GNU SETL Library Reference

Edition 0.1.2, for GNU SETL Version 0.1.2
Updated 9 July 2009

by dB

Table of Contents

1 The Library 1

1.1 Size of set; length of string or tuple 1
1.2 Numeric multiplication; set intersection; string or tuple replication 1
1.3 Exponentiation 2
1.4 Numeric affirmation or addition; set union; string or tuple concatenation 2
1.5 Numeric negation or subtraction; set difference 2
1.6 Numeric quotient 3
1.7 Equality and inequality 3
1.8 Order-based comparisons 3
1.9 Query (short-circuiting `om` test) 4
1.10 Numeric absolute value; integer value of character 5
1.11 Accept connection on socket 5
1.12 Arc cosine 5
1.13 Logical conjunction 5
1.14 Extract initial character using character set 6
1.15 Arbitrary element of set 6
1.16 Arc sine 6
1.17 Arc tangent 6
1.18 Arc tangent of quotient 6
1.19 Bitwise logical operators 7
1.20 Extract initial substring using character set 7
1.21 Indirect call 7
1.22 Call C function 7
1.23 Ceiling (lowest integer upper bound) 8
1.24 Character representation of small integer 8
1.25 Change directory 8
1.26 Clear system error indicator 8
1.27 Elapsed time in milliseconds 9
1.28 Close stream 9
1.29 Constants for use with `close` 10
1.30 Command-line arguments 10
1.31 Command name 10
1.32 Cosine 10
1.33 Hyperbolic cosine 10
1.34 Date and time of day 11
1.35 Type of denotation within string 11
1.36 Integer division 11
1.37 Domain of map 11
1.38 Duplicate file descriptor 11
1.39 Test for end of file 12
1.40 Test for integer divisible by 2 12

1.41 Replace current process 12
1.42 Natural exponential (e raised to a power) 12
1.43 Predefined "false" boolean value 13
1.44 Format date and time 13
1.45 Test for existence of file 13
1.46 Return filename of open stream 13
1.47 Return fd of open stream 13
1.48 Filter string through external command 14
1.49 Truncate floating-point number to integer 14
1.50 Format number with optional decimal point 14
1.51 Convert number to floating-point 15
1.52 Format number in scientific notation 15
1.53 Floor (greatest integer lower bound) 15
1.54 Flush output buffer 15
1.55 Fork process 16
1.56 Take arbitrary element from set 16
1.57 Take from beginning of string or tuple 16
1.58 Take from end of string or tuple 16
1.59 Size of file in bytes 17
1.60 Get one or more lines from `stdin` 17
1.61 Get one or more lines from input stream 17
1.62 Get one or more values from input stream 17
1.63 Get character from input stream 18
1.64 Get character from `stdin` 18
1.65 Get effective group ID 18
1.66 Get value of environment variable 18
1.67 Get effective user ID 18
1.68 Read stream up to the end 18
1.69 Get real group ID 19
1.70 Get line from input stream 19
1.71 Get fixed number of characters from input stream 19
1.72 Get process group ID 20
1.73 Direct-access read 20
1.74 Get real user ID 20
1.75 Current working directory 20
1.76 Pathname wildcard expansion 20
1.77 Find all occurrences of pattern in string 21
1.78 Replace patterns in string 21
1.79 Convert string to hexadecimal 21
1.80 Local host address 21
1.81 Local hostname 21
1.82 Integer equivalent of character 21
1.83 Implication 22
1.84 Membership test 22
1.85 Subset test 23
1.86 Integer quotient type switch 23
1.87 Internet host addresses 23
1.88 Internet hostnames 23

1.89 Type testers ... 23
1.90 Test for open stream ... 24
1.91 Concatenate tuple of strings, with delimiter ... 24
1.92 Send signal to process ... 24
1.93 Last error message from system routine ... 25
1.94 Extract initial substring by length ... 25
1.95 Set less one element ... 25
1.96 Map less one domain element ... 25
1.97 Test for existence of file or symlink ... 26
1.98 Create hard link ... 26
1.99 Natural logarithm ... 26
1.100 Pad string on left with blanks ... 26
1.101 Regular expression recognition switch ... 26
1.102 Find first occurrence of pattern in string ... 27
1.103 Extract initial substring by exact match ... 27
1.104 Numeric maximum ... 27
1.105 Allocate machine memory ... 27
1.106 Free machine memory ... 27
1.107 Reallocate machine memory ... 28
1.108 Copy machine memory ... 28
1.109 Machine memory read ... 28
1.110 Machine memory write ... 28
1.111 Numeric minimum ... 28
1.112 Integer modulus; symmetric set difference ... 29
1.113 Number of arguments passed by caller ... 29
1.114 Create new atom ... 29
1.115 Non-error message ... 29
1.116 Logical negation ... 29
1.117 Extract initial character using character set ... 29
1.118 Membership test ... 30
1.119 All subsets having a given size ... 30
1.120 Print to `stdout` with no trailing newline ... 30
1.121 Print to output stream with no trailing newline ... 30
1.122 Test for integer not divisible by 2 ... 30
1.123 The “undefined” value ... 31
1.124 Open a stream ... 31
1.124.1 Arguments to `open` ... 31
1.124.2 Compatibility of `open` with CIMS SETL and SETL2 ... 32
1.124.3 Results of `open` ... 32
1.124.4 Buffering ... 32
1.124.5 Testing for openness ... 33
1.124.6 Automatic opening (and closing) of files ... 33
1.124.7 TCP sockets ... 33
1.124.8 UDP sockets ... 33
1.124.9 Pumps and tty-pumps ... 33
1.124.10 Signal streams ... 34
1.124.11 Timer streams ... 34
1.124.12 Predefined streams (needing no `open`) ... 35

1.124.13 Alternative *how* arguments for `open` 35
1.125 Logical disjunction 37
1.126 Byte packing 37
1.127 Peek at next character in input stream 37
1.128 Peek at next character in `stdin` 38
1.129 Peer host address 38
1.130 Peer hostname 38
1.131 Peer port number 38
1.132 Test for existence of processes 38
1.133 Process ID of current or child process 39
1.134 Create primitive pipe 39
1.135 Pipe from child process 39
1.136 Pipe to child process 40
1.137 Retrieve port number 40
1.138 Power set 40
1.139 "Prettify" string 40
1.140 Print to `stdout` 40
1.141 Print to output stream 40
1.142 Bidirectional stream to child process 41
1.143 Put lines on `stdout` 41
1.144 Put lines on output stream 41
1.145 Put values on output stream 41
1.146 Put characters on output stream 42
1.147 Put characters on `stdout` 42
1.148 Set environment variable 42
1.149 Put characters on output stream 42
1.150 Put lines on output stream 42
1.151 Direct-access write 43
1.152 Pseudo-random numbers and selections 43
1.153 Range of map 43
1.154 Extract final character using character set 43
1.155 Extract final substring using character set 44
1.156 Extract final substring by length 44
1.157 Extract final substring by exact match 44
1.158 Extract final character using character set 44
1.159 Extract final substring using character set 44
1.160 Get one or more values from `stdin` 44
1.161 Get one or more values from input stream 45
1.162 Symbolic link referent 45
1.163 Get one or more values from a string 45
1.164 Receive datagram on UDP client socket 45
1.165 Receive datagram on UDP server socket 46
1.166 Receive file descriptor 46
1.167 Integer remainder 47
1.168 Reverse string 47
1.169 Rewind direct-access stream 47
1.170 Round to nearest integer 48
1.171 Create procedure reference 48

1.172 Pad string on right with blanks 48
1.173 Reposition direct-access stream 48
1.174 Wait for I/O event or timeout 49
1.175 Send datagram on UDP client socket 51
1.176 Send datagram on UDP server socket 51
1.177 Send file descriptor 51
1.178 Set effective group ID 51
1.179 Set environment variable 51
1.180 Set effective user ID 52
1.181 Set group ID 52
1.182 Set process group ID 52
1.183 Set random seed 52
1.184 Set user ID 52
1.185 Determine the type of integer quotients 53
1.186 Recognize regular expressions 53
1.187 Disable I/O in one or both directions 53
1.188 Constants for use with `shutdown` 54
1.189 Numeric sign 54
1.190 Sine 54
1.191 Hyperbolic sine 54
1.192 Extract initial substring by on character set 54
1.193 Split string into tuple 54
1.194 Square root 55
1.195 Child process termination status 55
1.196 Standard input, output, and error file descriptors 55
1.197 String representation of value 55
1.198 Radix-prefixed string representation of integer 56
1.199 Replace pattern in string 56
1.200 Subset test 56
1.201 Create symbolic link 57
1.202 Execute command in subshell 57
1.203 Low-level read 57
1.204 Low-level write 57
1.205 Trigonometric tangent 57
1.206 Hyperbolic tangent 58
1.207 Auto-flush output stream upon other stream input 58
1.208 Elapsed CPU time in milliseconds 58
1.209 Unique temporary filename 58
1.210 Convert string to lowercase 58
1.211 Convert string to uppercase 58
1.212 Calendar time in milliseconds 59
1.213 Predefined “true” boolean value 59
1.214 Type of SETL value 59
1.215 Set file creation mode mask 59
1.216 Push characters back on input stream 59
1.217 Push characters back on `stdin` 59
1.218 Convert from hexadecimal 59
1.219 Destroy file reference 60

1.220 Byte unpacking 60
1.221 “Unprettify” string 60
1.222 Remove environment variable definition 61
1.223 Read value from string 61
1.224 Read numeric value from string 61
1.225 Wait for any child process to terminate 61
1.226 Wait for child process to terminate 62
1.227 Format integer 62
1.228 Set plus one element 63
1.229 Write values to `stdout` 63
1.230 Write values to output stream 63

Index 64

1 The Library

This is an informal presentation of the functions and operators (the "intrinsics") built into GNU SETL.

In particular, SETL has no system of datatype declarations as of this writing, but the function prototypes (signatures) are presented here as if it were possible to declare the types of formal arguments and the type of the return value if any. Where an argument can be of any type, the keyword `var` is used.

These and most of the implicit polymorphism are plausible language extensions. But a further liberty is taken in a couple of cases where the return type depends on the *values* of the arguments, not just their types. For example, the type of `val`'s yield depends on what is in the input string. For another example, exponentiation where both arguments are of type `integer` returns a `real` when the second argument is negative. These operators are shown as pairs of prototypes, one returning `integer` and the other returning `real`. More realistic in terms of a likely language extension is that they could be declared to return a `number` that was defined as a union of numeric types.

If no return type is shown for a given intrinsic, it may be assumed to return `om` if it returns at all. Note also that `om` may be a possible return value even when the only return types shown are other than `om`. Such cases are identified in the accompanying descriptions, and usually correspond to non-fatal failures. (Language lawyers tend to dislike `om`, but it is of great practical value in the region between "wrong" and "well-defined". It also sounds nice when said slowly.)

The following intrinsics raise a run-time exception if called in "restricted" mode (see Section "`setl` '`--restricted`' option" in the *GNU SETL User Guide*): `callout`, `chdir`, `exec`, `fexists`, `filter`, `fsize`, `getegid`, `geteuid`, `getgid`, `getuid`, `getwd`, `kill` (except on processes spawned by `pump`, `pipe_from_child`, or `pipe_to_child`), `lexists`, `link`, `mem_alloc`, `mem_free`, `mem_realloc`, `mem_copy`, `mem_fetch_string`, `mem_fetch_c_string`, `mem_store_string`, `mem_store_c_string`, `setegid`, `seteuid`, `setgid`, `setpgrp`, `setuid`, `symlink`, `system`, `sys_read`, `sys_write`, `tmpnam`, `umask`, and `unlink`.

Also, `open` raises an exception in restricted mode unless the call is permitted by an '`--allow-...`' option (see Section "`setl` '`--allow-...`' option" in the *GNU SETL User Guide*) or is a request to open a timer, signal-catching, or signal-ignoring stream.

1.1 Size of set; length of string or tuple

```
op # (set) : integer
op # (string) : integer
op # (tuple) : integer
```

1.2 Numeric multiplication; set intersection; string or tuple replication

```
op * (integer, integer) : integer
op * (real, real) : real
op * (real, integer) : real
```

```
op * (integer, real) : real
op * (set, set) : set
op * (string, integer) : string
op * (integer, string) : string
op * (tuple, integer) : tuple
op * (integer, tuple) : tuple
```

For the forms involving a string or tuple, the result is the concatenation of however many copies of that string or tuple are indicated by the integer argument.

1.3 Exponentiation

```
op ** (integer, integer) : integer
op ** (integer, integer) : real
op ** (real, real) : real
op ** (real, integer) : real
op ** (integer, real) : real
```

When both arguments are of type `integer`, the return type is `real` if and only if the second argument is negative.

1.4 Numeric affirmation or addition; set union; string or tuple concatenation

```
op + (integer) : integer
op + (real) : real
op + (integer, integer) : integer
op + (real, real) : real
op + (real, integer) : real
op + (integer, real) : real
op + (set, set) : set
op + (string, string) : string
op + (tuple, tuple) : tuple
op + (string, string) : string
op + (string, var) : string
op + (var, string) : string
```

The binary forms in which one argument is a `real` and the other is an `integer` are treated as if the integer is "promoted" to a real before addition using `float` (which see for more information about floating-point overflow).

The binary forms in which one argument is a string and the other is *not* are treated as if `str` is first applied to the non-string argument to convert it preparatory to concatenation.

See also the "query" operator (`?`) regarding special-case treatment of the `+:=` operator when the left-hand argument has the initial value `om`.

1.5 Numeric negation or subtraction; set difference

```
op - (integer) : integer
op - (real) : real
op - (integer, integer) : integer
```

```
op - (real, real) : real
op - (real, integer) : real
op - (integer, real) : real
op - (set, set) : set
```

For sets, see also `less`.

1.6 Numeric quotient

```
op / (integer, integer) : real
op / (integer, integer) : integer
op / (real, real) : real
op / (real, integer) : real
op / (integer, real) : real
```

Note the return type of integer division here. By default, `integer / integer` gives a `real`.

You can force it to return a truncated (integer) result by setting `intslash` to `true`, or equivalently by calling `set_intslash` with an argument of `true`, but you do so at your peril. Consider a program that reads pairs of numbers and computes their quotients. Unless it is careful to ensure that each number (or at least one of each pair) has a `real` (floating-point) representation, say by using `float`, such a program will sometimes truncate quotients and sometimes not, depending on which input numbers happen to have decimal points in them.

The normal way to get truncated integer division in SETL is by using the `div` operator, as in the Algol family of languages. This makes explicit the fact that no fraction is expected in the result. (Even without dynamic run-time types, C and Fortran programmers often mistakenly code integer division when they intend "real" division, a frequent source of subtle bugs.)

