-close`, `process-alive-p` and `process-exit-code` can return stale information about a process, so should not be used.

`sb-ext:process-kill` *process signal* &optional (*whom :pid*) [Function]
Hand `signal` to `process`. If `whom` is `:pid`, use the kill Unix system call. If `whom` is `:process-group`, use the `killpg(1)` Unix system call. Returns `t` if successful, otherwise returns `nil` and error number (two values).

7.11 Unicode Support

SBCL provides support for working with Unicode text and querying the standard Unicode database for information about individual codepoints. Unicode-related functions are located in the `sb-unicode` package.
SBCL also extends ANSI character literal syntax to support Unicode codepoints. You can either specify a character by its Unicode name, with spaces replaced by underscores if a unique name exists or by giving its hexadecimal codepoint preceded by a `u`, an optional `+`, and an arbitrary number of leading zeros. You may also input the character directly into your source code if it can be encoded in your file. If a character had an assigned name in Unicode 1.0 that was distinct from its current name, you may also use that name (with spaces replaced by underscores) to specify the character, unless the name is already associated with a codepoint in the latest Unicode standard (such as `bell`).

> *Note*: Please note that the codepoint `u+1f5cf` (Page) introduced in Unicode 7.0 is named `unicode_page`, since the name *Page* is required to be assigned to form-feed (`u+0c`) by the ANSI standard.

For example, you can specify the codepoint `u+00e1` (*Latin Small Letter A With Acute*) as

- `#\latin_small_letter_a_with_acute`
- `#\latin_small_letter_a_acute`
- `#\á` (assuming a Unicode source file)
- `#\u00e1`
- `#\ue1`
- `#\u+00e1`

7.11.1 Unicode property access

The following functions can be used to find information about a Unicode codepoint.

`sb-unicode:general-category` *character* [Function]
Returns the general category of `character` as it appears in UnicodeData.txt

`sb-unicode:bidi-class` *character* [Function]
Returns the bidirectional class of `character`

`sb-unicode:combining-class` *character* [Function]
Returns the canonical combining class (CCC) of `character`

`sb-unicode:decimal-value` *character* [Function]
Returns the decimal digit value associated with `character` or `nil` if there is no such value.
The only characters in Unicode with a decimal digit value are those that are part of a range of characters that encode the digits 0-9. Because of this, `(decimal-digit c) <=> (digit-char-p c 10)` in `#+sb-unicode` builds

`sb-unicode:digit-value` *character* [Function]
Returns the Unicode digit value of `character` or `nil` if it doesn't exist.
Digit values are guaranteed to be integers between 0 and 9 inclusive. All characters with decimal digit values have the same digit value, but there are characters (such as digits of number systems without a 0 value) that have a digit value but no decimal digit value

`sb-unicode:numeric-value` *character* [Function]
Returns the numeric value of `character` or `nil` if there is no such value. Numeric value is the most general of the Unicode numeric properties. The only constraint on the numeric value is that it be a rational number.

`sb-unicode:mirrored-p` *character* [Function]
Returns `t` if `character` needs to be mirrored in bidirectional text. Otherwise, returns `nil`.

`sb-unicode:bidi-mirroring-glyph` *character* [Function]
Returns the mirror image of `character` if it exists. Otherwise, returns `nil`.

`sb-unicode:age` *character* [Function]
Returns the version of Unicode in which `character` was assigned as a pair of values, both integers, representing the major and minor version respectively. If `character` is not assigned in Unicode, returns `nil` for both values.

`sb-unicode:hangul-syllable-type` *character* [Function]
Returns the Hangul syllable type of `character`. The syllable type can be one of `:l`, `:v`, `:t`, `:lv`, or `:lvt`. If the character is not a Hangul syllable or Jamo, returns `nil`

`sb-unicode:east-asian-width` *character* [Function]
Returns the East Asian Width property of `character` as one of the keywords `:n` (Narrow), `:a` (Ambiguous), `:h` (Halfwidth), `:w` (Wide), `:f` (Fullwidth), or `:na` (Not applicable)

`sb-unicode:script` *character* [Function]
Returns the Script property of `character` as a keyword. If `character` does not have a known script, returns `:unknown`

`sb-unicode:char-block` *character* [Function]
Returns the Unicode block in which `character` resides as a keyword. If `character` does not have a known block, returns `:no-block`

`sb-unicode:unicode-1-name` *character* [Function]
Returns the name assigned to `character` in Unicode 1.0 if it is distinct from the name currently assigned to `character`. Otherwise, returns `nil`. This property has been officially obsoleted by the Unicode standard, and is only included for backwards compatibility.

`sb-unicode:proplist-p` *character property* [Function]
Returns `t` if `character` has the specified `property`. `property` is a keyword representing one of the properties from PropList.txt, with underscores replaced by dashes.

`sb-unicode:uppercase-p` *character* [Function]
Returns `t` if `character` has the Unicode property Uppercase and `nil` otherwise

`sb-unicode:lowercase-p` *character* [Function]
Returns `t` if `character` has the Unicode property Lowercase and `nil` otherwise

`sb-unicode:cased-p` *character* [Function]
Returns `t` if `character` has a (Unicode) case, and `nil` otherwise

`sb-unicode:case-ignorable-p` *character* [Function]
Returns `t` if `character` is Case Ignorable as defined in Unicode 6.3, Chapter 3

`sb-unicode:alphabetic-p` *character* [Function]
Returns `t` if `character` is Alphabetic according to the Unicode standard and `nil` otherwise

`sb-unicode:ideographic-p` *character* [Function]
Returns `t` if `character` has the Unicode property Ideographic, which loosely corresponds to the set of "Chinese characters"

`sb-unicode:math-p` *character* [Function]
Returns `t` if `character` is a mathematical symbol according to Unicode and `nil` otherwise

`sb-unicode:whitespace-p` *character* [Function]
Returns `t` if `character` is whitespace according to Unicode and `nil` otherwise

`sb-unicode:soft-dotted-p` *character* [Function]
Returns `t` if `character` has a soft dot (such as the dots on i and j) which disappears when accents are placed on top of it. and `nil` otherwise

`sb-unicode:hex-digit-p` *character* &key *ascii* [Function]
Returns `t` if `character` is a hexadecimal digit and `nil` otherwise. If `:ascii` is non-`nil`, fullwidth equivalents of the Latin letters A through F are excluded.

`sb-unicode:default-ignorable-p` *character* [Function]
Returns `t` if `character` is a Default*Ignorable*Code_Point

`sb-unicode:grapheme-break-class` *character* [Function]
Returns the grapheme breaking class of `character`, as specified in UAX #29.

`sb-unicode:word-break-class` *character* [Function]
Returns the word breaking class of `character`, as specified in UAX #29.

`sb-unicode:sentence-break-class` *character* [Function]
Returns the sentence breaking class of `character`, as specified in UAX #29.

`sb-unicode:line-break-class` *character* [Function]
Returns the line breaking class of `character`, as specified in the Unicode character database.

7.11.2 String operations

SBCL can normalize strings using:

`sb-unicode:normalize-string` *string* &optional (*form :nfd*) *filter* [Function]
Normalize `string` to the Unicode normalization form `form`. Acceptable values for form are `:nfd`, `:nfc`, `:nfkd`, and `:nfkc`. If `filter` is a function it is called on each decomposed character and only characters for which it returns `t` are collected.

`sb-unicode:normalized-p` *string* &optional (*form :nfd*) [Function]
Tests if `string` is normalized to `form`

SBCL implements the full range of Unicode case operations with the functions

`sb-unicode:uppercase` *string* &key *locale* [Function]
Returns the full uppercase of `string` according to the Unicode standard. The result is not guaranteed to have the same length as the input. If `:locale` is `nil`, no language-specific case transformations are applied. If `:locale` is a keyword representing a two-letter ISO country code, the case transforms of that locale are used. If `:locale` is `t`, the user's current locale is used (Unix and Win32 only).

`sb-unicode:lowercase` *string* &key *locale* [Function]
Returns the full lowercase of `string` according to the Unicode standard. The result is not guaranteed to have the same length as the input. `:locale` has the same semantics as the `:locale` argument to `uppercase`.

`sb-unicode:titlecase` *string* &key *locale* [Function]
Returns the titlecase of `string`. The resulting string can be longer than the input. `:locale` has the same semantics as the `:locale` argument to `uppercase`.

`sb-unicode:casefold` *string* [Function]
Returns the full casefolding of `string` according to the Unicode standard. Casefolding removes case information in a way that allows the results to be used for case-insensitive comparisons. The result is not guaranteed to have the same length as the input.

It also extends standard Common Lisp case functions such as `string-upcase` and `string-downcase` to support a subset of Unicode's casing behavior. Specifically, a character is `both-case-p` if its case mapping in Unicode is one-to-one and invertable.

The `sb-unicode` package also provides functions for collating/sorting strings according to the Unicode Collation Algorithm.

`sb-unicode:unicode<` *string1 string2* &key (*start1 0*) *end1* (*start2 0*) *end2* [Function]
Determines whether `string1` sorts before `string2` using the Unicode Collation Algorithm. The function uses an untailored Default Unicode Collation Element Table to produce the sort keys. The function uses the Shifted method for dealing with variable-weight characters, as described in UTS #10

`sb-unicode:unicode=` *string1 string2* &key (*start1 0*) *end1* (*start2 0*) *end2* (*strict t*) [Function]
Determines whether `string1` and `string2` are canonically equivalent according to Unicode. The `start` and `end` arguments behave like the arguments to `string=`. If `:strict` is `nil`, `unicode=` tests compatibility equivalence instead.

`sb-unicode:unicode-equal` *string1 string2* &key (*start1 0*) *end1* (*start2 0*) *end2* (*strict t*) [Function]
Determines whether `string1` and `string2` are canonically equivalent after casefolding (that is, ignoring case differences) according to Unicode. The `start` and `end` arguments behave like the arguments to `string=`. If `:strict` is `nil`, `unicode=` tests compatibility equivalence instead.

`sb-unicode:unicode<=` *string1 string2* &key (*start1 0*) *end1* (*start2 0*) *end2* [Function]
Tests if `string1` and `string2` are either `unicode<` or `unicode=`

`sb-unicode:unicode>` *string1 string2* &key (*start1 0*) *end1* (*start2 0*) *end2* [Function]
Tests if `string2` is `unicode<` `string1`.

`sb-unicode:unicode>=` *string1 string2* &key (*start1 0*) *end1* (*start2 0*) *end2* [Function]
Tests if `string1` and `string2` are either `unicode=` or `unicode>`

The following functions are provided for detecting visually confusable strings:

`sb-unicode:confusable-p` *string1 string2* &key (*start1 0*) *end1* (*start2 0*) *end2* [Function]
Determines whether `string1` and `string2` could be visually confusable according to the IDNA confusableSummary.txt table

7.11.3 Breaking strings

The `sb-unicode` package includes several functions for breaking a Unicode string into useful parts.

`sb-unicode:graphemes` *string* [Function]
Breaks `string` into graphemes according to the default grapheme breaking rules specified in UAX #29, returning a list of strings.

`sb-unicode:words` *string* [Function]
Breaks `string` into words according to the default word breaking rules specified in UAX #29. Returns a list of strings

`sb-unicode:sentences` *string* [Function]
Breaks `string` into sentences according to the default sentence breaking rules specified in UAX #29

`sb-unicode:lines` *string* &key (*margin *print-right-margin**) [Function]
Breaks `string` into lines that are no wider than `:margin` according to the line breaking rules outlined in UAX #14. Combining marks will always be kept together with their base characters, and spaces (but not other types of whitespace) will be removed from the end of lines. If `:margin` is unspecified, it defaults to 80 characters

7.12 Customization Hooks for Users

The toplevel repl prompt may be customized, and the function that reads user input may be replaced completely. See the `:toplevel` argument of `sb-ext:save-lisp-and-die`.

The behaviour of `require` when called with only one argument is implementation-defined. In SBCL, `require` behaves in the following way:

`require` *module-name* &optional *pathnames* [Function]
Loads a module, unless it already has been loaded. `pathnames`, if supplied, is a designator for a list of pathnames to be loaded if the module needs to be. If `pathnames` is not supplied, functions from the list `*module-provider-functions*` are called in order with `module-name` as an argument, until one of them returns non-`nil`. User code is responsible for calling `provide` to indicate a successful load of the module.

`sb-ext:*module-provider-functions*` [Variable]
See `require`.

Although SBCL does not provide a resident editor, the `ed` function can be customized to hook into user-provided editing mechanisms as follows:

`ed` &optional *x* [Function]
Starts the editor (on a file or a function if named). Functions from the list `*ed-functions*` are called in order with `x` as an argument until one of them returns non-`nil`; these functions are responsible for signalling a `file-error` to indicate failure to perform an operation on the file system.

`sb-ext:*ed-functions*` [Variable]
See `ed`.

Conditions of type `warning` and `style-warning` are sometimes signaled at runtime, especially during execution of Common Lisp defining forms such as `defun`, `defmethod`, etc. To muffle these warnings at runtime, SBCL provides a variable `sb-ext:*muffled-warnings*`:

`sb-ext:*muffled-warnings*` [Variable]
A type that ought to specify a subtype of `warning`. Whenever a warning is signaled, if the warning is of this type and is not handled by any other handler, it will be muffled.

7.13 Tools to Help Developers

SBCL provides a profiler and other extensions to the `trace` facility.
The debugger supports a number of options. Its documentation is accessed by typing `help` at the debugger prompt. See Section 5.1 [debugger], page 29.
Documentation for the command `inspect` is accessed by typing `help` at the `inspect` prompt.

7.14 Resolution of Name Conflicts

`clhs 11.1.1.2.5` requires that name conflicts in packages be resolvable in favour of any of the conflicting symbols. In the interactive debugger, this is achieved by prompting for the symbol in whose favour the conflict should be resolved; for programmatic use, the `sb-ext:resolve-conflict` restart should be invoked with one argument, which should be a member of the list returned by the condition accessor `sb-ext:name-conflict-symbols`.

7.15 Hash Table Extensions

hash table extensions supported by SBCL are all controlled by keyword arguments to `make-hash-table`.

`make-hash-table` &key (*test 'eql*) (*size 7*) (*rehash-size 1.5*) (*rehash-threshold 1*) *hash-function weakness synchronized* [Function]
Create and return a new hash table. The keywords are as follows:

- `:test`

  Determines how keys are compared. Must be a designator for one of the standard hash table tests, or a hash table test defined using `sb-ext:define-hash-table-test`. Additionally, when an explicit `hash-function` is provided (see below), any two argument equivalence predicate can be used as the `test`.

- `:size`

  A hint as to how many elements will be put in this hash table.

- `:rehash-size`

  Indicates how to expand the table when it fills up. If an integer, add space for that many elements. If a floating point number (which must be greater than 1.0), multiply the size by that amount.

- `:rehash-threshold`

  Indicates how dense the table can become before forcing a rehash. Can be any positive number <=1, with density approaching zero as the threshold approaches 0. Density 1 means an average of one entry per bucket.

- `:hash-function`

  If unsupplied, a hash function based on the `test` argument is used, which then must be one of the standardized hash table test functions, or one for which a default hash function has been defined using `sb-ext:define-hash-table-test`. If `hash-function` is specified, the `test` argument can be any two argument predicate consistent with it. The `hash-function` is expected to return a non-negative fixnum hash code. If `test` is neither standard nor defined by `define-hash-table-test`, then the `hash-function` must be specified.

- `:weakness`

  When `:weakness` is not `nil`, garbage collection may remove entries from the hash table. The value of `:weakness` specifies how the presence of a key or value in the hash table preserves their entries from garbage collection.

  Valid values are:

  - `:key` means that the key of an entry must be live to guarantee that the entry is preserved.
  - `:value` means that the value of an entry must be live to guarantee that the entry is preserved.
  - `:key-and-value` means that both the key and the value must be live to guarantee that the entry is preserved.
  - `:key-or-value` means that either the key or the value must be live to guarantee that the entry is preserved.
  - `nil` (the default) means that entries are always preserved.

- `:synchronized`

  If `nil` (the default), the hash-table may have multiple concurrent readers, but results are undefined if a thread writes to the hash-table concurrently with another reader or writer. If `t`, all concurrent accesses are safe, but note that CLHS 3.6 (Traversal Rules and Side Effects) remains in force. See also: `sb-ext:with-locked-hash-table`.

`sb-ext:define-hash-table-test` *name hash-function* [Macro]

Defines `name` as a new kind of hash table test for use with the `:test` argument to `make-hash-table`, and associates a default `hash-function` with it.

`name` must be a symbol naming a global two argument equivalence predicate. Afterwards both `'name` and `#'name` can be used with `:test` argument. In both cases `hash-table-test` will return the symbol `name`.

`hash-function` must be a symbol naming a global hash function consistent with the predicate, or be a `lambda` form implementing one in the current lexical environment. The hash function must compute the same hash code for any two objects for which `name` returns true, and subsequent calls with already hashed objects must always return the same hash code.

> *Note*: The `:hash-function` keyword argument to `make-hash-table` can be used to override the specified default hash-function.

Attempting to define `name` in a locked package as hash-table test causes a package lock violation.

Examples:

```
;; We want to use objects of type FOO as keys (by their
;; names.) EQUALP would work, but would make the names
;; case-insensitive -- which we don't want.
(defstruct foo (name nil :type (or null string)))

;; Define an equivalence test function and a hash function.
(defun foo-name= (f1 f2) (equal (foo-name f1) (foo-name f2)))
(defun sxhash-foo-name (f) (sxhash (foo-name f)))

(define-hash-table-test foo-name= sxhash-foo-name)

;; #'foo-name would work too.
(defun make-foo-table () (make-hash-table :test 'foo-name=))
(defun == (x y) (= x y))

(define-hash-table-test ==
  (lambda (x)
```

```
            ;; Hash codes must be consistent with test, so
            ;; not (SXHASH X), since
            ;;   (= 1 1.0)                   => T
            ;;   (= (SXHASH 1) (SXHASH 1.0)) => NIL
            ;; Note: this doesn't deal with complex numbers or
            ;; bignums too large to represent as double floats.
            (sxhash (coerce x 'double-float))))

        ;; #'== would work too
        (defun make-number-table () (make-hash-table :test '==))
```

`sb-ext:with-locked-hash-table` (*hash-table*) &body *body* [Macro]
Limits concurrent accesses to `hash-table` for the duration of `body`. If `hash-table` is synchronized, `body` will execute with exclusive ownership of the table. If `hash-table` is not synchronized, `body` will execute with other `with-locked-hash-table` bodies excluded – exclusion of hash-table accesses not surrounded by `with-locked-hash-table` is unspecified.

`sb-ext:hash-table-synchronized-p` *ht* [Function]
Returns `t` if `hash-table` is synchronized.

`sb-ext:hash-table-weakness` *ht* [Function]
Return the `weakness` of `hash-table` which is one of `nil`, `:key`, `:value`, `:key-and-value`, `:key-or-value`.

7.16 Random Number Generation

The initial value of `*random-state*` is the same each time SBCL is started. This makes it possible for user code to obtain repeatable pseudo random numbers using only standard-provided functionality. See `sb-ext:seed-random-state` below for an SBCL extension that allows to seed the random number generator from given data for an additional possibility to achieve this. Non-repeatable random numbers can always be obtained using (`make-random-state t`).
The sequence of numbers produced by repeated calls to `random` starting with the same random state and using the same sequence of `limit` arguments is guaranteed to be reproducible only in the same version of SBCL on the same platform, using the same code under the same evaluator mode and compiler optimization qualities. Just two examples of differences that may occur otherwise: calls to `random` can be compiled differently depending on how much is known about the `limit` argument at compile time, yielding different results even if called with the same argument at run time, and the results can differ depending on the machine's word size, for example for limits that are fixnums under 64-bit word size but bignums under 32-bit word size.

`sb-ext:seed-random-state` &optional *state* [Function]
Make a random state object. The optional `state` argument specifies a seed for deterministic pseudo-random number generation.
As per the Common Lisp standard for `make-random-state`,

- If `state` is `nil` or not supplied, return a copy of the default `*random-state*`.
- If `state` is a random state, return a copy of it.
- If `state` is `t`, return a randomly initialized state (using operating-system provided randomness where available, otherwise a poor substitute based on internal time and pid).

As a supported SBCL extension, we also support receiving as a seed an object of the following types:

- (`simple-array` (`unsigned-byte` 8) (*))
- `unsigned-byte`

While we support arguments of any size and will mix the provided bits into the random state, it is probably overkill to provide more than 256 bits worth of actual information.
This particular SBCL version also accepts an argument of the following type: (`simple-array` (`unsigned-byte` 32) (*))
This particular SBCL version uses the popular MT19937 PRNG algorithm, and its internal state only effectively contains about 19937 bits of information. `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html`

Some notes on random floats: The standard doesn't prescribe a specific method of generating random floats. The following paragraph describes SBCL's current implementation and should be taken as purely informational, that is, user code should not depend on any of its specific properties. The method used has been chosen because it is common, conceptually simple and fast.

To generate random floats, SBCL evaluates code that has an equivalent effect as

```
(* limit
   (float (/ (random (expt 2 23)) (expt 2 23)) 1.0f0))
```

(for `single-floats`) and correspondingly (with `52` and `1.0d0` instead of `23` and `1.0f0`) for `double-floats`. Note especially that this means that zero is a possible return value occurring with probability `(expt 2 -23)` and `(expt 2 -52)`, respectively. Also note that there exist twice as many equidistant floats between 0 and 1 as are generated. For example, the largest number that `(random 1.0f0)` ever returns is `(float (/ (1- (expt 2 23)) (expt 2 23)) 1.0f0)` while `(float (/ (1- (expt 2 24)) (expt 2 24)) 1.0f0)` is the largest `single-float` less than 1. This is a side effect of the fact that the implementation uses the fastest possible conversion from bits to floats.

SBCL currently uses the Mersenne Twister as its random number generator, specifically the 32-bit version under both 32- and 64-bit word size. The seeding algorithm has been improved several times by the authors of the Mersenne Twister; SBCL uses the third version (from 2002), which is still the most recent as of June 2012. The implementation has been tested to provide output identical to the recommended C implementation.

While the Mersenne Twister generates random numbers of much better statistical quality than other widely used generators, it uses only linear operations modulo 2 and thus fails some statistical tests.

(See chapter 7 *Testing widely used RNGs* in *TestU01: A C Library for Empirical Testing of Random Number Generators* by Pierre L'Ecuyer and Richard Simard, ACM Transactions on Mathematical Software, Vol. 33, article 22, 2007.)

For example, the distribution of ranks of (sufficiently large) random binary matrices is much distorted compared to the theoretically expected one when the matrices are generated by the Mersenne Twister. Thus, applications that are sensitive to this aspect should use a different type of generator.

7.17 Timeouts and Deadlines

SBCL supports three different ways of restricting the execution time available to individual operations or parts of computations:

- *Timeout Parameters*: Some operations such as thread synchronization primitives accept a `:timeout` parameter. See Section 7.17.1 [timeout parameters], page 83.
- *Synchronous Timeouts (Deadlines)*: Certain operations that may suspend execution for extended periods of time such as `cl:sleep`, thread synchronization primitives, IO and waiting for external processes respect deadlines established for a part of a computation. See Section 7.17.2 [synchronous timeouts], page 84.
- *Asynchronous Timeouts*: Asynchronous timeouts can interrupt most computations at (almost) any point. Thus, this kind of timeouts is the most versatile but it is also somewhat unsafe. See Section 7.17.3 [asynchronous timeouts], page 85.

7.17.1 Timeout Parameters

Certain operations accept `:timeout` keyword arguments. These only affect the specific operation and must be specified at each call site by passing a `:timeout` keyword argument and a corresponding timeout value to the respective operation. Expiration of the timeout before the operation completes results in either a normal return with a return value indicating the timeout or in the signaling of a specialized condition such as `sb-thread:join-thread-error`.

Example:

```
(defun join-thread-within-5-seconds (thread)
  (multiple-value-bind (value result)
      (sb-thread:join-thread thread :default nil :timeout 5)
    (when (eq result :timeout)
      (error "Could not join ~A within 5 seconds" thread))
    value))
```

The above code attempts to join the specified thread for up to five seconds, returning its value in case of success. If the thread is still running after the five seconds have elapsed, `sb-thread:join-thread` indicates

the timeout in its second return value. If a `:default` value was not provided, `sb-thread:join-thread` would signal a `sb-thread:join-thread-error` instead.

To wait for an arbitrary condition, optionally with a timeout, the `sb-ext:wait-for` macro can be used:

`sb-ext:wait-for` *test-form* &key *timeout* [Macro]

Wait until `test-form` evaluates to true, then return its primary value. If `timeout` is provided, waits at most approximately `timeout` seconds before returning `nil`.

If `with-deadline` has been used to provide a global deadline, signals a `deadline-timeout` if `test-form` doesn't evaluate to true before the deadline.

Experimental: subject to change without prior notice.

7.17.2 Synchronous Timeouts

Deadlines, in contrast to timeout parameters, are established for a dynamic scope using the `sb-sys:with-deadline` macro and indirectly affect operations within that scope. In case of nested uses, the effective deadline is the one that expires first unless an inner use explicitly overrides outer deadlines.

`sb-sys:with-deadline` (&key *seconds override*) &body *body* [Macro]

Arranges for a `timeout` condition to be signalled if an operation respecting deadlines occurs either after the deadline has passed, or would take longer than the time left to complete.

Currently only `sleep`, blocking IO operations, `sb-thread:get-mutex`, and `sb-thread:condition-wait` respect deadlines, but this includes their implicit uses inside SBCL itself.

Unless `override` is true, existing deadlines can only be restricted, not extended. Deadlines are per thread: children are unaffected by their parent's deadlines.

Experimental.

Expiration of deadlines set up this way only has an effect when it happens before or during the execution of a deadline-aware operation (Section 7.17.4 [operations supporting timeouts and deadlines], page 85). In this case, a `sb-sys:deadline-timeout` is signaled. A handler for this condition type may use the `sb-sys:defer-deadline` or `sb-sys:cancel-deadline` restarts to defer or cancel the deadline respectively and resume execution of the interrupted operation.

`sb-sys:deadline-timeout` [Condition]

Signaled when an operation in the context of a deadline takes longer than permitted by the deadline.

`sb-sys:defer-deadline` *seconds* &optional *condition* [Function]

Find the `defer-deadline` restart associated with `condition`, and invoke it with `seconds` as argument (deferring the deadline by that many seconds.) Otherwise return `nil` if the restart is not found.

`sb-sys:cancel-deadline` &optional *condition* [Function]

Find and invoke the `cancel-deadline` restart associated with `condition`, or return `nil` if the restart is not found.

When a thread is executing the debugger, signaling of `sb-sys:deadline-timeout` conditions for that thread is deferred until it exits the debugger.

Example:

```
(defun read-input ()
  (list (read-line) (read-line)))

(defun do-it ()
  (sb-sys:with-deadline (:seconds 5))
    (read-input)
    (sleep 2)
    (sb-ext:run-program "my-program"))
```

The above code establishes a deadline of five seconds within which the body of the `do-it` function should execute. All calls of deadline-aware functions in the dynamic scope, in this case two `read-line` calls, a `sleep` call and a `sb-ext:run-program` call, are affected by the deadline. If, for example, the first `read-line` call completes in one second and the second `read-line` call completes in three seconds, a `sb-sys:deadline-timeout` condition will be signaled after the `sleep` call has been executing for one second.

7.17.3 Asynchronous Timeouts

Asynchronous timeouts are established for a dynamic scope using the `sb-ext:with-timeout` macro:

`sb-ext:with-timeout` *expires* &body *body* [Macro]

Execute the body, asynchronously interrupting it and signalling a `timeout` condition after at least `expires` seconds have passed.

Note that it is never safe to unwind from an asynchronous condition. Consider:

```
(defun call-with-foo (function)
  (let (foo)
    (unwind-protect
       (progn
         (setf foo (get-foo))
         (funcall function foo))
      (when foo
        (release-foo foo)))))
```

If `timeout` occurs after `get-foo` has executed, but before the assignment, then `release-foo` will be missed. While individual sites like this can be made proof against asynchronous unwinds, this doesn't solve the fundamental issue, as all the frames potentially unwound through need to be proofed, which includes both system and application code – and in essence proofing everything will make the system uninterruptible.

Expiration of the timeout will cause the operation being executed at that moment to be interrupted by an asynchronously signaled `sb-ext:timeout` condition, (almost) irregardless of the operation and its context.

`sb-ext:timeout` [Condition]

Signaled when an operation does not complete within an allotted time budget.

7.17.4 Operations Supporting Timeouts and Deadlines

Operation	Timeout parameter	Affected by deadlines
cl:sleep	-	since SBCL 1.4.3
cl:read-line, etc.	no	yes
wait-for	yes	yes
process-wait	no	yes
grab-mutex	yes	yes
condition-wait	yes	yes
wait-on-semaphore	yes	yes
join-thread	yes	yes
receive-message	yes	yes?
wait-on-gate	yes	yes?
frlock-write	yes	yes?
grab-frlock-write-lock	yes	yes?

7.18 Miscellaneous Extensions

`sb-ext:array-storage-vector` *array* [Function]

Returns the underlying storage vector of `array`, which must be a non-displaced array.

In SBCL, if `array` is a of type (`simple-array` * (*)), it is its own storage vector. Multidimensional arrays, arrays with fill pointers, and adjustable arrays have an underlying storage vector with the same `array-element-type` as `array`, which this function returns.

> *Note*: the underlying vector is an implementation detail. Even though this function exposes it, changes in the implementation may cause this function to be removed without further warning.

`sb-ext:delete-directory` *pathspec* &key *recursive* [Function]

Deletes the directory designated by `pathspec` (a pathname designator). Returns the truename of the directory deleted.

If `recursive` is false (the default), signals an error unless the directory is empty. If `recursive` is true, first deletes all files and subdirectories. If `recursive` is true and the directory contains symbolic links, the links are deleted, not the files and directories they point to.

Signals an error if `pathspec` designates a file or a symbolic link instead of a directory, or if the directory could not be deleted for any reason.

Both

```
(DELETE-DIRECTORY "/tmp/foo")
(DELETE-DIRECTORY "/tmp/foo/")
```

delete the `"foo"` subdirectory of `"/tmp"`, or signal an error if it does not exist or if it is a file or a symbolic link.

`sb-ext:get-time-of-day` [Function]

Return the number of seconds and microseconds since the beginning of the UNIX epoch (January 1st 1970.)

`sb-ext:assert-version->=` &rest *subversions* [Function]

Asserts that the current SBCL is of version equal to or greater than the version specified in the arguments. A continuable error is signaled otherwise.

The arguments specify a sequence of subversion numbers in big endian order. They are compared lexicographically with the runtime version, and versions are treated as though trailed by an unbounded number of 0s.

For example, (`assert-version->=` 1 1 4) asserts that the current SBCL is version 1.1.4[.0.0...] or greater, and (`assert-version->=` 1) that it is version 1[.0.0...] or greater.

`sb-ext:unencapsulated-function` *function* [Function]

Return the innermost function within any encapsulations of the function designated by `function`. The identity of the returned function is not affected by encapsulations.

Note that the unencapsulated function may be `eq` to the designated function even in the presence of encapsulations. For generic functions, this is currently always the case.

`documentation` *object doc-type* [Generic function]

Return the documentation string of `doc-type` for `object`, or `nil` if none exists. In addition to the `doc-type`s and methods required by ANSI, SBCL's `documentation` (and its `setf`) supports methods with the following signatures:

- `(object symbol) (doc-type (eql declaration))`
- `(object sb-mop:slot-definition) (doc-type (eql t))`

Since `condition`s are implemented as classes in SBCL, the following also work:

- `(object condition) (doc-type (eql t))`
- `(object condition) (doc-type (eql 'type))`

Function documentation is stored separately for function names and objects: `defun`, `lambda`, &co create function objects with the specified documentation strings.

```
(setf (documentation name 'function) string)
```

sets the documentation string stored under the specified name, and

```
(setf (documentation func t) string)
```

sets the documentation string stored in the function object.

```
(documentation name 'function)
```

returns the documentation stored under the function name if any, and falls back on the documentation in the function object if necessary.

7.19 Stale Extensions

SBCL has inherited from CMUCL various hooks to allow the user to tweak and monitor the garbage collection process. These are somewhat stale code, and their interface might need to be cleaned up. If you have urgent need of them, look at the code in `src/code/gc.lisp` and bring it up on the developers' mailing list.

SBCL has various hooks inherited from CMUCL, like `sb-ext:float-denormalized-p`, to allow a program to take advantage of IEEE floating point arithmetic properties which aren't conveniently or efficiently expressible using the ANSI standard. These look good, and their interface looks good, but IEEE support is

slightly broken due to a stupid decision to remove some support for infinities (because it wasn't in the ANSI spec and it didn't occur to me that it was in the IEEE spec). If you need this stuff, take a look at the code and bring it up on the developers' mailing list.

8 External Formats

External formats determine the coding of characters from/to sequences of octets when exchanging data with the outside world. Examples of such exchanges are:

- Character streams associated with files, sockets and process input/output (see Section 11.1 [stream external formats], page 105, and Section 7.10.1 [running external programs], page 73)
- Names of files
- Foreign strings (see Section 9.2.2 [foreign types and lisp types], page 92)
- Posix interface (see Section 17.8 [sb posix], page 150)
- Hostname- and protocol-related functions of the BSD-socket interface (see Chapter 15 [networking], page 129)

Technically, external formats in SBCL are named objects describing coding of characters as well as policies in case de- or encoding is not possible. Each external format has a canonical name and zero or more aliases. User code mostly interacts with external formats by supplying external format designators to functions that use external formats internally.

8.1 The Default External Format

`sb-ext:*default-external-format*` [Variable]

Most functions interacting with external formats (`open`, notably) use this default.

`sb-ext:*default-source-external-format*` [Variable]

`sb-ext:*default-c-string-external-format*` [Variable]

8.2 External Format Designators

In situations where an external format designator is required, such as the `:external-format` argument in calls to `open` or `with-open-file`, users may supply the name of an encoding to denote the external format which is applying that encoding to Lisp characters.

In addition to the basic encoding for an external format, options controlling various special cases may be passed, by using a list (whose first element must be an encoding name and whose rest is a plist) as an external file format designator.

More specifically, external format designators can take the following forms:

- `:default`: Designates the current default external format (see Section 8.1 [default external format], page 88).
- `<keyword>`: Designates the supported external format that has `<keyword>` as one of its names (see Section 8.5 [supported external formats], page 89).
- `(<keyword> . <options-plist>)`: Designates an external format that is like the one designated by `<keyword>` with options as specified in `<options-plist>`.

Valid options for `<options-plist>` are:

- `:NEWLINE <newline>`

  An external format with an explicit `:newline` option is like its `<keyword>` parent but recognizes certain characters or character sequences as newlines. For `:lf` (the default), the `#\Linefeed` character is treated as `#\Newline` for both input and output. For `:cr`, `#\Return` is treated as `#\Newline`, while for `:crlf` the two-character sequence `#\Return #\Linefeed` is translated to and from `#\Newline`.

- `:REPLACEMENT <replacement>`

  An external format with an explicit `:replacement` option is like its `<keyword>` parent but does not signal an error in case a character or octet sequence cannot be en- or decoded. Instead, it inserts `<replacement>` at the position in question. `<replacement>` must be a string designator; that is, a character or a string.

For example:

```
(with-open-file (stream pathname :external-format '(:utf-8 :replacement #\?))
  (read-line stream))
```

will read the first line of `pathname`, replacing any octet sequence that is not valid in the UTF-8 external format with a question mark character.

8.3 Character Coding Conditions

De- or encoding characters using a given external format is not always possible:

- Decoding an octet vector using a given external format can fail if it contains an octet or sequence of octets that does not have an interpretation as a character according to the external format.
- Conversely, a string may contain characters that a given external format cannot encode. For example, the ASCII external format cannot encode the character `#\ö`.

Unless the external format governing the coding uses the `:replacement` option, SBCL will signal (continuable) errors under the above circumstances. The types of the condition signaled are not currently exported or documented but will be in future SBCL versions.

8.4 Converting between Strings and Octet Vectors

To encode Lisp strings as octet vectors and decode octet vectors as Lisp strings, the following SBCL-specific functions can be used:

`sb-ext:string-to-octets` *string* &key (*external-format :default*) (*start 0*) *end* *null-terminate* [Function]

Return an octet vector that is `string` encoded according to `external-format`.

If `external-format` is given, it must designate an external format.

If given, `start` and `end` must be bounding index designators and designate a subsequence of `string` that should be encoded.

If `null-terminate` is true, the returned octet vector ends with an additional 0 element that does not correspond to any part of `string`.

If some of the characters of `string` (or the subsequence bounded by `start` and `end`) cannot be encoded by `external-format` an error of a subtype of `sb-int:character-encoding-error` is signaled.

Note that for some values of `external-format` and `null-terminate` the length of the returned vector may be different from the length of `string` (or the subsequence bounded by `start` and `end`).

`sb-ext:octets-to-string` *vector* &key (*external-format :default*) (*start 0*) *end* [Function]

Return a string obtained by decoding `vector` according to `external-format`.

If `external-format` is given, it must designate an external format.

If given, `start` and `end` must be bounding index designators and designate a subsequence of `vector` that should be decoded.

If some of the octets of `vector` (or the subsequence bounded by `start` and `end`) cannot be decoded by `external-format` an error of a subtype of `sb-int:character-decoding-error` is signaled.

Note that for some values of `external-format` the length of the returned string may be different from the length of `vector` (or the subsequence bounded by `start` and `end`).

8.5 Supported External Formats

The following lists the external formats supported by SBCL in the form of the respective canonical name followed by the list of aliases:

- `:euc-jp`

  `:eucjp`, `:|eucJP|`
- `:gbk`

  `:cp936`
- `:shift_jis`

  `:sjis`, `:|Shift_JIS|`, `:cp932`
- `:ucs-2be`

  `:ucs2be`
- `:ucs-2le`

  `:ucs2le`
- `:ucs-4be`

  `:ucs4be`

- `:ucs-4le`
  `:ucs4le`
- `:utf-16be`
  `:utf16be`
- `:utf-16le`
  `:utf16le`
- `:utf-32be`
  `:utf32be`
- `:utf-32le`
  `:utf32le`

9 Foreign Function Interface

This chapter describes SBCL's interface to C programs and libraries (and, since C interfaces are a sort of *lingua franca* of the Unix world, to other programs and libraries in general).

> *Note*: In the modern Lisp world, the usual term for this functionality is Foreign Function Interface, or FFI, where despite the mention of *function* in this term, FFI also refers to direct manipulation of C data structures as well as functions. The traditional CMUCL terminology is Alien Interface, and while that older terminology is no longer used much in the system documentation, it still reflected in names in the implementation, notably in the name of the `sb-alien` package.

9.1 Introduction to the Foreign Function Interface

Because of Lisp's emphasis on dynamic memory allocation and garbage collection, Lisp implementations use non-C-like memory representations for objects. This representation mismatch creates friction when a Lisp program must share objects with programs which expect C data. There are three common approaches to establishing communication:

- The burden can be placed on the foreign program (and programmer) by requiring the knowledge and use of the representations used internally by the Lisp implementation. This can require a considerable amount of "glue" code on the C side, and that code tends to be sensitively dependent on the internal implementation details of the Lisp system.
- The Lisp system can automatically convert objects back and forth between the Lisp and foreign representations. This is convenient, but translation becomes prohibitively slow when large or complex data structures must be shared. This approach is supported by the SBCL FFI, and used automatically when passing integers and strings.
- The Lisp program can directly manipulate foreign objects through the use of extensions to the Lisp language.

SBCL, like CMUCL before it, relies primarily on the automatic conversion and direct manipulation approaches. The `sb-alien` package provides a facility wherein foreign values of simple scalar types are automatically converted and complex types are directly manipulated in their foreign representation. Additionally the lower-level System Area Pointers (or SAPs) can be used where necessary to provide untyped access to foreign memory.

Any foreign objects that can't automatically be converted into Lisp values are represented by objects of type `sb-alien-internals:alien-value` Since Lisp is a dynamically typed language, even foreign objects must have a run-time type; this type information is provided by encapsulating the raw pointer to the foreign data within an `sb-alien-internals:alien-value` object.

The type language and operations on foreign types are intentionally similar to those of the C language.

9.2 Foreign Types

Alien types have a description language based on nested list structure. For example the C type

```
struct foo {
    int a;
    struct foo *b[100];
};
```

has the corresponding SBCL FFI type

```
(struct foo
  (a int)
  (b (array (* (struct foo)) 100)))
```

9.2.1 Defining Foreign Types

Types may be either named or anonymous. With structure and union types, the name is part of the type specifier, allowing recursively defined types such as:

```
(struct foo (a (* (struct foo))))
```

An anonymous structure or union type is specified by using the name `nil`. The `with-alien` macro defines a local scope which *captures* any named type definitions. Other types are not inherently named, but can be given named abbreviations using the `define-alien-type` macro.

9.2.2 Foreign Types and Lisp Types

The foreign types form a subsystem of the SBCL type system. An `alien` type specifier provides a way to use any foreign type as a Lisp type specifier. For example,

```
(typep foo '(alien (* int)))
```

can be used to determine whether `foo` is a pointer to a foreign `int`. `alien` type specifiers can be used in the same ways as ordinary Lisp type specifiers (like `string`.) Alien type declarations are subject to the same precise type checking as any other declaration. See Section 4.3.2 [precise type checking], page 24.

Note that the type identifiers used in the foreign type system overlap with native Lisp type specifiers in some cases. For example, the type specifier `(alien single-float)` is identical to `single-float`, since foreign floats are automatically converted to Lisp floats. When `type-of` is called on an alien value that is not automatically converted to a Lisp value, then it will return an `alien` type specifier.

9.2.3 Foreign Type Specifiers

> *Note*: All foreign type names are exported from the `sb-alien` package. Some foreign type names are also symbols in the `common-lisp` package, in which case they are reexported from the `sb-alien` package, so that e.g. it is legal to refer to `single-float`.

These are the basic foreign type specifiers:

- The foreign type specifier `(* <foo>)` describes a pointer to an object of type `<foo>`. A pointed-to type `<foo>` of `t` indicates a pointer to anything, similar to `void *` in ANSI C. A null alien pointer can be detected with the `null-alien` function.
- The foreign type specifier `(array <foo> &rest <dimensions>)` describes array of the specified `<dimensions>`, holding elements of type `<foo>`. Note that (unlike in C) `(* <foo>)` and `(array <foo>)` are considered to be different types when type checking is done. If equivalence of pointer and array types is desired, it may be explicitly coerced using `cast`.

Arrays are accessed using `deref`, passing the indices as additional arguments. Elements are stored in row-major order (as in C), so the first dimension determines only the size of the memory block, and not the layout of the higher dimensions. An array whose first dimension is variable may be specified by using `nil` as the first dimension. Fixed-size arrays can be allocated as array elements, structure slots or `with-alien` variables. Dynamic arrays can only be allocated using `make-alien`.

- The foreign type specifier `(struct <name> &rest <fields>)` describes a structure type with the specified `<name>` and `<fields>`. Fields are allocated at the same offsets used by the implementation's C compiler, as guessed by the SBCL internals. An optional `:alignment` keyword argument can be specified for each field to explicitly control the alignment of a field. If `<name>` is `nil` then the structure is anonymous.

  If a named foreign `struct` specifier is passed to `define-alien-type` or `with-alien`, then this defines, respectively, a new global or local foreign structure type. If no `<fields>` are specified, then the fields are taken from the current (local or global) alien structure type definition of `<name>`.
- The foreign type specifier `(union <name> &rest <fields>)` is similar to `struct` but describes a union type. All fields are allocated at the same offset, and the size of the union is the size of the largest field. The programmer must determine which field is active from context.
- The foreign type specifier `(enum <name> &rest <specs>)` describes an enumeration type that maps between integer values and symbols. If `<name>` is `nil`, then the type is anonymous. Each element of the `<specs>` list is either a Lisp symbol, or a list `(<symbol> <value>)`. `<value>` is an integer. If `<value>` is not supplied, then it defaults to one greater than the value for the preceding spec (or to zero if it is the first spec).
- The foreign type specifier `(signed &optional <bits>)` specifies a signed integer with the specified number of `<bits>` precision. The upper limit on integer precision is determined by the machine's word size. If `<bits>` is not specified, the maximum size will be used.
- The foreign type specifier `(integer &optional <bits>)` is equivalent to the corresponding type specifier using `signed` instead of `integer`.
- The foreign type specifier `(unsigned &optional <bits>)` is like corresponding type specifier using `signed` except that the variable is treated as an unsigned integer.
- The foreign type specifier `(boolean &optional <bits>)` is similar to an enumeration type but maps from Lisp `nil` and `t` to C 0 and 1 respectively. `<bits>` determines the amount of storage allocated to hold the truth value.

- The foreign type specifier `single-float` describes a floating-point number in IEEE single-precision format.
- The foreign type specifier `double-float` describes a floating-point number in IEEE double-precision format.
- The foreign type specifier `(function <result-type> &rest <arg-types>)` describes a foreign function that takes arguments of the specified `<arg-types>` and returns a result of type `<result-type>`. Note that the only context where a foreign `function` type is directly specified is in the argument to `alien-funcall`. In all other contexts, foreign functions are represented by foreign function pointer types: `(* (function ...))`.
- The foreign type specifier `system-area-pointer` describes a pointer which is represented in Lisp as a `system-area-pointer` object. SBCL exports this type from `sb-alien` because CMUCL did, but tentatively (as of the first draft of this section of the manual, SBCL 0.7.6) it is deprecated, since it doesn't seem to be required by user code.
- The foreign type specifier `void` is used in function types to declare that no useful value is returned. Using `alien-funcall` to call a `void` foreign function will return zero values.
- The foreign type specifier `(c-string &key <external-format> <element-type> <not-null>)` is similar to `(* char)` but is interpreted as a null-terminated string, and is automatically converted into a Lisp string when accessed; or if the pointer is C `null` or 0, then accessing it gives Lisp `nil` unless `<not-null>` is true, in which case a `type-error` is signalled.

  external format conversion is automatically done when Lisp strings are passed to foreign code, or when foreign strings are passed to Lisp code. If the type specifier has an explicit `<external-format>`, that external format will be used. Otherwise `sb-ext:*default-c-string-external-format*` will be used. For example, when the following alien routine is called, the Lisp string given as argument is converted to an EBCDIC octet representation.

  ```
  (define-alien-routine test int (str (c-string :external-format :ebcdic-us)))
  ```

  Lisp strings of type `base-string` are stored with a trailing `NUL` termination, so no copying (either by the user or the implementation) is necessary when passing them to foreign code, assuming that the `<external-format>` and `<element-type>` of the `c-string` type are compatible with the internal representation of the string. For an SBCL built with Unicode support that means an `<external-format>` of `:ascii` and an `<element-type>` of `base-char`. Without Unicode support the `<external-format>` can also be `:iso-8859-1`, and the `<element-type>` can also be `character`. If `<external-format>` and `<element-type>` are not compatible, or the string is a `(simple-array character (*))`, this data is copied by the implementation as required.

  Assigning a Lisp string to a `c-string` structure field or variable stores the contents of the string to the memory already pointed to by that variable. When a foreign object of type `(* char)` is assigned to a `c-string`, then the `c-string` pointer is assigned to. This allows `c-string` pointers to be initialized. For example:

  ```
  (cl:in-package "CL-USER") ; which USEs package "SB-ALIEN"

  (define-alien-type nil (struct foo (str c-string)))

  (defun make-foo (str)
    (let ((my-foo (make-alien (struct foo))))
      (setf (slot my-foo 'str) (make-alien char (length str))
            (slot my-foo 'str) str)
      my-foo))
  ```

  Storing Lisp `nil` in a `c-string` writes C `NULL` to the variable.
- `sb-alien` also exports translations of these C type

specifiers as foreign type specifiers:
`char`, `short`, `int`, `long`, `unsigned-char`, `unsigned-short`, `unsigned-int`, `unsigned-long`, `float`, `double`, `size-t`, `off-t`

9.3 Operations On Foreign Values

This section describes how to read foreign values as Lisp values, how to coerce foreign values to different kinds of foreign values, and how to dynamically allocate and free foreign variables.

9.3.1 Accessing Foreign Values

`sb-alien:deref` *alien* &rest *indices* [Function]
Dereference an `alien` pointer or array. When dereferencing a pointer, an optional single index can be specified to give the equivalent of C pointer arithmetic; this index is scaled by the size of the type pointed to. When dereferencing an array, the number of indices must be the same as the number of dimensions in the array type. `setf`able.

`sb-alien:slot` *alien slot* [Function]
Extract the value of the slot named `slot` from a foreign `struct` or `union` `alien`. If `alien` is a pointer to a structure or union, then it is automatically dereferenced. `setf`able.

Note that `slot` is evaluated, and need not be a compile-time constant (but only constant slot accesses are efficiently compiled).

9.3.1.1 Untyped memory

As noted at the beginning of the chapter, the System Area Pointer facilities allow untyped access to foreign memory. SAPs can be converted to and from the usual typed foreign values using `sap-alien` and `alien-sap`, and also to and from integers (raw machine addresses). They should thus be used with caution; corrupting the Lisp heap or other memory with SAPs is trivial.

`sb-sys:int-sap` *x* [Function]
Creates a `sap` pointing at the virtual address `x`.

`sb-sys:sap-ref-32` *sap offset* [Function]
Access the value of the memory location at `offset` bytes from `sap`. `setf`able.

`sb-sys:sap=` *x y* [Function]
Compare the `sap`s `x` and `y` for equality.

Similarly named functions exist for accessing other sizes of word, other comparisons, and other conversions. The reader is invited to use `apropos` and `describe` for more details:

```
(apropos "sap" :sb-sys)
```

9.3.2 Coercing Foreign Values

`sb-alien:addr` *expr* [Macro]
Return an Alien pointer to the data addressed by `expr`, which must be a foreign variable, a call to `deref` or `slot`, or a use of `extern-alien`.

`sb-alien:cast` *alien type* [Macro]
Convert `alien` to an Alien of the specified `type` (not evaluated). Both types must be Alien array, pointer or function types.

Note that the resulting Lisp foreign variable object is not `eq` to the argument, but it points to the same foreign memory address.

`sb-alien:sap-alien` *sap type* [Macro]
Convert the `system-area-pointer` `sap` to an `alien` of the specified `type` (not evaluated). `type` must be pointer-like (foreign pointer, array, or record type).

`sb-alien:alien-sap` *alien* [Function]
Return a `system-area-pointer` pointing to `alien`'s data. `alien` must be of some foreign pointer, array, or record type.

9.3.3 Foreign Dynamic Allocation

Lisp code can call the C standard library functions `malloc` and `free` to dynamically allocate and deallocate foreign variables. The Lisp code uses the same allocator as foreign C code, so it's OK for foreign code to call `free` on the result of Lisp `make-alien`, or for Lisp code to call `free-alien` on foreign objects allocated by C code.

`sb-alien:make-alien` *type* &optional *size* [Macro]

Allocate an alien of type `type` in foreign heap, and return an alien pointer to it. The allocated memory is not initialized, and may contain garbage. The memory is allocated using `malloc(3)`, so it can be passed to foreign functions which use `free(3)`, or released using `free-alien`.

For alien stack allocation, see macro `with-alien`.

The `type` argument is not evaluated. If `size` is supplied, how it is interpreted depends on `type`:

- When `type` is a foreign array type, an array of that type is allocated, and a pointer to it is returned. Note that you must use `deref` to first access the array through the pointer.

  If supplied, `size` is used as the first dimension for the array.
- When `type` is any other foreign type, then an object for that type is allocated, and a pointer to it is returned. So (`make-alien int`) returns a (* `int`).

  If `size` is specified, then a block of that many objects is allocated, with the result pointing to the first one.

Examples:

```
(defvar *foo* (make-alien (array char 10)))
(type-of *foo*)                   ; => (alien (* (array (signed 8) 10)))
(setf (deref (deref *foo*) 0) 10) ; => 10

(make-alien char 12)              ; => (alien (* (signed 8)))
```

`sb-alien:make-alien-string` *string* &rest *rest* &key (*start 0*) *end* (*external-format :default*) (*null-terminate t*) [Function]

Copy part of `string` delimited by `start` and `end` into freshly allocated foreign memory, freeable using `free(3)` or `free-alien`. Returns the allocated string as a (* `char`) alien, and the number of bytes allocated as secondary value.

The string is encoded using `external-format`. If `null-terminate` is true (the default), the alien string is terminated by an additional null byte.

`sb-alien:free-alien` *alien* [Function]

Dispose of the storage pointed to by `alien`. The `alien` must have been allocated by `make-alien`, `make-alien-string` or `malloc(3)`.

9.4 Foreign Variables

Both local (stack allocated) and external (C global) foreign variables are supported.

9.4.1 Local Foreign Variables

`sb-alien:with-alien` *bindings* &body *body* [Macro]

Establish some local alien variables of dynamic extent. Each of `bindings` is of the form:

```
VAR TYPE [ ALLOCATION ] [ INITIAL-VALUE | EXTERNAL-NAME ]
```

`allocation` should be one of:

- `:local` (the default): The alien is allocated on the stack, and has dynamic extent.
- `:extern`: No alien is allocated, but `var` is established as a local name for the external alien given by `external-name`.

`var`s are established as symbol-macros; the bindings have lexical scope, and may be assigned with `setq` or `setf`.

The `with-alien` macro also establishes a new scope for named structures and unions. Any `type` specified for a variable may contain named structure or union types with the slots specified. Within the lexical scope of the binding specifiers and body, a locally defined foreign structure type `foo` can be referenced by its name using `(struct foo)`.

When a foreign function returns a structure by value, using `alien-funcall` as the `initial-value` allows the returned struct to be stack-allocated directly into the local variable's storage, avoiding heap allocation:

```
(with-alien ((result (struct point)
                     (alien-funcall
                      (extern-alien "make_point"
```

```
                                        (function (struct point) double double))
                              1.0d0 2.0d0)))
        (values (slot result 'x) (slot result 'y)))
```

9.4.2 External Foreign Variables

External foreign names are strings, and Lisp names are symbols. When an external foreign value is represented using a Lisp variable, there must be a way to convert from one name syntax into the other. The macros `extern-alien`, `define-alien-variable` and `define-alien-routine` use this conversion heuristic:

- Alien names are converted to Lisp names by uppercasing and replacing underscores with hyphens.
- Conversely, Lisp names are converted to alien names by lowercasing and replacing hyphens with underscores.
- Both the Lisp symbol and alien string names may be separately specified by using a list of the form

  `(<alien-string> <lisp-symbol>)`

`sb-alien:define-alien-variable` *name type* [Macro]

Define `name` as an external alien variable of type `type`. Neither is evaluated.

In its full form, `name` is `(<alien-name-string> <lisp-name-symbol>)`. If `name` is just a symbol or string, then the other name is guessed from the one supplied as described Section 9.4.2 [external foreign variables], page 96.

The Lisp name of the variable becomes a global alien variable. Global alien variables are effectively "global symbol macros"; a reference to the variable fetches the contents of the external variable. Similarly, setting the variable stores new contents – the new contents must be of the declared `type`. Someday, they may well be implemented using the ANSI `define-symbol-macro` mechanism, but as of SBCL 0.7.5, they are still implemented using an older more-or-less parallel mechanism inherited from CMUCL.

For example, to access a C-level counter `foo`, one could write

```
(define-alien-variable "foo" int)
;; Now it is possible to get the value of the C variable foo simply by
;; referencing that Lisp variable:
(print foo)
(setf foo 14)
(incf foo)
```

`sb-alien:get-errno` [Function]

Return the value of the C library pseudo-variable named `errno`.

Since in modern C libraries, `errno` is typically no longer a variable, but some bizarre artificial construct which behaves superficially like a variable within a given thread, it can no longer reliably be accessed through the ordinary `define-alien-variable` mechanism.

`sb-alien:extern-alien` *name type* [Macro]

Return an alien of `type` which points to an externally defined value of `name`. `name` is not evaluated and may be either a string or a symbol. `type` is an unevaluated alien type specifier. `setf`able.

9.5 Foreign Data Structure Examples

Now that we have alien types, operations and variables, we can manipulate foreign data structures. This C declaration

```
struct foo {
    int a;
    struct foo *b[100];
};
```

can be translated into the following alien type:

```
(define-alien-type nil
  (struct foo
    (a int)
    (b (array (* (struct foo)) 100))))
```

Once the `foo` alien type has been defined as above, the C expression

```
struct foo f;
```

```
f.b[7].a;
```

can be translated in this way:

```
(with-alien ((f (struct foo)))
  (slot (deref (slot f 'b) 7) 'a)
  ;;
  ;; Do something with f...
  )
```

Or consider this example of an external C variable and some accesses:

```
struct c_struct {
        short x, y;
        char a, b;
        int z;
        c_struct *n;
};
extern struct c_struct *my_struct;
my_struct->x++;
my_struct->a = 5;
my_struct = my_struct->n;
```

which can be manipulated in Lisp like this:

```
(define-alien-type nil
  (struct c-struct
          (x short)
          (y short)
          (a char)
          (b char)
          (z int)
          (n (* c-struct))))
(define-alien-variable "my_struct" (* c-struct))
(incf (slot my-struct 'x))
(setf (slot my-struct 'a) 5)
(setq my-struct (slot my-struct 'n))
```

9.6 Loading Shared Object Files

Foreign object files can be loaded into the running Lisp process by calling `load-shared-object`.

`sb-alien:load-shared-object` *pathname* &key *dont-save* [Function]

Load a shared library / dynamic shared object file / similar foreign container specified by designated `pathname`, such as a .so on an ELF platform.

Locating the shared object follows standard rules of the platform, consult the manual page for `dlopen(3)` for details. Typically paths specified by environment variables such as `LD_LIBRARY_PATH` are searched if the `pathname` has no directory, but on some systems (e.g. Mac OS X) search may happen even if `pathname` is absolute. (On Windows LoadLibrary is used instead of `dlopen(3)`.)

On non-Windows platforms calling `load-shared-object` again with a `pathname` `equal` to the designated pathname of a previous call will replace the old definitions; if a symbol was previously referenced through the object and is not present in the reloaded version an error will be signalled. Reloading may not work as expected if user or library-code has called `dlopen(3)` on the same shared object or running on a system where `dlclose(3)` is a noop.

`load-shared-object` interacts with `sb-ext:save-lisp-and-die`:

1. If `dont-save` is true (default is `nil`), the shared object will be dropped when `save-lisp-and-die` is called – otherwise shared objects are reloaded automatically when a saved core starts up. Specifying `dont-save` can be useful when the location of the shared object on startup is uncertain.

2. On most platforms references in compiled code to foreign symbols in shared objects (such as those generated by `define-alien-routine`) remain valid across `save-lisp-and-die`. On those platforms where this is not supported, a `warning` will be signalled when the core is saved – this is orthogonal from `dont-save`.

`sb-alien:unload-shared-object` *pathname* [Function]
Unloads the shared object loaded earlier using the designated `pathname` with `load-shared-object`, to the degree supported on the platform.
Experimental.

9.7 Foreign Function Calls

The foreign function call interface allows a Lisp program to call many functions written in languages that use the C calling convention.
Lisp sets up various signal handling routines and other environment information when it first starts up, and expects these to be in place at all times. The C functions called by Lisp should not change the environment, especially the signal handlers: the signal handlers installed by Lisp typically have interesting flags set (e.g to request machine context information, or for signal delivery on an alternate stack) which the Lisp runtime relies on for correct operation. Precise details of how this works may change without notice between versions; the source, or the brain of a friendly SBCL developer, is the only documentation. Users of a Lisp built with the `:sb-thread` feature should also read the section about threads, Chapter 13 [threading], page 116.

`sb-alien:alien-funcall` *alien* &rest *args* [Function]
Call the foreign function `alien` with `args` and return its C return value as a Lisp value. `alien`'s foreign type specifies the argument and result types. `alien` is typically an `extern-alien` or a value defined with `define-alien-routine`.
The type of `alien` must be `(alien (function ...))` or `(alien (* (function ...)))`. The function type is used to determine how to call the function (as though it was declared with a prototype). The type need not be known at compile time, but only known-type calls are efficiently compiled.
On x86-64 and ARM64 systems, structures may be passed and returned by value. The implementation follows the System V AMD64 / Microsoft x64 and AAPCS64 specifications, respectively.
Here is an example which allocates a `(struct foo)`, calls a foreign function to initialize it, then returns a Lisp vector of all the `(* (struct foo))` objects filled in by the foreign call:

```
;; Allocate a foo on the stack.
(with-alien ((f (struct foo)))
  ;; Call some C function to fill in foo fields.
  (alien-funcall (extern-alien "mangle_foo" (function void (* foo)))
                 (addr f))
  ;; Find how many foos to use by getting the A field.
  (let* ((num (slot f 'a))
         (result (make-array num)))
    ;; Get a pointer to the array so that we don't have to keep extracting it:
    (with-alien ((a (* (array (* (struct foo)) 100)) (addr (slot f 'b))))
      ;; Loop over the first N elements and stash them in the result vector.
      (dotimes (i num)
        (setf (svref result i) (deref (deref a) i)))
      ;; Voila.
      result)))
```

`sb-alien:alien-funcall-into` *alien result-buffer* &rest *args* [Function]
Call the foreign function `alien`, writing the struct result to `result-buffer`. Returns no values.
`result-buffer` should be a `system-area-pointer` to appropriately sized memory. Only supported on x86-64 and ARM64.
Here is an example that calls a C function returning a struct, writing the result to a stack-allocated buffer:

```
(define-alien-type nil (struct point (x double) (y double)))

(with-alien ((result (struct point)))
  (alien-funcall-into
   (extern-alien "make_point"
                 (function (struct point) double double))
   (alien-sap (addr result))
   1.0d0 2.0d0)
```

```
(values (slot result 'x) (slot result 'y)))
```

`sb-alien:define-alien-routine` *name result-type* &rest *args* [Macro]

Define a foreign interface function for the routine with the specified `name`. Also automatically `declaim` the `ftype` of the defined function. The semantics of the actual call are the same as for `alien-funcall`.

This macro is a convenience for automatically generating Lisp interfaces to simple foreign functions. The primary feature is the parameter style specification, which translates the C pass-by-reference idiom into additional return values.

`name` may be either a string, a symbol, or a list of the form `(<foreign-name-string> <lisp-name-symbol>)`.

`result-type` is the alien type for the function return value. `void` may be used to specify a function with no result.

`args` is a list of `(arg-name arg-type &optional style)` elements. `arg-name` is a symbol that names the argument, primarily for documentation. `arg-type` is the C type of the argument.

`style` specifies the way that the argument is passed:

- `:in`: An `:in` argument is simply passed by value. The value to be passed is obtained from argument(s) to the interface function. No values are returned for `:in` arguments. This is the default mode.
- `:out`: A pass-by-reference output value. The specified argument type must be a pointer to a fixed sized object. An object of the correct size is allocated on the stack, and its address is passed to the foreign function. When the function returns, the contents of this location are returned as one of the values of the Lisp function (and the location is automatically deallocated). `:out` and `:in-out` cannot be used with pointers to arrays, records or functions.
- `:copy`: This is similar to `:in`, except that the argument values are stored on the stack, and a pointer to the object is passed instead of the value itself.
- `:in-out`: This is a combination of `:out` and `:copy`. A pointer to the argument is passed, with the object being initialized from the supplied argument and the return value being determined by accessing the object on return.

  Note: Any efficiency-critical foreign interface function should be inline expanded, which can be done by preceding the `define-alien-routine` call with:

  ```
  (declaim (inline lisp-name))
  ```

  In addition to avoiding the Lisp call overhead, this allows pointers, word-integers and floats to be passed using unboxed representations, avoiding consing.

Consider the C function `cfoo` with the following calling convention:

```
void
cfoo (str, a, i)
    char *str;
    char *a; /* update */
    int *i; /* out */
{
  /* body of cfoo(...) */
}
```

This can be described by the following call to `define-alien-routine`:

```
(define-alien-routine "cfoo" void
  (str c-string)
  (a char :in-out)
  (i int :out))
```

The Lisp function `cfoo` will have two arguments (`str` and `a`) and two return values (`a` and `i`).

9.8 Calling Lisp From C

SBCL supports the calling of Lisp functions using the C calling convention. This is useful for both defining callbacks and for creating an interface for calling into Lisp as a shared library directly from C.