1.7 Equality and inequality

```
op = (var, var) : boolean
op /= (var, var) : boolean
```

1.8 Order-based comparisons

```
op < (integer, integer) : boolean
op < (real, real) : boolean
op < (real, integer) : boolean
op < (integer, real) : boolean
op < (string, string) : boolean
op < (tuple, tuple) : boolean
op > (integer, integer) : boolean
op > (real, real) : boolean
op > (real, integer) : boolean
op > (integer, real) : boolean
op > (string, string) : boolean
op > (tuple, tuple) : boolean
op <= (integer, integer) : boolean
op <= (real, real) : boolean
```

```
op <= (real, integer) : boolean
op <= (integer, real) : boolean
op <= (string, string) : boolean
op <= (tuple, tuple) : boolean
op >= (integer, integer) : boolean
op >= (real, real) : boolean
op >= (real, integer) : boolean
op >= (integer, real) : boolean
op >= (string, string) : boolean
op >= (tuple, tuple) : boolean
```

String comparisons are lexicographic, and tuple comparisons are recursive. If one string or tuple is a prefix of the other, the shorter one is considered to be the lesser.

1.9 Query (short-circuiting `om` test)

```
op ? (var, var) : var
```

The expression

```
x ? y
```

is equivalent to the expression

```
if x = om then y else x end
```

if *x* has no side-effects. Thus the query operator is "short-circuited" like `and` and `or`.

For a map `hist` used to count item occurrences, the sequence

```
hist(item) ?:= 0;  -- initialize if undefined
hist(item) +:= 1;  -- accumulate count
```

was once idiomatic in SETL, but the need for the explicit initialization has proven to be more of a nuisance than a help, so this can now be abbreviated to simply

```
hist(item) +:= 1;
```

where the implicit initialization depends on the type of the argument on the right-hand side of the `+:=`. Specifically, for `+:=`, when the left-hand side is `om`, it is initialized with the additive identity for the type on the right, namely `0` for `integer`, `0.0` for `real`, the empty string (`""`) for `string`, the empty set (`{}`) for `set`, and the empty tuple (`[]`) for `tuple`.

Note that this makes

```
x +:= "some string"
```

slightly different from

```
x := x + "some string"
```

when `x` is `om`, as the latter is then equivalent to

```
x := str om + "some string"
```

or in other words

```
x := "*some string"
```

Also,

```
n +:= 1
```

gives `n` the value 1 if it is initially `om`, but

```
n := n + 1
```

is treated as an error.

1.10 Numeric absolute value; integer value of character

```
op abs (integer) : integer
op abs (real) : real
op abs (string) : integer
```

For a number, `abs` returns the magnitude.

For a string operand, `abs` is equivalent to `ichar`.

1.11 Accept connection on socket

```
proc accept (integer) : integer
proc accept (string) : integer
```

The argument must be a TCP server socket opened by `open`. The `accept` function waits for a client to connect, and then returns a new socket for that connection. The `select` function can be used to test whether an `accept` would block on the given server socket, and `pump` or `fork` is often used to help programs serve clients concurrently instead of making them queue for service.

It is possible for `accept` to fail due to conditions arising between the time of a successful `select` and the issuing of the `accept` call. In this case `accept` returns `om`.

See also `http://cs.nyu.edu/~bacon/phd-thesis/` for more on network programming in SETL.

1.12 Arc cosine

```
op acos (real) : real
op acos (integer) : real
```

The argument must be in the range -1 to +1, and the result is in radians. See also `cos`.

1.13 Logical conjunction

```
op and (boolean, boolean) : boolean
```

The expression

```
x and y
```

is equivalent to the expression

```
if x then y else false end
```

for boolean *x* and *y*, which is to say that the `and` operator is "short-circuited": it only evaluates its second argument if necessary. This makes it suitable for use as a guard against erroneous evaluations such as subscripting a tuple with a nonpositive integer. For example,

```
if i > 0 then
  if t(i) = "marker" then
    ...
  end if;
end if;
```

can be replaced by the less cumbersome

```
if i > 0 and t(i) = "marker" then
  ...
end if;
```

See also or, the "query" operator (?), and the bitwise operators such as bit_and.

1.14 Extract initial character using character set

```
proc any (rw string s, string p) : string
```

If the first character of *s* occurs anywhere in the string *p*, that character is removed from *s* and returned as the function result. Otherwise, nothing happens to *s*, and the empty string ("") is returned.

See also break, len, match, notany, span, rany, rbreak, rlen, rmatch, rnotany, and rspan.

1.15 Arbitrary element of set

```
op arb (set) : var
```

An arbitrary (which is not to say *random* but rather *nondeterministically chosen*) element of the argument set is returned. If the set is empty, om is returned.

See also from and random.

1.16 Arc sine

```
op asin (real) : real
op asin (integer) : real
```

The argument must be in the range -1 to +1, and the result is in radians.

1.17 Arc tangent

```
op atan (real) : real
op atan (integer) : real
```

The result is in radians. See also atan2.

1.18 Arc tangent of quotient

```
op atan2 (real, real) : real
op atan2 (real, integer) : real
op atan2 (integer, real) : real
op atan2 (integer, integer) : real
```

For non-zero *x*, the expression

```
y atan2 x
```

is similar to the expression

```
atan (y/x)
```

When *x* is 0, however, only the atan2 form can be used, and returns $\pi/2$ or $-\pi/2$ depending on the sign of *y*.

SETL does not currently have built-in support for complex numbers, but `atan2` gives a convenient and reliable way to obtain the phase of a complex number whose real part is *x* and whose imaginary part is *y*.

[The "magnitude", `sqrt (x**2 + y**2)`, might also be useful to have as a built-in operator. In the C math library, it is called `hypot()`.]

1.19 Bitwise logical operators

```
op bit_and (integer, integer) : integer
op bit_not (integer) : integer
op bit_or (integer, integer) : integer
op bit_xor (integer, integer) : integer
```

These operators treat their operands as binary integers, that is to say as bit patterns. They are machine-dependent with respect to word size, but have some use in expressing certain algorithms that do specialized tricks with bit patterns.

See also `and`, `or`, and `not`.

1.20 Extract initial substring using character set

```
proc break (rw string s, string p) : string
```

If *s* contains a character that appears in *p*, the substring of *s* up to but not including that character is removed from the beginning of *s* and returned as the function result, while *s* itself is updated to reflect the loss. Otherwise (no character from *p* appears in *s*), the function result is the input value of *s*, and *s* is reduced to the empty string (`""`) by the operation.

See also `any`, `len`, `match`, `notany`, `span`, `rany`, `rbreak`, `rlen`, `rmatch`, `rnotany`, and `rspan`.

1.21 Indirect call

```
proc call (proc_ref, ...) : var
```

All arguments to `call` are read-only, so the procedure referenced through the `proc_ref` value must not have any `rw` or `wr` arguments. It may, however, *return* a result of any type, including `tuple`, so multiple values can be returned at relatively minor syntactic cost to the caller. Example:

```
f_ref := routine f;
  ...
[x, y, z] := call (f_ref, a, b);
  ...
proc f (v, w);
  ...
  return [p, q, r];
end proc f;
```

1.22 Call C function

```
proc callout (integer, om, tuple) : string
```

This is a SETL2 compatibility feature of dubious value. It is easier to use the SETL customization protocol, and you get a superior result (the interface you want) that way. The SETL2 callout interface is very awkward, and was invented so that the SETL2 interpreter could be extended without the need for any of its source code to be revealed. Basically, you had to supply a C function with a predetermined name and have it dispatch on the integer (C `int`) argument. You would then link your C function with the SETL2 interpreter ahead of (in place of) the default no-op function.

1.23 Ceiling (lowest integer upper bound)

```
op ceil (real) : integer
op ceil (integer) : integer
```

This operator returns the smallest integer that is greater than or equal to the given argument.

See also `floor`, `round`, `fix`, and `float`.

1.24 Character representation of small integer

```
op char (integer) : string
```

For example, `char 97 = "a"` in an ASCII environment.

Note that in SETL, it is safe to include NUL characters (`\0`) inside strings, so `char 0` is well-defined.

See also `ichar` and `abs`.

1.25 Change directory

```
proc chdir
proc chdir (string s)
```

The current working directory ("folder") is changed to *s* if given, otherwise to the user's home directory as indicated by the `HOME` environment variable.

See also `getwd`.

1.26 Clear system error indicator

```
proc clear_error
```

The system error indicator (the POSIX global `errno`) is cleared. It is a good idea to do a `clear_error` just before any call where you intend to check `last_error` after the call, as in this example:

```
clear_error;
kill (p);
if last_error = no_error then
  -- we are allowed to send a signal to p
  ...
else
  printa (stderr, "Error: ", last_error);
  ...
end if;
```

This approach can be used whenever you want to check for errors after calling a system routine like `kill` or `close`, which are normally silent about errors.

Contrast this with `open`, which returns `om` instead of a file descriptor if there is an error, allowing you to test the result directly.

The above example is for illustrative purposes only, because in practice, you would normally use `pexists` to test for process existence directly.

1.27 Elapsed time in milliseconds

```
proc clock : integer
```

This is the total amount of wall-clock ("real") time, in milliseconds, that has passed since the current process began. See also `time` and `tod`.

1.28 Close stream

```
proc close (string f)
proc close (integer f)
proc close (string f, integer how)
proc close (integer f, integer how)
```

The stream corresponding to the handle *f* that was previously passed to or returned by `open` (or returned by `accept`, `dup`, or `dup2`) is closed.

It is also permissible to `close` a predefined stream such as `stdin`, `stdout`, and `stderr` (or equivalently `0`, `1`, and `2` respectively), and to `close` a file descriptor (`integer f`) that is not open at the SETL level but may be open at the underlying system level. If it isn't even open at that level, `last_error` will be set.

The *how* argument, if present, must be one of the constants `close_await` (the default), `close_autoreap`, or `close_zombie`. It is meaningful for pipe, pump, and tty-pump streams, as follows.

If *how* is `close_await`, then `close` effectively does a `waitpid` on the child process ID, passing a *waitflag* of `true`. This causes `close` to block until the child process terminates. The termination status is then normally available in `status`, unless that status has already been "reaped", e.g. by a successful `waitpid` elsewhere in the program, in which case `status` is set to `om` in order to avoid a possible false interpretation.

If *how* is `close_autoreap`, then `close` does not block the program in a wait for the child process to terminate, but reaps (and discards) the status in the "background" if and when it does terminate.

Finally, if *how* is `close_zombie`, then `close` does not block the program in a wait, nor does it reap the termination status in the background. If the child process terminates, it will become a "zombie" (as defined by POSIX) until the equivalent of a `waitpid` successfully reaps its status. Normally, you would use this mode of operation when your program needs to maintain a relatively high degree of real-time "liveness" but also needs to get the termination status of a child process that might be slow to exit after the connection to it is closed. A good way to be apprised of the child process termination is to include a `"SIGCHLD"` signal stream in your main `select` set for input event processing. Upon receiving a line on that stream, you would try to reap the child's status using `waitpid`, normally with a *waitflag* of `false` in order to avoid blocking if some other child caused the signal.

See also shutdown.

1.29 Constants for use with close

```
close_await : integer
close_autoreap : integer
close_zombie : integer
```

See close.

1.30 Command-line arguments

```
command_line : tuple
```

This is a tuple of strings giving the command-line arguments that were passed to the SETL program at execution time.

In a script that uses the '#!' escape, or a SETL program compiled to "machine" code, these are all the parameters after (but not including) the command name.

If the program is being run by the standard POSIX driver (usually the setl command), command_line lists all the arguments after the '-x' or '--' flag.

See also command_name and the *GNU SETL User Guide*.

1.31 Command name

```
command_name : string
```

This string is the name by which the SETL program was invoked from the system. In POSIX, this is usually the name of a file containing a SETL script (see example below), or of a file containing the "machine"-code version of a SETL program.

If, however, the program was run by the standard POSIX driver (usually the setl command), then command_name returns the name of the SETL interpreter (usually setlrun).

As a POSIX example, suppose the standard executables of the SETL system have been installed in '/usr/bin'. Then if the following script is stored in the executable file 'Yhwh' and invoked, it will write 'I'm Yhwh' and a newline character to the standard output:

```
#! /usr/bin/setl -k
print ("I'm", command_name);  -- like shell's $0 or C's argv[0]
```

Note the use of the '#!' escape.

See also command_line and the *GNU SETL User Guide*.

1.32 Cosine

```
op cos (real) : real
op cos (integer) : real
```

The argument is in radians. See also acos.

1.33 Hyperbolic cosine

```
op cosh (real) : real
op cosh (integer) : real
```

A mis-speld abbr of "kosher".

1.34 Date and time of day

```
proc date : string
```

Returns the local date and time in a form that looks like '`Tue Nov 6 13:32:56 2007`'.

See also `fdate`, `tod`, `clock`, and `time`.

1.35 Type of denotation within string

```
op denotype (string s) : string
```

If *s* contains a denotation that would be acceptable to `unstr`, then `denotype` *s* `=` `type unstr` *s*, but if *s* is some other string, then the advantage of checking it with `denotype` first is that `denotype` returns `om` in that case instead of raising an exception as `unstr` would.

See also `val` and `str`.

1.36 Integer division

```
op div (integer, integer) : integer
```

SETL guarantees that `div` always truncates fractional results towards zero. (In C, it is machine-dependent for negative quotients.)

See also `mod` and `rem`.

1.37 Domain of map

```
op domain (set) : set
```

The argument must be a set of ordered pairs, that is, a set of tuples each of size 2. The result is the set of all first members of those tuples.

See also `range` and `lessf`.

1.38 Duplicate file descriptor

```
op dup (integer fd) : integer
op dup2 (integer fd1, integer fd2) : integer
```

These are direct interfaces to POSIX `dup()` and `dup2()`, useful when you need low-level control over system-level file descriptors, such as in `fork` and `exec` situations.

Note that the new fd produced by `dup` or `dup2` is not automatically open at the SETL level, though you can use `open` or one of the auto-opening intrinsics (see Section 1.124.6 [automatic opening], page 33) to arrange that subsequently). Also, in the case of `dup2`, there is an implicit POSIX `close()` of *fd2* before the duplication of *fd1* occurs. This is not tracked at the SETL level, so if *fd2* happens to refer to a stream that *is* open at the SETL level, it is possible that some buffered output intended for the original incarnation of *fd2* ends up going to wherever *fd1* points, unless care is taken to `flush (`*fd2*`)` first.

1.39 Test for end of file

```
proc eof : boolean
proc eof (string) : boolean
proc eof (integer) : boolean
```

Called with no arguments, `eof` indicates whether the last input operation was incomplete because the end of the stream was reached. Called with a single argument referring to an open stream, it returns the end-of-file status of that particular stream.

Note that input errors also cause `eof` to be asserted. These can be distinguished from normal end-of-file occurrences using `last_error`.

See also `get`, `geta`, `getb`, `getc`, `getchar`, `getline`, `getn`, `gets`, `peekc`, `peekchar`, `ungetc`, `ungetchar`, `read`, `reada`, `recv`, and `recvfrom`; but not `getfile`.

1.40 Test for integer divisible by 2

```
op even (integer) : boolean
```

1.41 Replace current process

```
proc exec (string cmd)
proc exec (string cmd, tuple argv)
proc exec (string cmd, tuple argv, tuple envp)
```

This is a low-level interface to POSIX `execvp()` or `execve()` depending on whether the *`envp`* argument is supplied.

If *`envp`* *is* present, `execve()` is used, which requires that *`cmd`* be a full pathname identifying a command.

If the *`envp`* argument is *not* given, then `execvp()` is used, implying that the `PATH` environment variable is searched for a directory containing an executable named *`cmd`* unless *`cmd`* contains a slash.

If the second argument (*`argv`*) appears, it must be a tuple of strings specifying the arguments that will be supplied to the command. Otherwise, *`argv`* defaults to a one-element tuple, namely `[`*`cmd`*`]`.

If there is an *`envp`*, it must be a tuple of strings defining the environment variables for the command, with each such string being of the form '*`name=value`*'.

If `exec` is successful, it does not return; the current process is overlaid by the new command.

Compare `filter`, `system`, and the `open` modes '`pump`', '`tty-pump`', '`pipe-from`', and '`pipe-to`', one of which will probably be able to achieve the effect you want more conveniently than `exec` does.

See also `fork` and `pump` for a couple of ways to create a child process suitable for overlaying with `exec`.

1.42 Natural exponential (e raised to a power)

```
op exp (real) : real
op exp (integer) : real
```

See also `log` and the general exponentiation operator (`**`).

1.43 Predefined "false" boolean value

```
false : boolean
```

1.44 Format date and time

```
proc fdate (integer ms, string fmt) : string
proc fdate (integer ms) : string
```

The *ms* argument represents some number of milliseconds since 1 January 1970 UTC, to be formatted as a date and time according to *fmt*, which defaults to '`%a %b %e %H:%M:%S.%s %Z %Y`'. For example, `fdate (936433255069)` is '`Sat Sep 4 04:20:55.069 EDT 1999`' in the timezone the author occupied at a certain moment in history, and `fdate (tod)` is a similarly fancy rendering of the current calendar time.

The %-sign patterns in *fmt* are those patterns defined for POSIX `strftime()` when applied to the result of applying POSIX `localtime()` to *ms* `div 1000`, together with one extension: '`%s`' will expand to the low-order 3 decimal digits of *ms*. (Note: that meaning of '`%s`' conflicts with a nonstandard GNU extension to `strftime()`, where it means the number of seconds since 1970 began.)

See also `date`, which should give the same result as the expression `fdate (tod, "%c")`.

1.45 Test for existence of file

```
op fexists (string) : boolean
```

Returns `true` or `false` according as POSIX `stat()` returns 0 or not.

Note that `stat()` *does* follow symbolic links, so `fexists` will only return `true` if the pathname ultimately indicated by the string argument exists (and the caller has sufficient access to all pathname components in reaching it).

Therefore `fexists` is stricter than `lexists`, which merely uses POSIX `lstat()`.

See also `link`, `symlink`, `readlink`, and `unlink`.

1.46 Return filename of open stream

```
op filename (string f) : string
op filename (integer f) : string
```

For a stream with handle *f* that is open at the SETL level, `filename` returns the string that was originally passed as the first argument to `open`, or `om` if in fact `open` was applied to an integer (a file descriptor).

It is an error to apply `filename` to an fd that is not open at the SETL level, even though it may be open at the system level.

1.47 Return fd of open stream

```
op fileno (string) : integer
op fileno (integer) : integer
```

The `fileno` operator returns the POSIX file descriptor (fd) associated with the open stream designated by the argument. Applied to an fd, it should merely return that fd, but applied to `om`, it will raise an exception.

The following idiom is common for programs that would rather crash immediately than continue with a non-fd result from `open`:

```
fd := fileno open (...);
```

The nominal use of `fileno`, however, is to obtain the fd associated with a `string` stream handle. (The use of the fd is preferable from the standpoint of uniqueness, and is also likely to be more efficient.)

1.48 Filter string through external command

```
proc filter (string cmd, string input) : string
proc filter (string cmd) : string
```

The *`cmd`* argument designates a command that reads from standard input and writes to standard output. The *`input`* argument is a string (default `""`) that is fed into the command's standard input, and the result string is the contents of the command's standard output.

The command itself is processed by the standard shell, `/bin/sh`. Thus it may contain arguments with patterns such as '`2>&1`' to redirect the standard error stream into the same destination as the standard output, like `dup2 (stdout, stderr)` does, and may also contain special characters such as '`*`' for filename "globbing".

See also `status`, `system`, `pump`, `pipe_from_child`, `pipe_to_child`, and the `open` modes '`pump`', '`tty-pump`', '`pipe-from`', and '`pipe-to`'.

1.49 Truncate floating-point number to integer

```
op fix (real) : integer
op fix (integer) : integer
```

Truncation is towards zero.

See also `ceil`, `floor`, `round`, and `float`.

1.50 Format number with optional decimal point

```
proc fixed (real x, integer w, integer a) : string
proc fixed (integer x, integer w, integer a) : string
```

The number *`x`* is converted to a string of length `abs w` or more, with `a` digits after the decimal point.

If `a` is zero, there is no decimal point; `a` must not be negative.

If `abs w` is larger than necessary, the string is padded with blanks on the left (for `w` greater than 0) or on the right (for negative `w`).

If `abs w` is too small, a longer string is produced as necessary to accommodate the number, with no padding.

If the conversion is not possible, the string '`nan`', '`inf`', or '`infinity`' will result.

See also `floating`, `whole`, `str`, and `strad`.

1.51 Convert number to floating-point

```
op float (integer) : real
op float (real) : real
```

Since integers in GNU SETL are unbounded, but reals are limited in an implementation-dependent way (typically to 64-bit IEEE floating-point), it is possible for this conversion to produce positive or negative floating-point infinity. Loss of precision is also possible for integers whose absolute value is larger than can fit in the mantissa of the floating-point representation: the mantissa is typically an effective 53 bits.

See also fix, ceil, floor, and round.

1.52 Format number in scientific notation

```
proc floating (real x, integer w, integer a) : string
proc floating (integer x, integer w, integer a) : string
```

The number *x* is converted to a string of length abs *w* or more in "scientific" notation, with one digit before the decimal point, a digits after it, and the string '**e+***d*' after that, where the latter stands for "times 10 to the power of *d*".

If a is zero, there is no decimal point; a must not be negative.

If abs *w* is larger than necessary, the string is padded with blanks on the left (for *w* greater than 0) or on the right (for negative *w*).

If abs *w* is too small, a longer string is produced as necessary to accommodate the number, with no padding.

If the conversion is not possible, the string '**nan**', '**inf**', or '**infinity**' will result.

See also fixed, whole, str, and strad.

1.53 Floor (greatest integer lower bound)

```
op floor (real) : integer
op floor (integer) : integer
```

This operator returns the largest integer that is less than or equal to the given argument.

See also ceil, round, fix, and float.

1.54 Flush output buffer

```
proc flush (string)
proc flush (integer)
```

All buffered output for the designated stream is written out.

This can be particularly important when streams are used to communicate between processes. Without a flush, data can remain in a stream's output buffer for arbitrarily long periods.

SETL flushes a bidirectional stream automatically whenever an input operation is initiated on that stream. This helps to avoid a common cause of deadlock in distributed systems.

See also tie, open, and fileno.

1.55 Fork process

```
proc fork : integer
```

This is a direct interface to POSIX `fork()`. The calling process splits into two processes.

In the calling ("parent") process, `fork` returns an integer representing the process ID of the "child" process.

In the child process, `fork` returns 0.

If the system cannot spawn a new process, `fork` returns `om`, and no child process is created.

Output buffers are `flush`ed before the attempt to spawn a child process is made.

In many cases, `pump` will be preferable to `fork` because `pump` also sets up communication with the child without the need to go through the customary low-level `pipe`, `dup2`, and `close` calls to set up the child's environment.

See also `exec`, `pipe_from_child`, `pipe_to_child`, `filter`, `system`, `pid`, `pexists`, `kill`, `wait`, `waitpid`, and the `open` modes '`pump`', '`tty-pump`', '`pipe-from`', and '`pipe-to`'.

1.56 Take arbitrary element from set

```
op from (wr var x, rw set s)
```

An arbitrary (which is not to say *random* but rather *nondeterministically chosen*) element of the argument set is removed from *`s`* and assigned to *`x`*.

If *`s`* is empty, *`x`* `:= om` instead.

Currently, `from` is actually a statement form, not an operator, but this may be extended in the future so that the extracted element (or `om`) is also returned as the result.

See also `fromb`, `frome`, `arb`, `less`, `lessf`, and the "minus" operator (`-`).

1.57 Take from beginning of string or tuple

```
op fromb (wr string x, rw string s)
op fromb (wr var x, rw tuple s)
```

The string or tuple *`s`* is stripped of its first element (a single character if *`s`* is a string), and that element is assigned to *`x`*.

If *`s`* is of length 0, *`x`* `:= om` instead.

Currently, `fromb` is actually a statement form, not an operator, but this may be extended in the future so that the extracted element (or `om`) is also returned as the result.

See also `from` and `frome`.

1.58 Take from end of string or tuple

```
op frome (wr string x, rw string s)
op frome (wr var x, rw tuple s)
```

The string or tuple *`s`* is stripped of its last element (a single character if *`s`* is a string), and that element is assigned to *`x`*.

If *`s`* is of length 0, *`x`* `:= om` instead.

Currently, `frome` is actually a statement form, not an operator, but this may be extended in the future so that the extracted element (or `om`) is also returned as the result.

See also `from` and `fromb`.

1.59 Size of file in bytes

```
op fsize (string f) : integer
op fsize (integer f) : integer
```

If *f* is a stream handle or a filename, the size of the referenced object in bytes (value of the st_size field of the POSIX struct stat) is returned.

Otherwise, om is returned. Thus fsize can be used to test for file existence and obtain file size in a single atomic operation.

Note that if *f* refers to a non-file stream (such as a socket connection, for example), fsize may still succeed and return a value such as 0.

See also fexists.

1.60 Get one or more lines from stdin

```
proc get (wr string...)
```

Equivalent to geta (stdin, ...).

Note that this signature for get follows that of SETL2, while geta looks like the old CIMS SETL get. This brings the signatures of get and geta into alignment with those of read and reada.

1.61 Get one or more lines from input stream

```
proc geta (string f, wr string...)
proc geta (integer f, wr string...)
```

Lines are read from the input stream *f* and assigned to the succeeding arguments in order. If the end of input is reached before all arguments have been satisfied, trailing arguments are set to om.

Lines are terminated by newline (\n), and there is no restriction on line length. The newline character is not delivered as part of the assigned string, and the final line before the end of input need not be terminated by a newline.

If *f* is not already open, an attempt is made to open it automatically for reading.

See also get, getline, getb, getc, getn, peekc, and reada.

1.62 Get one or more values from input stream

```
proc getb (string f, wr var...)
proc getb (integer f, wr var...)
```

Values are read from the input stream *f* and assigned to the succeeding arguments in order. If the end of input is reached "early", trailing arguments get om.

Values written by putb, except for atoms (see newat) and procedure references (see routine), are guaranteed to be readable by getb.

If *f* is not already open, an attempt is made to open it automatically for reading.

There is an important difference between getb and reada in that reada will always start reading at the beginning of a line, skipping ahead to just after the next newline (\n) character if necessary, whereas getb will simply start with the next available character

in the input stream. Both routines will happily cross newline boundaries to obtain more values, however.

See also geta.

1.63 Get character from input stream

```
op getc (string f) : string
op getc (integer f) : string
```

One character is read from the input stream *f* and returned as a string of length 1. If there are no more characters (end of input was reached), getc returns om instead.

If *f* is not already open, an attempt is made to open it automatically for reading.

See also getchar, getn, geta, and getb.

1.64 Get character from stdin

```
proc getchar : string
```

Equivalent to getc (stdin).

1.65 Get effective group ID

```
proc getegid : integer
```

Returns the result of calling the POSIX getegid() function.

See also getgid, setegid, setgid, geteuid, getuid, seteuid, and setuid (details and example).

1.66 Get value of environment variable

```
op getenv (string) : string
```

If the environment variable named by the argument exists, its value is returned; otherwise you get om.

See also putenv, setenv, and unsetenv.

1.67 Get effective user ID

```
proc geteuid : integer
```

Returns the result of calling the POSIX geteuid() function.

See also getuid, seteuid, setuid (details and example), getegid, getgid, setegid, and setgid.

1.68 Read stream up to the end

```
op getfile (string f) : string
op getfile (integer f) : string
```

Characters are read into a string from the input stream *f* until the end of input is reached. If the end-of-file condition is immediate, getfile returns the empty string ("").

Since getfile always reads until it reaches the end of input, it does not alter what eof returns.

If *f* is not already open, an attempt is made to open it automatically for reading, and if the file was automatically opened, it is automatically closed after the end of input has been reached.

The `getfile` intrinsic is unique among routines that attempt to auto-open a file for input in that if `getfile` fails on the auto-open attempt, it returns `om` rather than raising an exception. This is part of an effort to dissuade SETL programmers from writing racy code like

```
x := if fexists f then getfile f else "some default" end;
```

when the idiomatic

```
x := getfile f ? "some default";
```

is race-free. Of course, it would have been possible to write

```
fd := open (f, "r");
if fd /= om then
  x := getfile fd;
  close (fd);
else
  x := "some default";
end if;
```

but that is hardly in the spirit of a language which aims for conciseness, convenience, and readability.

See also `getc`, `getn`, `geta`, `getb`, `getline`, and `putfile`.

1.69 Get real group ID

```
proc getgid : integer
```

Returns the result of calling the POSIX `getgid()` function.

See also `getegid`, `setgid`, `setegid`, `getuid`, `geteuid`, `setuid` (details and example), and `seteuid`.

1.70 Get line from input stream

```
op getline (string f) : string
op getline (integer f) : string
```

This is an operator-form alternative to `geta` for reading a single line.

Characters through the next newline character are read, and the accumulated string is returned, without the newline.

If the end of input is reached with no characters being read, `om` is returned.

If *f* is not already open, an attempt is made to open it automatically for reading.

1.71 Get fixed number of characters from input stream

```
proc getn (string f, integer n) : string
proc getn (integer f, integer n) : string
```

Exactly *n* characters are read from the input stream *f* if at least that many remain before the end of the input. If fewer than *n* characters remain in the stream *f*, a shorter string is returned.

If *f* is not already open, an attempt is made to open it automatically for reading.

See also getc.

1.72 Get process group ID

```
proc getpgrp : integer
```

Retrieves the process group ID of the current process by calling POSIX `getpgrp()`. This is the process ID of the process group leader.

Note that process group IDs have nothing to do with user group IDs (contrast getgid).

See also setpgrp, pid, pexists, kill, and waitpid.

1.73 Direct-access read

```
proc gets (string f, integer start, integer n, wr string x)
proc gets (integer f, integer start, integer n, wr string x)
```

The direct-access stream *f* (see open modes '`r+`', '`w+`', '`a+`', and '`n+`') is viewed as a string, where *start* specifies the index (1 or higher) of the first character to read. The `gets` procedure reads *n* characters from *f* (fewer if the end of file is reached), and assigns the resulting string to *x*.

If *f* is not already open, an attempt is made to open it automatically in '`r+`' mode, which allows seeking, reading, and writing.

See also puts, seek, and rewind.