The `define-alien-callable` macro wraps Lisp code and creates a C foreign function which can be called with the C calling convention. On x86-64 and ARM64, callbacks may receive and return structures by value.

`sb-alien:define-alien-callable` *name result-type typed-lambda-list* &body *body* [Macro]
Define an alien function which can be called by alien code. The alien function returned by (`alien-callable-function name`) expects alien arguments of the specified `arg-type`s and returns an alien of type `result-type`.

`typed-lambda-list` is a list of `(arg-name arg-type)` elements, and `body` is `{doc-string} {decl}* {form}*`.

If (`alien-callable-function name`) already exists, its value is not changed (though it is arranged that an updated version of the Lisp callable function will be called, provided that the new type and the existing type are compatible). This feature allows for incremental redefinition of callable functions.

The `alien-callable-function` function returns the foreign callable value associated with any name defined by `define-alien-callable`, so that we can, for example, pass the callable value to C as a callback.

`sb-alien:alien-callable-function` *name* [Function]
Return the alien callable function associated with `name`.

The `with-alien-callable` macro wraps Lisp code and establishes local C foreign functions which can be called with the C calling convention. This macro is handy for passing callbacks which close over Lisp values into C.

`sb-alien:with-alien-callable` *definitions* &body *body* [Macro]
Establish some local alien functions. Each element of `definitions` is of the form:

```
NAME RESULT-TYPE {(ARG-NAME ARG-TYPE)}* {DOC-STRING} {DECL}* {FORM}*
```

The resulting alien callable value has dynamic extent.

Note that the garbage collector moves objects, and won't be able to fix up any references in C variables. There are three mechanisms for coping with this:

- `sb-ext:purify` moves all live Lisp data into static or read-only areas such that it will never be moved (or freed) again in the life of the Lisp session
- `sb-sys:with-pinned-objects` is a macro which arranges for some set of objects to be pinned in memory for the dynamic extent of its body forms. On ports which use the generational GC (most, as of this writing) this affects exactly the specified objects. On other ports it is implemented by turning off GC for the duration (so could be said to have a whole-world granularity).
- Disable GC, using the SB-EXT:WITHOUT-GCING macro.

9.8.1 Lisp as a Shared Library

SBCL supports the use of Lisp as a shared library that can be used by C programs using the `define-alien-callable` interface. See the `:callable-exports` argument of `sb-ext:save-lisp-and-die` for how to save the Lisp image in a way that allows a C program to initialize the Lisp runtime and the exported symbols. When SBCL is built as a library, it exposes the symbol `initialize_lisp` which can be used in conjunction with a core initializing global symbols to foreign callables as function pointers and with object code allocating those symbols to initialize the runtime properly. The arguments to `initialize_lisp` are the same as the arguments to the main `sbcl` program.

> *Note*: There is currently no way to run exit hooks or otherwise undo Lisp initialization gracefully from C.

9.9 Step-by-step Example of the Foreign Function Interface

This section presents a complete example of an interface to a somewhat complicated C function.
Suppose you have the following C function which you want to be able to call from Lisp in the file `test.c`:

```
struct c_struct
{
  int x;
  char *s;
};

struct c_struct *c_function (i, s, r, a)
    int i;
```

```
    char *s;
    struct c_struct *r;
    int a[10];
{
  int j;
  struct c_struct *r2;

  printf("i = %dn", i);
  printf("s = %sn", s);
  printf("r->x = %dn", r->x);
  printf("r->s = %sn", r->s);
  for (j = 0; j < 10; j++) printf("a[%d] = %d.n", j, a[j]);
  r2 = (struct c_struct *) malloc (sizeof(struct c_struct));
  r2->x = i + 5;
  r2->s = "a C string";
  return(r2);
};
```

It is possible to call this C function from Lisp using the file `test.lisp` containing

```
(cl:defpackage "TEST-C-CALL" (:use "CL" "SB-ALIEN" "SB-C-CALL"))
(cl:in-package "TEST-C-CALL")

;;; Define the record C-STRUCT in Lisp.
(define-alien-type nil
    (struct c-struct
            (x int)
            (s c-string)))

;;; Define the Lisp function interface to the C routine.  It returns a
;;; pointer to a record of type C-STRUCT.  It accepts four parameters:
;;; I, an int; S, a pointer to a string; R, a pointer to a C-STRUCT
;;; record; and A, a pointer to the array of 10 ints.
;;;
;;; The INLINE declaration eliminates some efficiency notes about heap
;;; allocation of alien values.
(declaim (inline c-function))
(define-alien-routine c-function
    (* (struct c-struct))
  (i int)
  (s c-string)
  (r (* (struct c-struct)))
  (a (array int 10)))

;;; a function which sets up the parameters to the C function and
;;; actually calls it
(defun call-cfun ()
  (with-alien ((ar (array int 10))
               (c-struct (struct c-struct)))
    (dotimes (i 10)                     ; Fill array.
      (setf (deref ar i) i))
    (setf (slot c-struct 'x) 20)
    (setf (slot c-struct 's) "a Lisp string")

    (with-alien ((res (* (struct c-struct))
                      (c-function 5 "another Lisp string" (addr c-struct) ar)))
      (format t "~&back from C function~%")
      (multiple-value-prog1
          (values (slot res 'x)
```

```
                (slot res 's))

          ;; Deallocate result. (after we are done referring to it:
          ;; "Pillage, *then* burn.")
          (free-alien res)))))
```

To execute the above example, it is necessary to compile the C routine, e.g. with `cc -c test.c && ld -shared -o test.so test.o`. In order to enable incremental loading with some linkers, you may need to say `cc -G 0 -c test.c`.

Once the C code has been compiled, you can start up Lisp and load it in: `sbcl`. Lisp should start up with its normal prompt.

Within Lisp, compile the Lisp file:

```
(compile-file "test.lisp")
```

This step can be done separately. You don't have to recompile every time.

Within Lisp, load the foreign object file to define the necessary symbols:

```
(load-shared-object "test.so")
```

Now you can load the compiled Lisp (fasl) file into Lisp:

```
(load "test.fasl")
```

And once the Lisp file is loaded, you can call the Lisp routine that sets up the parameters and calls the C function:

```
(test-c-call::call-cfun)
```

The C routine should print the following information to standard output:

```
i = 5
s = another Lisp string
r->x = 20
r->s = a Lisp string
a[0] = 0.
a[1] = 1.
a[2] = 2.
a[3] = 3.
a[4] = 4.
a[5] = 5.
a[6] = 6.
a[7] = 7.
a[8] = 8.
a[9] = 9.
```

After return from the C function, the Lisp wrapper function should print the following output:

```
back from C function
```

And upon return from the Lisp wrapper function, before the next prompt is printed, the Lisp read-eval-print loop should print the following return values:

```
10
"a C string"
```

10 Pathnames

10.1 Lisp Pathnames

There are many aspects of ANSI Common Lisp's pathname support which are implementation-defined and so need documentation.

10.1.1 Home Directory Specifiers

SBCL accepts the keyword `:home` and a list of the form `(:home "username")` as a directory component immediately following `:absolute`.

`:home` is represented in namestrings by `~/` and `(:home "username")` by `~username/` at the start of the namestring. Tilde-characters elsewhere in namestrings represent themselves.

Home directory specifiers are resolved to home directory of the current or specified user by `sb-ext:native-namestring`, which is used by the implementation to translate pathnames before passing them on to operating system specific routines.

Using `(:home "user")` form on Windows signals an error.

10.1.2 The SYS Logical Pathname Host

The logical pathname host named by `"SYS"` exists in SBCL. Its `logical-pathname-translations` may be set by the site or the user applicable to point to the locations of the system's sources; in particular, the core system's source files match the logical pathname `"SYS:SRC;**;*.*.*"`, and the contributed modules' source files match `"SYS:CONTRIB;**;*.*.*"`.

`sb-ext:set-sbcl-source-location` *pathname* [Function]

Initialize the `SYS` logical host based on `pathname`, which should be the top-level directory of the SBCL sources. This will replace any existing translations for `"SYS:SRC;"`, `"SYS:CONTRIB;"`, and `"SYS:OUTPUT;"`. Other `"SYS:"` translations are preserved.

10.2 Native Filenames

In some circumstances, what is wanted is a Lisp pathname object which corresponds to a string produced by the Operating System. In this case, some of the default parsing rules are inappropriate: most filesystems do not have a native understanding of wild pathnames; such functionality is often provided by shells above the OS, often in mutually-incompatible ways.

To allow the user to deal with this, the following functions are provided: `sb-ext:parse-native-namestring` and `sb-ext:native-pathname` return the closest equivalent Lisp pathname to a given string (appropriate for the Operating System), while `sb-ext:native-namestring` converts a non-wild pathname designator to the equivalent native namestring, if possible. Some Lisp pathname concepts (such as the `:back` directory component) have no direct equivalents in most Operating Systems; the behaviour of `sb-ext:native-namestring` is unspecified if an inappropriate pathname designator is passed to it. Additionally, note that conversion from pathname to native filename and back to pathname should not be expected to preserve equivalence under `equal`.

`sb-ext:parse-native-namestring` *thing* &optional *host* (*defaults* **default-pathname-defaults**) &key (*start* *0*) *end* *junk-allowed* *as-directory* [Function]

Convert `thing` into a pathname, using the native conventions appropriate for the pathname host `host`, or if not specified the host of `defaults`. If `thing` is a string, the parse is bounded by `start` and `end`, and error behaviour is controlled by `junk-allowed`, as with `parse-namestring`. For file systems whose native conventions allow directories to be indicated as files, if `as-directory` is true, return a pathname denoting `thing` as a directory.

`sb-ext:native-pathname` *pathspec* [Function]

Convert `pathspec` (a pathname designator) into a pathname, assuming the operating system native pathname conventions.

`sb-ext:native-namestring` *pathname* &key *as-file* [Function]

Construct the full native (name)string form of `pathname`. For file systems whose native conventions allow directories to be indicated as files, if `as-file` is true and the name, type, and version components of

`pathname` are all `nil` or `:unspecific`, construct a string that names the directory according to the file system's syntax for files.

Because some file systems permit the names of directories to be expressed in multiple ways, it is occasionally necessary to parse a native file name as a directory name or to produce a native file name that names a directory as a file. For these cases, PARSE-NATIVE-NAMESTRING accepts the keyword argument `:as-directory` to force a filename to parse as a directory, and `sb-ext:native-namestring` accepts the keyword argument `:as-file` to force a pathname to unparse as a file. For example,

```
; On Unix, the directory "/tmp/" can be denoted by "/tmp/" or "/tmp".
; Under the default rules for native filenames, these parse and
; unparse differently.
(defvar *p*)
(setf *p* (parse-native-namestring "/tmp/")) => #P"/tmp/"
(pathname-name *p*) => NIL
(pathname-directory *p*) => (:ABSOLUTE "tmp")
(native-namestring *p*) => "/tmp/"

(setf *p* (parse-native-namestring "/tmp")) => #P"/tmp"
(pathname-name *p*) => "tmp"
(pathname-directory *p*) => (:ABSOLUTE)
(native-namestring *p*) => "/tmp"

; A non-NIL AS-DIRECTORY argument to PARSE-NATIVE-NAMESTRING forces
; both the second string to parse the way the first does.
(setf *p* (parse-native-namestring "/tmp"
                                   nil *default-pathname-defaults*
                                   :as-directory t)) => #P"/tmp/"
(pathname-name *p*) => NIL
(pathname-directory *p*) => (:ABSOLUTE "tmp")

; A non-NIL AS-FILE argument to NATIVE-NAMESTRING forces the pathname
; parsed from the first string to unparse as the second string.
(setf *p* (parse-native-namestring "/tmp/")) => #P"/tmp/"
(native-namestring *p* :as-file t) => "/tmp"
```

11 Streams

Streams which read or write Lisp character data from or to the outside world – files, sockets or other external entities – require the specification of a conversion between the external, binary data and the Lisp characters. In ANSI Common Lisp, this is done by specifying the `:external-format` argument when the stream is created. The major information required is an *encoding*, specified by a keyword naming that encoding; however, it is also possible to specify refinements to that encoding as additional options to the external format designator.

In addition, SBCL supports various extensions of ANSI Common Lisp streams:

- *Bivalent Streams*: A type of stream that can read and write both `character` and `(unsigned-byte 8)` values.
- *Gray Streams*: User-overloadable CLOS classes whose instances can be used as Lisp streams (e.g. passed as the first argument to `format`).
- *Simple Streams*: The bundled contrib module `sb-simple-streams` implements a subset of the Franz Allegro simple-streams proposal.

11.1 Stream External Formats

The function `stream-external-format` returns the canonical name of the external format (see Chapter 8 [external formats], page 88) used by the stream for character-based input and/or output.

When constructing file streams, for example using `open` or `with-open-file`, the external format to use is specified via the `:external-format` argument which accepts an external format designator (see Section 8.2 [external format designators], page 88).

11.2 Bivalent Streams

A *bivalent stream* can be used to read and write both `character` and `(unsigned-byte 8)` values. A bivalent stream is created by calling `open` with the argument `:element-type :default`. On such a stream, both binary and character data can be read and written with the usual input and output functions.

Streams are *not* created bivalent by default for performance reasons. Bivalent streams are incompatible with `fast-read-char`, an internal optimization in SBCL's stream machinery that bulk-converts octets to characters and implements a fast path through `read-char`.

11.3 Gray Streams

The Gray Streams interface is a widely supported extension that provides for definition of CLOS-extensible stream classes. Gray stream classes are implemented by adding methods to generic functions analogous to Common Lisp's standard I/O functions. Instances of Gray stream classes may be used with any I/O operation where a non-Gray stream can, provided that all required methods have been implemented suitably.

11.3.1 Gray Streams classes

The defined Gray Stream classes are these:

`sb-gray:fundamental-stream` [Class]
Base class for all Gray streams.

`sb-gray:fundamental-input-stream` [Class]
Superclass of all Gray input streams.

The function `input-stream-p` will return true of any generalized instance of `sb-gray:fundamental-input-stream`.

`sb-gray:fundamental-output-stream` [Class]
Superclass of all Gray output streams.

The function `output-stream-p` will return true of any generalized instance of `sb-gray:fundamental-output-stream`.

`sb-gray:fundamental-binary-stream` [Class]
Superclass of all Gray streams whose element-type is a subtype of unsigned-byte or signed-byte.

Note that instantiable subclasses of `sb-gray:fundamental-binary-stream` should provide (or inherit) an applicable method for the generic function `stream-element-type`.

`sb-gray:fundamental-character-stream` [Class]
Superclass of all Gray streams whose element-type is a subtype of character.

`sb-gray:fundamental-binary-input-stream` [Class]
Superclass of all Gray input streams whose element-type is a subtype of unsigned-byte or signed-byte.

`sb-gray:fundamental-binary-output-stream` [Class]
Superclass of all Gray output streams whose element-type is a subtype of unsigned-byte or signed-byte.

`sb-gray:fundamental-character-input-stream` [Class]
Superclass of all Gray input streams whose element-type is a subtype of character.

`sb-gray:fundamental-character-output-stream` [Class]
Superclass of all Gray output streams whose element-type is a subtype of character.

11.3.2 Methods common to all streams

These generic functions can be specialized on any generalized instance of fundamental-stream.

`stream-element-type` *stream* [Generic function]
Return a type specifier for the kind of object returned by the `stream`. The class `sb-gray:fundamental-character-stream` provides a default method which returns `character`.

`close` *stream* &key *abort* [Generic function]
Close the given `stream`. No more I/O may be performed, but inquiries may still be made. If `:abort` is true, an attempt is made to clean up the side effects of having created the stream.

`sb-gray:stream-file-position` *stream* &optional *position-spec* [Generic function]
Used by `file-position`. Returns or changes the current position within `stream`.

11.3.3 Input stream methods

These generic functions may be specialized on any generalized instance of fundamental-input-stream.

`sb-gray:stream-clear-input` *stream* [Generic function]
This is like `cl:clear-input`, but for Gray streams, returning `nil`. The default method does nothing.

`sb-gray:stream-read-sequence` *stream seq* &optional *start end* [Generic function]
This is like `cl:read-sequence`, but for Gray streams.

11.3.4 Character input stream methods

These generic functions are used to implement subclasses of `sb-gray:fundamental-input-stream`:

`sb-gray:stream-peek-char` *stream* [Generic function]
This is used to implement `peek-char`; this corresponds to `peek-type` of `nil`. It returns either a character or `:eof`. The default method calls `stream-read-char` and `stream-unread-char`.

`sb-gray:stream-read-char-no-hang` *stream* [Generic function]
This is used to implement `read-char-no-hang`. It returns either a character, or `nil` if no input is currently available, or `:eof` if end-of-file is reached. The default method provided by `fundamental-character-input-stream` simply calls `stream-read-char`; this is sufficient for file streams, but interactive streams should define their own method.

`sb-gray:stream-read-char` *stream* [Generic function]
Read one character from the stream. Return either a character object, or the symbol `:eof` if the stream is at end-of-file. Every subclass of `fundamental-character-input-stream` must define a method for this function.

`sb-gray:stream-read-line` *stream* [Generic function]
This is used by `read-line`. A string is returned as the first value. The second value is true if the string was terminated by end-of-file instead of the end of a line. The default method uses repeated calls to `stream-read-char`.

`sb-gray:stream-listen` *stream* [Generic function]
This is used by `listen`. It returns true or false. The default method uses `stream-read-char-no-hang` and `stream-unread-char`. Most streams should define their own method since it will usually be trivial and will always be more efficient than the default method.

`sb-gray:stream-unread-char` *stream character* [Generic function]
Undo the last call to `stream-read-char`, as in `unread-char`. Return `nil`. Every subclass of `fundamental-character-input-stream` must define a method for this function.

11.3.5 Output stream methods

These generic functions are used to implement subclasses of `sb-gray:fundamental-output-stream`:

`sb-gray:stream-clear-output` *stream* [Generic function]
This is like `cl:clear-output`, but for Gray streams: clear the given output `stream`. The default method does nothing.

`sb-gray:stream-finish-output` *stream* [Generic function]
Attempts to ensure that all output sent to the Stream has reached its destination, and only then returns false. Implements `finish-output`. The default method does nothing.

`sb-gray:stream-force-output` *stream* [Generic function]
Attempts to force any buffered output to be sent. Implements `force-output`. The default method does nothing.

`sb-gray:stream-write-sequence` *stream seq* &optional *start end* [Generic function]
This is like `cl:write-sequence`, but for Gray streams.

11.3.6 Character output stream methods

These generic functions are used to implement subclasses of `sb-gray:fundamental-character-output-stream`:

`sb-gray:stream-advance-to-column` *stream column* [Generic function]
Write enough blank space so that the next character will be written at the specified column. Returns true if the operation is successful, or `nil` if it is not supported for this stream. This is intended for use by by `pprint` and `format ~t`. The default method uses `stream-line-column` and repeated calls to `stream-write-char` with a #SPACE character; it returns `nil` if `stream-line-column` returns `nil`.

`sb-gray:stream-fresh-line` *stream* [Generic function]
Outputs a new line to the Stream if it is not positioned at the beginning of a line. Returns `t` if it output a new line, nil otherwise. Used by `fresh-line`. The default method uses `stream-start-line-p` and `stream-terpri`.

`sb-gray:stream-line-column` *stream* [Generic function]
Return the column number where the next character will be written, or `nil` if that is not meaningful for this stream. The first column on a line is numbered 0. This function is used in the implementation of `pprint` and the `format ~t` directive. For every character output stream class that is defined, a method must be defined for this function, although it is permissible for it to always return `nil`.

`sb-gray:stream-line-length` *stream* [Generic function]
Return the stream line length or `nil`.

`sb-gray:stream-start-line-p` *stream* [Generic function]
Is `stream` known to be positioned at the beginning of a line? It is permissible for an implementation to always return `nil`. This is used in the implementation of `fresh-line`. Note that while a value of 0 from `stream-line-column` also indicates the beginning of a line, there are cases where `stream-start-line-p` can be meaningfully implemented although `stream-line-column` can't be. For example, for a window using variable-width characters, the column number isn't very meaningful, but the beginning of the line does have a clear meaning. The default method for `stream-start-line-p` on class `fundamental-character-output-stream` uses `stream-line-column`, so if that is defined to return `nil`, then a method should be provided for either `stream-start-line-p` or `stream-fresh-line`.

`sb-gray:stream-terpri` *stream* [Generic function]
Writes an end of line, as for `terpri`. Returns `nil`. The default method does (`stream-write-char` stream `#\Newline`).

`sb-gray:stream-write-char` *stream character* [Generic function]
Write `character` to `stream` and return `character`. Every subclass of `fundamental-character-output-stream` must have a method defined for this function.

`sb-gray:stream-write-string` *stream string* &optional *start end* [Generic function]
This is used by `write-string`. It writes the string to the stream, optionally delimited by start and end, which default to 0 and `nil`. The string argument is returned. The default method provided by `fundamental-character-output-stream` uses repeated calls to `stream-write-char`.

11.3.7 Binary stream methods

The following generic functions are available for subclasses of `sb-gray:fundamental-binary-stream`:

`sb-gray:stream-read-byte` *stream* [Generic function]
Used by `read-byte`; returns either an integer, or the symbol `:eof` if the stream is at end-of-file.

`sb-gray:stream-write-byte` *stream integer* [Generic function]
Implements `write-byte`; writes the integer to the stream and returns the integer as the result.

11.3.8 Gray Streams Examples

Below are two classes of stream that can be conveniently defined as wrappers for Common Lisp streams. These are meant to serve as examples of minimal implementations of the protocols that must be followed when defining Gray streams. Realistic uses of the Gray Streams API would implement the various methods that can do I/O in batches, such as `sb-gray:stream-read-line`, `sb-gray:stream-write-string`, `sb-gray:stream-read-sequence`, and `sb-gray:stream-write-sequence`.

11.3.8.1 Character Counting Input Stream

It is occasionally handy for programs that process input files to count the number of characters and lines seen so far, and the number of characters seen on the current line, so that useful messages may be reported in case of parsing errors, etc. Here is a character input stream class that keeps track of these counts. Note that all character input streams must implement `sb-gray:stream-read-char` and `sb-gray:stream-unread-char`.

```
(defclass wrapped-stream (fundamental-stream)
  ((stream :initarg :stream :reader stream-of)))

(defmethod stream-element-type ((stream wrapped-stream))
  (stream-element-type (stream-of stream)))

(defmethod close ((stream wrapped-stream) &key abort)
  (close (stream-of stream) :abort abort))

(defclass wrapped-character-input-stream
    (wrapped-stream fundamental-character-input-stream)
  ())

(defmethod stream-read-char ((stream wrapped-character-input-stream))
  (read-char (stream-of stream) nil :eof))

(defmethod stream-unread-char ((stream wrapped-character-input-stream)
                               char)
  (unread-char char (stream-of stream)))

(defclass counting-character-input-stream
    (wrapped-character-input-stream)
  ((char-count :initform 1 :accessor char-count-of)
   (line-count :initform 1 :accessor line-count-of)
```

```
   (col-count :initform 1 :accessor col-count-of)
   (prev-col-count :initform 1 :accessor prev-col-count-of)))

(defmethod stream-read-char ((stream counting-character-input-stream))
  (with-accessors ((inner-stream stream-of) (chars char-count-of)
                   (lines line-count-of) (cols col-count-of)
                   (prev prev-col-count-of)) stream
      (let ((char (call-next-method)))
        (cond ((eql char :eof)
               :eof)
              ((char= char #Newline)
               (incf lines)
               (incf chars)
               (setf prev cols)
               (setf cols 1)
               char)
              (t
               (incf chars)
               (incf cols)
               char)))))

(defmethod stream-unread-char ((stream counting-character-input-stream)
                               char)
  (with-accessors ((inner-stream stream-of) (chars char-count-of)
                   (lines line-count-of) (cols col-count-of)
                   (prev prev-col-count-of)) stream
      (cond ((char= char #Newline)
             (decf lines)
             (decf chars)
             (setf cols prev))
            (t
             (decf chars)
             (decf cols)
             char))
      (call-next-method)))
```

The default methods for `sb-gray:stream-read-char-no-hang`, `sb-gray:stream-peek-char`, `sb-gray:stream-listen`, `sb-gray:stream-clear-input`, `sb-gray:stream-read-line`, and `sb-gray:stream-read-sequence` should be sufficient (though the last two will probably be slower than methods that forwarded directly).

Here's a sample use of this class:

```
(with-input-from-string (input "1 2
 3 :foo  ")
  (let ((counted-stream (make-instance 'counting-character-input-stream
                         :stream input)))
    (loop for thing = (read counted-stream) while thing
       unless (numberp thing) do
         (error "Non-number ~S (line ~D, column ~D)" thing
                (line-count-of counted-stream)
                (- (col-count-of counted-stream)
                   (length (format nil "~S" thing))))
       end
       do (print thing))))
```

Output:

```
1
2
3
Non-number :FOO (line 2, column 5)
```

```
[Condition of type SIMPLE-ERROR]
```

11.3.8.2 Output Prefixing Character Stream

One use for a wrapped output stream might be to prefix each line of text with a timestamp, e.g. for a logging stream. Here's a simple stream that does this, though without any fancy line-wrapping. Note that all character output stream classes must implement `sb-gray:stream-write-char` and `sb-gray:stream-line-column`.

```
(defclass wrapped-stream (fundamental-stream)
  ((stream :initarg :stream :reader stream-of)))

(defmethod stream-element-type ((stream wrapped-stream))
  (stream-element-type (stream-of stream)))

(defmethod close ((stream wrapped-stream) &key abort)
  (close (stream-of stream) :abort abort))

(defclass wrapped-character-output-stream
    (wrapped-stream fundamental-character-output-stream)
  ((col-index :initform 0 :accessor col-index-of)))

(defmethod stream-line-column ((stream wrapped-character-output-stream))
  (col-index-of stream))

(defmethod stream-write-char ((stream wrapped-character-output-stream)
                              char)
  (with-accessors ((inner-stream stream-of) (cols col-index-of)) stream
    (write-char char inner-stream)
    (if (char= char #Newline)
        (setf cols 0)
        (incf cols))))

(defclass prefixed-character-output-stream
    (wrapped-character-output-stream)
  ((prefix :initarg :prefix :reader prefix-of)))

(defgeneric write-prefix (prefix stream)
  (:method ((prefix string) stream) (write-string prefix stream))
  (:method ((prefix function) stream) (funcall prefix stream)))

(defmethod stream-write-char ((stream prefixed-character-output-stream)
                              char)
  (with-accessors ((inner-stream stream-of) (cols col-index-of)
                   (prefix prefix-of)) stream
    (when (zerop cols)
      (write-prefix prefix inner-stream))
    (call-next-method)))
```

As with the example input stream, this implements only the minimal protocol. A production implementation should also provide methods for at least `sb-gray:stream-write-string`, `sb-gray:stream-write-sequence`.

And here's a sample use of this class:

```
(flet ((format-timestamp (stream)
         (apply #'format stream "[~2@*~2,' D:~1@*~2,'0D:~0@*~2,'0D] "
                (multiple-value-list (get-decoded-time)))))
  (let ((output (make-instance 'prefixed-character-output-stream
                               :stream *standard-output*
                               :prefix #'format-timestamp)))
    (loop for string in '("abc" "def" ")ghi") do
```

```
              (write-line string output)
              (sleep 1))))
```

Output:

```
[ 0:30:05] abc
[ 0:30:06] def
[ 0:30:07] ghi
NIL
```

11.4 Simple Streams

Simple streams are an extensible streams protocol that avoids some problems with Section 11.3 [gray streams], page 105.

Documentation about simple streams is available at:

`http://www.franz.com/support/documentation/6.2/doc/streams.htm`

The implementation should be considered Alpha-quality; the basic framework is there, but many classes are just stubs at the moment.

See `SYS:CONTRIB;SB-SIMPLE-STREAMS;SIMPLE-STREAM-TEST.LISP` for things that should work.

Known differences to the ACL behaviour:

- `sb-simple-streams:open` does not return a `simple-stream` by default. See its `:class` argument.
- `write-vector` is unimplemented.

12 Package Locks

None of the following sections apply to SBCL built without package locking support.

The interface described here is experimental: incompatible changes in future SBCL releases are possible, even expected: the concept of *implementation packages* and the associated operators may be renamed; more operations (such as naming restarts or catch tags) may be added to the list of operations violating package locks.

12.1 Package Lock Concepts

Package locks protect against unintentional modifications of a package: they provide similar protection to user packages as is mandated to `common-lisp` package by the ANSI specification. They are not, and should not be used as, a security measure.

Newly created packages are by default unlocked (see the `:lock` option to `defpackage`).

The package `common-lisp` and SBCL internal implementation packages are locked by default, including `sb-ext`.

It may be beneficial to lock `common-lisp-user` as well, to ensure that various libraries don't pollute it without asking, but this is not currently done by default.

12.1.1 Implementation Packages

Each package has a list of associated implementation packages. A locked package, and the symbols whose home package it is, can be modified without violating package locks only when `*package*` is bound to one of the implementation packages of the locked package.

Unless explicitly altered by `defpackage`, `sb-ext:add-implementation-package`, or `sb-ext:remove-implementation-package`, each package is its own (only) implementation package.

12.1.2 Package Lock Violations

12.1.2.1 Lexical Bindings and Declarations

Lexical bindings or declarations that violate package locks cause a compile-time warning, and a runtime `program-error` when the form that violates package locks would be executed.

A complete listing of operators affected by this is: `let`, `let*`, `flet`, `labels`, `macrolet`, `symbol-macrolet`, `declare`.

Package locks affecting both lexical bindings and declarations can be disabled locally with the `sb-ext:disable-package-locks` declaration, and re-enabled with the `sb-ext:enable-package-locks` declaration.

Example:

```
(in-package :locked)

(defun foo () ...)

(defmacro with-foo (&body body)
  `(locally (declare (disable-package-locks locked:foo))
     (flet ((foo () ...))
       (declare (enable-package-locks locked:foo)) ; re-enable for body
       ,@body)))
```

12.1.2.2 Other Operations

If a non-lexical operation violates a package lock, a continuable error that is of a subtype of `sb-ext:package-lock-violation` (subtype of `package-error`) is signalled when the operation is attempted.

Additional restarts may be established for continuable package lock violations for interactive use.

The actual type of the error depends on circumstances that caused the violation: operations on packages signal errors of type `sb-ext:package-locked-error`, and operations on symbols signal errors of type `sb-ext:symbol-package-locked-error`.

12.1.3 Package Locks in Compiled Code

If file-compiled code contains interned symbols, then loading that code into an image without the said symbols will not cause a package lock violation, even if the packages in question are locked.

With the exception of interned symbols, behaviour is unspecified if package locks affecting compiled code are not the same during loading of the code or execution.

Specifically, code compiled with packages unlocked may or may not fail to signal package-lock-violations even if the packages are locked at runtime, and code compiled with packages locked may or may not signal spurious package-lock-violations at runtime even if the packages are unlocked.

In practice all this means that package-locks have a negligible performance penalty in compiled code as long as they are not violated.

12.1.4 Operations Violating Package Locks

12.1.4.1 Operations on Packages

The following actions cause a package lock violation if the package operated on is locked, and `*package*` is not an implementation package of that package, and the action would cause a change in the state of the package (so e.g. exporting already external symbols is never a violation). Package lock violations caused by these operations signal errors of type `sb-ext:package-locked-error`.

- Shadowing a symbol in a package.
- Importing a symbol to a package.
- Uninterning a symbol from a package.
- Exporting a symbol from a package.
- Unexporting a symbol from a package.
- Changing the packages used by a package.
- Renaming a package.
- Deleting a package.
- Adding a new package local nickname to a package.
- Removing an existing package local nickname from a package.

12.1.4.2 Operations on Symbols

Following actions cause a package lock violation if the home package of the symbol operated on is locked, and `*package*` is not an implementation package of that package. Package lock violations caused by these action signal errors of type `sb-ext:symbol-package-locked-error`.

These actions cause only one package lock violation per lexically apparent violated package.

Example:

```
;;; Packages FOO and BAR are locked.
;;;
;;; Two lexically apparent violated packages: exactly two
;;; package-locked-errors will be signalled.

(defclass foo:point ()
  ((x :accessor bar:x)
   (y :accessor bar:y)))
```

- Binding or altering its value lexically or dynamically, or establishing it as a symbol-macro.

  Exceptions:

  - If the symbol is not defined as a constant, global symbol-macro or a global dynamic variable, it may be lexically bound or established as a local symbol macro.
  - If the symbol is defined as a global dynamic variable, it may be assigned or bound.
- Defining, undefining, or binding it, or its setf name as a function.

  Exceptions:

  - If the symbol is not defined as a function, macro, or special operator it and its setf name may be lexically bound as a function.

- Defining, undefining, or binding it as a macro or compiler macro.

  Exceptions:

  - If the symbol is not defined as a function, macro, or special operator it may be lexically bound as a macro.
- Defining it as a type specifier or structure.
- Defining it as a declaration with a declaration proclamation.
- Declaring or proclaiming it special.
- Declaring or proclaiming its type or ftype.

  Exceptions:

  - If the symbol may be lexically bound, the type of that binding may be declared.
  - If the symbol may be lexically bound as a function, the ftype of that binding may be declared.
- Defining a setf expander for it.
- Defining it as a method combination type.
- Using it as the `class-name` argument to (`setf find-class`).
- Defining it as a hash table test using `sb-ext:define-hash-table-test`.

12.2 Package Lock Dictionary

`sb-ext:disable-package-locks` [Declaration]

Syntax: `(sb-ext:disable-package-locks &rest symbols)`

Disables package locks affecting the named symbols during compilation in the lexical scope of the declaration. Disabling locks on symbols whose home package is unlocked, or disabling an already disabled lock, has no effect.

`sb-ext:enable-package-locks` [Declaration]

Syntax: `(sb-ext:enable-package-locks &rest symbols)`

Re-enables package locks affecting the named symbols during compilation in the lexical scope of the declaration. Enabling locks that were not first disabled with `sb-ext:disable-package-locks` declaration, or enabling locks that are already enabled has no effect.

`sb-ext:package-lock-violation` [Condition]

Subtype of `cl:package-error`. A subtype of this error is signalled when a package-lock is violated.

`sb-ext:package-locked-error` [Condition]

Subtype of `sb-ext:package-lock-violation`. An error of this type is signalled when an operation on a package violates a package lock.

`sb-ext:symbol-package-locked-error` [Condition]

Subtype of `sb-ext:package-lock-violation`. An error of this type is signalled when an operation on a symbol violates a package lock. The symbol that caused the violation is accessed by the function `sb-ext:package-locked-error-symbol`.

`sb-ext:package-locked-error-symbol` *condition* [Function]

Return the symbol that caused the `symbol-package-locked-error` condition.

`sb-ext:package-locked-p` *package* [Function]

Returns `t` when `package` is locked, `nil` otherwise. Signals an error if `package` doesn't designate a valid package.

`sb-ext:lock-package` *package* [Function]

Locks `package` and returns `t`. Has no effect if `package` was already locked. Signals an error if `package` is not a valid package designator.

`sb-ext:unlock-package` *package* [Function]

Unlocks `package` and returns `t`. Has no effect if `package` was already unlocked. Signals an error if `package` is not a valid package designator.

`sb-ext:package-implemented-by-list` *package* [Function]
Returns a list containing the implementation packages of `package`. Signals an error if `package` is not a valid package designator.

`sb-ext:package-implements-list` *package* [Function]
Returns the packages that `package` is an implementation package of. Signals an error if `package` is not a valid package designator.

`sb-ext:add-implementation-package` *packages-to-add* &optional (*package* **package**) [Function]
Adds `packages-to-add` as implementation packages of `package`. Signals an error if `package` or any of the `packages-to-add` is not a valid package designator.

`sb-ext:remove-implementation-package` *packages-to-remove* &optional (*package* **package**) [Function]
Removes `packages-to-remove` from the implementation packages of `package`. Signals an error if `package` or any of the `packages-to-remove` is not a valid package designator.

`sb-ext:without-package-locks` &body *body* [Macro]
Ignores all runtime package lock violations during the execution of body. Body can begin with declarations.

`sb-ext:with-unlocked-packages` (&rest *packages*) &body *forms* [Macro]
Unlocks `packages` for the dynamic scope of the body. Signals an error if any of `packages` is not a valid package designator.

The `defpackage` options are extended to include the following:

- `:lock <boolean>` (defaults to `nil`)

  If the argument to `:lock` is `t`, the package is locked, else it is unlocked. Existing package are also affected.

- `:implement <package-designator>*`

  The package is added as an implementation package to the packages named. If `:implement` is not provided, it defaults to the package itself.

Example:

```
(defpackage "FOO" (:export "BAR") (:lock t) (:implement))
(defpackage "FOO-INT" (:use "FOO") (:implement "FOO" "FOO-INT"))

;;; is equivalent to

(defpackage "FOO") (:export "BAR"))
(lock-package "FOO")
(remove-implementation-package "FOO" "FOO")

(defpackage "FOO-INT" (:use "BAR"))
(add-implementation-package "FOO-INT" "FOO")
```

13 Threading

SBCL supports a fairly low-level threading interface that maps onto the host operating system's concept of threads or lightweight processes. This means that threads may take advantage of hardware multiprocessing on machines that have more than one CPU, but it does not allow Lisp control of the scheduler. This is found in the `sb-thread` package.

Threads are part of the default build on x86[-64]/ARM64 Linux and Windows.

They are also supported on: x86[-64] Darwin (Mac OS X), x86[-64] FreeBSD, x86 SunOS (Solaris), PPC Linux, ARM64 Linux, RISC-V Linux. On these platforms threads must be explicitly enabled at build-time, see `install` for directions.

13.1 Threading Basics

```
(make-thread (lambda () (write-line "Hello, world")))
```

13.1.1 Thread Objects

`sb-thread:thread` [Structure]
Thread type. Do not rely on threads being structs as it may change in future versions.

`sb-thread:*current-thread*` [Variable]
Bound in each thread to the thread itself.

`sb-thread:list-all-threads` [Function]
Return a list of the live threads. Note that the return value is potentially stale even before the function returns, as new threads may be created and old ones may exit at any time.

`sb-thread:thread-alive-p` *thread* [Function]
Return `t` if `thread` is still alive. Note that the return value is potentially stale even before the function returns, as the thread may exit at any time.

`sb-thread:thread-name` *thread* [Function]
Name of the thread. Can be assigned to using `setf`. A thread name must be a simple-string (not necessarily unique) or `nil`.

`sb-thread:main-thread-p` &optional (*thread sb-thread:*current-thread**) [Function]
True if `thread`, defaulting to current thread, is the main thread of the process.

`sb-thread:main-thread` [Function]
Returns the main thread of the process.

13.1.2 Running Threads

`sb-thread:make-thread` *function* &key *name arguments* [Function]
Create a new thread of `name` that runs `function` with the argument list designator provided (defaults to no argument). Thread exits when the function returns. The return values of `function` are kept around and can be retrieved by `join-thread`.

Invoking the initial `abort` restart established by `make-thread` terminates the thread.

`sb-thread:return-from-thread` *values-form* &key *allow-exit* [Macro]
Unwinds from and terminates the current thread, with values from `values-form` as the results visible to `join-thread`.

If current thread is the main thread of the process (see `main-thread-p`), signals an error unless `allow-exit` is true, as terminating the main thread would terminate the entire process. If `allow-exit` is true, returning from the main thread is equivalent to calling `sb-ext:exit` with `:code` 0 and `:abort nil`.

`sb-thread:abort-thread` &key *allow-exit* [Function]
Unwinds from and terminates the current thread abnormally, causing `join-thread` on current thread to signal an error unless a default-value is provided.

If current thread is the main thread of the process (see `main-thread-p`), signals an error unless `allow-exit` is true, as terminating the main thread would terminate the entire process. If `allow-exit` is true, aborting the main thread is equivalent to calling `sb-ext:exit` code 1 and `:abort nil`.

Invoking the initial `abort` restart established by `make-thread` is equivalent to calling `abort-thread` in other than main threads. However, whereas `abort` restart may be rebound, `abort-thread` always unwinds the entire thread. (Behaviour of the initial `abort` restart for main thread depends on the `:toplevel` argument to `sb-ext:save-lisp-and-die`.)

`sb-thread:join-thread` *thread* &key *default timeout* [Function]

Suspend current thread until `thread` exits. Return the result values of the thread function.

If `thread` does not exit within `timeout` seconds and `default` is supplied, return two values: 1) `default` 2) `:timeout`. If `default` is not supplied, signal a `join-thread-error` with `join-thread-problem` equal to `:timeout`.

If `thread` does not exit normally (i.e. aborted) and `default` is supplied, return two values: 1) `default` 2) `:abort`. If `default` is not supplied, signal a `join-thread-error` with `join-thread-problem` equal to `:abort`.

If `thread` is the current thread, signal a `join-thread-error` with `join-thread-problem` equal to `:self-join`.

Trying to join the main thread causes `join-thread` to block until `timeout` occurs or the process exits: when the main thread exits, the entire process exits.

Users should not rely on the ability to join a chosen `thread` from more than one other thread simultaneously. Future changes to `join-thread` may directly call the underlying thread library, and not all threading implementations consider such usage to be well-defined.

Note: Return convention in case of a timeout is experimental and subject to change.

`sb-thread:thread-yield` [Function]

Yield the processor to other threads.

13.1.3 Asynchronous Operations

`sb-thread:interrupt-thread` *thread function* [Function]

Interrupt `thread` and make it run `function`.

The interrupt is asynchronous, and can occur anywhere with the exception of sections protected using `sb-sys:without-interrupts`.

`function` is called with interrupts disabled, under `sb-sys:allow-with-interrupts`. Since functions such as `grab-mutex` may try to enable interrupts internally, in most cases `function` should either enter `sb-sys:with-interrupts` to allow nested interrupts, or `sb-sys:without-interrupts` to prevent them completely.

When a thread receives multiple interrupts, they are executed in the order they were sent – first in, first out.

This means that a great degree of care is required to use `interrupt-thread` safely and sanely in a production environment. The general recommendation is to limit uses of `interrupt-thread` for interactive debugging, banning it entirely from production environments – it is simply exceedingly hard to use correctly.

With those caveats in mind, what you need to know when using it:

- If calling `function` causes a non-local transfer of control (i.e. an unwind), all normal cleanup forms will be executed.

  However, if the interrupt occurs during cleanup forms of an `unwind-protect`, it is just as if that had happened due to a regular `go`, `throw`, or `return-from`: the interrupted cleanup form and those following it in the same `unwind-protect` do not get executed.

  SBCL tries to keep its own internals asynch-unwind-safe, but this is frankly an unreasonable expectation for third party libraries, especially given that asynch-unwind-safety does not compose: a function calling only asynch-unwind-safe function isn't automatically asynch-unwind-safe.

  This means that in order for an asynch unwind to be safe, the entire callstack at the point of interruption needs to be asynch-unwind-safe.

- In addition to asynch-unwind-safety you must consider the issue of reentrancy. `interrupt-thread` can cause function that are never normally called recursively to be re-entered during their dynamic contour, which may cause them to misbehave. (Consider binding of special variables, values of global variables, etc.)

Taken together, these two restrict the "safe" things to do using `interrupt-thread` to a fairly minimal set. One useful one – exclusively for interactive development use is using it to force entry to debugger to inspect the state of a thread:

```
(interrupt-thread thread #'break)
```

Short version: be careful out there.

`sb-thread:terminate-thread` *thread* [Function]

Terminate the thread identified by `thread`, by interrupting it and causing it to call `sb-thread:abort-thread` with `:allow-exit t`.

The unwind caused by `terminate-thread` is asynchronous, meaning that e.g. thread executing

```
(let (foo)
   (unwind-protect
       (progn
          (setf foo (get-foo))
          (work-on-foo foo))
     (when foo
       ;; An interrupt occurring inside the cleanup clause
       ;; will cause cleanups from the current UNWIND-PROTECT
       ;; to be dropped.
       (release-foo foo))))
```

might miss calling `release-foo` despite GET-FOO having returned true if the interrupt occurs inside the cleanup clause, e.g. during execution of `release-foo`.

Thus, in order to write an asynch unwind safe `unwind-protect` you need to use `without-interrupts`:

```
(let (foo)
  (sb-sys:without-interrupts
    (unwind-protect
        (progn
          (setf foo (sb-sys:allow-with-interrupts
                      (get-foo)))
          (sb-sys:with-local-interrupts
            (work-on-foo foo)))
      (when foo
        (release-foo foo)))))
```

Since most libraries using `unwind-protect` do not do this, you should never assume that unknown code can safely be terminated using `terminate-thread`.

13.1.4 Miscellaneous Operations

`sb-thread:symbol-value-in-thread` *symbol thread* &optional (*errorp t*) [Function]

Return the local value of `symbol` in `thread`, and a secondary value of `t` on success.

If the value cannot be retrieved (because the thread has exited or because it has no local binding for `name`) and `errorp` is true signals an error of type `symbol-value-in-thread-error`; if `errorp` is false returns a primary value of `nil`, and a secondary value of `nil`.

Can also be used with `setf` to change the thread-local value of `symbol`.

`symbol-value-in-thread` is primarily intended as a debugging tool, and not as a mechanism for inter-thread communication.

13.1.5 Error Conditions

`sb-thread:thread-error` [Condition]

Conditions of type `thread-error` are signalled when thread operations fail. The offending thread is initialized by the `:thread` initialization argument and read by the function `thread-error-thread`.

`sb-thread:thread-error-thread` *condition* [Function]

Return the offending thread that the `thread-error` pertains to.

`sb-thread:symbol-value-in-thread-error` [Condition]
Signalled when `symbol-value-in-thread` or its `setf` version fails due to e.g. the symbol not having a thread-local value, or the target thread having exited. The offending symbol can be accessed using `cell-error-name`, and the offending thread using `thread-error-thread`.

`sb-thread:interrupt-thread-error` [Condition]
Signalled when interrupting a thread fails because the thread has already exited. The offending thread can be accessed using `thread-error-thread`.

`sb-thread:join-thread-error` [Condition]
Signalled when joining a thread fails due to abnormal exit of the thread to be joined. The offending thread can be accessed using `thread-error-thread`.

13.2 Special Variables

The interaction of special variables with multiple threads is mostly as one would expect, with behaviour very similar to other implementations.

- Global special values are visible across all threads.
- Bindings (e.g. using `let`) are local to the thread.
- Threads do not inherit dynamic bindings from the parent thread.

The last point means that

```
(defparameter *x* 0)
(let ((*x* 1))
  (sb-thread:make-thread (lambda () (print *x*))))
```

prints `0` and not `1`.

Note, however, that there is a hard limit on the number of distinct symbols that can be bound dynamically in threaded builds (see `--tls-limit` in Section 3.3.1 [runtime options], page 11). Exceeding this limit invokes Section 5.2 [ldb], page 39, with the error `Thread local storage exhausted.`

13.3 Atomic Operations

Following atomic operations are particularly useful for implementing lockless algorithms.

`sb-ext:atomic-decf` *place* &optional (*diff 1*) [Macro]
Atomically decrements `place` by `diff`, and returns the value of `place` before the decrement.

`place` must access one of the following:

- a `defstruct` slot with declared type `(unsigned-byte 64)` or `aref` of a `(simple-array (unsigned-byte 64) (*))` (the type `sb-ext:word` can be used for these purposes)
- `car` or `cdr` (respectively `first` or `rest`) of a `cons`,
- a variable defined using `defglobal` with a proclaimed type of `fixnum`.

Macroexpansion is performed on `place` before expanding `atomic-decf`.

Decrementing is done using modular arithmetic, which is well-defined over two different domains:

- For structures and arrays, the operation accepts and produces an `(unsigned-byte 64)`, and `diff` must be of type `(signed-byte 64)`. `atomic-decf` of `#x0` by one results in `#xFFFFFFFFFFFFFFFF` being stored in `place`.
- For other places, the domain is `fixnum`, and `diff` must be a `fixnum`. `atomic-decf` of `#x4000000000000000` by one results in `#x3FFFFFFFFFFFFFFF` being stored in `place`.

`diff` defaults to 1.

EXPERIMENTAL: Interface subject to change.

`sb-ext:atomic-incf` *place* &optional (*diff 1*) [Macro]
Atomically increments `place` by `diff`, and returns the value of `place` before the increment.

`place` must access one of the following:

- a `defstruct` slot with declared type `(unsigned-byte 64)` or `aref` of a `(simple-array (unsigned-byte 64) (*))` The type `sb-ext:word` can be used for these purposes.

- `car` or `cdr` (respectively `first` or `rest`) of a `cons`.
- a variable defined using `defglobal` with a proclaimed type of `fixnum`. Macroexpansion is performed on `place` before expanding `atomic-incf`.

Incrementing is done using modular arithmetic, which is well-defined over two different domains:

- For structures and arrays, the operation accepts and produces an (`unsigned-byte` 64), and `diff` must be of type (`signed-byte` 64). `atomic-incf` of `#xFFFFFFFFFFFFFFFF` by one results in #x0 being stored in `place`.
- For other places, the domain is `fixnum`, and `diff` must be a `fixnum`. `atomic-incf` of `#x3FFFFFFFFFFFFFFF` by one results in `#x-4000000000000000` being stored in `place`.

`diff` defaults to 1.

EXPERIMENTAL: Interface subject to change.

`sb-ext:atomic-pop` *place* [Macro]

Like `pop`, but atomic. `place` may be read multiple times before the operation completes – the write does not occur until such time that no other thread modified `place` between the read and the write.

Works on all `cas`able places.

`sb-ext:atomic-push` *obj place* [Macro]

Like `push`, but atomic. `place` may be read multiple times before the operation completes – the write does not occur until such time that no other thread modified `place` between the read and the write.

Works on all `cas`able places.

`sb-ext:atomic-update` *place update-fn* &rest *arguments* [Macro]

Updates `place` atomically to the value returned by calling function designated by `update-fn` with `arguments` and the previous value of `place`.

`place` may be read and `update-fn` evaluated and called multiple times before the update succeeds: atomicity in this context means that the value of `place` did not change between the time it was read, and the time it was replaced with the computed value.

`place` can be any place supported by `sb-ext:compare-and-swap`.

Examples:

```
;;; Conses T to the head of FOO-LIST.
(defstruct foo list)
(defvar *foo* (make-foo))
(atomic-update (foo-list *foo*) #'cons t)

(let ((x (cons :count 0)))
   (mapc #'sb-thread:join-thread
         (loop repeat 1000
               collect (sb-thread:make-thread
                        (lambda ()
                          (loop repeat 1000
                                do (atomic-update (cdr x) #'1+)
                                   (sleep 0.00001))))))
   ;; Guaranteed to be (:COUNT . 1000000) -- if you replace
   ;; atomic update with (INCF (CDR X)) above, the result becomes
   ;; unpredictable.
   x)
```

`sb-ext:compare-and-swap` *place old new* [Macro]

Atomically stores `new` in `place` if `old` matches the current value of `place`. Two values are considered to match if they are `eq`. Returns the previous value of `place`: if the returned value is `eq` to `old`, the swap was carried out.

`place` must be an `cas`-able place. Built-in `cas`-able places are accessor forms whose `car` is one of the following:

`car`, `cdr`, `first`, `rest`, `svref`, `symbol-plist`, `symbol-value`, `slot-value` `sb-mop:standard-instance-access`, `sb-mop:funcallable-standard-instance-access`,

or the name of a `defstruct` created accessor for a slot whose storage type is not raw. (Refer to the the "Efficiency" chapter of the manual for the list of raw slot types. Future extensions to this macro may allow it to work on some raw slot types.)

In case of `slot-value`, if the slot is unbound, `slot-unbound` is called unless `old` is `eq` to `sb-pcl:+slot-unbound+` in which case `sb-pcl:+slot-unbound+` is returned and `new` is assigned to the slot. Additionally, the results are unspecified if there is an applicable method on either `sb-mop:slot-value-using-class`, (`setf sb-mop:slot-value-using-class`), or `sb-mop:slot-boundp-using-class`.

Additionally, the `place` can be a anything for which a `cas`-function has been defined.

Our `sb-ext:compare-and-swap` is user-extensible by defining functions named `(cas <place>)`, allowing users to add CAS support to new places.

`sb-ext:cas` *place old new* [Macro]

Synonym for `compare-and-swap`.

Additionally `defun`, `defgeneric`, `defmethod`, `flet`, and `labels` can be also used to define `cas`-functions analogously to `setf`-functions:

```
(defvar *foo* nil)

(defun (cas foo) (old new)
  (cas (symbol-value '*foo*) old new))
```

First argument of a `cas` function is the expected old value, and the second argument of is the new value. Note that the system provides no automatic atomicity for `cas` functions, nor can it verify that they are atomic: it is up to the implementor of a `cas` function to ensure its atomicity.

EXPERIMENTAL: Interface subject to change.

`sb-ext:get-cas-expansion` *place* &optional *environment* [Function]

Analogous to `get-setf-expansion`. Returns the following six values:

- list of temporary variables
- list of value-forms whose results those variable must be bound
- temporary variable for the old value of `place`
- temporary variable for the new value of `place`
- form using the aforementioned temporaries which performs the compare-and-swap operation on `place`
- form using the aforementioned temporaries with which to perform a volatile read of `place`

Example:

```
(get-cas-expansion '(car x))
; => (#:CONS871), (X), #:OLD872, #:NEW873,
;    (SB-KERNEL:%COMPARE-AND-SWAP-CAR #:CONS871 #:OLD872 :NEW873).
;    (CAR #:CONS871)

(defmacro my-atomic-incf (place &optional (delta 1) &environment env)
  (multiple-value-bind (vars vals old new cas-form read-form)
      (get-cas-expansion place env)
   (let ((delta-value (gensym "DELTA")))
     `(let* (,@(mapcar 'list vars vals)
             (,old ,read-form)
             (,delta-value ,delta)
             (,new (+ ,old ,delta-value)))
        (loop until (eq ,old (setf ,old ,cas-form))
              do (setf ,new (+ ,old ,delta-value)))
        ,new))))
```

EXPERIMENTAL: Interface subject to change.

13.4 Mutex Support

Mutexes are used for controlling access to a shared resource. One thread is allowed to hold the mutex, others which attempt to take it will be made to wait until it's free. Threads are woken in the order that they go to sleep.

```
(defpackage :demo (:use "CL" "SB-THREAD" "SB-EXT"))

(in-package :demo)

(defvar *a-mutex* (make-mutex :name "my lock"))

(defun thread-fn ()
  (format t "Thread ~A running ~%" *current-thread*)
  (with-mutex (*a-mutex*)
    (format t "Thread ~A got the lock~%" *current-thread*)
    (sleep (random 5)))
  (format t "Thread ~A dropped lock, dying now~%" *current-thread*))

(make-thread #'thread-fn)
(make-thread #'thread-fn)
```

`sb-thread:mutex` [Structure]
Mutex type.

`sb-thread:with-mutex` (*mutex* &key (*wait-p t*) *timeout value*) &body *body* [Macro]
Acquire `mutex` for the dynamic scope of `body`. If `wait-p` is true (the default), and the `mutex` is not immediately available, sleep until it is available.

If `timeout` is given, it specifies a relative timeout, in seconds, on how long the system should try to acquire the lock in the contended case.

If the mutex isn't acquired successfully due to either `wait-p` or `timeout`, `body` is not executed, and `with-mutex` returns `nil`.

Otherwise body is executed with the mutex held by current thread, and `with-mutex` returns the values of `body`.

Historically `with-mutex` also accepted a `value` argument, which when provided was used as the new owner of the mutex instead of the current thread. This is no longer supported: if `value` is provided, it must be either `nil` or the current thread.

`sb-thread:with-recursive-lock` (*mutex* &key (*wait-p t*) *timeout*) &body *body* [Macro]
Acquire `mutex` for the dynamic scope of `body`.

If `wait-p` is true (the default), and the `mutex` is not immediately available or held by the current thread, sleep until it is available.

If `timeout` is given, it specifies a relative timeout, in seconds, on how long the system should try to acquire the lock in the contended case.

If the mutex isn't acquired successfully due to either `wait-p` or `timeout`, `body` is not executed, and `with-recursive-lock` returns `nil`.

Otherwise body is executed with the mutex held by current thread, and `with-recursive-lock` returns the values of `body`.

Unlike `with-mutex`, which signals an error on attempt to re-acquire an already held mutex, `with-recursive-lock` allows recursive lock attempts to succeed.

`sb-thread:make-mutex` &key *name* [Function]
Create a `mutex`.

`sb-thread:mutex-name` *instance* [Function]
The name of the `mutex`. `setf`able.

`sb-thread:mutex-owner` *mutex* [Function]
Current owner of `mutex`, `nil` if the mutex is free. Naturally, this is racy by design (another thread may acquire the mutex after this function returns), it is intended for informative purposes. For testing whether the current thread is holding a mutex see `holding-mutex-p`.

`sb-thread:mutex-value` *mutex* [Function]
Current owner of `mutex`, `nil` if the mutex is free. May return a stale value, use `mutex-owner` instead.

`sb-thread:grab-mutex` *mutex* &key (*waitp t*) *timeout* [Function]
Acquire `mutex` for the current thread. If `waitp` is true (the default) and the mutex is not immediately available, sleep until it is available.

If `timeout` is given, it specifies a relative timeout, in seconds, on how long `grab-mutex` should try to acquire the lock in the contended case.

If `grab-mutex` returns `t`, the lock acquisition was successful. In case of `waitp` being `nil`, or an expired `timeout`, `grab-mutex` may also return `nil` which denotes that `grab-mutex` did -not- acquire the lock.

Notes:

- `grab-mutex` is not interrupt safe. The correct way to call it is:

```
(without-interrupts
  ...
  (allow-with-interrupts (grab-mutex ...))
  ...)
```

  `without-interrupts` is necessary to avoid an interrupt unwinding the call while the mutex is in an inconsistent state while `allow-with-interrupts` allows the call to be interrupted from sleep.

- `(grab-mutex <mutex> :timeout 0.0)` differs from `(grab-mutex <mutex> :waitp nil)` in that the former may signal a `deadline-timeout` if the global deadline was due already on entering `grab-mutex`.

  The exact interplay of `grab-mutex` and deadlines are reserved to change in future versions.

- It is recommended that you use `with-mutex` instead of calling `grab-mutex` directly.

`sb-thread:release-mutex` *mutex* &key (*if-not-owner :punt*) [Function]
Release `mutex` and wake up any other thread waiting for it.

`release-mutex` is not interrupt safe: interrupts should be disabled around calls to it.

The `if-not-owner` keyword dictates behavior when the current thread does not own the mutex. Do nothing and silently return if `:punt`, signal a `warning` or `error` if `:warn` or `:error` respectively, or release the mutex anyway if `:force`.

13.5 Semaphores

Semaphores are among other things useful for keeping track of a countable resource, e.g. messages in a queue, and sleep when the resource is exhausted.

`sb-thread:semaphore` [Structure]
Semaphore type. The fact that a `semaphore` is a `structure-object` should be considered an implementation detail, and may change in the future.

`sb-thread:make-semaphore` &key *name* (*count 0*) [Function]
Create a semaphore with the supplied `count` and `name`.

`sb-thread:signal-semaphore` *semaphore* &optional (*n 1*) [Function]
Increment the count of `semaphore` by `n`. If there are threads waiting on this semaphore, then `n` of them is woken up.

`sb-thread:wait-on-semaphore` *semaphore* &key (*n 1*) *timeout notification* [Function]
Decrement the count of `semaphore` by `n` if the count would not be negative.

Else blocks until the semaphore can be decremented. Returns the new count of `semaphore` on success.

If `timeout` is given, it is the maximum number of seconds to wait. If the count cannot be decremented in that time, returns `nil` without decrementing the count.

If `notification` is given, it must be a `semaphore-notification` object whose `semaphore-notification-status` is `nil`. If `wait-on-semaphore` succeeds and decrements the count, the status is set to `t`.

`sb-thread:try-semaphore` *semaphore* &optional (*n 1*) *notification* [Function]
Try to decrement the count of `semaphore` by `n`. If the count were to become negative, punt and return `nil`, otherwise return the new count of `semaphore`.

If `notification` is given it must be a semaphore notification object with `semaphore-notification-status` of `nil`. If the count is decremented, the status is set to `t`.

`sb-thread:semaphore-count` *semaphore* [Function]
Returns the current count of `semaphore`.

`sb-thread:semaphore-name` *semaphore* [Function]
The name of the semaphore `instance`. `setf`able.

`sb-thread:semaphore-notification` [Structure]
Semaphore notification object. Can be passed to `wait-on-semaphore` and `try-semaphore` as the `:notification` argument. Consequences are undefined if multiple threads are using the same notification object in parallel.

`sb-thread:make-semaphore-notification` [Function]
Constructor for `semaphore-notification` objects. `semaphore-notification-status` is initially `nil`.

`sb-thread:semaphore-notification-status` *semaphore-notification* [Function]
Returns `t` if a `wait-on-semaphore` or `try-semaphore` using `semaphore-notification` has succeeded since the notification object was created or cleared.

`sb-thread:clear-semaphore-notification` *semaphore-notification* [Function]
Resets the `semaphore-notification` object for use with another call to `wait-on-semaphore` or `try-semaphore`.

13.6 Waitqueue/condition variables

These are based on the POSIX condition variable design, hence the annoyingly CL-conflicting name. For use when you want to check a condition and sleep until it's true. For example: you have a shared queue, a writer process checking *queue is empty* and one or more readers that need to know when *queue is not empty*. It sounds simple but is astonishingly easy to deadlock if another process runs when you weren't expecting it to.

There are three components:

- the condition itself (not represented in code)
- the condition variable (a.k.a. waitqueue) which proxies for it
- a lock to hold while testing the condition

Important stuff to be aware of:

- when calling condition-wait, you must hold the mutex. condition-wait will drop the mutex while it waits, and obtain it again before returning for whatever reason;
- likewise, you must be holding the mutex around calls to `sb-thread:condition-notify`;
- a process may return from `sb-thread:condition-wait` in several circumstances: it is not guaranteed that the underlying condition has become true. You must check that the resource is ready for whatever you want to do to it.

```
(defvar *buffer-queue* (make-waitqueue))
(defvar *buffer-lock* (make-mutex :name "buffer lock"))

(defvar *buffer* (list nil))

(defun reader ()
  (with-mutex (*buffer-lock*)
    (loop
     (condition-wait *buffer-queue* *buffer-lock*)
     (loop
      (unless *buffer* (return))
      (let ((head (car *buffer*)))
        (setf *buffer* (cdr *buffer*))
        (format t "reader ~A woke, read ~A~%"
                *current-thread* head))))))
```

```
(defun writer ()
  (loop
   (sleep (random 5))
   (with-mutex (*buffer-lock*)
     (let ((el (intern
                (string (code-char
                         (+ (char-code #A) (random 26)))))))
       (setf *buffer* (cons el *buffer*)))
     (condition-notify *buffer-queue*))))

(make-thread #'writer)
(make-thread #'reader)
(make-thread #'reader)
```

`sb-thread:waitqueue` [Structure]
Waitqueue type.

`sb-thread:make-waitqueue` &key *name* [Function]
Create a `waitqueue`.

`sb-thread:waitqueue-name` *instance* [Function]
The name of the waitqueue. `setf`able.

`sb-thread:condition-wait` *queue mutex* &key *timeout* [Function]
Atomically release `mutex` and start waiting on `queue` until another thread wakes us up using either `condition-notify` or `condition-broadcast` on `queue`, at which point we re-acquire `mutex` and return `t`.

Spurious wakeups are possible.

If `timeout` is given, it is the maximum number of seconds to wait, including both waiting for the wakeup and the time to re-acquire `mutex`. When neither a wakeup nor a re-acquisition occurs within the given time, returns `nil` without re-acquiring `mutex`.

If `condition-wait` unwinds, it may do so with or without `mutex` being held.

Important: Since `condition-wait` may return without `condition-notify` or `condition-broadcast` having occurred, the correct way to write code that uses `condition-wait` is to loop around the call, checking the associated data:

```
(defvar *data* nil)
(defvar *queue* (make-waitqueue))
(defvar *lock* (make-mutex))

;; Consumer
(defun pop-data (&optional timeout)
  (with-mutex (*lock*)
    (loop until *data*
          do (or (condition-wait *queue* *lock* :timeout timeout)
                 ;; Lock not held, must unwind without touching *data*.
                 (return-from pop-data nil)))
    (pop *data*)))

;; Producer
(defun push-data (data)
  (with-mutex (*lock*)
    (push data *data*)
    (condition-notify *queue*)))
```

`sb-thread:condition-notify` *queue* &optional (*n 1*) [Function]
Notify `n` threads waiting on `queue`.