1.74 Get real user ID

```
proc getuid : integer
```

Returns the result of calling the POSIX `getuid()` function.

See also geteuid, setuid (details and example), seteuid, getgid, getegid, setgid, and setegid.

1.75 Current working directory

```
proc getwd : string
```

Current working directory of the process.

See also chdir.

1.76 Pathname wildcard expansion

```
op glob (string) : tuple
```

Using the POSIX `glob()` function, expand the pathname pattern given in the `string` operand to produce a tuple of strings. An tuple of length 0 is produced if there is no match, or if there is an error (so if the empty tuple is unexpected, consult last_error).

For example, if the current directory contains 3 `.h` files, then

```
glob '*.h'
```

might equal the tuple

```
['a.h', 'b.h', 'c.h']
```

See also getwd and chdir.

1.77 Find all occurrences of pattern in string

```
proc gmark (string s, string p) : tuple
proc gmark (string s, tuple p) : tuple
```

All non-overlapping leftmost occurrences of the POSIX extended regular expression (ERE) pattern *p* are found in *s*, and the result is returned as a tuple of pairs of integers `[i, j]` such that each matched substring of *s* can be addressed as *s*`(i..j)`.

If *p* is a `tuple`, it must consist of a pair of `string` patterns *p1*, *p2*. In that case, it means everything from *p1* up to the first subsequent occurrence of *p2*.

See also `mark`, `gsub`, and `sub`.

1.78 Replace patterns in string

```
proc gsub (rw string s, string p) : tuple
proc gsub (rw string s, tuple p) : tuple
proc gsub (rw string s, string p, string r) : tuple
proc gsub (rw string s, tuple p, string r) : tuple
```

All non-overlapping leftmost occurrences in *s* of the POSIX extended regular expression (ERE) pattern *p* are replaced by *r*, which defaults to the empty string (`""`). The original substrings of *s* replaced by this operation are returned as a tuple of strings.

If *p* is a `tuple`, it must consist of a pair of `string` patterns *p1*, *p2*. In that case, it means everything from *p1* up to the first subsequent occurrence of *p2*.

See also `sub`, `gmark`, and `mark`.

1.79 Convert string to hexadecimal

```
op hex (string) : string
```

For example, `hex "djB" = "646A42"` in an ASCII environment.

See also `unhex`.

1.80 Local host address

```
proc hostaddr : string
```

Primary Internet (IP) address of the current host, if it can be found; otherwise `om`.

See also `hostname`, `ip_addresses`, `ip_names`, `peer_address`, and `peer_name`.

1.81 Local hostname

```
proc hostname : string
```

Primary Internet (DNS) name of the current host.

See also `hostaddr`, `ip_names`, `ip_addresses`, `peer_name`, and `peer_address`.

1.82 Integer equivalent of character

```
op ichar (string) : integer
```

For example, `ichar "a" = 97` in an ASCII environment. The argument must be 1 character in length.

See also `char` and `abs`.

1.83 Implication

```
op impl (boolean, boolean) : boolean
```

Here is the customary "truth table" defining this operator:

```
true  impl true   =  true
true  impl false  =  false
false impl true   =  true
false impl false  =  true
```

This seldom-used operator could have been "short-circuited" like `and`, `or`, and the "query" operator (`?`), but currently isn't. That is to say, the right-hand side of `impl` is always evaluated.

For most programming applications, it is more natural to write

```
q or not p
```

or

```
(not p) or q
```

than the almost equivalent (except for the lack of short-circuiting)

```
p impl q
```

because in programming, in contrast to mathematics, one is usually testing logical expressions rather than proving universal truths. (Even in mathematics, the implication operator does not often appear in subclauses, but very often at the "top level" in statements that are true over some universe of discourse.) The surface resemblance between the "if-then" control structure of programming and the "if-then" alternative syntax for implication in mathematics corresponds to a semantic resemblance: in programming, the execution of the second part follows from the truth of the first part, while in mathematical implication, it is the truth of the second part that follows from that of the first part.

1.84 Membership test

```
op in (var x, set s) : boolean
op in (var x, tuple s) : boolean
op in (string x, string s) : boolean
```

The keyword `in` is also used in a common iterator form, as in

```
for x in s loop
  ...
end loop;
```

and

```
squares := {x*x : x in s};
```

This use of `in` as an iterator should not be confused with its use as a membership test, where it is simply a boolean-valued binary operator.

If *s* is a `set` and *x* is `om`, the expression

```
x in s
```

is `false`. But if *s* is a `tuple` (with *x* `= om`), the result is `true` if and only if the tuple has any "holes" (non-trailing `om` values) in it.

When *s* and *x* are strings, the expression *x* `in` *s* is `true` if and only if *x* is a substring of *s*.

See also `arb`, `from`, and `notin`.

1.85 Subset test

```
op incs (set s, set ss) : boolean
```

Returns `true` if *ss* is a subset of *s*. Thus

```
s incs ss
```

has the same truth value as

```
ss subset s
```

1.86 Integer quotient type switch

```
intslash : boolean
```

By default, the result of dividing two `integer` values in SETL is `real`, as in Pascal and in the Algol family (and as vehemently opposed to the Fortran/C/SETL2 family). This default corresponds to `intslash = false`. See the discussion of the "slash" operator (`/`) for a heated argument in favour of leaving it this way, and see also `set_intslash` for the preferred way of changing it to `true` should there arise some compelling reason to do so.

1.87 Internet host addresses

```
proc ip_addresses : set
proc ip_addresses (string) : set
```

Called with no arguments, `ip_addresses` returns the set of all Internet (IP) addresses of the machine hosting the current process, as strings like '`128.122.129.66`'. Otherwise, it returns a set of such strings for the host whose name or IP address is given in dotted notation by the argument.

See also `ip_names`, `hostaddr`, `hostname`, `peer_address`, and `peer_name`.

1.88 Internet hostnames

```
proc ip_names : set
proc ip_names (string) : set
```

Called with no arguments, `ip_names` returns the set of all Internet (IP) names of the machine hosting the current process, as strings like '`GALT.CS.NYU.EDU`'. Otherwise, it returns a set of such strings for the host whose name or IP address is given in dotted notation by the argument.

See also `ip_addresses`, `hostname`, `hostaddr`, `peer_name`, and `peer_address`.

1.89 Type testers

```
op is_atom (var) : boolean
op is_boolean (var) : boolean
op is_integer (var) : boolean
op is_map (var) : boolean
```

```
op is_mmap (var) : boolean
op is_numeric (var) : boolean
op is_om (var) : boolean
op is_real (var) : boolean
op is_routine (var) : boolean
op is_set (var) : boolean
op is_smap (var) : boolean
op is_string (var) : boolean
op is_tuple (var) : boolean
```

The operator `is_map` (or equivalently `is_mmap`, for "multi-valued map") returns `true` if its argument is a set consisting entirely of ordered pairs (tuples of length 2).

The operator `is_smap` (for "single-valued map") adds the further condition that given a map *f*, `#domain` *f* `= #`*f*; that is, that *f* takes each domain element to one range element.

See also `type` and `denotype`.

1.90 Test for open stream

```
op is_open (string f) : boolean
op is_open (integer f) : boolean
```

Returns `true` if *f* is one of the pre-opened `stdin`, `stdout`, or `stderr`; a stream opened by `open`; or an automatically opened stream.

1.91 Concatenate tuple of strings, with delimiter

```
proc join (tuple t, string delim) : string
```

All elements of the tuple *t* must be strings, and they are concatenated together, separated by the delimiter string *delim*. As a special case, if *t* is the empty tuple (`[]`), the empty string (`""`) is returned. Note that *delim* is not used when `#`*t* is 0 or 1, but must still be a string.

For `#`*t* `> 0`,

```
join (t, delim) = ([] +/ [delim+s : s in t])(#delim+1..)
```

See also `split`.

1.92 Send signal to process

```
proc kill (integer p)
proc kill (integer p, integer signal)
proc kill (integer p, string signal)
```

Calls POSIX `kill()` on *p*. Among processes to which the caller of `kill` has permission to send a signal, *p* is interpreted as follows.

If *p* is greater than 0, the signal is sent to the process with a process ID equal to *p*.

If *p* is 0, the signal is sent to every process whose process group ID is equal to that of the caller.

If *p* is negative and not equal to -1, then `-`*p* is a process group ID, and the signal is sent to every process in that group.

If *p* is -1, the signal is sent to every process to which the caller has permission to send a signal. (For non-privileged processes, this is the set of processes owned by the user.)

If *p* indicates a nonexistent process or process group, the call has no effect except to set last_error.

If *signal* is omitted, it defaults to 'TERM', or equivalently 'SIGTERM'. Signals may be specified as integers or more portably as strings. Case is not significant. The signal names HUP, INT, QUIT, ILL, ABRT, FPE, KILL, SEGV, PIPE, ALRM, TERM, USR1, USR2, CHLD, CONT, STOP, TSTP, TTIN, and TTOU are defined by POSIX. Local system documentation or C header files may define additional ones, such as WINCH and PWR.

As a special case, if *signal* is 0, no signal is sent, but the validity of *p* is checked (and the result is reflected in last_error).

See slso pid, pexists, getpgrp, setpgrp, fork, pump, pipe_from_child, pipe_to_child, system, and the open modes 'pump', 'tty-pump', 'pipe-from', and 'pipe-to' for more on processes; and the example under clear_error for more on detecting and handling errors arising from system calls.

1.93 Last error message from system routine

```
last_error : string
```

This is the error message corresponding to the last setting of the global errno by a POSIX routine.

See also clear_error and no_error.

1.94 Extract initial substring by length

```
proc len (rw string s, integer n) : string
```

The lesser of *n* and #*s* characters are removed from the beginning of *s* and returned as the function result.

See also any, break, match, notany, span, rany, rbreak, rlen, rmatch, rnotany, and rspan.

1.95 Set less one element

```
op less (set s, var x) : set
```

Definition: *s* less *x* = *s* - {*x*}.

Note that if *x* is om, the result is just *s*.

See the set difference ("minus") operator (-), and also see lessf, from, and with.

1.96 Map less one domain element

```
op lessf (set s, var x) : set
```

The set *s* must be a map. The lessf operator returns a copy of the map in which all pairs having *x* as a domain element are removed.

See also less and from.

1.97 Test for existence of file or symlink

```
op lexists (string) : boolean
```

Return `true` if POSIX `lstat()` returns 0; otherwise `false`.

Note that `lstat()` does *not* follow symbolic links, so `lexists` tests only for the accessibility of the pathname indicated by the string argument.

Therefore `lexists` is less strict than `fexists`, which uses POSIX `stat()`.

See also `link`, `symlink`, `readlink`, and `unlink`.

1.98 Create hard link

```
proc link (string existing, string new)
```

Atomically create a "hard link" *`new`* to the existing file *`existing`* using POSIX `link()`, if *`new`* does not exist before the call. There is no return value, but calling `clear_error` before the operation and inspecting `last_error` after it can be used to determine atomically whether the operation was successful. Thus `link` can be used to implement a "test and set" mutex lock, assuming *`existing`* exists: if *`new`* already exists, the operation will fail; and if it doesn't exist, it is created and the calling process then "owns" the lock. To relinquish it, the caller later calls `unlink` on *`new`*; or another process can "forcibly" remove it.

The `open` modes '`n`' and '`n+`' provide a similar mutual exclusion capability.

If `link` succeeds, *`new`* and *`existing`* refer to exactly the same file.

See also `symlink`.

1.99 Natural logarithm

```
op log (real) : real
op log (integer) : real
```

The argument must be greater than 0.

See also `exp`.

1.100 Pad string on left with blanks

```
proc lpad (string s, integer n) : string
```

If *`n`* `>` `#`*`s`*, the returned string is a copy of *`s`* padded on the left with blanks to length *`n`*. Otherwise, a copy of *`s`* is returned.

See also `rpad`.

1.101 Regular expression recognition switch

```
magic : boolean
```

By default, `magic` is `true`, meaning that subscripting and slicing of strings by pattern strings causes the pattern strings to be interpreted as regular expressions. This also affects `sub`, `gsub`, `mark`, `gmark`, and `split` when its second argument is given explicitly. You can assign `magic := false` to turn off this behaviour, so that pattern strings are interpreted literally

See also `set_magic` for an alternative way of setting `magic`.

1.102 Find first occurrence of pattern in string

```
proc mark (string s, string p) : tuple
proc mark (string s, tuple p) : tuple
```

The leftmost occurrence if any of the POSIX extended regular expression (ERE) pattern *p* is found in *s*, and the result is returned as a pair of integers `[i, j]` such that the matched substring of *s* can be addressed as *s*`(`*i*`..`*j*`)`.

If there is no such occurrence, `om` is returned.

If *p* is a `tuple`, it must consist of a pair of `string` patterns *p1*, *p2*. In that case, it means everything from *p1* up to the first subsequent occurrence of *p2*.

See also `gmark`, `sub`, and `gsub`.

1.103 Extract initial substring by exact match

```
proc match (rw string s, string p) : string
```

If *p*'s value is the initial substring of *s*'s, it is removed from *s* and returned as the function result. Otherwise, nothing happens to *s* and the empty string (`""`) is returned.

See also `any`, `break`, `len`, `notany`, `span`, `rany`, `rbreak`, `rlen`, `rmatch`, `rnotany`, and `rspan`.

1.104 Numeric maximum

```
op max (integer, integer) : integer
op max (integer, real) : real
op max (real, integer) : real
op max (real, real) : real
```

See also `min`.

1.105 Allocate machine memory

```
proc mem_alloc (integer n) : integer
```

This is a low-level interface to POSIX `malloc()`. It allocates *n* bytes of system memory and returns the address of that memory block.

See also `mem_free`, `mem_realloc`, `mem_copy`, `mem_fetch_...`, and `mem_store_...`.

1.106 Free machine memory

```
proc mem_free (integer)
```

This is a low-level interface to POSIX `free()`. Pass it an address returned by `mem_alloc` when you are sure that the memory block associated with that address can be released.

Bad things can happen if you try to free a given block twice, or try to refer to a block you have already freed.

See also `mem_alloc`, `mem_realloc`, `mem_copy`, `mem_fetch_...`, and `mem_store_...`.

1.107 Reallocate machine memory

```
proc mem_realloc (integer address, integer n): integer
```

This is a low-level interface to POSIX `realloc()`. Pass it the *`address`* of an existing memory block, and a new length *`n`*.

See also `mem_alloc`, `mem_free`, and `mem_copy`.

1.108 Copy machine memory

```
proc mem_copy (integer dst, integer src, integer n)
```

This is a low-level procedure for copying machine memory. It copies *`n`* bytes starting at address *`src`* to consecutive locations starting at address *`dst`*. If the memory regions overlap, the consequences are not defined.

See also `mem_alloc`, `mem_free`, `mem_realloc`, `mem_fetch_...`, and `mem_store_...`.

1.109 Machine memory read

```
proc mem_fetch_string (integer address, integer n) : string
proc mem_fetch_c_string (integer address) : string
```

These are low-level functions to fetch data from machine memory. The `integer` *`address`* gives a machine address from which some number *n* of bytes will be read.