IMPORTANT: The same mutex that is used in the corresponding `condition-wait` must be held by this thread during this call.

`sb-thread:condition-broadcast` *queue* [Function]
Notify all threads waiting on `queue`.
IMPORTANT: The same mutex that is used in the corresponding `condition-wait` must be held by this thread during this call.

13.7 Barriers

These are based on the Linux kernel barrier design, which is in turn based on the Alpha CPU memory model. They are presently implemented for x86, x86-64, PPC, ARM64, and RISC-V systems, and behave as compiler barriers on all other CPUs.
In addition to explicit use of the `sb-thread:barrier` macro, the following functions and macros also serve as `:memory` barriers:

- `sb-ext:atomic-decf`, `sb-ext:atomic-incf`, `sb-ext:atomic-push`, and `sb-ext:atomic-pop`
- `sb-ext:compare-and-swap`
- `sb-thread:grab-mutex`, `sb-thread:release-mutex`, `sb-thread:with-mutex` and `sb-thread:with-recursive-lock`
- `sb-thread:signal-semaphore`, `sb-thread:try-semaphore` and `sb-thread:wait-on-semaphore`
- `sb-thread:condition-wait`, `sb-thread:condition-notify` and `sb-thread:condition-broadcast`.

`sb-thread:barrier` (*kind*) &body *forms* [Macro]
Insert a barrier in the code stream, preventing some sort of reordering.
`kind` should be one of:

- `:compiler`: Prevent the compiler from reordering memory access across the barrier.
- `:memory`: Prevent the CPU from reordering any memory access across the barrier.
- `:read`: Prevent the CPU from reordering any read access across the barrier.
- `:write`: Prevent the cpu from reordering any write access across the barrier.
- `:data-dependency`: Prevent the cpu from reordering dependent memory reads across the barrier (requiring reads before the barrier to complete before any reads after the barrier that depend on them). This is a weaker form of the `:read` barrier.

`forms` is an implicit `progn`, evaluated before the barrier. `barrier` returns the values of the last form in `forms`.
The file `memory-barriers.txt` in the Linux kernel documentation is highly recommended reading for anyone programming at this level.

13.8 Sessions/Debugging

If the user has multiple views onto the same Lisp image (for example, using multiple terminals, or a windowing system, or network access) they are typically set up as multiple *sessions* such that each view has its own collection of foreground, background, and stopped threads. A thread which wishes to create a new session can use `sb-thread:with-new-session` to remove itself from the current session (which it shares with its parent and siblings) and create a fresh one.

`sb-thread:with-new-session` *args* &body *forms* [Macro]

`sb-thread:make-listener-thread` *tty-name* [Function]
Within a single session, threads arbitrate between themselves for the user's attention. A thread may be in one of three notional states: foreground, background, or stopped. When a background process attempts to print a repl prompt or to enter the debugger, it will stop and print a message saying that it has stopped. The user at his leisure may switch to that thread to find out what it needs. If a background thread enters the debugger, selecting any restart will put it back into the background before it resumes. Arbitration for the input stream is managed by calls to `sb-thread:get-foreground` (which may block) and `sb-thread:release-foreground`.

`sb-thread:get-foreground` [Function]

`sb-thread:release-foreground` &optional *next* [Function]
Background this thread. If `next` is supplied, arrange for it to have the foreground next.

13.9 Foreign threads

Direct calls to `pthread_create(3)` (instead of `sb-thread:make-thread`) create threads that SBCL is not aware of, these are called foreign threads. Currently, it is not possible to run Lisp code in such threads. This means that the Lisp side signal handlers cannot work. The best solution is to start foreign threads with signals blocked, but since third party libraries may create threads, it is not always feasible to do so. As a workaround, upon receiving a signal in a foreign thread, SBCL changes the thread's sigmask to block all signals that it wants to handle and resends the signal to the current process which should land in a thread that does not block it, that is, a Lisp thread.

The resignalling trick cannot work for synchronously triggered signals (`sigsegv` and co), take care not to trigger any. Resignalling for synchronously triggered signals in foreign threads is subject to `--lose-on-corruption`, see Section 3.3.1 [runtime options], page 11.

13.10 Implementation on Linux x86oids

Threading is implemented using pthreads and some Linux specific bits like futexes.

On x86, the per-thread local bindings for special variables is achieved using the `%fs` segment register to point to a per-thread storage area. This may cause interesting results if you link to foreign code that expects threading or creates new threads, and the thread library in question uses %fs in an incompatible way. On x86-64 the r12 register has a similar role.

Queues require the `futex(2)` system call to be available: this is the reason for the NPTL requirement. We test at runtime that this system call exists.

Garbage collection is done with the existing Conservative Generational GC. Allocation is done in small (typically 8k) regions: each thread has its own region so this involves no stopping. However, when a region fills, a lock must be obtained while another is allocated, and when a collection is required, all processes are stopped. This is achieved by sending them signals, which may make for interesting behaviour if they are interrupted in system calls. The streams interface is believed to handle the required system call restarting correctly, but this may be a consideration when making other blocking calls e.g. from foreign library code.

Large amounts of the SBCL library have not been inspected for thread-safety. Some of the obviously unsafe areas have large locks around them, so compilation and fasl loading, for example, cannot be parallelized. Work is ongoing in this area.

A new thread by default is created in the same POSIX process group and session as the thread it was created by. This has an impact on keyboard interrupt handling: pressing your terminal's intr key (typically `Control-C`) will interrupt all processes in the foreground process group, including Lisp threads that SBCL considers to be notionally *background*. This is undesirable, so background threads are set to ignore the `sigint` signal.

`sb-thread:make-listener-thread` in addition to creating a new Lisp session makes a new POSIX session, so that pressing `Control-C` in one window will not interrupt another listener - this has been found to be embarrassing.

14 Timers

SBCL supports a system-wide event scheduler implemented on top of `setitimer(2)` that also works with threads but does not require a separate scheduler thread.

The following example schedules a timer that writes `Hello, world` after two seconds.

```
(schedule-timer (make-timer (lambda ()
                              (write-line "Hello, world")
                              (force-output)))
                2)
```

It should be noted that writing timer functions requires special care, as the dynamic environment in which they run is unpredictable: dynamic variable bindings, locks held, etc, all depend on whatever code was running when the timer fired. The following example should serve as a cautionary tale:

```
(defvar *foo* nil)

(defun show-foo ()
  (format t "~&foo=~S~%" *foo*)
  (force-output t))

(defun demo ()
  (schedule-timer (make-timer #'show-foo) 0.5)
  (schedule-timer (make-timer #'show-foo) 1.5)
  (let ((*foo* t))
    (sleep 1.0))
  (let ((*foo* :surprise!))
    (sleep 2.0)))
```

`sb-ext:timer` [Structure]
Timer type. Do not rely on timers being structs as it may change in future versions.

`sb-ext:make-timer` *function* &key *name* (*thread sb-thread:*current-thread**) [Function]
Create a timer that runs `function` when triggered.

If a `thread` is supplied, `function` is run in that thread. If `thread` is `t`, a new thread is created for `function` each time the timer is triggered. If `thread` is `nil`, `function` is run in an unspecified thread.

When `thread` is not `t`, `sb-thread:interrupt-thread` is used to run `function` and the ordering guarantees of `sb-thread:interrupt-thread` apply. In that case, `function` runs with interrupts disabled but `with-interrupts` is allowed.

`sb-ext:timer-name` *timer* [Function]
Return the name of `timer`.

`sb-ext:timer-scheduled-p` *timer* &key (*delta 0*) [Function]
See if `timer` will still need to be triggered after `delta` seconds from now. For timers with a repeat interval it returns true.

`sb-ext:schedule-timer` *timer time* &key *repeat-interval absolute-p catch-up* [Function]
Schedule `timer` to be triggered at `time`. If `absolute-p` then `time` is universal time, but non-integral values are also allowed, else `time` is measured as the number of seconds from the current time.

If `repeat-interval` is given, `timer` is automatically rescheduled upon expiry.

If `repeat-interval` is non-`nil`, the Boolean `catch-up` controls whether `timer` will "catch up" by repeatedly calling its function without delay in case calls are missed because of a clock discontinuity such as a suspend and resume cycle of the computer. The default is `nil`, i.e. do not catch up.

`sb-ext:unschedule-timer` *timer* [Function]
Cancel `timer`. Once this function returns it is guaranteed that `timer` shall not be triggered again and there are no unfinished triggers.

`sb-ext:list-all-timers` [Function]
Return a list of all timers in the system.

15 Networking

The `sb-bsd-sockets` module provides a thinly disguised BSD socket API for SBCL. Ideas have been stolen from the BSD socket API for C and Graham Barr's `IO::Socket` classes for Perl.

Sockets are represented as CLOS objects, and the API naming conventions attempt to balance between the BSD names and good lisp style.

15.1 Sockets Overview

Most of the functions are modelled on the BSD socket API. BSD sockets are widely supported, portably (by Unix standards, at least) available on a variety of systems, and documented. There are some differences in approach where we have taken advantage of some of the more useful features of Common Lisp – briefly:

- Where the C API would typically return -1 and set `errno`, `sb-bsd-sockets` signals an error. All the errors are subclasses of `sb-bsd-sockets:socket-error` and generally correspond one for one with possible `errno` values.
- We use multiple return values in many places where the C API would use pass-by-reference values.
- We can often avoid supplying an explicit length argument to functions because we already know how long the argument is.
- IP addresses and ports are represented in slightly friendlier fashion than "network-endian integers".

15.2 General Sockets

`sb-bsd-sockets:socket` [Class]
Common superclass of all sockets, not meant to be directly instantiated.

`sb-bsd-sockets:socket-bind` *socket* &rest *address* [Function]
Bind `socket` to `address`, which may vary according to socket family. For the INET family, pass `address` and `port` as two arguments; for local address family sockets, pass the filename string. See also `bind(2)`.

`sb-bsd-sockets:socket-accept` *socket* [Function]
Perform the `accept(2)` call, returning a newly-created connected socket and the peer address as multiple values

`sb-bsd-sockets:socket-connect` *socket* &rest *address* [Function]
Perform the `connect(2)` call to connect `socket` to a remote `peer`. No useful return value.

`sb-bsd-sockets:socket-peername` *socket* [Function]
Return `socket`'s peer; depending on the address family this may return multiple values

`sb-bsd-sockets:socket-name` *socket* [Function]
Return the address (as vector of bytes) and port that `socket` is bound to, as multiple values.

`sb-bsd-sockets:socket-receive` *socket buffer length* &key *oob peek waitall dontwait element-type* [Function]
Read `length` octets from `socket` into `buffer` (or a freshly-consed buffer if `nil`), using `recvfrom(2)`. If `length` is `nil`, the length of `buffer` is used, so at least one of these two arguments must be non-`nil`. If `buffer` is supplied, it had better be of an element type one octet wide. Returns the buffer, its length, and the address of the peer that sent it, as multiple values. On datagram sockets, sets `MSG_TRUNC` so that the actual packet length is returned even if the buffer was too small.

`sb-bsd-sockets:socket-send` *socket buffer length* &key *address external-format oob eor dontroute dontwait nosignal confirm more* [Function]
Send `length` octets from `buffer` into `socket`, using `sendto(2)`. If `buffer` is a string, it will converted to octets according to `external-format`. If `length` is `nil`, the length of the octet buffer is used. The format of `address` depends on the socket type (for example for INET domain sockets it would be a list of an IP address and a port). If no socket address is provided, `send(2)` will be called instead. Returns the number of octets written.

`sb-bsd-sockets:socket-listen` *socket backlog* [Function]
Mark `socket` as willing to accept incoming connections. The integer `backlog` defines the maximum length that the queue of pending connections may grow to before new connection attempts are refused. See also `listen(2)`.

`sb-bsd-sockets:socket-open-p` *socket* [Function]
Return true if `socket` is open; otherwise, return false.

`sb-bsd-sockets:socket-close` *socket* &key *abort* [Function]
Close `socket`, unless it was already closed.
If `socket-make-stream` has been called, calls `close` using `abort` on that stream. Otherwise closes the socket file descriptor using `close(2)`.

`sb-bsd-sockets:socket-shutdown` *socket* &key *direction* [Function]
Indicate that no communication in `direction` will be performed on `socket`.
`direction` has to be one of `:input`, `:output` or `:io`.
After a shutdown, no input and/or output of the indicated `direction` can be performed on `socket`.

`sb-bsd-sockets:socket-make-stream` *socket* &key *input output element-type external-format buffering timeout auto-close serve-events* [Function]
Find or create a `stream` that can be used for IO on `socket` (which must be connected). Specify whether the stream is for `input`, `output`, or both (it is an error to specify neither).
`element-type` and `external-format` are as per `open`.
`timeout` specifies a read timeout for the stream.

`sb-bsd-sockets:socket-make-stream` (*socket sb-bsd-sockets:socket*) &key *input output* (*element-type 'character*) (*buffering :full*) (*external-format :default*) *timeout auto-close serve-events* [Method]
Default method for `socket` objects.
`element-type` defaults to `character`, to construct a bivalent stream, capable of both binary and character IO use `:default`.
Acceptable values for `buffering` are `:full`, `:line` and `:none`, default is `:full`, i.e. output is buffered till it is explicitly flushed using `close` or `finish-output`. (`force-output` forces some output to be flushed: to ensure all buffered output is flushed use `finish-output`.)
Streams have no `timeout` by default. If one is provided, it is the number of seconds the system will at most wait for input to appear on the socket stream when trying to read from it.
If `auto-close` is true, the underlying OS socket is automatically closed after the stream and the socket have been garbage collected. Default is false.
If `serve-events` is true, blocking IO on the socket will dispatch to the recursive event loop. Default is false.
The stream for `socket` will be cached, and a second invocation of this method will return the same stream. This may lead to oddities if this function is invoked with inconsistent arguments (e.g. one might request an input stream and get an output stream in response).

`sb-bsd-sockets:socket-error` *where* &optional *errno* [Function]
Signal an appropriate error for syscall `where` and `errno`.
`where` should be a string naming the failed function.
When supplied, `errno` should be the UNIX error number associated to the failed call. The default behavior is to use the current value of the errno variable.

`sb-bsd-sockets:non-blocking-mode` *socket* [Function]
Is `socket` in non-blocking mode?

15.3 Socket Options

A subset of socket options are supported, using a fairly general framework which should make it simple to add more as required – see `SYS:CONTRIB;SB-BSD-SOCKETS:SOCKOPT.LISP` for details. The name mapping from C is fairly straightforward: `SO_RCVLOWAT` becomes `sb-bsd-sockets:sockopt-receive-low-water` and `(setf sb-bsd-sockets:sockopt-receive-low-water)`.

`sb-bsd-sockets:sockopt-reuse-address` *socket* [Function]
Return the value of the SO-REUSEADDR socket option for `socket`. This can also be updated with `setf`.

`sb-bsd-sockets:sockopt-keep-alive` *socket* [Function]
Return the value of the SO-KEEPALIVE socket option for `socket`. This can also be updated with `setf`.

`sb-bsd-sockets:sockopt-oob-inline` *socket* [Function]
Return the value of the SO-OOBINLINE socket option for `socket`. This can also be updated with `setf`.

`sb-bsd-sockets:sockopt-bsd-compatible` *socket* [Function]
Return the value of the SO-BSDCOMPAT socket option for `socket`. This can also be updated with `setf`. Available only on Linux.

`sb-bsd-sockets:sockopt-pass-credentials` *socket* [Function]
Return the value of the SO-PASSCRED socket option for `socket`. This can also be updated with `setf`. Available only on Linux.

`sb-bsd-sockets:sockopt-debug` *socket* [Function]
Return the value of the SO-DEBUG socket option for `socket`. This can also be updated with `setf`.

`sb-bsd-sockets:sockopt-dont-route` *socket* [Function]
Return the value of the SO-DONTROUTE socket option for `socket`. This can also be updated with `setf`.

`sb-bsd-sockets:sockopt-broadcast` *socket* [Function]
Return the value of the SO-BROADCAST socket option for `socket`. This can also be updated with `setf`.

`sb-bsd-sockets:sockopt-tcp-nodelay` *socket* [Function]
Return the value of the TCP-NODELAY socket option for `socket`. This can also be updated with `setf`.

15.4 INET Domain Sockets

The TCP and UDP sockets that you know and love. Some representation issues:

- IPv4 Internet addresses are represented by vectors of `(unsigned-byte 8)` (e.g. `#(127 0 0 1)`). Ports are just integers. No conversion between network- and host-order data is needed from the user of this package.
- IPv6 Internet addresses are represented by length 16 vectors of `(unsigned-byte 8)` (e.g. `#(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1)`. Ports are just integers. As for IPv4 addresses, no conversion between network- and host-order data is needed from the user of this package.
- Socket addresses are represented by the two values for address and port, so for example, `(sb-bsd-sockets:socket-connect socket #(192 168 1 1) 80)` for IPv4 and `(sb-bsd-sockets:socket-connect socket #(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1) 80)` for IPv6.

`sb-bsd-sockets:inet-socket` [Class]
Class representing TCP and UDP over IPv4 sockets.
Examples:

```
(make-instance 'sb-bsd-sockets:inet-socket :type :stream :protocol :tcp)

(make-instance 'sb-bsd-sockets:inet-socket :type :datagram :protocol :udp)
```

`sb-bsd-sockets:inet6-socket` [Class]
Class representing TCP and UDP over IPv6 sockets.
Examples:

```
(make-instance 'sb-bsd-sockets:inet6-socket :type :stream :protocol :tcp)

(make-instance 'sb-bsd-sockets:inet6-socket :type :datagram :protocol :udp)
```

`sb-bsd-sockets:make-inet-address` *dotted-quads* [Function]
Return a vector of octets given a string `dotted-quads` in the format `"127.0.0.1"`. Signals an error if the string is malformed.

`sb-bsd-sockets:make-inet6-address` *colon-separated-integers* [Function]
Return a vector of octets given a string representation of an IPv6 address `colon-separated-integers`. Signal an error if the string is malformed.

`sb-bsd-sockets:get-protocol-by-name` *name* [Function]
Given a protocol name, return the protocol number, the protocol name, and a list of protocol aliases.

15.5 Local Domain Sockets

Local domain (`AF_LOCAL`) sockets are also known as Unix-domain sockets but were renamed by POSIX presumably on the basis that they may be available on other systems too.

A local socket address is a string, which is used to create a node in the local filesystem. This means of course that they cannot be used across a network.

`sb-bsd-sockets:local-socket` [Class]
Class representing local domain (`AF_LOCAL`) sockets, also known as Unix-domain sockets.

A local abstract socket address is also a string the scope of which is the local machine. However, in contrast to a local socket address, there is no corresponding filesystem node.

`sb-bsd-sockets:local-abstract-socket` [Class]
Class representing local domain (`AF_LOCAL`) sockets with addresses in the abstract namespace.

15.6 Name Service

Presently name service is implemented by calling out to the `getaddrinfo(3)` and `gethostinfo(3)`, or to `gethostbyname(3)` and `gethostbyaddr(3)` on platforms where the preferred functions are not available. The exact details of the name resolving process (for example the choice of whether DNS or a hosts file is used for lookup) are platform dependent.

`sb-bsd-sockets:host-ent` [Class]
This class represents the results of an address lookup.

`sb-bsd-sockets:get-host-by-name` *host-name* [Function]
Returns a `host-ent` instance for `host-name` or signals a `name-service-error`.

Another `host-ent` instance containing zero, one or more IPv6 addresses may be returned as a second return value.

`host-name` may also be an IP address in dotted quad notation or some other weird stuff - see getaddrinfo(3) for the details.

`sb-bsd-sockets:get-host-by-address` *address* [Function]
Returns a `host-ent` instance for `address`, which should be a vector of (integer 0 255) with 4 elements in case of an IPv4 address and 16 elements in case of an IPv6 address, or signals a `name-service-error`. See gethostbyaddr(3) for details.

`sb-bsd-sockets:host-ent-address` *host-ent* [Function]
Return some valid address for `host-ent`.

16 Profiling

SBCL includes both a deterministic profiler, that can collect statistics on individual functions, and a more "modern", statistical profiler.
Inlined functions do not appear in the results reported by either.

16.1 Deterministic Profiler

The package `sb-profile` provides a classic, per-function-call profiler.

> **Warning**: When profiling code executed by multiple threads in parallel, the consing attributed to each function is inaccurate.

`sb-profile:profile` &rest *names* [Macro]
If no names are supplied, return the list of profiled functions.
If names are supplied, wrap profiling code around the named functions. As in `trace`, the names are not evaluated. A symbol names a function. A string names all the functions named by symbols in the named package. If a function is already profiled, then unprofile and reprofile (useful to notice function redefinition.) If a name is undefined, then we give a warning and ignore it. See also `unprofile`, `report` and `reset`.

`sb-profile:unprofile` &rest *names* [Macro]
Unwrap any profiling code around the named functions, or if no names are given, unprofile all profiled functions. A symbol names a function. A string names all the functions named by symbols in the named package. `names` defaults to the list of names of all currently profiled functions.

`sb-profile:report` &key *limit* (*print-no-call-list* *t*) [Function]
Report results from profiling. The results are approximately adjusted for profiling overhead. The compensation may be rather inaccurate when bignums are involved in runtime calculation, as in a very-long-running Lisp process.
If `limit` is set to an integer, only the top `limit` results are reported. If `print-no-call-list` is `t` (the default) then a list of uncalled profiled functions are listed.

`sb-profile:reset` [Function]
Reset the counters for all profiled functions.

16.2 Statistical Profiler

The `sb-sprof` module, loadable by

```
(require :sb-sprof)
```

provides an alternate profiler which works by taking samples of the program execution at regular intervals, instead of instrumenting functions as `sb-profile:profile` does. You might find `sb-sprof` more useful than the deterministic profiler when profiling functions in the `common-lisp` package, SBCL internals, or code where the instrumenting overhead is excessive.
Additionally `sb-sprof` includes a limited deterministic profiler which can be used for reporting the amounts of calls to some functions during
Example usage:

```
(in-package :cl-user)

(require :sb-sprof)

(declaim (optimize speed))

(defun cpu-test-inner (a i)
  (logxor a
          (* i 5)
          (+ a i)))

(defun cpu-test (n)
  (let ((a 0))
```

```
    (dotimes (i (expt 2 n) a)
      (setf a (cpu-test-inner a i)))))

;;;; CPU profiling

;;; Take up to 1000 samples of running (CPU-TEST 26), and give a flat
;;; table report at the end. Profiling will end one the body has been
;;; evaluated once, whether or not 1000 samples have been taken.
(sb-sprof:with-profiling (:max-samples 1000
                          :report :flat
                          :loop nil)
  (cpu-test 26))

;;; Record call counts for functions defined on symbols in the CL-USER
;;; package.
(sb-sprof:profile-call-counts "CL-USER")

;;; Take 1000 samples of running (CPU-TEST 24), and give a flat
;;; table report at the end. The body will be re-evaluated in a loop
;;; until 1000 samples have been taken. A sample count will be printed
;;; after each iteration.
(sb-sprof:with-profiling (:max-samples 1000
                          :report :flat
                          :loop t
                          :show-progress t)
  (cpu-test 24))

;;;; Allocation profiling

(defun foo (&rest args)
  (mapcar (lambda (x) (float x 1d0)) args))

(defun bar (n)
  (declare (fixnum n))
  (apply #'foo (loop repeat n collect n)))

(sb-sprof:with-profiling (:max-samples 10000
                          :mode :alloc
                          :report :flat)
  (bar 1000))
```

Output:

The flat report format will show a table of all functions that the profiler encountered on the call stack during sampling, ordered by the number of samples taken while executing that function.

```
            Self        Total        Cumul
  Nr  Count     %  Count     %  Count     %    Calls  Function
------------------------------------------------------------------------
   1     69  24.4     97  34.3     69  24.4 67108864  CPU-TEST-INNER
   2     64  22.6     64  22.6    133  47.0        -  SB-VM::GENERIC-+
   3     39  13.8    256  90.5    172  60.8        1  CPU-TEST
   4     31  11.0     31  11.0    203  71.7        -  SB-KERNEL:TWO-ARG-XOR
```

For each function, the table will show three absolute and relative sample counts. The `Self` column shows samples taken while directly executing that function. The `Total` column shows samples taken while executing that function or functions called from it (sampled to a platform-specific depth). The `Cumul` column shows the sum of all `Self` columns up to and including that line in the table.

Additionally the `Calls` column will record the amount of calls that were made to the function during the profiling run. This value will only be reported for functions that have been explicitly marked for call counting with `sb-sprof:profile-call-counts`.

The profiler also hooks into the disassembler such that instructions which have been sampled are annotated with their relative frequency of sampling. This information is not stored across different sampling runs.

```
;      6CF:       702E            JO L4              ; 6/242 samples
;      6D1:       D1E3            SHL EBX, 1
;      6D3:       702A            JO L4
;      6D5: L2:   F6C303          TEST BL, 3         ; 2/242 samples
;      6D8:       756D            JNE L8
;      6DA:       8BC3            MOV EAX, EBX       ; 5/242 samples
;      6DC: L3:   83F900          CMP ECX, 0         ; 4/242 samples
```

Platform support

Allocation profiling is only supported on SBCL builds that use the generational GC. Tracking of call stacks at a depth of more than two levels is only supported on x86 and x86-64.

Macros

`sb-sprof:with-profiling` (&key (*sample-interval 'sb-sprof:*sample-interval**) *alloc-interval* (*max-samples 'sb-sprof:*max-samples**) *reset* (*mode 'sb-sprof:*sampling-mode**) *loop max-depth show-progress* (*threads :all*) *report*) &body *body* [Macro]

Evaluate `body` with statistical profiling turned on. If `loop` is true, loop around the `body` until a sufficient number of samples has been collected. Returns the values from the last evaluation of `body`.

The following keyword args are recognized:

- `:sample-interval <n>`

  Take a sample every `<n>` seconds. Default is `*sample-interval*`.

- `:mode <mode>`

  If `:cpu`, run the profiler in CPU profiling mode. If `:alloc`, run the profiler in allocation profiling mode. If `:time`, run the profiler in wallclock profiling mode.

- `:max-samples <max>`

  If `:loop` is `nil` (the default), collect no more than `<max>` samples. If `:loop` is `t`, repeat evaluating body until `<max>` samples are taken. Default is `*max-samples*`.

- `:report <type>`

  If specified, call `report` with `:type <type>` at the end.

- `:reset <bool>`

  If true, call `reset` at the beginning.

- `:threads <list-form>`

  Form that evaluates to the list threads to profile, or `:all` to indicate that all threads should be profiled. Defaults to all threads.

  `:threads` has no effect on call-counting at the moment.

  On some platforms (e.g. Darwin) the signals used by the profiler are not properly delivered to threads in proportion to their CPU usage when doing `:cpu` profiling. If you see empty call graphs, or are obviously missing several samples from certain threads, you may be falling afoul of this. In this case using `:mode :time` is likely to work better.

- `:loop <bool>`

  If false (the default), evaluate `body` only once. If true repeatedly evaluate `body`.

`sb-sprof:with-sampling` (&optional (*on t*)) &body *body* [Macro]

Evaluate body with statistical sampling turned on or off in the current thread.

Functions

`sb-sprof:map-traces` *function samples* [Function]

Call `function` on each trace in `samples`

The signature of `function` must be compatible with (thread trace).

`function` is called once for each trace where `thread` is the `sb-thread:thread` instance that was sampled to produce `trace`, and `trace` is an opaque object to be passed to `map-trace-pc-locs`.

EXPERIMENTAL: Interface subject to change.

`sb-sprof:sample-pc` *info pc-or-offset* [Function]
Extract and return program counter from `info` and `pc-or-offset`.
Can be applied to the arguments passed by `map-trace-pc-locs` and `map-all-pc-locs`.
EXPERIMENTAL: Interface subject to change.

`sb-sprof:report` &key (*type :graph*) *max min-percent call-graph* (*sort-by sb-sprof:*report-sort-by**) (*sort-order sb-sprof:*report-sort-order**) (*stream *standard-output**) *show-progress* [Function]
Report statistical profiling results. The following keyword args are recognized:

- `:type <type>`
  Specifies the type of report to generate. If `:flat`, show flat report, if `:graph` show a call graph and a flat report. If nil, don't print out a report.
- `:stream <stream>`
  Specify a stream to print the report on. Default is `*standard-output*`.
- `:max <max>`
  Don't show more than `<max>` entries in the flat report.
- `:min-percent <min-percent>`
  Don't show functions taking less than `<min-percent>` of the total time in the flat report.
- `:sort-by <column>`
  If `:samples`, sort flat report by number of samples taken. If `:cumulative-samples`, sort flat report by cumulative number of samples taken (shows how much time each function spent on stack.) Default is `*report-sort-by*`.
- `:sort-order <order>`
  If `:descending`, sort flat report in descending order. If `:ascending`, sort flat report in ascending order. Default is `*report-sort-order*`.
- `:show-progress <bool>`
  If true, print progress messages while generating the call graph.
- `:call-graph <graph>`
  Print a report from `<graph>` instead of the latest profiling results.

Value of this function is a `call-graph` object representing the resulting call-graph, or `nil` if there are no samples (e.g. right after calling `reset`.)
Profiling is stopped before the call graph is generated.

`sb-sprof:reset` [Function]
Reset the profiler.

`sb-sprof:start-profiling` &key (*max-samples sb-sprof:*max-samples**) (*mode sb-sprof:*sampling-mode**) (*sample-interval sb-sprof:*sample-interval**) *alloc-interval max-depth* (*threads :all*) [Function]
Start profiling statistically in the current thread if not already profiling. The following keyword args are recognized:

- `:sample-interval <n>`
  Take a sample every `<n>` seconds. Default is `*sample-interval*`.
- `:mode <mode>`
  If `:cpu`, run the profiler in CPU profiling mode. If `:alloc`, run the profiler in allocation profiling mode. If `:time`, run the profiler in wallclock profiling mode.
- `:max-samples <max>`
  Maximum number of stack traces to collect. Default is `*max-samples*`.
- `:threads <list>`
  List threads to profile, or `:all` to indicate that all threads should be profiled. Defaults to `:all`.
  `:threads` has no effect on call-counting at the moment.
  On some platforms (e.g. Darwin) the signals used by the profiler are not properly delivered to threads in proportion to their CPU usage when doing `:cpu` profiling. If you see empty call graphs, or are obviously missing several samples from certain threads, you may be falling afoul of this.

`sb-sprof:stop-profiling` [Function]
Stop profiling if profiling.

`sb-sprof:profile-call-counts` &rest *names* [Function]
Mark the functions named by `names` as being subject to call counting during statistical profiling. If a string is used as a name, it will be interpreted as a package name. In this case call counting will be done for all functions with names like `x` or `(setf x)`, where `x` is a symbol with the package as its home package.

`sb-sprof:unprofile-call-counts` [Function]
Clear all call counting information. Call counting will be done for no functions during statistical profiling.

Variables

`sb-sprof:*max-samples*` [Variable]
Default maximum number of stack traces collected.

`sb-sprof:*sample-interval*` [Variable]
Default number of seconds between samples.

Credits

`sb-sprof` is an SBCL port, with enhancements, of Gerd Moellmann's statistical profiler for CMUCL.

17 Contributed Modules

SBCL comes with a number of modules that are not part of the core system. These are loaded via `(require :<modulename>)` (see Section 7.12 [customization hooks for users], page 79). This section contains documentation (or pointers to documentation) for some of the contributed modules.

17.1 sb-aclrepl

The `sb-aclrepl` module offers an Allegro CL-style Read-Eval-Print Loop for SBCL, with integrated inspector. Adding a debugger interface is planned.

Allegro CL is a registered trademark of Franz Inc.

17.1.1 Usage

To start `sb-aclrepl` as your read-eval-print loop, put the form

```
(require 'sb-aclrepl)
```

in your `~/.sbclrc`, one of your Section 3.4 [initialization files], page 13.

17.1.2 Customization

The following customization variables are available:

`sb-aclrepl:*command-char*` [Variable]
Prefix character for a top-level command

`sb-aclrepl:*prompt*` [Variable]
The current prompt string or formatter function.

`sb-aclrepl:*exit-on-eof*` [Variable]
If `t`, then exit when the EOF character is entered.