For `mem_fetch_string`, *n* is given by the second argument (*`n`*).

For `mem_fetch_c_string`, *n* is deduced using the C convention of a NUL (`char 0`) terminating character.

See also `mem_store_...`, `pack_...`, `unpack_...`, `mem_copy`, `mem_alloc`, and `mem_free`.

1.110 Machine memory write

```
proc mem_store_string (string s, integer address)
proc mem_store_c_string (string s, integer address)
```

These are low-level procedures to clobber machine memory. The integer *`address`* gives an address to which some number *n* of bytes will be written.

For `mem_store_string`, *n* is `#s`.

For `mem_store_c_string`, *n* is 1 more than the lesser of `#s` and the number of bytes before the first NUL character (`\0`) in *`s`* (a trailing NUL character is always written by `mem_store_c_string`).

See also `mem_fetch_...`, `pack_...`, `unpack_...`, `mem_copy`, `mem_alloc`, and `mem_free`.

1.111 Numeric minimum

```
op min (integer, integer) : integer
op min (integer, real) : real
op min (real, integer) : real
op min (real, real) : real
```

See also `max`.

1.112 Integer modulus; symmetric set difference

```
op mod (integer, integer) : integer
op mod (set, set) : set
```

SETL guarantees a non-negative remainder as the result of `mod`, following the usual mathematical "clock arithmetic" definition. The sign of the denominator is immaterial, so:

```
 5 mod  3 = 2
-5 mod  3 = 1
 5 mod -3 = 2
-5 mod -3 = 1
```

The use of a single operator name for both the non-commutative, non-associative operation of integer modulus and the commutative, associative operation of symmetric set difference is a regrettable matter of SETL history.

The set-theoretic `mod` is analogous to the logical "exclusive or".

See also `rem` and `div`.

1.113 Number of arguments passed by caller

```
nargs : integer
```

This is particularly useful in cases where there is a possibility of trailing `om` and/or writable parameters on procedures that take a variable number of arguments.

Note that `nargs` is the *total* number of arguments passed to the currently active routine, including the required ones.

1.114 Create new atom

```
proc newat : atom
```

This creates a unique `atom`, whose salient property is merely that it is different from all other `atoms` created by the current process.

See also `is_atom`.

1.115 Non-error message

```
no_error : string
```

This is the value `last_error` has immediately after a call to `clear_error`. It corresponds to the POSIX `errno` value of 0.

1.116 Logical negation

```
op not (boolean) : boolean
```

See also `and`, `or`, and the bitwise operators such as `bit_not`.

1.117 Extract initial character using character set

```
proc notany (rw string s, string p) : string
```

If the first character of *s* does not occur anywhere in the string *p*, that character is removed from *s* and returned as the function result. Otherwise, nothing happens to *s*, and the empty string (`""`) is returned.

See also any, break, len, match, span, rany, rbreak, rlen, rmatch, rnotany, and rspan.

1.118 Membership test

```
op notin (var x, set s) : boolean
op notin (var x, tuple s) : boolean
op notin (string x, string s) : boolean
```

Definition: *x* notin *s* = not (*x* in *s*).

1.119 All subsets having a given size

```
op npow (integer n, set s) : set
op npow (set s, integer n) : set
```

Definition: *s* npow *n* = *n* npow *s* = {ss in pow *s* | #ss = *n*}.

This is the set of all $\binom{\sharp s}{n}$ subsets of *s* that have *n* members.

1.120 Print to stdout with no trailing newline

```
proc nprint (...)
```

Equivalent to nprinta (stdout, ...).

1.121 Print to output stream with no trailing newline

```
proc nprinta (string f, ...)
proc nprinta (integer f, ...)
```

There can be 0 or more arguments after *f*, of any type. They are sent in sequence to the stream *f*, separated by single spaces. String arguments are written directly; all others are written as if they had been passed through str first.

If *f* is not already open, an attempt is made to open it automatically for writing.

Note that the output of the program

```
nprinta (stderr, 1, 2);
```

is '1 2', which is not the same as the output of the program

```
nprinta (stderr, 1);
nprinta (stderr, 2);
```

which is '12'.

See also printa.

1.122 Test for integer not divisible by 2

```
op odd (integer) : boolean
```

1.123 The "undefined" value

```
om
```

This is the default value of all uninitialized SETL variables, undefined set, range, or tuple elements, the implicit return value of all `proc` routines that are not defined to return anything else, and the default result of many operations when they do not succeed in obtaining a primary result but are not upset enough to raise an exception about it.

Ideal if nothing is what you want. See nothing, hear nothing, and say `om`.

See also `type`, `denotype`, `str`, and `unstr`.

1.124 Open a stream

```
proc open (string f, string how) : integer
proc open (integer f, string how) : integer
```

This function tries to return a file descriptor (fd) for *`f`*, where *`f`* may be a string or may be an integer fd that is already open at the operating system level but not at the SETL level.

1.124.1 Arguments to `open`

Valid values of the (case-insensitive) "mode" argument *`how`*, and their meanings, are:

'r'	sequential input
'w'	sequential output
'a'	sequential output, append to file
'n'	sequential output, new file
'r+'	direct access
'w+'	direct access, empty file first
'a+'	direct access, append to file
'n+'	direct access, new file
'rw'	bidirectional I/O
'pipe-from'	input from command
'pipe-to'	output to command
'pump'	input/output from/to command
'tty-pump'	input/output on pty-wrapped command
'socket'	TCP client socket
'server-socket'	TCP server socket
'udp-client-socket'	can `send` and `recv`
'udp-server-socket'	can `recvfrom` and `sendto`
'signal'	get newline per received signal
'ignore'	signal to be ignored
'real-ms'	get newline per real time period

For 'a', 'n', and all four modes ending in '+', the *`f`* argument must designate a file. For 'r', 'w', and 'rw', it may designate a file, an fd that is already open at the system level but not at the SETL level (see Section 1.124.4 [buffering], page 32), or a fifo or socket in the filesystem.

For modes 'w' through 'n+' above, if *`f`* is a string, the file will be created if it does not exist (assuming this is permitted). For modes 'n' and 'n+', *`f`* *must* be a string.

For '`pipe-from`', '`pipe-to`', '`pump`', and '`tty-pump`', the *`f`* argument must be a string giving a command to be executed by the standard shell, `/bin/sh`. The POSIX-level `FD_CLOEXEC` flag is set on the resulting stream, just as the POSIX `popen()` does.

For '`socket`' and '`udp-client-socket`', *`f`* must be of the form '*`host`*`:`*`port`*', where *`host`* is the name or IP address (in dotted notation) of an Internet host, and *`port`* is a port number or a service name as looked up from the local '`/etc/services`' file by POSIX `getservbyname()`.

For '`server-socket`' and '`udp-server-socket`', *`f`* must be a string giving a port number or service name. In the special case where *`f`* is '`0`', the system chooses an "ephemeral" port number which can be retrieved (as an integer) using `port`.

For '`signal`' and '`ignore`', *`f`* must be a signal name. See Section 1.124.10 [signal streams], page 34.

Finally, for '`real-ms`', the *`f`* argument to `open` must be a decimal digit string giving a number in milliseconds.

1.124.2 Compatibility of `open` with CIMS SETL and SETL2

This `open` is almost upwardly compatible with theCIMS SETL **open**, and completely compatible for programs which ignored the return value, because all the I/O routines accept as a *stream handle* either the argument that was originally passed on a successful `open` call or the file descriptor (fd) that was returned by it. This `open` is also compatible with SETL2 in that the fd serves as a unique handle.

1.124.3 Results of `open`

If `open` cannot open the file, command, socket, fifo, signal, timer, or fd identified by *`f`*, it returns `om`. In the case of clear programming errors such as an invalid mode or an attempt to open an fd in mode '`n`' or '`n+`', it raises an exception instead.

For modes '`n`' and '`n+`', if the file *`f`* already exists, `open` returns `om`. The existence check against and the creation of *`f`* are performed by calling POSIX `open()` with flags `O_CREAT` and `O_EXCL` set. This guarantees that the combined check and creation operation is atomic with respect to all other processes in the system that are trying to do the same thing at the same time, providing a convenient and portable mutual exclusion mechanism using the filesystem.

A common idiom for turning all `open` errors into exceptions (such as when a needed file is not found) is

```
fd := fileno open (f, ...);
```

because `fileno` raises an exception on `om` but returns its argument when that argument is a valid fd.

1.124.4 Buffering

A significant difference between file descriptors that are open at the SETL level and the "raw" POSIX file descriptors that underlie them at the system level is that at the SETL level, every fd returned by `open` is subject to buffering. (In C, buffering is usually done by the "stdio" layer using a pointer to a `FILE` object, and the fd is contained within that object. In SETL, by contrast, the fd is used as a handle for both the buffering structure and the underlying system fd.)

The main consequence of this implicit buffering in SETL is that for programs which communicate with other programs, it is important to `flush` output before attempting input, or a deadlock is likely to result.

Fortunately, SETL takes care of most of this automatically. In particular, for a bidirectional fd such as a socket or pump stream, `flush` is implicitly called whenever an output operation on that fd is followed by an input one.

For the common situation where a child process communicates with its parent over `stdin` and `stdout` (i.e., a child process designed to be used through a '`pump`' stream), the call `tie (stdin, stdout)` can be very helpful, as this causes `stdout` to be `flush`ed whenever an output operation on `stdout` is followed by an input operation on `stdin`. Indeed, the presence of precisely that call is often the mark of such a process.

1.124.5 Testing for openness

The `is_open` operator can be used to test whether a stream is open without otherwise disturbing it, and is in fact the only thing you can call without error or side-effects on a stream that is not open.

1.124.6 Automatic opening (and closing) of files

The routines `geta`, `getb`, `getc`, `getfile`, `getline`, `getn`, `gets`, `nprinta`, `peekc`, `printa`, `puta`, `putb`, `putc`, `putfile`, `putline`, `puts`, `reada`, `rewind`, `seek`, `ungetc`, and `writea` open files automatically on first reference if they can.

Any file that has been automatically opened for input will be automatically closed on end-of-file. The only routine that auto-closes an auto-opened output file is `putfile`.

Similarly, file descriptors that are open at the system level but not at the SETL level are automatically opened on a first reference by any of the above routines. Auto-closing follows the same rules as for auto-opened files. For mere file descriptors, this means closing only at the SETL level. For an fd that is *not* opened automatically, but rather explicitly (at the SETL level using `open` on the fd), this has the interesting consequence that one `close` operation will only close the fd at the SETL level. To close it at the system level too, a second `close` is required.

1.124.7 TCP sockets

The call to `open` a TCP connection (mode '`socket`') can block for an unspecified length of time.

1.124.8 UDP sockets

The only I/O (data-transferring) operations allowed on UDP client sockets are `send` and `recv`, and the only ones allowed on UDP server sockets are `sendto` and `recvfrom`. And conversely, `recv` and `send` can only be used on UDP client sockets, and `recvfrom` and `sendto` can only be used on UDP server sockets.

1.124.9 Pumps and tty-pumps

For a '`pump`' stream, the standard input and output of the child process are connected to the caller of `open` through a socket created by POSIX `socketpair()`.

For a '`tty-pump`' stream, however, its standard input and output are connected to the slave side of a pseudo-terminal (pty) in "raw" mode, while the `open` caller gets the fd of the master.

Giving the child process a terminal-like environment will normally cause it to line-buffer its output by default. This can be very helpful in allowing off-the-shelf "filtering" programs to be used as line-by-line processors, or to drive programs that are nominally intended to interact with users at a terminal (like the `expect` command on some systems does).

1.124.10 Signal streams

When the *`how`* argument to `open` is '`signal`' or '`ignore`', the *`f`* arument must be one of the following (case-insensitive) signal names, with or without the '`SIG`' prefix:

signal name	**default action**	**usual meaning**
'`SIGHUP`'	terminate process	modem hangup, or reread config file
'`SIGINT`'	terminate process	interrupt from keyboard (e.g., `ctrl-C`)
'`SIGQUIT`'	terminate process	quit from keyboard (e.g., `ctrl-\`)
'`SIGUSR1`'	terminate process	user-defined signal 1
'`SIGUSR2`'	terminate process	user-defined signal 2
'`SIGPIPE`'	terminate process	write to pipe or socket with no readers
'`SIGTERM`'	terminate process	software termination request
'`SIGCHLD`'	ignore	child process stopped or terminated
'`SIGCONT`'	ignore	continue after stoppage
'`SIGPWR`'	ignore	low battery, or power failure imminent
'`SIGWINCH`'	ignore	terminal window size change

Whenever a signal is received and there is at least one signal stream open on that signal type, a line is delivered to every such stream. Currently the line is empty, but applications should anticipate future information content by reading an arbitrary line, e.g. by using `getline` or `geta`.

Otherwise, when a signal is received and there is at least one signal-ignoring stream open on that signal type, the signal is ignored.

Otherwise, the signal's effect defaults to the action specified in the table above. All sequential input routines and `select` can be used on signal streams just as they can on regular input streams.

When the last (signal or signal-ignoring) stream for a given signal type is closed, the signal disposition reverts to its default.

SIGPWR and SIGWINCH are not required by POSIX, but are widely supported.

In the GNU SETL implementation, SIGCHLD is always caught internally; ignoring or receiving it at the SETL level is transparently simulated.

1.124.11 Timer streams

When `open` is called with *`how`* = '`real-ms`', the resulting fd delivers a newline every `val` *`f`* milliseconds.

That fd can also be used with `select`.

1.124.12 Predefined streams (needing no `open`)

There are three predefined streams with the following aliases:

fd	aliases	meaning
`stdin` (0)	`""`, `"-"`, `"stdin"`, `"STDIN"`, `"input"`, `"INPUT"`	standard input
`stdout` (1)	`""`, `"-"`, `"stdout"`, `"STDOUT"`, `"output"`, `"OUTPUT"`	standard output
`stderr` (2)	`"stderr"`, `"STDERR"`, `"error"`, `"ERROR"`	standard error

Files whose actual names are '`input`', '`ERROR`', etc. may still be referred to by explicitly opening them before starting I/O on them. This will cause such names not to act as standard aliases again until they are closed as streams.

The empty string (`""`) acts as `stdin` or `stdout` depending on the direction of the stream operation. Likewise for the hyphen (`"-"`).

You can `close` `stdin`, `stdout`, or `stderr` at any time, and by the rules of POSIX, the next `open` will choose the lowest fd, providing a mechanism by which you can implement redirection *à la* shell. See also `dup2`.