`sb-aclrepl:*use-short-package-name*` [Variable]
When `t`, use the shortest package nickname in a prompt.

`sb-aclrepl:*max-history*` [Variable]
Maximum number of history commands to remember

17.1.3 Example Initialization

Here's a longer example of a `~/.sbclrc` file that shows off some of the features of sb-aclrepl:

```
(ignore-errors (require 'sb-aclrepl))

(when (find-package 'sb-aclrepl)
  (push :aclrepl cl:*features*))
#+aclrepl
(progn
  (setq sb-aclrepl:*max-history* 100)
  (setf (sb-aclrepl:alias "asdc")
        #'(lambda (sys) (asdf:operate 'asdf:compile-op sys)))
  (sb-aclrepl:alias "l" (sys) (asdf:operate 'asdf:load-op sys))
  (sb-aclrepl:alias "t" (sys) (asdf:operate 'asdf:test-op sys))
  ;; The 1 below means that two characaters ("up") are required
  (sb-aclrepl:alias ("up" 1 "Use package") (package) (use-package package))
  ;; The 0 below means only the first letter ("r") is required,
  ;; such as ":r base64"
  (sb-aclrepl:alias ("require" 0 "Require module") (sys) (require sys))
  (setq cl:*features* (delete :aclrepl cl:*features*)))
```

Questions, comments, or bug reports should be sent to Kevin Rosenberg (kevin@rosenberg.net).

17.2 sb-concurrency

Additional data structures, synchronization primitives and tools for concurrent programming. Similiar to Java's `java.util.concurrent` package.

17.2.1 Queue

`sb-concurrency:queue` is a lock-free, thread-safe FIFO queue datatype.

The implementation is based on *An Optimistic Approach to Lock-Free FIFO Queues* by Edya Ladan-Mozes and Nir Shavit.

Before SBCL 1.0.38, this implementation resided in its own contrib (see Section 17.9 [sb queue], page 153), which is still provided for backwards-compatibility, but which has since been deprecated.

`sb-concurrency:queue` [Structure]
Lock-free thread safe FIFO queue.

Use `enqueue` to add objects to the queue, and `dequeue` to remove them.

`sb-concurrency:dequeue` *queue* [Function]
Retrieves the oldest value in `queue` and returns it as the primary value, and `t` as secondary value. If the queue is empty, returns `nil` as both primary and secondary value.

`sb-concurrency:enqueue` *value queue* [Function]
Adds `value` to the end of `queue`. Returns `value`.

`sb-concurrency:list-queue-contents` *queue* [Function]
Returns the contents of `queue` as a list without removing them from the `queue`. Mainly useful for manual examination of queue state, as the list may be out of date by the time it is returned, and concurrent dequeue operations may in the worse case force the queue-traversal to be restarted several times.

`sb-concurrency:make-queue` &key *name initial-contents* [Function]
Returns a new `queue` with `name` and contents of the `initial-contents` sequence enqueued.

`sb-concurrency:queue-count` *queue* [Function]
Returns the number of objects in `queue`. Mainly useful for manual examination of queue state, and in `print-object` methods: inefficient as it must walk the entire queue.

`sb-concurrency:queue-empty-p` *queue* [Function]
Returns `t` if `queue` is empty, `nil` otherwise.

`sb-concurrency:queue-name` *instance* [Function]
Name of a `queue`. Can be assigned to using `setf`. Queue names can be arbitrary printable objects, and need not be unique.

`sb-concurrency:queuep` *object* [Function]
Returns true if argument is a `queue`, `nil` otherwise.

17.2.2 Mailbox (lock-free)

`sb-concurrency:mailbox` is a lock-free message queue where one or multiple ends can send messages to one or multiple receivers. The difference to Section 17.2.1 [sb concurrency queue], page 139, is that the receiving end may block until a message arrives.

Built on top of the Section 17.2.1 [sb concurrency queue], page 139, implementation.

`sb-concurrency:mailbox` [Structure]
Mailbox aka message queue.

`send-message` adds a message to the mailbox, `receive-message` waits till a message becomes available, whereas `receive-message-no-hang` is a non-blocking variant, and `receive-pending-messages` empties the entire mailbox in one go.

Messages can be arbitrary objects.

`sb-concurrency:list-mailbox-messages` *mailbox* [Function]
Returns a fresh list containing all the messages in `mailbox`. Does not remove messages from the mailbox.

`sb-concurrency:mailbox-count` *mailbox* [Function]
Returns the number of messages currently in `mailbox`.

`sb-concurrency:mailbox-empty-p` *mailbox* [Function]
Returns true if `mailbox` is currently empty, `nil` otherwise.

`sb-concurrency:mailbox-name` *instance* [Function]
Name of a `mailbox`. `setf`able.

`sb-concurrency:mailboxp` *object* [Function]
Returns true if argument is a `mailbox`, `nil` otherwise.

`sb-concurrency:make-mailbox` &key *name initial-contents* [Function]
Returns a new `mailbox` with messages in `initial-contents` enqueued.

`sb-concurrency:receive-message` *mailbox* &key *timeout* [Function]
Removes the oldest message from `mailbox` and returns it as the primary value, and a secondary value of `t`. If `mailbox` is empty waits until a message arrives.
If `timeout` is provided, and no message arrives within the specified interval, returns primary and secondary value of `nil`.

`sb-concurrency:receive-message-no-hang` *mailbox* [Function]
The non-blocking variant of `receive-message`. Returns two values, the message removed from `mailbox`, and a flag specifying whether a message could be received.

`sb-concurrency:receive-pending-messages` *mailbox* &optional *n* [Function]
Removes and returns all (or at most `n`) currently pending messages from `mailbox`, or returns `nil` if no messages are pending.

> *Note*: Concurrent threads may be snarfing messages during the run of this function, so even `x` and `y` appearing right next to each other in the result does not necessarily mean that `y` was the message sent right after `x`.

`sb-concurrency:send-message` *mailbox message* [Function]
Adds a `message` to `mailbox`. Message can be any object.

17.2.3 Gates

`sb-concurrency:gate` is a synchronization object suitable for when multiple threads must wait for a single event before proceeding.

`sb-concurrency:gate` [Structure]
`gate` type. Gates are synchronization constructs suitable for making multiple threads wait for single event before proceeding.
Use `wait-on-gate` to wait for a gate to open, `open-gate` to open one, and `close-gate` to close an open gate. `gate-open-p` can be used to test the state of a gate without blocking.

`sb-concurrency:close-gate` *gate* [Function]
Closes `gate`. Returns `t` if the gate was previously open, and `nil` if the gate was already closed.

`sb-concurrency:gate-name` *instance* [Function]
Name of a `gate`. `setf`able.

`sb-concurrency:gate-open-p` *gate* [Function]
Returns true if `gate` is open.

`sb-concurrency:gatep` *object* [Function]
Returns true if the argument is a `gate`.

`sb-concurrency:make-gate` &key *name open* [Function]
Makes a new gate. Gate will be initially open if `open` is true, and closed if `open` is `nil` (the default.) `name`, if provided, is the name of the gate, used when printing the gate.

`sb-concurrency:open-gate` *gate* [Function]
Opens `gate`. Returns `t` if the gate was previously closed, and `nil` if the gate was already open.

`sb-concurrency:wait-on-gate` *gate* &key *timeout* [Function]
Waits for `gate` to open, or `timeout` seconds to pass. Returns `t` if the gate was opened in time, and `nil` otherwise.

17.2.4 Frlocks, aka Fast Read Locks

`sb-concurrency:frlock` [Structure]

FRlock, aka Fast Read Lock.

Fast Read Locks allow multiple readers and one potential writer to operate in parallel while providing for consistency for readers and mutual exclusion for writers.

Readers gain entry to protected regions without waiting, but need to retry if a writer operated inside the region while they were reading. This makes frlocks very efficient when readers are much more common than writers.

FRlocks are *not* suitable when it is not safe at all for readers and writers to operate on the same data in parallel: they provide consistency, not exclusion between readers and writers. Hence using an frlock to e.g. protect an SBCL hash-table is unsafe. If multiple readers operating in parallel with a writer would be safe but inconsistent without a lock, frlocks are suitable.

The recommended interface to use is `frlock-read` and `frlock-write`, but those needing it can also use a lower-level interface.

Example:

```
;; Values returned by FOO are always consistent so that
;; the third value is the sum of the two first ones.
(let ((a 0)
      (b 0)
      (c 0)
      (lk (make-frlock)))
  (defun foo ()
     (frlock-read (lk) a b c))
  (defun bar (x y)
     (frlock-write (lk)
       (setf a x
             b y
             c (+ x y)))))
```

`sb-concurrency:frlock-read` (*frlock*) &body *value-forms* [Macro]

Evaluates `value-forms` under `frlock` till it obtains a consistent set, and returns that as multiple values.

`sb-concurrency:frlock-write` (*frlock* &key (*wait-p* *t*) *timeout*) &body *body* [Macro]

Executes `body` while holding `frlock` for writing.

`sb-concurrency:make-frlock` &key *name* [Function]

Returns a new `frlock` with `name`.

`sb-concurrency:frlock-name` *instance* [Function]

Name of an `frlock`. `setf`able.

`sb-concurrency:frlock-read-begin` *frlock* [Function]

Start a read sequence on `frlock`. Returns a read-token and an epoch to be validated later.

Using `frlock-read` instead is recommended.

`sb-concurrency:frlock-read-end` *frlock* [Function]

Ends a read sequence on `frlock`. Returns a token and an epoch. If the token and epoch are `eql` to the read-token and epoch returned by `frlock-read-begin`, the values read under the `frlock` are consistent and can be used: if the values differ, the values are inconsistent and the read must be restated.

Using `frlock-read` instead is recommended.

Example:

```
(multiple-value-bind (t0 e0) (frlock-read-begin *fr*)
  (let ((a (get-a))
        (b (get-b)))
    (multiple-value-bind (t1 e1) (frlock-read-end *fr*)
      (if (and (eql t0 t1) (eql e0 e1))
          (list :a a :b b)
          :aborted))))
```

`sb-concurrency:grab-frlock-write-lock` *frlock* &key (*wait-p t*) *timeout* [Function]
Acquires `frlock` for writing, invalidating existing and future read-tokens for the duration. Returns `t` on success, and `nil` if the lock wasn't acquired due to e.g. a timeout. Using `frlock-write` instead is recommended.

`sb-concurrency:release-frlock-write-lock` *frlock* [Function]
Releases `frlock` after writing, allowing valid read-tokens to be acquired again. Signals an error if the current thread doesn't hold `frlock` for writing. Using `frlock-write` instead is recommended.

17.3 sb-cover

The `sb-cover` module provides a code coverage tool for SBCL. The tool has support for expression coverage, and for some branch coverage. Coverage reports are only generated for code compiled using `compile-file` with the value of the `sb-cover:store-coverage-data` optimization quality set to 3.
As of SBCL 1.0.6, `sb-cover` is still experimental, and the interfaces documented here might change in later versions.
How to use it:

```
;;; Load SB-COVER
(require :sb-cover)

;;; Turn on generation of code coverage instrumentation in the compiler
(declaim (optimize sb-cover:store-coverage-data))

;;; Load some code, ensuring that it's recompiled with the new optimization
;;; policy.
(asdf:oos 'asdf:load-op :cl-ppcre-test :force t)

;;; Run the test suite.
(cl-ppcre-test:test)

;;; Produce a coverage report
(sb-cover:report "/tmp/report/")

;;; Turn off instrumentation
(declaim (optimize (sb-cover:store-coverage-data 0)))
```

`sb-cover:report` *directory* &key (*form-mode :whole*) (*if-matches 'identity*) (*external-format :default*) [Function]
Print a code coverage report of all instrumented files into `directory`. If `directory` does not exist, it will be created. The main report will be printed to the file cover-index.html. The external format of the source files can be specified with the `external-format` parameter.
If the keyword argument `:form-mode` has the value `:car`, the annotations in the coverage report will be placed on the `car`s of any cons-forms, while if it has the value `:whole` the whole form will be annotated (the default). The former mode shows explicitly which forms were instrumented, while the latter mode is generally easier to read.
The keyword argument `if-matches` should be a designator for a function of one argument, called for the namestring of each file with code coverage info. If it returns true, the file's info is included in the report, otherwise ignored. The default value is `cl:identity`.

`sb-cover:reset-coverage` &optional *object* [Function]
Reset all coverage data back to the `Not executed` state.

`sb-cover:clear-coverage` [Function]
Clear all files from the coverage database. The files will be re-entered into the database when the fasl files (produced by compiling `store-coverage-data` optimization policy set to 3) are loaded again into the image.

`sb-cover:save-coverage` [Function]
Returns an opaque representation of the current code coverage state. The only operation that may be done on the state is passing it to `restore-coverage`. The representation is guaranteed to be readably

printable. A representation that has been printed and read back will work identically in `restore-coverage`.

`sb-cover:save-coverage-in-file` *pathname* [Function]
Call `save-coverage` and write the results of that operation into the file designated by `pathname`.

`sb-cover:restore-coverage` *coverage-state* [Function]
Restore the code coverage data back to an earlier state produced by `save-coverage`.

`sb-cover:restore-coverage-from-file` *pathname* [Function]
`read` the contents of the file designated by `pathname` and pass the result to `restore-coverage`.

`sb-cover:merge-coverage` *coverage-state* [Function]
Merge the code coverage data to include covered code from an earlier state produced by `save-coverage`.

`sb-cover:merge-coverage-from-file` *pathname* [Function]
`read` the contents of the file designated by `pathname` and pass the result to `merge-coverage`.

17.4 sb-grovel

The `sb-grovel` module helps in generation of foreign function interfaces. It aids in extracting constants' values from the C compiler and in generating sb-alien structure and union types, Section 9.2.1 [defining foreign types], page 91.

The ASDF (`http://www.cliki.net/ASDF`) component type GROVEL-CONSTANTS-FILE has its `asdf:perform` operation defined to write out a C source file, compile it, and run it. The output from this program is Lisp, which is then itself compiled and loaded.

`sb-grovel` is used in a few contributed modules, and it is currently compatible only to SBCL. However, if you want to use it, here are a few directions.

17.4.1 Using sb-grovel in Your Own ASDF System

- Create a Lisp package for the foreign constants/functions to go

into.

- Make your system depend on the `sb-grovel` system.
- Create a grovel-constants data file – for an example, see `example-constants.lisp` in the `contrib/sb-grovel/` directory in the SBCL source distribution.
- Add it as a component in your system. For example:

```
(eval-when (:compile-toplevel :load-toplevel :execute)
  (require :sb-grovel))

(defpackage :example-package.system
            (:use :cl :asdf :sb-grovel :sb-alien))

(in-package :example-package.system)

(defsystem example-system
    :depends-on (sb-grovel)
    :components
    ((:module "sbcl"
              :components
              ((:file "defpackage")
               (grovel-constants-file "example-constants"
                                      :package :example-package)))))
```

Make sure to specify the package you chose in step 1.

- Build stuff.

17.4.2 Contents of a grovel-constants-file

The grovel-constants-file, typically named `constants.lisp`, comprises lisp expressions describing the foreign things that you want to grovel for. A `constants.lisp` file contains two sections:

- a list of headers to include in the C program, for example:

```
("sys/types.h" "sys/socket.h" "sys/stat.h" "unistd.h" "sys/un.h"
 "netinet/in.h" "netinet/in_systm.h" "netinet/ip.h" "net/if.h"
 "netdb.h" "errno.h" "netinet/tcp.h" "fcntl.h" "signal.h")
```

- A list of sb-grovel clauses describing the things you want to grovel from the C compiler, for example:

```
((:integer af-local
           #+(or sunos solaris) "AF_UNIX"
           #-(or sunos solaris) "AF_LOCAL"
           "Local to host (pipes and file-domain).")
 (:structure stat ("struct stat"
                   (integer dev "dev_t" "st_dev")
                   (integer atime "time_t" "st_atime")))
 (:function getpid ("getpid" int )))
```

There are two types of things that sb-grovel can sensibly extract from the C compiler: constant integers and structure layouts. It is also possible to define foreign functions in the constants.lisp file, but these definitions don't use any information from the C program; they expand directly to `sb-alien:define-alien-routine` forms.

Here's how to use the grovel clauses:

- `:integer`: constant expressions in C. Used in this form:

```
(:integer lisp-variable-name "C expression" &optional doc export)
```

  `"C expression"` will be typically be the name of a constant, but other forms are possible.

- `:enum`:

```
(:enum lisp-type-name ((lisp-enumerated-name c-enumerated-name) ...)))
```

  An `sb-alien:enum` type with name `lisp-type-name` will be defined. The symbols are the `lisp-enumerated-name`s, and the values are grovelled from the `c-enumerated-name`s.

- `:structure`: alien structure definitions look like this:

```
(:structure lisp-struct-name ("struct c_structure"
                              (type-designator lisp-element-name
                               "c_element_type" "c_element_name"
                               :distrust-length nil)
                              ; ...
                              ))
```

  `type-designator` is a reference to a type whose size (and type constraints) will be groveled for. sb-grovel accepts a form of type designator that doesn't quite conform to either lisp nor sb-alien's type specifiers. Here's a list of type designators that sb-grovel currently accepts:

  - `integer`: a C integral type; sb-grovel will infer the exact type from size information extracted from the C program. All common C integer types can be grovelled for with this type designator, but it is not possible to grovel for bit fields yet.
  - `(unsigned n)`: an unsigned integer variable that is `n` bytes long. No size information from the C program will be used.
  - `(signed n)`: an signed integer variable that is `n` bytes long. No size information from the C program will be used.
  - `c-string`: an array of `char` in the structure. sb-grovel will use the array's length from the C program, unless you pass it the `:distrust-length` keyword argument with non-`nil` value (this might be required for structures such as solaris's `struct dirent`).
  - `sb-grovel::c-string-pointer`: a pointer to a C string, corresponding to the `sb-alien:c-string` type (see Section 9.2.3 [foreign type specifiers], page 92).
  - `(array alien-type)`: an array of the previously-declared `alien-type`. The array's size will be determined from the output of the C program and the alien type's size.

- `(array alien-type n)`: an array of the previously-declared `alien-type`. The array's size will be assumed as being `n`.

Note that `c-string` and `sb-grovel::c-string-pointer` do not have the same meaning. If you declare that an element is of type `c-string`, it will be treated as if the string is a part of the structure, whereas if you declare that the element is of type `sb-grovel::c-string-pointer`, a *pointer to a string* will be the structure member.

- `:function`: alien function definitions are similar to `define-alien-routine` definitions, because they expand to such forms when the lisp program is loaded. See Section 9.7 [foreign function calls], page 98.

```
(:function lisp-function-name
           ("alien_function_name" alien-return-type
                                  (argument alien-type)
                                  (argument2 alien-type)))
```

17.4.3 Programming with sb-grovel's Structure Types

Let us assume that you have a grovelled structure definition:

```
(:structure mystruct ("struct my_structure"
                      (integer myint "int" "st_int")
                      (c-string mystring "char[]" "st_str")))
```

What can you do with it? Here's a short interface document:

- Creating and destroying objects:
  - Function `(allocate-mystruct)` allocates an object of type `mystruct` and returns a system area pointer to it.
  - Macro `(with-mystruct var ((member init) [...]) &body body)` allocates an object of type `mystruct` that is valid in `body`. If `body` terminates or performs an non-local exit, the object pointed to by `var` will be deallocated.
- Accessing structure members:
  - `(mystruct-myint var)` and `(mystruct-mystring var)` return the value of the respective fields in `mystruct`.
  - `(setf (mystruct-myint var) new-val)` and `(setf (mystruct-mystring var) new-val)` sets the value of the respective structure member to the value of `new-val`. Notice that in `(setf (mystruct-mystring var) new-val)`'s case, `new-val` is a lisp string.

17.4.4 Traps and Pitfalls

Basically, you can treat functions and data structure definitions that sb-grovel spits out as if they were alien routines and types. This has a few implications that might not be immediately obvious (especially if you have programmed in a previous version of sb-grovel that didn't use alien types):

- You must take care of grovel-allocated structures yourself. They are alien types, so the garbage collector will not collect them when you drop the last reference.
- If you use the `with-mystruct` macro, be sure that no references to the variable thus allocated leaks out. It will be deallocated when the block exits.

17.5 sb-introspect

The `sb-introspect` module is about finding definitions, as well as querying their properties and relationships in the running image.

17.5.1 Finding Definitions

`sb-introspect:definition-source` [Structure]
This structure identifies a sexp in a compiled file. Despite the name, the source location may not correspond to a definition but to e.g. a function call (see `who-calls`).

`sb-introspect:definition-source-pathname` *instance* [Function]
Pathname of the source file. This is `nil` if the source location is not in a compiled file.

`sb-introspect:definition-source-form-path` *instance* [Function]
List of indices that identify the sexp in the file given by `definition-source-pathname`. The first element in the list is the index of the top-level form that contains the sexp. If the file was compiled at a high enough debug level, then the rest of the elements recursively index into the list structure of the top-level form.

Thus, the form path is somewhat stable regarding edits in the file, but it gets invalidated by, for example, inserting a new top-level form before the sexp in question.

`sb-introspect:definition-source-form-number` *instance* [Function]
Depth-first index of the sexp within the top-level form identified by the first element of `definition-source-form-path`. That is, this is the index of the sexp in the list of subexpressions of the top-level form ordered according to depth-first traversal. 0 corresponds to the top-level form itself.

When combined with the index of the top-level form (given by the first element of `definition-source-form-path`), the form number allows reconstruction of the rest of the form path, which may be missing. This requires parsing the source file. Currently, this job is delegated to e.g. SLIME.

`sb-introspect:definition-source-character-offset` *instance* [Function]
Character offset of the top-level form containing the sexp.

`sb-introspect:definition-source-file-write-date` *instance* [Function]
`file-write-date` of `definition-source-pathname` at the time of compilation. `nil` if not compiled from a file.

`sb-introspect:definition-source-plist` *instance* [Function]
The `source-plist` from `with-compilation-unit` in effect when the file was compiled.

`sb-introspect:find-definition-source` *object* [Function]
Return the `definition-source` corresponding to the definition of `object` or `nil` if there is no corresponding definition. `object` must be a `package`, `function`, `method`, `method-combination`, `sb-mop:slot-definition`, `standard-object`, `structure-object`, `condition`, `class`, `structure-class`, or a subclass of `condition`. An error is signalled for other types.

A `definition-source` object is always returned for definitions that exist, but the source location (e.g. `definition-source-pathname`) may be missing.

For definitions that do not define an object (e.g. `defvar`), use `find-definition-sources-by-name`.

`sb-introspect:find-definition-sources-by-name` *name type* [Function]
Returns a list of `definition-source`s for definitions of `name` with the given definition `type`. A `definition-source` object is always returned for definitions that exist, but the source location (e.g. `definition-source-pathname`) may be missing. `type` can currently be one of the following.

- Public definition types:

  `:class` `:compiler-macro` `:condition` `:constant` `:function` `:generic-function` `:macro` `:method` `:method-combination` `:package` `:setf-expander` `:structure` `:symbol-macro` `:type` `:alien-type` `:alien-callback` `:variable` `:declaration`

- Internal definition types:

  `:optimizer` `:source-transform` `:transform` `:vop` `:ir1-convert`

Definition types are disjoint. For example, `:type` refers to `deftype`s but not `class`es or `sb-alien:define-alien-type`, as those are of definition type `:class` and `:alien-type`, respectively. `:function` does not include `:generic-function`, `:class` does not include `:structure`, etc. `:variable` refers to non-constant dynamic variables (e.g. those defined with `defvar`, `defparameter`, `sb-ext:defglobal` or `sb-alien:define-alien-variable` but not with `defconstant`).

Valid `name`s are generally `symbol`s with the following exceptions:

- For `:compiler-macro`, `:function`, `:generic-function` and `:method`, anything that's `valid-function-name-p` is valid.
- For `:package`, string designators are valid.

If an unsupported `type` is requested or `name` is invalid, this function returns `nil`.

17.5.2 Special Variables

`sb-introspect:who-binds` *symbol* [Function]
Find the source locations where the special variable `symbol` is bound, and return them as an alist of function or macro name, `definition-source` pairs.

`sb-introspect:who-references` *symbol* [Function]
Find the source locations where the special variable `symbol` is read, and return them as an alist of function or macro name, `definition-source` pairs.

`sb-introspect:who-sets` *symbol* [Function]
Find the source locations where the special variable `symbol` is set, and return them as an alist of function or macro name, `definition-source` pairs.

17.5.3 Functions

`sb-introspect:function-lambda-list` *function* [Function]
Return the lambda list of `function`. `function` must be a function object or a function name in the sense of `valid-function-name-p`. Works for special operators, macros, simple functions, interpreted functions, and generic functions.

The second return value indicates whether the lambda list could not be determined (e.g. because the function was compiled with `debug` 0).

`sb-introspect:function-type` *function-designator* [Function]
Returns the ftype of `function-designator` or `nil`.

`sb-introspect:method-combination-lambda-list` *method-combination* [Function]
Return the lambda list of the `method-combination` designator. `method-combination` can be a method combination object, or a method combination name.

`sb-introspect:valid-function-name-p` *name* [Function]
See if `name` is a valid function name. In addition to the ANSI definition of function name, which is symbols plus lists like (`setf symbol`), SBCL allows (`sb-ext:cas symbol`) and various internal constructs.

`sb-introspect:find-function-callers` *function* &optional (*spaces* '(*:all*)) [Function]
List functions that call `function` by searching `spaces` for code objects. This can make previously garbage objects live.

`spaces` should be a list of the symbols `:dynamic`, `:static`, `:read-only`, or `:immobile` on `#+immobile-space`. The shorthand (`:all`) is also accepted.

`sb-introspect:find-function-callees` *function* [Function]
Return functions called by `function`.

`sb-introspect:who-calls` *function-name* [Function]
Find the source locations where the global function `function-name` is called, and return them as an alist of function or macro name, `definition-source` pairs.

`sb-introspect:who-macroexpands` *macro-name* [Function]
Find the source locations where the macro `macro-name` is expanded, and return them as an alist of function or macro name, `definition-source` pairs.

17.5.4 Types and Classes

`sb-introspect:deftype-lambda-list` *type-specifier-name* [Function]
Returns the lambda list of `type-specifier-name` as the first return value, and a flag whether the arglist could be found as the second value.

`type-specifier-name` must be a symbol. This function can find the lambda list of derived type specifiers (e.g. those defined with `deftype`) and classes with compound type specifier syntaxes (e.g. the class `float`). It returns `nil`, `nil` for other type specifiers (e.g. `and`, `or`, `not`) and types (e.g. `list`).

`sb-introspect:who-specializes-directly` *class-designator* [Function]
Find the source locations of methods directly specializing on `class-designator`, and return them as an alist of generic function name, `definition-source` pairs.

A method matches the criterion either if it specializes on the same class as `class-designator` designates, or if it eql-specializes on an instance of the designated class.

Experimental.

`sb-introspect:who-specializes-generally` *class-designator* [Function]
Find the source locations of methods specializing on `class-designator` or a subclass of it, and return them as an alist of generic function name, `definition-source` pairs. `definition-source-description` identifies the method.

A method matches the criterion either if it specializes on the designated class itself or a subclass of it (this includes CLASS-EQ specializers), or if it eql-specializes on an instance of the designated class or a subclass of it.

Experimental.

17.5.5 Allocation

`sb-introspect:allocation-information` *object* [Function]
Returns information about the allocation of `object`. The primary return value indicates the general type of allocation: `:immediate`, `:heap`, `:stack`, or `:foreign`.

Non-NIL secondary return values provide additional information about the allocation.

For `:heap` objects the secondary value is a plist:

`:space` Indicates the heap segment the object is allocated in.

`:generation` The current generation of the object: 0 for nursery, 6 for pseudo-static generation loaded from core. (GENCGC and `:space :dynamic` only.)

`:large` Indicates a "large" object subject to non-copying promotion. (GENCGC and `:space :dynamic` only.)

`:boxed` Indicates that the object is allocated in a boxed region. Unboxed allocation is used for e.g. specialized arrays after they have survived one collection. (GENCGC and `:space :dynamic` only.)

`:pinned` Indicates that the page(s) on which the object resides are kept live due to conservative references. Note that object may reside on a pinned page even if `:pinned` is `nil` if the GC has not had the need to mark the page as pinned. (GENCGC and `:space :dynamic` only.)

`:write-protected` Indicates that the page on which the object starts is write-protected, which indicates for `:boxed` objects that it hasn't been written to since the last GC of its generation. (GENCGC and `:space :dynamic` only.)

`:page` The index of the page the object resides on. (GENCGC and `:space :dynamic` only.)

For `:stack` objects, the secondary value is the thread on whose stack the object is allocated.

Expected use-cases include introspection to gain insight into allocation and GC behaviour and restricting memoization to heap-allocated arguments.

Experimental: interface subject to change.

`sb-introspect:map-root` *function object* &key *simple* (*ext t*) [Function]
Call `function` with all non-immediate objects pointed to by `object`. Returns `object`.

If `simple` is true (default is `nil`), elides those pointers that are not notionally part of certain built-in objects but backpointers to a conceptual parent: e.g. elides the pointer from a `symbol` to the corresponding `package`.

If `ext` is true (default is `t`), includes some pointers that are not actually contained in the object but found in certain well-known indirect containers: `fdefinitions`, `eql` specializers, classes, and thread-local symbol values in other threads fall into this category.

> *Note*: calling `map-root` with a `sb-thread:thread` does not currently map over conservative roots from the thread registers and interrupt contexts.

Experimental: interface subject to change.

17.6 sb-manual

The `sb-manual` module has the SBCL user manual in forms mimicking `pax:defsection`:

```
(defsection @example (:title "Example")
  "This is an example, but see the real @SB-MANUAL."
  (print function)
  (@subexample section))
```

The names of the variables holding the documentation are exported from the `sb-manual` package. Since sections are basically variables, in Slime, `M-.` on `"@SB-MANUAL"`, `"print"`, or on `"@subexample"` will take you to the respective definition. This makes it easy to navigate the documentation. Normal Lisp definition docstrings and section docstrings reference sections following the usual convention of uppercasing the name. Docstrings are in a subset of Markdown and use very little markup in general, so they are easy to read directly in the source.

The official manual in Info, HTML and PDF formats is generated via Texinfo generated from these definitions.

17.6.1 Using PAX

However, `sb-manual::defsection` is but a dummy implementation of `pax:defsection` to avoid a hard dependency on PAX.

See the `mgl-pax` `asdf:system` or `https://github.com/melisgl/mgl-pax/`.

When PAX is loaded, the dummy `defsection` definitions are made real, so that PAX can work with them.

`sb-manual:use-pax` [Function]

Ensure that exported variables are `pax:section`s. It is an error if the `mgl-pax` library is not loaded.

Calling this function explicitly is rarely necessary because it is called automatically:

- when `sb-manual` is loaded, if `pax` is present;
- when `pax:document` (more precisely, `dref:locate`) is called on an `sb-manual` section.

The latter feature requires v0.4.12 of `pax`. See the `mgl-pax` asdf:system.

17.6.2 Browsing Live with PAX

With PAX, you can browse the manual live. The documentation of this feature is available at online (`https://melisgl.github.io/mgl-pax-world/pax-manual.html#MGL-PAX:@BROWSING-LIVE-DOCUMENTATION%20MGL-PAX:SECTION`).

If you are browsing this manual live right now, here is the equivalent live link: `pax::@browsing-live-documentation`.

Notable features:

- Autolinks within the manual: if `sb-ext:exit` is mentioned, then it's linked to its documentation. You basically get links to where `M-.` would go in the sources.
- Autolinks to the CLHS.
- View the documentation of any Lisp definition or section without generating the entire manual.
- Locatives (e.g. the `"[function]"` in `"- [function] SB-EXT:EXIT"`) are also links in live browsing: they tell Slime to visit the definition.

  For this to work, you need to allow Slime to evaluate Elisp sent from SBCL:

  ```
  (setq slime-enable-evaluate-in-emacs t)
  ```

  and maybe your window manager focus stealing configuration needs tweaking as well.

Live browsing can greatly reduce the latency of Edit-Compile-View Loop, when working on documentation.

17.6.3 Fancy Documentation with PAX

PAX can generate dead documentation, too. In the SBCL sources, `contrib/sb-manual/make-pax-docs.sh` generates the manual in plain text, Markdown, PDF, and HTML formats. These differ from those generated via Texinfo in that they are autolinked (like when Section 17.6.2 [browsing live with pax], page 149).