1.124.13 Alternative *`how`* arguments for `open`

The following synonyms for the *`how`* argument also exist, but are not particularly recommended:

'`a`'	'`AB`'
'`a`'	'`APPEND`'
'`a`'	'`BINARY-APPEND`'
'`a`'	'`CODED-APPEND`'
'`a`'	'`OUTPUT-APPEND`'
'`a`'	'`PRINT-APPEND`'
'`a`'	'`TEXT-APPEND`'
'`a+`'	'`A+B`'
'`a+`'	'`AB+`'
'`ignore`'	'`IGNORE-SIGNAL`'
'`ignore`'	'`SIGNAL-IGNORE`'
'`n`'	'`BINARY-NEW`'
'`n`'	'`CODED-NEW`'
'`n`'	'`NB`'
'`n`'	'`NEW`'
'`n`'	'`NEW-BINARY`'
'`n`'	'`NEW-CODED`'
'`n`'	'`NEW-TEXT`'
'`n`'	'`NEW-W`'
'`n`'	'`TEXT-NEW`'
'`n+`'	'`BINARY-DIRECT-NEW`'
'`n+`'	'`BINARY-RANDOM-NEW`'
'`n+`'	'`DIRECT-BINARY-NEW`'
'`n+`'	'`DIRECT-NEW`'
'`n+`'	'`N+B`'
'`n+`'	'`NB+`'

'n+'	'NEW+'
'n+'	'NEW-BINARY-DIRECT'
'n+'	'NEW-BINARY-RANDOM'
'n+'	'NEW-DIRECT'
'n+'	'NEW-DIRECT-BINARY'
'n+'	'NEW-R+'
'n+'	'NEW-RANDOM'
'n+'	'NEW-RANDOM-BINARY'
'n+'	'NEW-W+'
'n+'	'RANDOM-BINARY-NEW'
'n+'	'RANDOM-NEW'
'pipe-from'	'PIPE-IN'
'pipe-to'	'PIPE-OUT'
'r'	'BINARY'
'r'	'BINARY-IN'
'r'	'CODED'
'r'	'CODED-IN'
'r'	'INPUT'
'r'	'RB'
'r'	'TEXT'
'r'	'TEXT-IN'
'r+'	'BINARY-DIRECT'
'r+'	'BINARY-RANDOM'
'r+'	'DIRECT'
'r+'	'DIRECT-BINARY'
'r+'	'R+B'
'r+'	'RANDOM'
'r+'	'RANDOM-BINARY'
'r+'	'RB+'
'rw'	'BIDIRECTIONAL'
'rw'	'INPUT-OUTPUT'
'rw'	'READ-WRITE'
'rw'	'TWO-WAY'
'rw'	'TWOWAY'
'server-socket'	'TCP-SERVER-SOCKET'
'signal'	'SIGNAL-IN'
'socket'	'CLIENT-SOCKET'
'socket'	'TCP-CLIENT-SOCKET'
'tty-pump'	'LINE-PUMP'
'w'	'BINARY-OUT'
'w'	'CODED-OUT'
'w'	'OUTPUT'
'w'	'PRINT'
'w'	'TEXT-OUT'
'w'	'WB'
'w+'	'W+B'
'w+'	'WB+'

1.125 Logical disjunction

```
op or (boolean, boolean) : boolean
```

The expression

```
x ord y
```

is equivalent to the expression

```
if x then true else y end
```

for boolean `x` and `y`, which is to say that the `or` operator is "short-circuited" like `and` and the "query" operator (`?`), making it likewise suitable for use as a guard.

See also the bitwise operators such as `bit_or`.

1.126 Byte packing

```
op pack_short (integer) : string
op pack_unsigned_short (integer) : string
op pack_int (integer) : string
op pack_unsigned_int (integer) : string
op pack_long (integer) : string
op pack_unsigned_long (integer) : string
op pack_integer (integer) : string
op pack_double (real) : string
op pack_real (real) : string
```

These are low-level, platform-dependent operators for obtaining strings that internally represent numeric C types.

Except for `pack_integer`, the packers that expect an `integer` check to make sure the argument will fit in the C type. They then pack the integer value into a string of the C type's size, using 2's complement for signed types.

For example, the following program will print '`6C00`' on a little-endian machine where `short` is 2 bytes wide, and '`006C`' on a similar but big-endian machine:

```
print (hex pack_short 16#6c);
```

(If this seems backwards to you, remember that in forming the string, the `pack_...` operator reads bytes starting with the byte numbered 0.)

The `pack_integer` and `pack_real` operators give a representation of the argument that can be inverted without loss of information by `unpack_integer` and `unpack_real` respectively.

Finally, `pack_double` converts the `real` argument to a C `double` and returns a string containing that C representation. The 64-bit IEEE 754 format is widely used.

See also `mem_fetch_...`, `mem_store_...`, and `reverse`.

1.127 Peek at next character in input stream

```
op peekc (string f) : string
op peekc (integer f) : string
```

The next available character, if any, in the input stream `f` is returned as a string of length 1, just as with `getc`. However, the character also remains in the input stream as if it had

been "pushed back" by ungetc. If there are no more characters (the end of the input was reached), peekc behaves exactly the same as getc, returning om.

If *f* is not already open, an attempt will automatically be made to open it for reading.

See also peekchar, ungetc, and ungetchar.

1.128 Peek at next character in stdin

```
proc peekchar : string
```

Equivalent to peekc (stdin).

1.129 Peer host address

```
proc peer_address (integer f): string
proc peer_address (string f): string
```

If *f* refers to a socket in a "connected" state, peer_address returns the remote peer's Internet (IP) address in dotted notation.

See also open, peer_name, peer_port, ip_addresses, ip_names, hostaddr, and hostname.

1.130 Peer hostname

```
proc peer_name (integer f): string
proc peer_name (string f): string
```

If *f* refers to a socket in a "connected" state, peer_name returns the remote peer's Internet (DNS) name. If the name cannot be found, peer_name returns om.

See also open, peer_address, peer_port, ip_names, ip_addresses, hostname, and hostaddr.

1.131 Peer port number

```
proc peer_port (integer f): integer
proc peer_port (string f): integer
```

If *f* refers to a socket in a "connected" state, peer_port returns the remote peer's TCP or UDP port number.

See also open, port, peer_address, and peer_name.

1.132 Test for existence of processes

```
op pexists (integer p) : boolean
```

Tests whether the process or set of processes identified by *p* exists, according to the same rules as for the *p* argument to kill.

Note that if *p* exists, pexists (*p*) will return true even if the caller does not have permission to send a signal to it. These cases can, however, be distinguished by using clear_error, last_error, and no_error:

```
clear_error;
if pexists p then
  if last_error = no_error then
    -- we can send a signal to p
    ...
  else
    -- p exists, but we cannot signal it
    ...
  end if;
else
  -- p does not exist
  ...
end if;
```

See also pid.

1.133 Process ID of current or child process

```
proc pid : integer
proc pid (string) : integer
proc pid (integer) : integer
```

Called with no argument, pid returns the POSIX process ID of the current process. Called with an argument that refers to an open pipe, pump, or tty-pump stream, it returns the child's process ID.

See also getpgrp, setpgrp, pipe_from_child, pipe_to_child, pump, pexists, kill, and the open modes 'pump', 'tty-pump', 'pipe-from', and 'pipe-to'.

1.134 Create primitive pipe

```
proc pipe : tuple
```

This is a low-level interface to POSIX socketpair().

It returns a pair of linked bidirectional file descriptors (integers) that are open at the system level but not at the SETL level, such that output to one fd is presented as input to the other, and vice versa.

This is chiefly only of use when you are (perhaps as a classroom exercise) managing processes at the level of fork, dup2, close, exec, and waitpid. Otherwise, the higher-level primitives pump, pipe_from_child, pipe_to_child, filter, system, and the open modes 'pump', 'tty-pump', 'pipe-from', and 'pipe-to' are likely to cover your needs more conveniently.

1.135 Pipe from child process

```
proc pipe_from_child : integer
```

The pipe_from_child primitive is a degenerate form of pump (which see for further details on failure, flushing and FD_CLOEXEC).

It creates a child process, and returns to the calling process a readable fd connected to the standard output of that child. In the child process, pipe_from_child returns -1.

See also pipe_to_child and the open mode 'pipe-from'.

1.136 Pipe to child process

```
proc pipe_to_child : integer
```

The `pipe_to_child` primitive is a degenerate form of `pump` (which see for further details on failure, flushing and `FD_CLOEXEC`).

It creates a child process, and returns to the calling process a writeable fd connected to the standard input of that child. In the child process, `pipe_to_child` returns -1.

See also `pipe_from_child` and the `open` mode '`pipe-to`'.

1.137 Retrieve port number

```
op port (string f) : integer
op port (integer f) : integer
```

Returns the local TCP or UDP port number associated with a client or server socket *`f`*.

See also `peer_port` and `open`.

1.138 Power set

```
op pow (set s) : set
```

Returns the set of all 2 sup ♯*s* subsets of *`s`*, including the empty set `{}` and *`s`* itself.

See also `npow`.

1.139 "Prettify" string

```
op pretty (var) : string
```

If the argument is not already a `string`, the `pretty` operator first converts it to one as if by calling `str`. It then returns a copy of that `string` in which the 95 characters that ASCII considers "printable" are left unmolested, except for the apostrophe (single quote, `'`), which becomes two apostrophes in a row, and the backslash (\), which becomes two backslashes in a row. An apostrophe is also added at each end. Among the other codes, the audible alarm, backspace, formfeed, newline, return, horizontal tab, and vertical tab are converted to `\a`, `\b`, `\f`, `\n`, `\r`, `\t`, and `\v` respectively (these are the same as the C conventions), and all remaining codes are converted to *`\ooo`* (backslash followd by 3 octal digits).

See also `unpretty` and `unstr`.

1.140 Print to `stdout`

```
proc print (...)
```

Equivalent to `printa (stdout, ...)`.

1.141 Print to output stream

```
proc printa (string f, ...)
proc printa (integer f, ...)
```

There can be 0 or more arguments after *`f`*, of any type. They are sent in sequence to the stream *`f`*, separated by single spaces and followed by a newline character. String arguments are written directly; all others are written as if they had been passed through `str`.

If *f* is not already open, an attempt is made to open it automatically for writing.

See also print and nprinta (which omits the trailing newline).

1.142 Bidirectional stream to child process

```
proc pump : integer
```

The pump primitive creates a child process as fork does, but returns in the parent a bidirectional stream that is connected to the child's standard input and output. The POSIX-level FD_CLOEXEC flag is set on this stream, in sympathy with POSIX popen(). Other streams that are open in the parent are left open in the child, and pump returns -1 in the child.

If the system cannot create a new process, or if it cannot create the requisite bidirectional stream, pump returns om.

Output buffers are flushed before the attempt to spawn a child process is made.

See also open (particularly the 'pump' and 'tty-pump' modes), tie, close, shutdown, pipe_from_child, pipe_to_child, and filter.

1.143 Put lines on stdout

```
proc put (...)
```

Equivalent to puta (stdout, ...).

Note that this signature for put follows that of SETL2, and puta looks like the CIMS SETL put. This brings the signatures of put and puta into alignment with those of print and printa.

1.144 Put lines on output stream

```
proc puta (string f, ...)
proc puta (integer f, ...)
```

There can be 0 or more string arguments after the stream handle *f*. They are written directly to *f*, with a newline character after each.

If *f* is not already open, an attempt is made to open it automatically for writing.

A synonym for puta is putline.

See also printa.

1.145 Put values on output stream

```
proc putb (string f, ...)
proc putb (integer f, ...)
```

There can be 0 or more arguments after *f*, of any type. They are sent in sequence to the stream *f*, separated by single spaces and followed by a newline character. All of them are written as if they had been passed through str first, with no exception for strings (contrast printa).

Values written by putb, except for atoms (see newat) and procedure references (see routine), can be read by getb.

If *f* is not already open, an attempt is made to open it automatically for writing.

This procedure is functionally identical to writea.

See also puta.

1.146 Put characters on output stream

```
proc putc (string f, string c)
proc putc (integer f, string c)
```

The 0 or more characters in *c* are sent to the stream *f*.

If *f* is not already open, an attempt is made to open it automatically for writing.

See also putfile.

1.147 Put characters on stdout

```
proc putchar (string c)
```

The call putchar (*c*) is the same as putc (stdout, *c*).

1.148 Set environment variable

```
proc putenv (string)
```

The argument should have the form '*name*=*value*' as in POSIX putenv(). However, it is possible on some systems to "unset" an environment variable by passing just the '*name*' part. The use of setenv and unsetenv is recommended in preference to the rather old-fashioned putenv.

See also getenv.

1.149 Put characters on output stream

```
proc putfile (string f, string c)
proc putfile (integer f, string c)
```

The 0 or more characters in *c* are sent to the stream *f*.

If *f* is not already open, an attempt is made to open it automatically for writing.

The putfile routine is equivalent to putc, except that putfile will automatically close a file that was automatically opened.

See also getfile.

1.150 Put lines on output stream

```
proc putline (string f, ...)
proc putline (integer f, ...)
```

There can be 0 or more string arguments after the stream handle *f*. They are written directly to *f*, with a newline character after each.

If *f* is not already open, an attempt is made to open it automatically for writing.

A synonym for putline is puta.

See also printa.

1.151 Direct-access write

```
proc puts (string f, integer start, string x)
proc puts (integer f, integer start, string x)
```

The direct-access stream *f* (see open modes 'r+', 'w+', 'a+', and 'n+') is viewed as a string, where *start* specifies the index (1 or higher) of the first character to write. The puts procedure writes *n* characters from *f*, increasing the size of the file as necessary.

If *f* is not already open, an attempt is made to open it automatically in 'r+' mode, which allows seeking, reading, and writing.

See also gets, seek, and rewind.

1.152 Pseudo-random numbers and selections

```
op random (integer i) : integer
op random (real r) : real
op random (string s) : string
op random (set t) : var
op random (tuple t) : var
```

For an integer *i* >= 0, random *i* returns a pseudo-random integer in {0..*i*}, or in (non-SETL) algebraic terms, the closed interval [0,*i*]. For *i* < 0, the return value is in {-*i*..0}.

For a positive real (floating-point) *r*, random *r* returns a real in the half-open interval [0,*r*); and for negative *r*, in (*r*,0]. As a special case, random 0.0 returns 0.0.

The discrepancy between the use of a closed interval in the integer case and the half-closed interval in the real case, and the special treatment of 0.0, is an unfortunate SETL idiosyncrasy. The use of a closed interval at all in this context is in fact dissonant with the semantics of most other programming languages.

For a string *s*, random *s* returns a pseudo-randomly chosen character from *s*, or om if *s* is the empty string ("").

For a set or tuple *t*, random *t* returns a pseudo-randomly chosen element from *t*, or om if #*t* = 0.

See also setrandom.

1.153 Range of map

```
op range (set) : set
```

The argument must be a set of ordered pairs, that is, a set of tuples each of size 2. The result is the set of all second members of those tuples.

See also domain.

1.154 Extract final character using character set

```
proc rany (rw string s, string p) : string
```

If the last character of *s* occurs anywhere in the string *p*, that character is removed from *s* and returned as the function result. Otherwise, nothing happens to *s*, and the empty string ("") is returned.

See also any, break, len, match, notany, span, rbreak, rlen, rmatch, rnotany, and rspan.

1.155 Extract final substring using character set

```
proc rbreak (rw string s, string p) : string
```

Starting from the right, if *s* contains a character that appears in *p*, the substring of *s* up to but not including that character is removed from the end of *s* and returned as the function result, while *s* itself is updated to reflect the loss. Otherwise (no character from *p* appears in *s*), the function result is the input value of *s*, and *s* is reduced to the empty string ("") by the operation.

See also any, break, len, match, notany, span, rany, rlen, rmatch, rnotany, and rspan.

1.156 Extract final substring by length

```
proc rlen (rw string s, integer n) : string
```

The lesser of *n* and #*s* characters are removed from the end of *s* and returned as the function result.

See also any, break, len, match, notany, span, rany, rbreak, rmatch, rnotany, and rspan.

1.157 Extract final substring by exact match

```
proc rmatch (rw string s, string p) : string
```

If *p* occurs as the final substring in *s*, it is removed from *s* and returned as the function result. Otherwise, nothing happens to *s* and the empty string ("") is returned.

See also any, break, len, match, notany, span, rany, rbreak, rlen, rnotany, and rspan.

1.158 Extract final character using character set

```
proc rnotany (rw string s, string p) : string
```

If the last character of *s* does not occur anywhere in the string *p*, that character is removed from *s* and returned as the function result. Otherwise, nothing happens to *s*, and the empty string ("") is returned.

See also any, break, len, match, notany, span, rany, rbreak, rlen, rmatch, and rspan.

1.159 Extract final substring using character set

```
proc rspan (rw string s, string p) : string
```

The longest trailing substring of *s* consisting of characters that are in the string *p* is removed from *s* and returned as the function result. Otherwise, nothing happens to *s*, and the empty string ("") is returned.

See also any, break, len, match, notany, span, rany, rbreak, rlen, rmatch, and rnotany.

1.160 Get one or more values from stdin

```
proc read (wr var...)
```

Equivalent to reada (stdin, ...).

1.161 Get one or more values from input stream

```
proc reada (string f, wr var...)
proc reada (integer f, wr var...)
```

Values are read from the input stream *f* and assigned to the succeeding arguments in order. If the end of input is reached "early", trailing arguments get om.

Values written by writea, except for atoms (see newat) and procedure references (see routine), are guaranteed to be readable by reada.

If *f* is not already open, an attempt is made to open it automatically for reading.

There is an important difference between reada and getb in that reada will always start reading at the beginning of a line, skipping ahead to just after the next newline (\n) character if necessary, whereas getb will simply start with the next available character in the input stream. Both routines will happily cross newline boundaries to obtain more values, however.

See also read, geta, getc, getn, unstr, and val.

1.162 Symbolic link referent

```
op readlink (string f) : string
```

When *f* names a file that is really a symbolic link, readlink returns the string associated with *f*. The resulting string may or may not name another existing file.

Contrast this with a regular input operation on *f*, which will fail if *f* is a symbolic link to a file that doesn't exist.

If *f* itself doesn't exist, or is not a symbolic link, readlink returns om, and last_error indicates which case applies.

See also lexists, fexists, symlink, link, and unlink.

1.163 Get one or more values from a string

```
proc reads (string s, wr var...)
```

Values are "read" from the string *s* and assigned to the succeeding arguments in order. If the end of the string is reached "early", trailing arguments get om. The rules for value recognition and conversion are the same as for reada and unstr.

See also val.

1.164 Receive datagram on UDP client socket

```
op recv (string f) : string
op recv (integer f) : string
```

A datagram is read from the UDP client socket *f* and returned as a string. The select function can be used to check or wait for input of this kind.

See the open mode 'udp-client-socket', and see also recvfrom, send, and sendto.

1.165 Receive datagram on UDP server socket

```
op recvfrom (string f) : tuple
op recvfrom (integer f) : tuple
```

A datagram is read from the UDP server socket *f*, and the sender's IP address and port number are sensed. These are all bundled together as the pair of strings [*source*, *datagram*] and returned, with *source* formatted as '*IP_address*:*portnum*'.

The select function can be used to check or wait for input of this kind.

See the open mode 'udp-server-socket', and see also sendto, recv, and send.

1.166 Receive file descriptor

```
op recv_fd (string f) : integer
op recv_fd (integer f) : integer
```

A file descriptor (fd) sent by a process executing a send_fd is returned and left open at the underlying system level. It is not immediately opened at the SETL level, but will be auto-opened in the usual way by the first input or output operation (see Section 1.124.6 [automatic opening], page 33) if it is not opened explicitly. The stream handle *f* should refer to a Unix-domain socket, such as is created by pump, pipe_from_child, pipe_to_child pipe, and the open modes 'pump', 'tty-pump', 'pipe-from', and 'pipe-to'. Other types of sockets may also be used in some (here unspecified) circumstances on some systems.

The integer value of the returned fd is chosen by recv_fd in the manner of dup, and refers to the same system-level object as the fd that was passed to send_fd.

The select function can be used to test or wait for the presence of a file descriptor ready to be received on *f*.

Note that there is no "rendezvous" between send_fd and recv_fd, so in general, the caller of send_fd should take care not to close the fd before it has some positive indication from the caller of recv_fd that the fd has actually been received (this will be true by the time recv_fd returns, so any acknowledgement sent by the recv_fd caller and received by the send_fd caller after that point will do).

On some systems (particularly message-passing systems such as QNX Neutrino), this is more than a merely theoretical concern, because if the fd of a connection to a resource that is being passed in this way has already been closed by the time the receiver tries to make a "copy" of it, the fd will appear to be "bad".

The reason such an interlock cannot in general be provided by a SETL implementation is that the channel over which the fd is to be passed may be only unidirectional.

Here is an example in which a child process opens files and passes their file descriptors up to its parent. It is a skeletal design pattern for a situation where the child process is opening something trickier than mere local files (e.g., crash-prone remote connections or resources requiring special privileges). We begin with the parent process, which expects a list of filenames on the command line, uses the child to open them, and prints their contents on stderr:

```
-- fd-pass-main.setl
pd := fileno open ('setl fd-pass-pump.setl', 'pump');
for a in command_line loop
  printa (stderr, '<<', a, '>>');  -- print filename
  printa (pd, a);  -- pass filename to child
  fd := recv_fd pd;  -- get fd from child
  printa (stderr, getfile fd);  -- get and print file
  close (fd);  -- be tidy and don't leak
end loop;
close (pd);  -- close connection to child
```

Now for the child program. The "rendezvous" is accomplished by not closing the fd until the next filename or end-of-file is received from the parent, by which time the parent has certainly completed its `recv_fd` call (indeed, by then it has even finished *using* the fd):

```
-- fd-pass-pump.setl
tie (stdin, stdout);  -- flush stdout on read from stdin
fd := om;  -- initial state
while (s := getline stdin) /= om loop
  if fd /= om then
    close (fd);  -- close old fd
  end if;
  fd := open (s, 'r');  -- connect to file
  send_fd (stdout, fd);  -- send fd to parent
end loop;
close (fd);  -- final close, for show
```

1.167 Integer remainder

```
op rem (integer n, integer d) : integer
```

For non-zero *d*,

```
n rem d = n - ((n div d) * d
```

so, for example:

```
 5 rem  3 =  2
-5 rem  3 = -2
 5 rem -3 =  2
-5 rem -3 = -2
```

In contrast with `mod`, the sign of the result follows that of the numerator, and its magnitude depends only on the magnitudes of the operands (not on their signs).

1.168 Reverse string

```
op reverse (string) : string
```

Characters in reverse order.

1.169 Rewind direct-access stream

```
proc rewind (string f)
```

```
proc rewind (integer f)
```

Equivalent to `seek (f, 0)`.

1.170 Round to nearest integer

```
op round (real) : integer
op round (integer) : integer
```

Numbers ending in .5 are rounded away from zero.

See also `floor`, `ceil`, `fix`, and `float`.

1.171 Create procedure reference

```
op routine (proc_name) : proc_ref
```

This pseudo-operator produces a value that can subsequently be passed to `call` in order to perform an indirect procedure call. The typenames in the signature shown here do not really exist as SETL keywords, but suggest how this operator is used: you pass it the name of a procedure in your program, and `routine` returns an opaque handle which you can save for later use.

For example, it is sometimes convenient to use a mapping to associate strings with procedure references, as is illustrated by the "callback" style of programming in the example at `select`.

Another, possibly more familiar, application is the generic numerical integration function to which you pass a reference to the function you wish to integrate over:

```
area := integrate (routine g, 0, 1, 100);

proc g(x);
  return sin x;  -- test example
end g;

proc integrate (f, x0, x1, n);  -- simple and crude
  const dx = (x1 - x0) / n;
  return +/[dx * call (f, x0 + dx*(i - 0.5)) : i in {1..n}];
end integrate;
```

1.172 Pad string on right with blanks

```
proc rpad (string s, integer n) : string
```

If *n* > #*s*, the returned string is a copy of *s* padded on the right with blanks to length *n*. Otherwise, a copy of *s* is returned.

See also `lpad`.

1.173 Reposition direct-access stream

```
proc seek (string f, integer offset)
proc seek (integer f, integer offset)
```

The direct-access stream *f* (see `open` modes '`r+`', '`w+`', '`a+`', and '`n+`') is repositioned so that the next ordinary read or write operation will start at *offset* bytes past the beginning of

the file. Note that *offset* should be 0 or more, consistent with the conventions of POSIX `fseek()`, unlike the *start* parameter of `gets` and `puts`, which is 1-origined. Thus *offset* is equivalent to *start* `- 1`.

If *f* is not already open, an attempt is made to open it automatically in '`r+`' mode, which allows seeking, reading, and writing.

See also `rewind`.

1.174 Wait for I/O event or timeout

```
proc select (tuple fds) : tuple
proc select (tuple fds, integer ms) : tuple
```

This is an extended interface to POSIX `select()`, which allows a program to wait for I/O events on multiple streams simultaneously, optionally taking a timeout or immediate return.

Because interprocess communication, signals, and interval timers are all wrapped as I/O streams in SETL too, this is the fundamental function for event-driven programming in SETL.

The *fds* argument contains up to 3 sets of stream handles, where *fds*`(1)` lists streams that may produce input (the meaning of this is actually extended to included TCP server sockets that are ready to `accept` without blocking, UDP sockets that have datagrams ready to be received by `recv` or `recvfrom`, and Unix-domain sockets on which `recv_fd` can be called without blocking), *fds*`(2)` lists streams that take output (including UDP sockets ready for `send` or `sendto` operations, and Unix-domain sockets ready for `send_fd`), and *fds*`(3)` lists streams that can generate exceptional conditions (the meaning of which is system-dependent). For convenience, an empty *fds* tuple can be indicated by passing *fds* as `om`.

The *ms* argument, if present, gives the number of milliseconds `select` should wait for at least one of the stream handles given in *fds* to become ready. If *ms* is 0, the file descriptors are polled without waiting. If *ms* is absent, `select` waits indefinitely.

The return value from `select` is a 3-tuple of fd sets corresponding respectively to input, output, and exceptional streams that are ready. In the case of a timeout, all 3 sets will be empty.

Because the contract of `select` only states that *some* output can be written when an fd appears in the second set of the returned triple, but not *how much* output, testing for output readiness is not deterministic enough to be often useful in practice, given that for sockets there is buffering at both the SETL level and the system level.

Testing for input, on the other hand, works well with trusted peers on the local network because input actually has to be received before the input fd will become ready, and it is generally very straightforward (especially in SETL) to arrange a simple protocol that ensures this only happens when the sender has a "whole" message ready to send. Indeed, a common design pattern for dealing with remote peers is to interpose a small local "pumping" child process that handshakes with the parent and deals with the peer no matter whether the communication with the peer is pure input, pure output, or both. The parent uses `select` to be notified by the (local, trusted) child process that a whole piece of input has been received, and/or to be notified that the child process is ready for the next whole piece of output. As long as the agreement includes a limit on the sizes of these pieces, the parent

can then easily be engineered to remain responsive even in the face of misbehaving remote peers, without having to resort to threads or asynchronous I/O, neither of which complex and error-prone features exist in SETL.

The "callback" style of event-driven programming can be supported quite easily by passing control to a routine that repeatedly calls `select` and looks up what function to call from a map over file descriptors. Here is an example, beginning with some global variables and the event dispatcher:

```
var callbacks := {};
 ...
proc dispatcher;
  var cb_map, ready, fd;
  loop
    cb_map := callbacks;
    [ready] := select ([domain cb_map]);
    for fd in ready loop
      call (cb_map(fd), fd);
    end loop;
    done_round;
  end loop;
end dispatcher;
```

Once called, this dispatch loop executes indefinitely until some callback causes the program to terminate.

To "register" a callback procedure `p` to be called upon input from a source `fd`:

```
callbacks(fd) := routine p;
```

To "deregister" the callback for a source `fd` that has been closed:

```
callbacks(fd) := om;
```

Notice that care is taken in the dispatcher to iterate over a fixed local copy of the `callbacks` map, so that updates of the global map due to registrations and deregistrations do not lead to incorrect calls being made. (If the global map was used directly, and a ready fd was closed and deregistered by some callback for another fd, then reused in the natural course of fd allocation and registered against a different callback, the latter could be called for the new incarnation of the fd because it still appeared in the "ready" set.) Even with this protection against a silly though subtle mistake being made by the general-purpose dispatcher, however, the callback routines themselves need to be aware of the consequences of opening and closing file descriptors in various orders, especially given how those small integers are used and reused. Because the order in which callback routines are called is by definition arbitrary, they may sometimes need to schedule work rather than doing it right away. And since by the same token, there is no way to know which callback routine will be called last, a call to some user-defined `done_round` routine also appears in the example dispatcher above.

These are the normal hazards of programming with callbacks in any language, and are arguably a high price to pay for the ostensible convenience of callbacks (allowing you to avoid writing your own main loop). In SETL, armed with this rather user-friendly `select`, it is often much simpler and clearer to write that main loop directly and avoid a host of potential subtle interactions among callbacks by testing for ready file descriptors in an order

you define and can easily understand and analyze, calling routines and/or executing small pieces of in-line code depending on the required complexity of the response.

1.175 Send datagram on UDP client socket

```
proc send (string f, string datagram)
proc send (integer f, string datagram)
```

The *datagram* is sent to the UDP client socket *f*.

The `select` function can be used to check or wait for *f* to be ready to take a datagram.

See the `open` mode '`udp-client-socket`', and see also `recv`, `sendto`, and `recvfrom`.

1.176 Send datagram on UDP server socket

```
proc sendto (string f, string address, string datagram)
proc sendto (integer f, string address, string datagram)
```

The *datagram* is sent via the UDP server socket *f* to the destination *address*, which should be formatted as a string of the form '*IP_address*`:`*portnum*'.

The `select` function can be used to check or wait for *f* to be ready to take a datagram.

See the `open` mode '`udp-server-socket`', and see also `recvfrom`, `send`, and `recv`.

1.177 Send file descriptor

```
proc send_fd (string f, integer fd)
proc send_fd (integer f, integer fd)
```

The file descriptor *fd* is sent to a process that has called (or will call) `recv_fd`. The stream handle *f* should refer to a Unix-domain socket, though other types of sockets may also be used in some (here unspecified) circumstances on some systems.

The *fd* should not be `close`d until after the corresponding `recv_fd` has verifiably completed.

The `select` function can be used to check or wait for *f* to be ready to take a file descriptor without blocking.

1.178 Set effective group ID

```
proc setegid (integer gid)
```

Calls POSIX `setegid()` on the group ID *gid*.