Also, you can generate documentation yourself with e.g.

```
(pax:document sb-manual:@sbcl-manual :format :markdown)
```

There is seldom a need to generate these manually, as `https://fixnum.com` generates them for the master branch of SBCL every few hours.

17.7 sb-md5

The `sb-md5` module implements the RFC1321 MD5 Message Digest Algorithm.

`sb-md5:md5sum-file` *pathname* [Function]
Calculate the MD5 message-digest of the file specified by `pathname`.

`sb-md5:md5sum-sequence` *sequence* &key (*start 0*) *end* [Function]
Calculate the MD5 message-digest of data in `sequence`, which should be a 1d `simple-array` with element type (`unsigned-byte` 8). On CMUCL and SBCL non-simple and non-1d arrays with this element-type are also supported.

`sb-md5:md5sum-stream` *stream* [Function]
Calculate an MD5 message-digest of the contents of `stream`. Its element-type has to be (`unsigned-byte` 8). Use on character streams is DEPRECATED, as this will not work correctly on implementations with `char-code-limit` > 256 and ignores character coding issues.

`sb-md5:md5sum-string` *string* &key (*external-format :default*) (*start 0*) *end* [Function]
Calculate the MD5 message-digest of the binary representation of `string` (as octets) in the external format specified by `external-format`. The boundaries `start` and `end` refer to character positions in the string, not to octets in the resulting binary representation. The permissible external format specifiers are determined by the underlying implementation.

The implementation for CMUCL was largely done by Pierre Mai, with help from members of the `cmucl-help` mailing list. Since CMUCL and SBCL are similar in many respects, it was not too difficult to extend the low-level implementation optimizations for CMUCL to SBCL. Following this, SBCL's compiler was extended to implement efficient compilation of modular arithmetic (Section 6.8 [modular arithmetic], page 56), which enabled the implementation to be expressed in portable arithmetical terms, apart from the use of Section 17.10 [sb rotate byte], page 153, for bitwise rotation.

17.8 sb-posix

Sb-posix is the supported interface for calling out to the operating system.

> *Note*: The functionality contained in the package `sb-unix` is for SBCL internal use only; its contents are likely to change from version to version.

The scope of this interface is "operating system calls on a typical Unixlike platform". This is section 2 of the Unix manual, plus section 3 calls that are (a) typically found in libc, but (b) not part of the C standard. For example, we intend to provide support for `opendir(3)` and `readdir(3)` but not for `printf(3)`. That said, if your favourite system call is not included yet, you are encouraged to submit a patch to the SBCL mailing list.

Some facilities are omitted where they offer absolutely no additional use over some portable function, or would be actively dangerous to the consistency of Lisp. Not all functions are available on all platforms.

Sb-posix functions do not implicitly take measures to provide thread-safety or reentrancy beyond whatever the underlying C library does, except in cases where doing so is necessary to maintain the consistency of the Lisp image. For example, the bindings to the user and group database accessing functions are neither thread-safe nor reentrant unless the underlying libc happens to make them so (but see Section 17.8.7 [sb posix extensions to posix], page 153).

17.8.1 Lisp Names for C Names

All symbols are in the `sb-posix` package. This package contains a Lisp function for each supported Unix system call or function, a variable or constant for each supported Unix constant, an object type for each supported Unix structure type, and a slot name for each supported Unix structure member. A symbol name is derived from the C binding's name, by (a) uppercasing, then (b) removing leading underscores (`#\_`) then replacing remaining underscore characters with the hyphen (`#\-`). The requirement to uppercase is so that in a standard upcasing reader the user may write `sb-posix:creat` instead of `sb-posix:|creat|` as would otherise be required.

No other changes to "Lispify" symbol names are made, so `creat` becomes `CREAT`, not `CREATE`.

The user is encouraged not to `(use-package :sb-posix)` but instead to use the `sb-posix:` prefix on all references, as some of the symbols symbols contained in the `sb-posix` package have the same name as CL symbols (e.g. `open`, `close`, `signal`). Also, see Section 7.3 [package local nicknames], page 61.

17.8.2 Types

Generally, marshalling between Lisp and C data types is done using SBCL's FFI. See Chapter 9 [foreign function interface], page 91.

Some functions accept objects such as filenames or file descriptors. In the C binding to POSIX, these are represented as strings and small integers respectively. For the Lisp programmer's convenience we introduce designators such that CL pathnames or open streams can be passed to these functions. For example, `sb-posix:rename` accepts both pathnames and strings as its arguments.

17.8.2.1 File-descriptors

`sb-posix:file-descriptor` [Type]

A `fixnum` designating a native file descriptor.

`sb-sys:make-fd-stream` can be used to construct a `file-stream` associated with a native file descriptor.

Note that mixing I/O operations on a `file-stream` with operations directly on its descriptor may produce unexpected results if the stream is buffered.

`sb-posix:file-descriptor-designator` [Type]

Designator for a `file-descriptor`: either a fixnum designating itself, or a `file-stream` designating the underlying file-descriptor.

`sb-posix:file-descriptor` *file-descriptor* [Function]

Converts `file-descriptor-designator` into a `file-descriptor`.

17.8.2.2 Filenames

`sb-posix:filename` [Type]

A `string` designating a filename in native namestring syntax.

Note that native namestring syntax is distinct from Lisp namestring syntax:

```
(pathname "/foo*/bar")
```

is a wild pathname with a pattern-matching directory component. `sb-ext:parse-native-namestring` may be used to construct Lisp pathnames that denote POSIX filenames as understood by system calls, and `sb-ext:native-namestring` can be used to coerce them into strings in the native namestring syntax.

Note also that POSIX filename syntax does not distinguish the names of files from the names of directories: in order to parse the name of a directory in POSIX filename syntax into a pathname `my-defaults` for which

```
(merge-pathnames (make-pathname :name "FOO" :case :common)
                 my-defaults)
```

returns a pathname that denotes a file in the directory, supply a true `:as-directory` argument to `sb-ext:parse-native-namestring`. Likewise, to supply the name of a directory to a POSIX function in non-directory syntax, supply a true `:as-file` argument to `sb-ext:native-namestring`.

`sb-posix:filename-designator` [Type]

Designator for a `filename`: a `string` designating itself, or a designator for a `pathname` designating the corresponding native namestring.

`sb-posix:filename` *filename* [Function]

Converts `filename-designator` into a `filename`.

17.8.3 Function Parameters

The calling convention is modelled after that of CMUCL's `unix` package: in particular, it's like the C interface except that:

- Length arguments are omitted or optional where the sensible value is obvious. For example, `read` would be defined this way:

  ```
  (read fd buffer &optional (length (length buffer))) => bytes-read
  ```

- Where C simulates "out" parameters using pointers (for instance, in `pipe(2)` or `socketpair(2)`), these may be optional or omitted in the Lisp interface: if not provided, appropriate objects will be allocated and returned (using multiple return values if necessary).

- Some functions accept objects such as filenames or file descriptors. Wherever these are specified as such in the C bindings, the Lisp interface accepts designators for them as specified in the Section 17.8.2 [sb posix types], page 151, section above.
- A few functions have been included in sb-posix that do not correspond exactly with their C counterparts. These are described in Section 17.8.6 [sb posix idiosyncracies], page 152.

17.8.4 Function Return Values

The return value is usually the same as for the C binding, except in error cases: where the C function is defined as returning some sentinel value and setting `errno` on error, we instead signal an error of type `sb-posix:syscall-error`. The actual error value (`errno`) is stored in this condition and can be accessed with `sb-posix:syscall-errno`.

We do not automatically translate the returned value into lispy objects – for example, `sb-posix:open` returns a small integer, not a stream. Exception: boolean-returning functions (or, more commonly, macros) do not return a C integer but instead a Lisp boolean.

17.8.5 Lisp Objects and C structures

Sb-posix provides various Lisp object types to stand in for C structures in the POSIX library. Lisp bindings to C functions that accept, manipulate, or return C structures accept, manipulate, or return instances of these Lisp types instead of instances of alien types.

The names of the Lisp types are chosen according to the general rules described above. For example Lisp objects of type `sb-posix:stat` stand in for C structures of type `struct stat`.

Accessors are provided for each standard field in the structure. These are named `<structure-name>-<field-name>` where the two components are chosen according to the general name conversion rules, with the exception that in cases where all fields in a given structure have a common prefix, that prefix is omitted. For example, `stat.st_dev` in C becomes `stat-dev` in Lisp.

Because sb-posix might not support all semi-standard or implementation-dependent members of all structure types on your system (patches welcome), here is an enumeration of all supported Lisp objects corresponding to supported POSIX structures, and the supported slots for those structures.

`sb-posix:flock` [Class]
Class representing locks used in `fcntl(2)`.

`sb-posix:passwd` [Class]
Instances of this class represent entries in the system's user database.

`sb-posix:group` [Class]
Instances of this class represent entries in the system's group database.

`sb-posix:stat` [Class]
Instances of this class represent POSIX file metadata.

`sb-posix:termios` [Class]
Instances of this class represent I/O characteristics of the terminal.

`sb-posix:timeval` [Class]
Instances of this class represent time values.

17.8.6 Functions with Idiosyncratic Bindings

A few functions in sb-posix don't correspond directly to their C counterparts.

`sb-posix:getcwd` [Function]
Returns the process's current working directory as a string.

`sb-posix:readlink` *pathspec* [Function]
Returns the resolved target of a symbolic link as a string.

`sb-posix:syslog` *priority format* &rest *args* [Function]
Send a message to the syslog facility, with severity level `priority`. The message will be formatted as by `cl:format` (rather than C's `printf`) with format string `format` and arguments `args`.

17.8.7 Extensions to POSIX

Some of POSIX's standardized operators are not safe to use on their own, so `sb-posix` exports a few helpers that do not correspond exactly to functionality present in the POSIX standard.

The user and group database accessing routines are not required to be thread-safe or reentrant and so can only be used safely if all clients coordinate around their use. Since it would be logically impossible for independently developed programs to coordinate, `sb-posix` exports two iteration macros, `sb-posix:do-passwds` and `sb-posix:do-groups`, each of which iterates over the respective database while preventing the keyed accesses (`sb-posix:getpwnam`, `sb-posix:getpwuid`, `sb-posix:getgrnam`, `sb-posix:getgrgid`) from running until iteration completes.

`sb-posix:do-passwds` (*passwd* &optional *result*) &body *body* [Macro]
Evaluate `body` with `passwd` bound to successive entries from the passwd database, and return `result`. An implicit block named `nil` surrounds the form; an implicit `tagbody` surrounds `body`. It is unspecified whether `passwd` is assigned, rebound, or destructively modified upon each iteration. It is an error to use any operator that accesses the `passwd` database during the dynamic extent of `do-passwds`.

`sb-posix:do-groups` (*group* &optional *result*) &body *body* [Macro]
Evaluate `body` with `group` bound to successive entries from the group database, and return `result`. An implicit block named `nil` surrounds the form; an implicit `tagbody` surrounds `body`. It is unspecified whether `group` is assigned, rebound, or destructively modified upon each iteration. It is an error to use any operator that accesses the `group` database during the dynamic extent of `do-groups`.

17.9 sb-queue

Since SBCL 1.0.38, the `sb-queue` module has been merged into the `sb-concurrency` module. See Section 17.2 [sb concurrency], page 139.

17.10 sb-rotate-byte

The `sb-rotate-byte` module offers an interface to bitwise rotation, with an efficient implementation for operations which can be performed directly using the platform's arithmetic routines. It implements the specification at `http://www.cliki.net/ROTATE-BYTE`.

Bitwise rotation is a component of various cryptographic or hashing algorithms: MD5, SHA-1, etc.; often these algorithms are specified on 32-bit rings.

`sb-rotate-byte:rotate-byte` *count bytespec integer* [Function]
Rotates a field of bits within `integer`; specifically, returns an integer that contains the bits of `integer` rotated `count` times leftwards within the byte specified by `bytespec`, and elsewhere contains the bits of `integer`.

18 Deprecation

In order to support evolution of interfaces in SBCL as well as in user code, SBCL allows declaring functions, variables and types as deprecated. Users of deprecated things are notified by means of warnings while the deprecated thing in question is still available.

This chapter documents the interfaces for being notified when using deprecated thing and declaring things as deprecated, the deprecation process used for SBCL interfaces, and lists legacy interfaces in various stages of deprecation.

Deprecation in this context should not be confused with those things the ANSI Common Lisp standard calls *deprecated*: the entirety of ANSI CL is supported by SBCL, and none of those interfaces are subject to censure.

18.1 Why Deprecate?

While generally speaking we try to keep SBCL changes as backwards compatible as feasible, there are situations when existing interfaces are deprecated:

- **Broken Interfaces**

  Sometimes it turns out that an interface is sufficiently misdesigned that fixing it would be worse than deprecating it and replacing it with another.

  This is typically the case when fixing the interface would change its semantics in ways that could break user code subtly: in such cases we may end up considering the obvious breakage caused by deprecation to be preferable.

  Another example are functions or macros whose current signature makes them hard or impossible to extend in the future: backwards compatible extensions would either make the interface intolerably hairy, or are sometimes outright impossible.

- **Internal Interfaces**

  SBCL has several internal interfaces that were never meant to be used in user code – or at least never meant to be used in user code unwilling to track changes to SBCL internals.

  Ideally, we'd like to be free to refactor our own internals as we please, without even going through the hassle of deprecating things. Sometimes, however, it turns out that our internal interfaces have several external users who aren't using them advisedly, but due to misunderstandings regarding their status or stability.

  Consider a deprecated internal interface a reminder for SBCL maintainers not to delete the thing just yet, even though it is seems unused – because it has external users.

  When internal interfaces are deprecated we try our best to provide supported alternatives.

- **Aesthetics & Ease of Maintenance**

  Sometimes an interface isn't broken or internal but just inconsistent somehow.

  This mostly happens only with historical interfaces inherited from CMUCL which often haven't been officially supported in SBCL before, or with new extensions to SBCL that haven't been around for very long in the first place.

  The alternative would be to keep the suboptimal version around forever, possibly alongside an improved version. Sometimes we may do just that, but because every line of code comes with a maintenance cost, sometimes we opt to deprecate the suboptimal version instead: SBCL doesn't have infinite developer resources.

  We also believe that sometimes cleaning out legacy interfaces helps keep the whole system more comprehensible to users, and makes introspective tools such as `apropos` more useful.

18.2 The Deprecation Pipeline

SBCL uses a *deprecation pipeline* with multiplestages: as time time goes by, deprecated things move from earlier stages of deprecation to later stages before finally being removed. The intention is making users aware of necessary changes early but allowing a migration to new interfaces at a reasonable pace.

Deprecation proceeds in three stages, each lasting approximately a year. In some cases it might move slower or faster, but one year per stage is what we aim at in general. During each stage warnings (and errors) of

increasing severity are signaled, which note that the interface is deprecated, and point users towards any replacements when applicable.

- **Early Deprecation**

  During early deprecation the interface is kept in working condition. However, when a thing in this deprecation stage is used, an `sb-ext:early-deprecation-warning`, which is a `style-warning`, is signaled at compile-time.

  The internals may change at this stage: typically because the interface is re-implemented on top of its successor. While we try to keep things as backwards-compatible as feasible (taking maintenance costs into account), sometimes semantics change slightly.

  For example, when the spinlock API was deprecated, spinlock objects ceased to exist, and the whole spinlock API became a synonym for the mutex API – so code using the spinlock API continued working but silently switched to mutexes instead. However, if someone relied on

  ```
  (typep lock 'spinlock)
  ```

  returning `nil` for a mutexes, trouble could ensue.

- **Late Deprecation**

  During late deprecation the interface remains as it was during early deprecation, but the compile-time warning is upgraded: when a thing in this deprecation stage is used, a `sb-ext:late-deprecation-warning`, which is a full `warning`, is signaled at compile-time.

- **Final Deprecation**

  During final deprecation the symbols still exist. However, when a thing in this deprecation stage is used, a `sb-ext:final-deprecation-warning`, which is a full `warning`, is signaled at compile-time and an `error` is signaled at run-time.

- **After Final Deprecation**

  The interface is deleted entirely.

18.3 Deprecation Conditions

`sb-ext:deprecation-condition` is the superclass of all deprecation-related warning and error conditions. All common slots and readers are defined in this condition class.

`sb-ext:deprecation-condition` [Condition]
Superclass for deprecation-related error and warning conditions.

`sb-ext:early-deprecation-warning` [Condition]
This warning is signaled when the use of a variable, function, type, etc. in `:early` deprecation is detected at compile-time. The use will work at run-time with no warning or error.

`sb-ext:late-deprecation-warning` [Condition]
This warning is signaled when the use of a variable, function, type, etc. in `:late` deprecation is detected at compile-time. The use will work at run-time with no warning or error.

`sb-ext:final-deprecation-warning` [Condition]
This warning is signaled when the use of a variable, function, type, etc. in `:final` deprecation is detected at compile-time. An error will be signaled at run-time.

`sb-ext:deprecation-error` [Condition]
This error is signaled at run-time when an attempt is made to use a thing that is in `:final` deprecation, i.e. call a function or access a variable.

18.4 Introspecting Deprecation Information

The deprecation status of functions and variables can be inspected using the `sb-cltl2:function-information` and `sb-cltl2:variable-information` functions provided by the `sb-cltl2` contributed module.

18.5 Deprecation Declaration

The `sb-ext:deprecated` declaration can be used to declare objects in various namespaces as deprecated.

Note: See the `namespace` `clhs` glossary entry in the glossary of the Common Lisp HyperSpec.)

`sb-ext:deprecated` [Declaration]

Syntax: `(sb-ext:deprecated stage since &rest object-clauses)`

stage ::= {`:early` | `:late` | `:final`}

since ::= {`<version>` | (`<software>` `<version>`)}

object-clause ::= (namespace `<name>` [`:replacement` `<replacement>`])

namespace ::= {`cl:variable` | `cl:function` | `cl:type`}

where the terminal `<name>` is the name of the deprecated thing, `<version>` and `<software>` are strings describing the version in which the thing has been deprecated and `<replacement>` is a name or a list of names designating things that should be used instead of the deprecated thing.

Currently the following namespaces are supported:

- `cl:function`: Declare functions, compiler-macros or macros as deprecated.

  When declaring a function to be in `:final` deprecation, there should be no actual definition of the function as the declaration emits a stub function that signals a `sb-ext:deprecation-error` at run-time when called.

- `cl:variable`: Declare special and global variables, constants and symbol-macros as deprecated.

  When declaring a variable to be in `:final` deprecation, there should be no actual definition of the variable as the declaration emits a symbol-macro that signals a `sb-ext:deprecation-error` at run-time when accessed.

- `cl:type`: Declare named types (i.e. defined via `deftype`), standard classes, structure classes and condition classes as deprecated.

18.6 Deprecation Examples

Marking functions as deprecated:

```
(defun foo ())
(defun bar ())
(declaim (deprecated :early ("my-system" "1.2.3")
                     (function foo :replacement bar)))

;; Remember: do not define the actual function or variable in case of
;; :final deprecation:
(declaim (deprecated :final ("my-system" "1.2.3")
                     (function fez :replacement whoop)))
```

Attempting to use the deprecated functions:

```
(defun baz ()
  (foo))
| STYLE-WARNING: The function CL-USER::FOO has been deprecated...
=> BAZ
(baz)
=> NIL ; no error

(defun danger ()
  (fez))
| WARNING: The function CL-USER::FEZ has been deprecated...
=> DANGER
(danger)
|- ERROR: The function CL-USER::FEZ has been deprecated...
```

18.7 Deprecated Interfaces in SBCL

This sections lists legacy interfaces in various stages of deprecation.

18.7.1 List of Deprecated Interfaces

18.7.1.1 Early Deprecation

- `sockint::win32-*`

  Deprecated in favor of the corresponding prefix-less functions (e.g. `sockint::bind` replaces `sockint::win32-bind`) as of 1.2.10 in March 2015. Expected to move into late deprecation in August 2015.

- `sb-unix:unix-exit`

  Deprecated as of 1.0.56.55 in May 2012. Expected to move into late deprecation in May 2013.

  When the SBCL process termination was refactored, `sb-unix:unix-exit` ceased to be used internally. Since `sb-unix` is an internal package not intended for user code to use, and since we're slowly in the process of refactoring things to be less Unix-oriented, `sb-unix:unix-exit` was initially deleted as it was no longer used. Unfortunately it became apparent that it was used by several external users, so it was re-instated in deprecated form.

  While the cost of keeping `sb-unix:unix-exit` indefinitely is trivial, the ability to refactor our internals is important, so its deprecation was taken as an opportunity to highlight that `sb-unix` is an internal package and `sb-posix` should be used by user-programs instead – or alternatively calling the foreign function directly if the desired interface doesn't for some reason exist in `sb-posix`.

  Remedy

  For code needing to work with legacy SBCLs, use e.g. `system-exit`. In modern SBCLs, simply call either `sb-posix:exit` or `sb-ext:exit` with appropriate arguments.

- `sb-c::merge-tail-calls` compiler policy

  Deprecated as of 1.0.53.74 in November 2011. Expected to move into late deprecation in November 2012.

  This compiler policy was never functional: SBCL has always merged Section 6.2 [tail call], page 46s when it could, regardless of this policy setting. (It was also never officially supported, but several code-bases have historically used it.)

  Remedy

  Simply remove the policy declarations. They were never necessary: SBCL always merged tail-calls when possible. To disable tail merging, structure the code to avoid the tail position instead.

- The Spinlock API

  Deprecated as of 1.0.53.11 in August 2011. Expected to move into late deprecation in August 2012.

  Spinlocks were an internal interface but had a number of external users and were hence deprecated instead of being simply deleted.

  Affected symbols: `sb-thread::spinlock`, `sb-thread::make-spinlock`, `sb-thread::with-spinlock`, `sb-thread::with-recursive-spinlock`, `sb-thread::get-spinlock`, `sb-thread::release-spinlock`, `sb-thread::spinlock-value`, and `sb-thread::spinlock-name`.

  Remedy

  Use the mutex API instead, or implement spinlocks suiting your needs on top of `sb-ext:compare-and-swap`, `sb-ext:spin-loop-hint`, etc.

- `sockint::handle->fd`, `sockint::fd->handle`

  Internally deprecated in 2012. Declared deprecated as of 1.2.10 in March 2015. Expected to move into final deprecation in August 2015.

18.7.1.2 Late Deprecation

- `sb-thread:join-thread-error-thread` and `sb-thread:interrupt-thread-error-thread`

  Deprecated in favor of `sb-thread:thread-error-thread` as of 1.0.29.17 in June 2009. Expected to move into final deprecation in June 2012.

  Remedy

  For code that needs to support legacy SBCLs, use e.g.:

  ```
  (defun get-thread-error-thread (condition)
  ```

```
#+#.(cl:if (cl:find-symbol "THREAD-ERROR-THREAD" :sb-thread)
           '(and) '(or))
(sb-thread:thread-error-thread condition)
#-#.(cl:if (cl:find-symbol "THREAD-ERROR-THREAD" :sb-thread)
           '(and) '(or))
(etypecase condition
 (sb-thread:join-thread-error
  (sb-thread:join-thread-error-thread condition))
 (sb-thread:interrupt-thread-error
  (sb-thread:interrupt-thread-error-thread condition))))
```

- `sb-introspect:function-arglist`

  Deprecated in favor of `sb-introspect:function-lambda-list` as of 1.0.24.5 in January 2009. Expected to move into final deprecation in January 2012.

  Renamed for consistency and aesthetics. Functions have lambda-lists, not arglists.

  Remedy

  For code that needs to support legacy SBCLs, use e.g.:

  ```
  (defun get-function-lambda-list (function)
    #+#.(cl:if (cl:find-symbol "FUNCTION-LAMBDA-LIST" :sb-introspect)
               '(and) '(or))
    (sb-introspect:function-lambda-list function)
    #-#.(cl:if (cl:find-symbol "FUNCTION-LAMBDA-LIST" :sb-introspect)
               '(and) '(or))
    (sb-introspect:function-arglist function))
  ```

- Stack Allocation Policies

  Deprecated in favor of `sb-ext:*stack-allocate-dynamic-extent*` as of 1.0.19.7 in August 2008, and are expected to be removed in August 2012.

  Affected symbols: `sb-c::stack-allocate-dynamic-extent`, `sb-c::stack-allocate-vector`, and `sb-c::stack-allocate-value-cells`.

  These compiler policies were never officially supported, and turned out the be a flawed design.

  Remedy

  For code that needs stack-allocation in legacy SBCLs, conditionalize using:

  ```
  #-#.(cl:if (cl:find-symbol "*STACK-ALLOCATE-DYNAMIC-EXTENT*" :sb-ext)
             '(and) '(or))
  (declare (optimize sb-c::stack-allocate-dynamic-extent))
  ```

  However, unless stack allocation is essential, we recommend simply removing these declarations. Refer to documentation on `sb-ext:*stack-allocate-dynamic*` for details on stack allocation control in modern SBCLs.

- `sb-sys:output-raw-bytes`

  Deprecated as of 1.0.8.16 in June 2007. Expected to move into final deprecation in June 2012.

  Internal interface with some external users. Never officially supported, deemed unnecessary in presence of `write-sequence` and bivalent streams.

  Remedy

  Use streams with element-type (`unsigned-byte` 8) or `:default` – the latter allowing both binary and character IO – in conjunction with `write-sequence`.

18.7.1.3 Final Deprecation

No interfaces are currently in final deprecation.

18.7.2 Historical Interfaces

The following is a partial list of interfaces present in historical versions of SBCL, which have since then been deleted.

- `sb-kernel:instance-lambda`

  Historically needed for CLOS code. Deprecated as of 0.9.3.32 in August 2005. Deleted as of 1.0.47.8 in April 2011. Plain `lambda` can be used where `sb-kernel:instance-lambda` used to be needed.

- `sb-alien:def-alien-routine`, `sb-alien:def-alien-variable`, `sb-alien:def-alien-type`

  Inherited from CMUCL, naming convention not consistent with preferred SBCL style. Deprecated as of 0.pre7.90 in December 2001. Deleted as of 1.0.9.17 in September 2007. Replaced by `sb-alien:define-alien-routine`, `sb-alien:define-alien-variable`, and `sb-alien:define-alien-type`.

19 Further Resources

19.1 The SBCL Project

19.1.1 Volunteer Support

Your primary source of SBCL support should probably be the mailing list `sbcl-help`: in addition to other users, SBCL developers monitor this list and are available for advice. As an anti-spam measure subscription is required for posting:

`https://lists.sourceforge.net/lists/listinfo/sbcl-help`

Remember that the people answering your question are volunteers, so you stand a much better chance of getting a good answer if you ask a good question.

Before sending mail, check the list archives at either

- `http://sourceforge.net/mailarchive/forum.php?forum_name=sbcl-help` or
- the `gmane.lisp.steel-bank.general` news group on `news.gmane.io`

to see if your question has been answered already. Checking the bug database is also worth it (see Section 19.1.3 [reporting bugs], page 160), to see if the issue is already known.

For general advice on asking good questions, see

`http://www.catb.org/~esr/faqs/smart-questions.html`.

19.1.2 Commercial Support

There is no formal organization developing SBCL, but if you need a paid support arrangement or custom SBCL development, we maintain the list of companies and consultants below. Use it to identify service providers with appropriate skills and interests, and contact them directly.

The SBCL project cannot verify the accuracy of the information or the competence of the people listed, and they have provided their own blurbs below: you must make your own judgement of suitability from the available information – refer to the links they provide, the `CREDITS` file, mailing list archives, commit messages, and so on. Please feel free to ask for advice through Section 19.1.1 [volunteer support], page 160.

(At present, no companies or consultants wish to advertise paid support or custom SBCL development in this manual.)

19.1.3 Reporting Bugs

SBCL uses Launchpad to track bugs. The bug database is available at

`https://bugs.launchpad.net/sbcl`

Reporting bugs there requires registering at Launchpad. However, bugs can also be reported on the mailing list `sbcl-bugs`, which is moderated but does *not* require subscribing. Simply send email to `sbcl-bugs@lists.sourceforge.net` (`mailto:sbcl-bugs@lists.sourceforge.net`) to have the bug checked and added to Launchpad by SBCL maintainers.

See the `HACKING` file on how to send patches.

19.1.3.1 How to Report Bugs Effectively

Please include enough information in a bug report that someone reading it can reproduce the problem. That is, don't write

```
Subject: apparent bug in PRINT-OBJECT (or *PRINT-LENGTH*?)
PRINT-OBJECT doesn't seem to work with *PRINT-LENGTH*. Is this a bug?
```

but instead

```
Subject: apparent bug in PRINT-OBJECT (or *PRINT-LENGTH*?)
In sbcl-1.2.3 running under OpenBSD 4.5 on my Alpha box, when
I compile and load the file
   (DEFSTRUCT (FOO (:PRINT-OBJECT (LAMBDA (X Y)
                                    (LET ((*PRINT-LENGTH* 4))
                                      (PRINT X Y)))))
     X Y)
```

```
then at the command line type
   (MAKE-FOO)
the program loops endlessly instead of printing the object.
```

A more in-depth discussion on reporting bugs effectively can be found at `http://www.chiark.greenend.org.uk/~sgtatham/bugs.html`.

19.1.3.2 Getting Backtraces for Low-level Errors

If you get a fatal errors such as `signal N is [un]blocked` or SBCL just hang and you want to send a useful bug report, then:

- Make sure SBCL is compiled and run with Section 5.2 [ldb], page 39, enabled.
- Isolate a smallish test case, run it.
- If SBCL hangs, kill it with `sigabrt`: `kill -ABRT <pidof sbcl>`, which by default enters `ldb`.
- Print the backtrace from `ldb` by typing `ba`.
- Next, attach gdb with `gdb -p <pidof sbcl>`, and get backtraces for all threads with `thread apply all ba`.
- If multiple threads are in play, then – still in gdb – try to get Lisp backtrace for all threads: `thread apply all call backtrace_from_fp($ebp, 100, 0)` on x86. Substitute `$ebp` with `$rbp` on x86-64. The backtraces will appear in the stdout of the SBCL process.
- Send a report with the backtraces and the output (both stdout and stderr) produced by SBCL.
- Don't forget to include OS and SBCL version.
- If available, include information on outcome of the same test with other versions of SBCL, OS, etc.

19.1.4 Editor Integration

Though SBCL can be used running "bare", the recommended mode of development is with an editor connected to SBCL, supporting not only basic Lisp editing (paren-matching, etc) but providing among other features an integrated debugger, interactive compilation, and automated documentation lookup.

Currently *SLIME* (Superior Lisp Interaction Mode for Emacs) together with Emacs is recommended for use with SBCL, though other options exist as well. SLIME can be downloaded from `https://slime.common-lisp.dev/`.

19.1.5 Internals Documentation

If you're interested in the development of the SBCL system itself, then subscribing to sbcl-devel@lists.sourceforge.net (`mailto:sbcl-devel@lists.sourceforge.net`) is a good idea.

The Texinfo sources for the SBCL Internals Manual are available in `doc/internals/`.

Some low-level information describing the programming details of the conversion from CMUCL to SBCL is available in the `doc/FOR-CMUCL-DEVELOPERS` file.

19.1.6 History and Implementation of SBCL

You can work productively with SBCL without knowing or understanding anything about where it came from, how it is implemented, or how it extends the ANSI Common Lisp standard. However, a little knowledge can be helpful in order to understand error messages, to troubleshoot problems, to understand why some parts of the system are better debugged than others, and to anticipate which known bugs, known performance problems, and missing extensions are likely to be fixed, tuned, or added.

SBCL is descended from CMUCL, which is itself descended from Spice Lisp, including early implementations for the Mach operating system on the IBM RT, back in the 1980s. Some design decisions from that time are still reflected in the current implementation:

- Some parts of the system expect to be loaded into a fixed-at-compile-time location in virtual memory.
- The system is implemented as a C program which is responsible for supplying low-level services and loading a Lisp `.core` file.

SBCL also inherited some newer architectural features from CMUCL. The most important is that on some architectures it has a generational GC, which has various implications (mostly good) for performance. These are discussed in another chapter, Chapter 6 [efficiency], page 42.

SBCL has diverged from CMUCL in that SBCL is now essentially a compiler-only implementation of Common Lisp. This is a change in implementation strategy, taking advantage of the freedom "any of these facilities might share the same execution strategy" guaranteed in `clhs 3.1` (Evaluation). It does not mean SBCL can't be used interactively, and in fact the change is largely invisible to the casual user, since SBCL still can and does execute code interactively by compiling it on the fly. (It is visible if you know how to look, like using `compiled-function-p`; and it is visible in the way that SBCL doesn't have many bugs which behave differently in interpreted code than in compiled code.) What it means is that in SBCL, the `eval` function only truly "interprets" a few easy kinds of forms, such as symbols which are `boundp`. More complicated forms are evaluated by calling `compile` and then calling `funcall` on the returned result.

The direct ancestor of SBCL is the x86 port of CMUCL. This port was in some ways the most cobbled-together of all the CMUCL ports, since a number of strange changes had to be made to support the register-poor x86 architecture. Some things (like tracing and debugging) do not work particularly well there. SBCL should be able to improve in these areas (and has already improved in some other areas), but it takes a while.

On the x86 SBCL – like the x86 port of CMUCL – uses a *conservative GC*. This means that it doesn't maintain a strict separation between tagged and untagged data, instead treating some untagged data (e.g. raw floating point numbers) as possibly-tagged data and so not collecting any Lisp objects that they point to. This has some negative consequences for average time efficiency (though possibly no worse than the negative consequences of trying to implement an exact GC on a processor architecture as register-poor as the x86) and also has potentially unlimited consequences for worst-case memory efficiency. In practice, conservative GCs work reasonably well, not getting anywhere near the worst case. But they can occasionally cause odd patterns of memory usage.

The fork from CMUCL was based on a major rewrite of the system bootstrap process. CMUCL has for many years tolerated a very unusual "build" procedure which doesn't actually build the complete system from scratch but instead progressively overwrites parts of a running system with new versions. This quasi-build procedure can cause various bizarre bootstrapping hangups, especially when a major change is made to the system. It also makes the connection between the current source code and the current executable more tenuous than in other software systems – it's easy to accidentally build a CMUCL system containing characteristics not reflected in the current version of the source code. See the paper *SBCL: A Sanely Bootstrappable Common Lisp* (`https://research.gold.ac.uk/id/eprint/2336/`).

Other major changes since the fork from CMUCL include:

- SBCL has removed many CMUCL extensions, (e.g. IP networking, remote procedure call, Unix system interface, and X11 interface) from the core system. Most of these are available as contributed modules (distributed with SBCL) or third-party modules instead.
- SBCL has deleted or deprecated some nonstandard features and code complexity which helped efficiency at the price of maintainability. For example, the SBCL compiler no longer implements memory pooling internally (and so is simpler and more maintainable, but generates more garbage and runs more slowly).

19.2 Common Lisp Resources

19.2.1 Language Reference

CLHS (Common Lisp HyperSpec) is a hypertext version of the ANSI standard, made freely available by LispWorks – an invaluable reference.

See `https://www.lispworks.com/documentation/HyperSpec/Front/index.htm`.

19.2.2 Internet Community

IRC channels on `https://libera.chat/`:

- `#common-lisp`: *Common Lisp, the #1=(programmable . #1#) programming language*
- `#lispcafe`: *The Lisp Café; sit down, have a drink, chat about anything, and enjoy your stay.* | `https://www.cliki.net/lispcafe` | *Be insuperable to each other*
- `#sbcl`: *Steel Bank Common Lisp Dev Hangout*

You can use `https://web.libera.chat` or a normal IRC client.

Also, see `https://www.reddit.com/r/Common_Lisp/`, as well as `https://www.lisp.org` and `https://cliki.net`, which contain numerous pointers places in the net where lispers talk shop.

19.2.3 Third-party Libraries

For a wealth of information about free Common Lisp libraries and tools we recommend checking out *CLiki*: `https://cliki.net/`.
The most popular library manager is Quicklisp: `https://www.quicklisp.org/beta/`.

19.2.4 Common Lisp Books

If you're not a programmer and you're trying to learn, many introductory Lisp books are available. However, we don't have any standout favorites.
If you are an experienced programmer in other languages but need to learn about Common Lisp, some books stand out:

- *Practical Common Lisp* by Peter Seibel

  An excellent introduction to the language, covering both the basics and "advanced topics" like macros, CLOS, and packages. Available both in print format and on the web: `https://gigamonkeys.com/book/`.

- *Paradigms Of Artificial Intelligence Programming* by Peter Norvig

  Good information on general Common Lisp programming, and many nontrivial examples. Whether or not your work is AI, it's a very good book to look at.

- *On Lisp* by Paul Graham

  An in-depth treatment of macros, but not recommended as a first Common Lisp book, since it is slightly pre-ANSI so you need to be on your guard against non-standard usages, and since it doesn't really even try to cover the language as a whole, focusing solely on macros. Downloadable from `https://www.paulgraham.com/onlisp.html`.

- *Object-Oriented Programming In Common Lisp* by Sonya Keene

  With the exception of *Practical Common Lisp*, most introductory books don't emphasize CLOS. This one does. Even if you're very knowledgeable about object oriented programming in the abstract, it's worth looking at this book if you want to do any OO in Common Lisp. Some abstractions in CLOS (especially multiple dispatch) go beyond anything you'll see in most OO systems, and there are a number of lesser differences as well. This book tends to help with the culture shock.

- *Art Of Metaobject Programming* by Gregor Kiczales et al.

  Currently the prime source of information on the Common Lisp Metaobject Protocol, which is supported by SBCL. Section 2 (Chapters 5 and 6) are freely available at `http://mop.lisp.se/www.alu.org/mop/`.

Appendix A Concept Index

A

actual source 20, 21
arithmetic, hardware 56
arithmetic, modular 56
asynchronous timeout 84

B

basic block 35
block compilation 49
block compilation declarations 50
block compilation, debugger implications 31
block, basic 35
block, start location 35
breakpoint 36

C

call closure 48
call, full 47, 49
call, inline 50
call, let 47
call, local 47, 49
call, local self-recursive 47
call, local tail 48
call, tail 31, 46, 48
CheneyGC 57, 162
cleanup, stack frame kind 31
closure 33, 48, 53
closure, call 48
code location unknown 32
compatibility with other Lisps 24
compilation, block 49
compilation-speed, optimization quality .. 25, 33, 36, 52
compile-time type error 26
compiler message 19
compiler message severity 19
compiler message verbosity 22
compiler read error 27
conservative garbage collector 57, 162
constant folding 43
control optimization 43

D

dead code elimination 43, 44
deadline 84
debug variable, value availabilty 33
debug variables 32
debug, optimization quality . 5, 25, 32, 33, 35, 36, 37, 46
debugger 29
debugger policy 35
debugger, disabling 39
debugger, enabling 39
debugging 28
decimal syntax for rationals 60
declaration 112
deprecation 153
disabling debugger 39
disabling LDB 11, 39

E

efficicency 41
enabling debugger 39
enabling LDB 39
entry point, external 31
error, run-time 32
existing programs, getting them to run 24
expansion, inline 50
extended package prefix syntax 60
external entry point 31
external format 87, 92, 105
external, stack frame kind 31

F

finalization 62
format, external 87, 92, 105
full call 47, 49
function call, full 47, 49
function call, inline 50
function call, let 47
function call, local 47, 49
function call, local self-recursive 47
function call, local tail 48
function call, tail 31, 46, 48
function, tracing 39

G

garbage collection 62
garbage collector, conservative 57, 162
garbage collector, generational 57, 100, 135, 161
GENCGC 57, 100, 135, 161
generational garbage collector 57, 100, 135, 161

H

hash table 80

I

inline expansion 27, 50
inline expansion, recording 51
inline, semi 36, 52
inlining 50
interning symbols 60
interpreter 28
interrupt 32

L

LDB, disabling ... 11, 39
LDB, enabling ... 39
let call ... 47
let optimization ... 42
local call ... 47, 49
local call, self-recursive ... 47
local call, tail ... 48
lock, package ... 60, 111
logical pathname ... 103

M

macroexpansion ... 22
macroexpansion, errors during ... 27
message, compiler ... 19
modular arithmetic ... 56
multiple values optimization ... 45

N

NFKC ... 60
nicknames, package-local ... 61
normalization form compatibility composition ... 60
normalization of strings ... 78
normalization of symbol name ... 60

O

open-coding ... 27
optimization quality compilation-speed ... 25, 33, 36, 52
optimization quality debug ... 5, 25, 32, 33, 35, 36, 37, 46
optimization quality safety ... 23, 24, 25, 32, 54
optimization quality space ... 25, 36, 52
optimization quality speed ... 25, 33, 35, 36, 52
optimization, constant folding ... 43
optimization, control ... 43
optimization, dead code elimination ... 43, 44
optimization, inlining ... 50
optimization, let ... 42
optimization, multiple values ... 45
optimization, tail call ... 31, 35, 46
optimization, unreachable code deletion ... 44
optimization, unused expression elimination ... 43
optional, stack frame kind ... 31
original source ... 20, 21

P

package lock ... 60, 111
package prefix syntax, extended ... 60
package-local nicknames ... 61
pathname ... 102
pathname, logical ... 103
policy, debugger ... 35
precise type checking ... 23
processing path ... 20, 21
profiling ... 132
profiling, deterministic ... 133
profiling, statistical ... 133

R

random number generation ... 82
rational, decimal syntax ... 60
read error, compiler ... 27
Read-Eval-Print Loop ... 13, 16, 138
reader extensions ... 60
recursion, tail ... 46
REPL ... 13, 16, 138
return values ... 49
run-time error ... 32

S

safety, optimization quality ... 23, 24, 25, 32, 54
semi-inline ... 36, 52
severity of compiler message ... 19
single-stepping ... 38
slot access ... 52
slot, unbound ... 66, 68
source location in debugger ... 34
source optimization ... 42
source transform ... 22
source, actual ... 20, 21
source, original ... 20, 21
space, optimization quality ... 25, 36, 52
speed, optimization quality ... 25, 33, 35, 36, 52
stack frame ... 30
static functions ... 27
stepper ... 38
symbol name normalization ... 60
symbols, interning ... 60
synchronous timeout ... 84

T

tail call ... 31, 46, 48
tail call optimization ... 31, 35, 46
tail recursion ... 46
TCO ... 31, 35, 46
timeout parameters ... 83
timeout, asynchronous ... 84
timeout, synchronous ... 84
tracing ... 39
type checking, precise ... 23
type error, compile-time ... 26
types, portability ... 24

U

unbound slot ... 66, 68
Unicode ... 60, 76
unknown code location ... 32
unreachable code deletion ... 44
unused expression elimination ... 43

V

variable, debugger access ... 32
variable, value availabilty in debugger ... 33
verbosity of compiler messages ... 22

W

weak pointers 63

Appendix B Function and Macro Index

A

abort-thread [sb-thread] 116
add-implementation-package [sb-ext] 115
add-package-local-nickname [sb-ext] 61
addr [sb-alien] 94
adjust-sequence [sb-sequence] 69
age [sb-unicode] 77
alien-callable-function [sb-alien] 100
alien-funcall [sb-alien] 98
alien-funcall-into [sb-alien] 98
alien-sap [sb-alien] 94
allocation-information [sb-introspect] 148
alphabetic-p [sb-unicode] 77
array-storage-vector [sb-ext] 85
assert-version->= [sb-ext] 86
atomic-decf [sb-ext] 119
atomic-incf [sb-ext] 119
atomic-pop [sb-ext] 120
atomic-push [sb-ext] 120
atomic-update [sb-ext] 120

B

barrier [sb-thread] 126
bidi-class [sb-unicode] 76
bidi-mirroring-glyph [sb-unicode] 77
bytes-consed-between-gcs [sb-ext] 63

C

cancel-deadline [sb-sys] 84
cancel-finalization [sb-ext] 63
cas [sb-ext] 121
case-ignorable-p [sb-unicode] 77
cased-p [sb-unicode] 77
casefold [sb-unicode] 78
cast [sb-alien] 94
char-block [sb-unicode] 77
clear-coverage [sb-cover] 142
clear-semaphore-notification [sb-thread] 124
close [common-lisp] 106
close-gate [sb-concurrency] 140
combining-class [sb-unicode] 76
compare-and-swap [sb-ext] 120
compile [common-lisp] 17
compile-file [common-lisp] 17
concatenate [sb-sequence] 70
condition-broadcast [sb-thread] 126
condition-notify [sb-thread] 125
condition-wait [sb-thread] 125
confusable-p [sb-unicode] 79
copy-seq [sb-sequence] 70
count [sb-sequence] 70
count-if [sb-sequence] 70
count-if-not [sb-sequence] 70

D

decimal-value [sb-unicode] 76
default-ignorable-p [sb-unicode] 78
defer-deadline [sb-sys] 84
defglobal [sb-ext] 56
define-alien-callable [sb-alien] 99
define-alien-routine [sb-alien] 99
define-alien-variable [sb-alien] 96
define-hash-table-test [sb-ext] 81
define-source-context [sb-ext] 23
definition-source-character-offset [sb-introspect] 146
definition-source-file-write-date [sb-introspect] 146
definition-source-form-number [sb-introspect] 146
definition-source-form-path [sb-introspect] 146
definition-source-pathname [sb-introspect] 145
definition-source-plist [sb-introspect] 146
deftype-lambda-list [sb-introspect] 147
delete [sb-sequence] 71
delete-directory [sb-ext] 85
delete-duplicates [sb-sequence] 71
delete-if [sb-sequence] 71
delete-if-not [sb-sequence] 71
dequeue [sb-concurrency] 139
deref [sb-alien] 94
describe-compiler-policy [sb-ext] 26
digit-value [sb-unicode] 76
disable-debugger [sb-ext] 39
do-groups [sb-posix] 153
do-passwds [sb-posix] 153
documentation [common-lisp] 86
dosequence [sb-sequence] 69
dynamic-space-size [sb-ext] 63

E

east-asian-width [sb-unicode] 77
ed [common-lisp] 80
elt [sb-sequence] 69
emptyp [sb-sequence] 69
enable-debugger [sb-ext] 39
enqueue [sb-concurrency] 139
exit [sb-ext] 73
extern-alien [sb-alien] 96

F

file-descriptor [sb-posix] 151
filename [sb-posix] 151
fill [sb-sequence] 70
finalize [sb-ext] 62
find [sb-sequence] 70
find-definition-source [sb-introspect] 146
find-definition-sources-by-name [sb-introspect] 146
find-function-callees [sb-introspect] 147
find-function-callers [sb-introspect] 147
find-if [sb-sequence] 71
find-if-not [sb-sequence] 71
free-alien [sb-alien] 95
frlock-name [sb-concurrency] 141
frlock-read [sb-concurrency] 141
frlock-read-begin [sb-concurrency] 141
frlock-read-end [sb-concurrency] 141
frlock-write [sb-concurrency] 141
function-lambda-list [sb-introspect] 147
function-type [sb-introspect] 147

G

gate-name [sb-concurrency] 140
gate-open-p [sb-concurrency] 140
gatep [sb-concurrency] 140
gc [sb-ext] 62
gc-logfile [sb-ext] 64
general-category [sb-unicode] 76
generation-average-age [sb-ext] 64
generation-bytes-allocated [sb-ext] 64
generation-bytes-consed-between-gcs [sb-ext] . 64
generation-minimum-age-before-gc [sb-ext] 64
generation-number-of-gcs [sb-ext] 64
generation-number-of-gcs-before-promotion [sb-ext] 64
get-bytes-consed [sb-ext] 64
get-cas-expansion [sb-ext] 121
get-errno [sb-alien] 96
get-foreground [sb-thread] 126
get-host-by-address [sb-bsd-sockets] 132
get-host-by-name [sb-bsd-sockets] 132
get-protocol-by-name [sb-bsd-sockets] 132
get-time-of-day [sb-ext] 86
getcwd [sb-posix] 152
grab-frlock-write-lock [sb-concurrency] 142
grab-mutex [sb-thread] 123
grapheme-break-class [sb-unicode] 78
graphemes [sb-unicode] 79

H

hangul-syllable-type [sb-unicode] 77
hash-table-synchronized-p [sb-ext] 82
hash-table-weakness [sb-ext] 82
hex-digit-p [sb-unicode] 78
host-ent-address [sb-bsd-sockets] 132

I

ideographic-p [sb-unicode] 77
int-sap [sb-sys] 94
interrupt-thread [sb-thread] 117
iterator-copy [sb-sequence] 72
iterator-element [sb-sequence] 72
iterator-endp [sb-sequence] 72
iterator-index [sb-sequence] 72
iterator-step [sb-sequence] 72

J

join-thread [sb-thread] 117

L

length [sb-sequence] 69
line-break-class [sb-unicode] 78
lines [sb-unicode] 79
list-all-threads [sb-thread] 116
list-all-timers [sb-ext] 128
list-mailbox-messages [sb-concurrency] 139
list-queue-contents [sb-concurrency] 139
load-shared-object [sb-alien] 97
lock-package [sb-ext] 114
lowercase [sb-unicode] 78
lowercase-p [sb-unicode] 77

M

mailbox-count [sb-concurrency] 140
mailbox-empty-p [sb-concurrency] 140
mailbox-name [sb-concurrency] 140
mailboxp [sb-concurrency] 140
main-thread [sb-thread] 116
main-thread-p [sb-thread] 116
make-alien [sb-alien] 94
make-alien-string [sb-alien] 95
make-frlock [sb-concurrency] 141
make-gate [sb-concurrency] 140
make-hash-table [common-lisp] 80
make-inet-address [sb-bsd-sockets] 131
make-inet6-address [sb-bsd-sockets] 132
make-listener-thread [sb-thread] 126
make-mailbox [sb-concurrency] 140
make-mutex [sb-thread] 122
make-queue [sb-concurrency] 139
make-semaphore [sb-thread] 123
make-semaphore-notification [sb-thread] 124
make-sequence-iterator [sb-sequence] 71
make-sequence-like [sb-sequence] 69
make-simple-sequence-iterator [sb-sequence] ... 72
make-thread [sb-thread] 116
make-timer [sb-ext] 128
make-waitqueue [sb-thread] 125
make-weak-pointer [sb-ext] 63
map [sb-sequence] 70
map-root [sb-introspect] 148
map-traces [sb-sprof] 135
math-p [sb-unicode] 77
md5sum-file [sb-md5] 150

md5sum-sequence [sb-md5] 150
md5sum-stream [sb-md5] 150
md5sum-string [sb-md5] 150
merge [sb-sequence] 70
merge-coverage [sb-cover] 143
merge-coverage-from-file [sb-cover] 143
method-combination-lambda-list
[sb-introspect] 147
mirrored-p [sb-unicode] 77
mismatch [sb-sequence] 70
mutex-name [sb-thread] 122
mutex-owner [sb-thread] 122
mutex-value [sb-thread] 123

N

native-namestring [sb-ext] 103
native-pathname [sb-ext] 103
non-blocking-mode [sb-bsd-sockets] 130
normalize-string [sb-unicode] 78
normalized-p [sb-unicode] 78
nreverse [sb-sequence] 70
nsubstitute [sb-sequence] 71
nsubstitute-if [sb-sequence] 71
nsubstitute-if-not [sb-sequence] 71
numeric-value [sb-unicode] 77

O

octets-to-string [sb-ext] 89
open-gate [sb-concurrency] 140

P

package-implemented-by-list [sb-ext] 115
package-implements-list [sb-ext] 115
package-local-nicknames [sb-ext] 61
package-locally-nicknamed-by-list [sb-ext] 61
package-locked-error-symbol [sb-ext] 114
package-locked-p [sb-ext] 114
parse-native-namestring [sb-ext] 103
position [sb-sequence] 71
position-if [sb-sequence] 71
position-if-not [sb-sequence] 71
posix-environ [sb-ext] 73
posix-getenv [sb-ext] 73
process-alive-p [sb-ext] 75
process-close [sb-ext] 76
process-core-dumped [sb-ext] 76
process-error [sb-ext] 75
process-exit-code [sb-ext] 76
process-input [sb-ext] 75
process-kill [sb-ext] 76
process-output [sb-ext] 75
process-p [sb-ext] 75
process-status [sb-ext] 75
process-wait [sb-ext] 75
profile [sb-profile] 133
profile-call-counts [sb-sprof] 137
proplist-p [sb-unicode] 77

Q

queue-count [sb-concurrency] 139
queue-empty-p [sb-concurrency] 139
queue-name [sb-concurrency] 139
queuep [sb-concurrency] 139

R

readlink [sb-posix] 152
readtable-normalization [sb-ext] 60
receive-message [sb-concurrency] 140
receive-message-no-hang [sb-concurrency] 140
receive-pending-messages [sb-concurrency] 140
reduce [sb-sequence] 70
release-foreground [sb-thread] 126
release-frlock-write-lock [sb-concurrency] 142
release-mutex [sb-thread] 123
remove [sb-sequence] 71
remove-duplicates [sb-sequence] 71
remove-if [sb-sequence] 71
remove-if-not [sb-sequence] 71
remove-implementation-package [sb-ext] 115
remove-package-local-nickname [sb-ext] 62
replace [sb-sequence] 70
report [sb-cover] 142
report [sb-profile] 133
report [sb-sprof] 136
require [common-lisp] 79
reset [sb-profile] 133
reset [sb-sprof] 136
reset-coverage [sb-cover] 142
restore-coverage [sb-cover] 143
restore-coverage-from-file [sb-cover] 143
restrict-compiler-policy [sb-ext] 26
return-from-thread [sb-thread] 116
reverse [sb-sequence] 70
rotate-byte [sb-rotate-byte] 153
run-program [sb-ext] 73

S

sample-pc [sb-sprof] 136
sap-alien [sb-alien] 94
sap-ref-32 [sb-sys] 94
sap= [sb-sys] 94
save-coverage [sb-cover] 142
save-coverage-in-file [sb-cover] 143
save-lisp-and-die [sb-ext] 14
schedule-timer [sb-ext] 128
script [sb-unicode] 77
search [sb-sequence] 70
search-roots [sb-ext] 64
seed-random-state [sb-ext] 82
semaphore-count [sb-thread] 124
semaphore-name [sb-thread] 124
semaphore-notification-status [sb-thread] 124
send-message [sb-concurrency] 140
sentence-break-class [sb-unicode] 78
sentences [sb-unicode] 79
set-sbcl-source-location [sb-ext] 103
signal-semaphore [sb-thread] 123
slot [sb-alien] 94

socket-accept [sb-bsd-sockets] 129
socket-bind [sb-bsd-sockets] 129
socket-close [sb-bsd-sockets] 130
socket-connect [sb-bsd-sockets] 129
socket-error [sb-bsd-sockets] 130
socket-listen [sb-bsd-sockets] 130
socket-make-stream [sb-bsd-sockets] 130
socket-name [sb-bsd-sockets] 129
socket-open-p [sb-bsd-sockets] 130
socket-peername [sb-bsd-sockets] 129
socket-receive [sb-bsd-sockets] 129
socket-send [sb-bsd-sockets] 129
socket-shutdown [sb-bsd-sockets] 130
sockopt-broadcast [sb-bsd-sockets] 131
sockopt-bsd-compatible [sb-bsd-sockets] 131
sockopt-debug [sb-bsd-sockets] 131
sockopt-dont-route [sb-bsd-sockets] 131
sockopt-keep-alive [sb-bsd-sockets] 131
sockopt-oob-inline [sb-bsd-sockets] 131
sockopt-pass-credentials [sb-bsd-sockets] 131
sockopt-reuse-address [sb-bsd-sockets] 130
sockopt-tcp-nodelay [sb-bsd-sockets] 131
soft-dotted-p [sb-unicode] 78
sort [sb-sequence] 70
stable-sort [sb-sequence] 70
start-profiling [sb-sprof] 136
step [common-lisp] 38
stop-profiling [sb-sprof] 137
stream-advance-to-column [sb-gray] 107
stream-clear-input [sb-gray] 106
stream-clear-output [sb-gray] 107
stream-element-type [common-lisp] 106
stream-file-position [sb-gray] 106
stream-finish-output [sb-gray] 107
stream-force-output [sb-gray] 107
stream-fresh-line [sb-gray] 107
stream-line-column [sb-gray] 107
stream-line-length [sb-gray] 107
stream-listen [sb-gray] 107
stream-peek-char [sb-gray] 106
stream-read-byte [sb-gray] 108
stream-read-char [sb-gray] 106
stream-read-char-no-hang [sb-gray] 106
stream-read-line [sb-gray] 106
stream-read-sequence [sb-gray] 106
stream-start-line-p [sb-gray] 107
stream-terpri [sb-gray] 108
stream-unread-char [sb-gray] 107
stream-write-byte [sb-gray] 108
stream-write-char [sb-gray] 108
stream-write-sequence [sb-gray] 107
stream-write-string [sb-gray] 108
string-to-octets [sb-ext] 89
subseq [sb-sequence] 70
substitute [sb-sequence] 71
substitute-if [sb-sequence] 71
substitute-if-not [sb-sequence] 71
symbol-value-in-thread [sb-thread] 118
syslog [sb-posix] 152

T

terminate-thread [sb-thread] 118
thread-alive-p [sb-thread] 116
thread-error-thread [sb-thread] 118
thread-name [sb-thread] 116
thread-yield [sb-thread] 117
timer-name [sb-ext] 128
timer-scheduled-p [sb-ext] 128
titlecase [sb-unicode] 78
trace [common-lisp] 39
try-semaphore [sb-thread] 123

U

unencapsulated-function [sb-ext] 86
unicode-1-name [sb-unicode] 77
unicode-equal [sb-unicode] 79
unicode< [sb-unicode] 78
unicode<= [sb-unicode] 79
unicode= [sb-unicode] 79
unicode> [sb-unicode] 79
unicode>= [sb-unicode] 79
unload-shared-object [sb-alien] 98
unlock-package [sb-ext] 114
unprofile [sb-profile] 133
unprofile-call-counts [sb-sprof] 137
unschedule-timer [sb-ext] 128
untrace [common-lisp] 41
uppercase [sb-unicode] 78
uppercase-p [sb-unicode] 77
use-pax [sb-manual] 149

V

valid-function-name-p [sb-introspect] 147
var [sb-debug] 32

W

wait-for [sb-ext] 84
wait-on-gate [sb-concurrency] 140
wait-on-semaphore [sb-thread] 123
waitqueue-name [sb-thread] 125
weak-pointer-value [sb-ext] 63
whitespace-p [sb-unicode] 77
who-binds [sb-introspect] 147
who-calls [sb-introspect] 147
who-macroexpands [sb-introspect] 147
who-references [sb-introspect] 147
who-sets [sb-introspect] 147
who-specializes-directly [sb-introspect] 148
who-specializes-generally [sb-introspect] 148
with-alien [sb-alien] 95
with-alien-callable [sb-alien] 100
with-compilation-unit [common-lisp] 18
with-deadline [sb-sys] 84
with-locked-hash-table [sb-ext] 82
with-mutex [sb-thread] 122
with-new-session [sb-thread] 126
with-profiling [sb-sprof] 135
with-recursive-lock [sb-thread] 122

with-sampling [sb-sprof] 135
with-sequence-iterator [sb-sequence] 72
with-sequence-iterator-functions
[sb-sequence] 72
with-timeout [sb-ext] 85
with-unlocked-packages [sb-ext] 115
without-package-locks [sb-ext] 115
word-break-class [sb-unicode] 78
words [sb-unicode] 79

Appendix C Variable and Constant Index

A

after-gc-hooks [sb-ext] 62

B

block-compile-default [sb-ext] 18

C

command-char [sb-aclrepl] 138
compile-progress [sb-ext] 23
compiler-print-variable-alist [sb-ext] 22
core-pathname [sb-ext] 15
current-thread [sb-thread] 116

D

debug-beginner-help-p [sb-debug] 29
debug-print-variable-alist [sb-ext] 30
default-c-string-external-format [sb-ext] ... 88
default-external-format [sb-ext] 88
default-source-external-format [sb-ext] 88

E

ed-functions [sb-ext] 80
efficiency-note-cost-threshold [sb-ext] 23
efficiency-note-limit [sb-ext] 23
enclosing-source-cutoff [sb-ext] 23
evaluator-mode [sb-ext] 28
exit-hooks [sb-ext] 14
exit-on-eof [sb-aclrepl] 138

G

gc-real-time [sb-ext] 63
gc-run-time [sb-ext] 63

I

init-hooks [sb-ext] 14
inline-expansion-limit [sb-ext] 50
invoke-debugger-hook [sb-ext] 29

M

max-history [sb-aclrepl] 138
max-samples [sb-sprof] 137
max-trace-indentation [sb-debug] 41
module-provider-functions [sb-ext] 80
muffled-warnings [sb-ext] 80

O

on-package-variance [sb-ext] 62

P

posix-argv [sb-ext] 73
prompt [sb-aclrepl] 138

S

sample-interval [sb-sprof] 137
save-hooks [sb-ext] 15
stack-allocate-dynamic-extent [sb-ext] 53
sysinit-pathname-function [sb-ext] 13

T

trace-encapsulate-default [sb-debug] 41
trace-indentation-step [sb-debug] 41
trace-report-default [sb-debug] 41

U

undefined-warning-limit [sb-ext] 23
use-short-package-name [sb-aclrepl] 138
userinit-pathname-function [sb-ext] 13

Appendix D Type Index

C

code-deletion-note [sb-ext] 19
compiler-note [sb-ext] 19

D

deadline-timeout [sb-sys] 84
definition-source [sb-introspect] 145
deprecation-condition [sb-ext] 155
deprecation-error [sb-ext] 155

E

early-deprecation-warning [sb-ext] 155

F

file-descriptor [sb-posix] 151
file-descriptor-designator [sb-posix] 151
filename [sb-posix] 151
filename-designator [sb-posix] 151
final-deprecation-warning [sb-ext] 155
flock [sb-posix] 152
frlock [sb-concurrency] 141
fundamental-binary-input-stream [sb-gray] 106
fundamental-binary-output-stream [sb-gray] ... 106
fundamental-binary-stream [sb-gray] 105
fundamental-character-input-stream [sb-gray] . 106
fundamental-character-output-stream [sb-gray] 106
fundamental-character-stream [sb-gray] 106
fundamental-input-stream [sb-gray] 105
fundamental-output-stream [sb-gray] 105
fundamental-stream [sb-gray] 105

G

gate [sb-concurrency] 140
group [sb-posix] 152

H

host-ent [sb-bsd-sockets] 132

I

inet-socket [sb-bsd-sockets] 131
inet6-socket [sb-bsd-sockets] 131
interrupt-thread-error [sb-thread] 119

J

join-thread-error [sb-thread] 119

L

late-deprecation-warning [sb-ext] 155
local-abstract-socket [sb-bsd-sockets] 132
local-socket [sb-bsd-sockets] 132

M

mailbox [sb-concurrency] 139
mutex [sb-thread] 122

P

package-lock-violation [sb-ext] 114
package-locked-error [sb-ext] 114
passwd [sb-posix] 152
protocol-unimplemented [sb-sequence] 69

Q

queue [sb-concurrency] 139

S

semaphore [sb-thread] 123
semaphore-notification [sb-thread] 124
socket [sb-bsd-sockets] 129
stat [sb-posix] 152
symbol-package-locked-error [sb-ext] 114
symbol-value-in-thread-error [sb-thread] 119

T

termios [sb-posix] 152
thread [sb-thread] 116
thread-error [sb-thread] 118
timeout [sb-ext] 85
timer [sb-ext] 128
timeval [sb-posix] 152

W

waitqueue [sb-thread] 125

Appendix E Declaration Index

A

always-bound [sb-ext] 57

D

deprecated [sb-ext] 156
disable-package-locks [sb-ext] 114

E

enable-package-locks [sb-ext] 114
end-block [sb-ext] 50

G

global [sb-ext] 56

M

maybe-inline [sb-ext] 52
muffle-conditions [sb-ext] 22

S

start-block [sb-ext] 50

U

unmuffle-conditions [sb-ext] 22

Colophon

This manual is maintained as part of the Section 17.6 [sb manual], page 149, contrib. `SB-MANUAL` can generate Texinfo, which is in turn converted into other formats (e.g. HTML or pdf). If you're *reading* this manual in one of these, that's fine, but if you want to *modify* this manual, you are strongly advised to seek the sources (most live under `contrib/sb-manual/doc/`) and submit a patch.

Also, see `https://fixnum.com/` for the unofficial, PAX-generated version with a lot more links and with initial values of variables.