See also `setgid`, `getegid`, `getgid`, `seteuid`, `setuid` (details and example), `geteuid`, and `getuid`.

1.179 Set environment variable

```
proc setenv (string name, string value)
proc setenv (string name)
```

The call `setenv` *name*`,` *value* gives the environment variable *name* the value *value*. Omitting *value* is equivalent to specifying the empty string (`""`).

See also `getenv`, `unsetenv`, and the mildly deprecated `putenv`.

1.180 Set effective user ID

```
proc seteuid (integer uid)
```

Calls POSIX `seteuid()` on the user ID *`uid`*.

See also `setuid` (details and example), `geteuid`, `getuid`, `setegid`, `setgid`, `getegid`, and `getgid`.

1.181 Set group ID

```
proc setgid (integer gid)
```

Calls POSIX `setgid()` on the group ID *`gid`*.

See also `setegid`, `getgid`, `getegid`, `setuid` (details and example), `seteuid`, `getuid`, and `geteuid`.

1.182 Set process group ID

```
proc setpgrp
```

Make the current process a process group leader by making its process group ID equal to its process ID. This is done by calling POSIX `setpgid()` with both arguments 0, not by calling the XSI `setpgrp()`.

Note that process group IDs have nothing to do with user group IDs (contrast `setgid`).

In POSIX, *job control* shells place each successive command in a new process group. A command may in general contain multiple processes, and signals coming from the *controlling terminal* get sent only to the *foreground* command, of which there is at most one. There can be many "background" commands, however, and the job control shell can move commands between background and foreground. In order to avoid interfering with all this, the SETL programmer does well to avoid using `setpgrp` in programs intended to be run interactively, except in very special circumstances where a group of processes is to be backgrounded in a way that deliberately removes it from job control management.

See also `getpgrp`, `pid`, `pexists`, `kill`, and `waitpid`.

1.183 Set random seed

```
proc setrandom (integer seed)
```

Establishes a starting point for pseudo-random number generation. If *`seed`* is `0`, the starting point is supposed to be as unpredictable as the implementation can make it, given available sources of "entropy".

See also `random`.

1.184 Set user ID

```
proc setuid (integer uid)
```

Calls POSIX `setuid()` on the user ID *`uid`*.

See also `seteuid`, `getuid`, `geteuid`. `setgid`, `setegid`, `getgid`, and `getegid`.

A "setuid executable" file in the POSIX world is one that has the set-user-ID mode bit set (the POSIX command `chmod u+s ...` sets this bit, `chmod u-s ...` clears it, and `ls -l ...` displays whether it is set).

When a user (called the "real" user) other than the owner of such a file executes it, the "effective" user ID and "saved" user ID are initially defined to be those of the file's owner. The program executes with the privileges of the effective user, which in particular implies that upon starting, the program can manipulate the owner's files.

But the program has another important privilege too, namely the ability to switch the effective user ID back and forth between the real and the saved ID, using `seteuid`. (For a regular user, `setuid` is equivalent to `seteuid`, but for a user with appropriate privileges, historically called a "superuser", `setuid` sets all 3 ids and is in that respect like a one-way door.)

For example, suppose you are a professor, and you want your students to be able to submit homework programs and other files safely. Naturally, your IT department does not trust you, so you don't have special privileges. Armed with `seteuid`, however, it is easy to set up what you want: simply write your `submit` program such that when run by a student, it uses the student's privileges to fetch the files, and your privileges to store them in a private area you designate.

The situation for group ids and "setgid executables" is exactly analogous to that for users.

1.185 Determine the type of integer quotients

```
proc set_intslash (boolean) : boolean
```

Calling `set_intslash` is equivalent to retrieving the current value of `intslash` and then setting it to the value given by the `boolean` argument.

1.186 Recognize regular expressions

```
proc set_magic (boolean) : boolean
```

Calling `set_magic` is equivalent to retrieving the current value of `magic` and then setting it to the value given by the `boolean` argument.

1.187 Disable I/O in one or both directions

```
proc shutdown (string f, integer how)
proc shutdown (integer f, integer how)
```

If *f* designates a bidirectional stream, `shutdown` disables the I/O direction(s) indicated in the *how* argument (which may be any one of the predefined constants `shut_rd`, `shut_wr`, or `shut_rdwr`), using POSIX `shutdown()`.

The underlying file descriptor remains open, however.

For a TCP stream, `shutdown` can perform a "half-close", which can be used, for example, to tell a peer that you have finished sending data (thus making the peer see an end-of-file condition on its input side) but that you would still like to receive a reply on the same (still "half-open") connection.

It is permissible to call `shutdown` on a file descriptor that is not open at the SETL level but is open at the underlying system level, as with `close`.

See also `open`, `pump`, and `last_error`.

1.188 Constants for use with `shutdown`

```
shut_rd : integer
shut_wr : integer
shut_rdwr : integer
```

See `shutdown`.

1.189 Numeric sign

```
op sign (integer x) : integer
op sign (real x) : integer
```

Returns `-1`, `0`, or `1` according as *x* `<` `0`, *x* `=` `0`, or *x* `>` `0` respectively.

1.190 Sine

```
op sin (real) : real
op sin (integer) : real
```

The argument is in radians. See also `asin`.

1.191 Hyperbolic sine

```
op sinh (real) : real
op sinh (integer) : real
```

Enlarged nasal passage.

1.192 Extract initial substring by on character set

```
proc span (rw string s, string p) : string
```

The longest initial substring of *s* consisting of characters that are in the string *p* is removed from *s* and returned as the function result. Otherwise, nothing happens to *s*, and the empty string (`""`) is returned.

See also `any`, `break`, `len`, `match`, `notany`, `rany`, `rbreak`, `rlen`, `rmatch`, `rnotany`, and `rspan`.

1.193 Split string into tuple

```
proc split (string s, string p) : tuple
proc split (string s, tuple p) : tuple
proc split (string s) : tuple
```

Substrings of *s* are returned as a `tuple`, where the regular expression *p* is a "delimiter" pattern defaulting to whitespace, '`[ \f\n\r\t\v]+`'.

The subject string *s* is considered to be surrounded by strings satisfying the delimiter pattern *p*, and `split` returns the strings between the delimiters. So for example `split ("David Bacon::WGP:", ":")` is `["David Bacon", "", "WGP", ""]`, but when a line is split into whitespace-delimited words, extra whitespace on either end of the line does not appear as a empty string in the result of `split`. This is because the leading or trailing whitespace merges with the added surrounding whitespace from the delimiter pattern's point of view, and indeed there will be some end-merging whenever *p* matches 2 or more copies of a leading or trailing substring of *s*.

Note that `split ("", p) = []` for any pattern *p*: splitting the empty string yields the empty tuple.

See also `join`, `magic`, and *set_magic*.

1.194 Square root

```
op sqrt (real) : real
op sqrt (integer) : real
```

Where squares come from.

1.195 Child process termination status

```
status : integer
```

Status of the child process that was last successfully waited for by `filter`, `system`, `wait`, or `waitpid`, or by `close` as applied to a pipe, pump, or tty-pump stream.

If the child process exited "normally" (e.g., by executing a SETL `stop` statement or by reaching the end of the main program), `status` is non-negative; but if it was killed by a signal, `status` gives the negative of the signal number that terminated it.

See also `kill`, `pump`, `pipe_from_child`, `pipe_to_child`, and `open` modes '`pump`', '`tty-pump`', '`pipe-from`', and '`pipe-to`'.

1.196 Standard input, output, and error file descriptors

```
stdin : integer
stdout : integer
stderr : integer
```

The constant fd `stdin` is `0`, `stdout` is `1`, and `stderr` is `2`.

1.197 String representation of value

```
op str (var x) : string
```

The argument *x* is converted to a `string` such that (for most types) the result can be converted back to the same value using `unstr`.

Some loss of precision is possible in the case of floating-point (`real`) *x*. This can be overcome by the use of `fixed` or `floating` in place of `str`.

Procedure references are converted to '`<ROUTINE>`', and atoms are converted to strings of the form '`#n`', where *n* is some arbitrary sequence of decimal digits. Neither of these forms is "back-convertible" by `unstr`, as `str` and `unstr` are intended primarily for use in communication between programs (via files, pipes, or sockets).

For a string *x* that does not have the form of a SETL identifier (alphabetic character followed by 0 or more alphanumeric or underscore characters), `str x` is a copy of *x* except that each apostrophe (') is twinned (producing two apostrophes in a row), and an apostrophe is added at each end. If a string *x* *does* have the form of a SETL identifier, however, `str x = unstr x = x`.

See also also `pretty`, `unpretty`, `whole`, `strad`, `val`, and `printa`.

1.198 Radix-prefixed string representation of integer

```
proc strad (integer x, integer radix) : string
```

The integer argument *x* is converted to a string representing its value with an explicit radix. The *radix* argument, which must be an integer in the range 2 to 36, gives the "base" for the denotation, which is produced in the form '*radix#digits*', where the radix is represented in decimal and the digits after the sharp sign are drawn from the characters `0` through `9` and lowercase `a` through `z`. Examples:

```
strad (10, 10) = "10#10"
strad (10, 16) = "16#a"
strad (10, 2) = "2#1010"
strad (-899, 36) = "36#oz"
```

Strings produced by `strad` can be converted back to integers by `val`, `unstr`, and `reads`.

See also `whole`, `str`, `fixed`, and `floating`.

1.199 Replace pattern in string

```
proc sub (rw string s, string p) : string
proc sub (rw string s, tuple p) : string
proc sub (rw string s, string p, string r) : string
proc sub (rw string s, tuple p, string r) : string
```

The leftmost occurrence in *s*, if any, of the pattern given by the regular expression *p* is replaced by *r*, which defaults to the empty string (`""`). The substring of *s* replaced by this operation is returned as a `string`, unless *p* did not occur in *s*, in which case `om` is returned and *s* is left unmodified.

If *p* is a `tuple`, it must consist of a pair of `string` patterns *p1*, *p2*. In that case, it means everything from *p1* up to the first subsequent occurrence of *p2*.

Note that

```
sub (s, p, r)
```

is very similar to

```
s(p) := r
```

if *p* is a string, or

```
s(p1..p2) := r
```

if *p* is the tuple `[`*p1*`, `*p2*`]`.

See also `gsub`, `mark`, and `gmark`.

1.200 Subset test

```
op subset (set ss, set s) : boolean
```

Returns `true` if *ss* is a subset of *s*.

See also `incs`.

1.201 Create symbolic link

```
proc symlink (string f, string new)
```

Atomically create a symbolic link to *f* under the filename *new* using POSIX `symlink()`, if *new* does not exist before the call. There is no return value, but calling `clear_error` before the operation and inspecting `last_error` after it can be used to determine atomically whether the operation was successful. Thus `symlink` can be used to implement a "test and set" mutex lock in the filesystem: if *new* already exists, the operation will fail; and if it doesn't exist, it is created and the calling process then "owns" the lock. To relinquish it, the caller later calls `unlink` on *new*; or another process can "forcibly" remove it.

The `open` modes '`n`' and '`n+`' provide a similar mutual exclusion capability.

For symbolic links, in contradistinction to hard links, it does not matter whether *f* refers to an existing file or not in order for the `symlink` call to succeed. However, if *f* does exist and the `symlink` call does succeed, subsequent attempts to read or write *new* will be redirected into attempts to read or write *f*.

See also `link`, `readlink`, `lexists`, and `fexists`.

1.202 Execute command in subshell

```
proc system (string cmd) : integer
```

The command *cmd* is passed to POSIX `system()`. The resulting status is returned as the function result, and also stored in `status` (which see for details).

See also `filter` and the `open` modes '`pump`', '`tty-pump`', '`pipe-from`', and '`pipe-to`'.

1.203 Low-level read

```
proc sys_read (integer fd, integer n) : string
```

This function bypasses SETL buffering and calls POSIX `read()` directly. The *fd* may or may not be open at the SETL level (see Section 1.124.4 [buffering], page 32).

Up to *n* bytes are read from the *fd* and returned as a string. The string may be less than *n* bytes long if the end of the input is encountered before *n* bytes have been read, or if the *fd* is a socket (or pipe or pump) where the sender does a `sys_write` or equivalent after sending fewer than *n* bytes.

1.204 Low-level write

```
proc sys_write (integer fd, string s) : integer
```

This function bypasses SETL buffering and calls POSIX `write()` directly. The *fd* may or may not be open at the SETL level (see Section 1.124.4 [buffering], page 32).

An attempt is made to write all `#`*s* bytes of *s*. The actual number of bytes written is returned.

1.205 Trigonometric tangent

```
op tan (real) : real
op tan (integer) : real
```

The argument is in radians. See also `atan` and `atan2`.

1.206 Hyperbolic tangent

```
op tanh (real) : real
op tanh (integer) : real
```

An extreme diversion.

1.207 Auto-flush output stream upon other stream input

```
proc tie (string, string)
proc tie (string, integer)
proc tie (integer, string)
proc tie (integer, integer)
```

The arguments designate streams open at the SETL level. After the call to `tie`, whenever an input operation such as `reada` or `geta` is requested on one of the two streams, all buffered output on the other is written out (`flush`ed) first. Section 1.124.4 [buffering], page 32.

There is no `untie`; it appeals to the crude sense of symmetry but does not seem useful.

1.208 Elapsed CPU time in milliseconds

```
proc time : integer
```

This gives the total amount of CPU time used by the current process and all its child processes that have terminated and been waited for (automatically or by `wait` or `waitpid`), in milliseconds. This includes both "user" time and time spent by the operating system on behalf of the user.

See also `clock` and `tod`.

1.209 Unique temporary filename

```
proc tmpnam : string
```

This is an interface to POSIX `tmpnam()`, and returns a new, unique temporary filename each time it is called.

1.210 Convert string to lowercase

```
op to_lower (string) : string
```

A string of length equal to that of the argument is returned. Characters other than `A-Z` are unaffected.

1.211 Convert string to uppercase

```
op to_upper (string) : string
```

A string of length equal to that of the argument is returned. Characters other than `a-z` are unaffected.

1.212 Calendar time in milliseconds

```
proc tod : integer
```

This is the total number of milliseconds that have elapsed in the "epoch" beginning 1 January 1970 UTC. Abject apologies to Herr Wittgenstein.

See also clock, time, date, and fdate.

1.213 Predefined "true" boolean value

```
true : boolean
```

1.214 Type of SETL value

```
op type (var) : string
```

Returns 'ATOM', 'BOOLEAN', 'INTEGER', 'REAL', 'SET', 'STRING', 'TUPLE', 'PROC_REF', or 'OM'.

See also the is_... type-testing operators and denotype.

1.215 Set file creation mode mask

```
proc umask : integer
proc umask (integer) : integer
```

This is an interface to POSIX umask(). Both forms of the call return the current value of the mask, and the second form changes it to a new value. For example,

```
umask (8#022);
```

arranges that files created by the program and its child processes will not be writable by other users or groups unless subsequently made so by the POSIX chmod command.

1.216 Push characters back on input stream

```
proc ungetc (string f, string c)
proc ungetc (integer f, string c)
```

Immediately after any input operation on the stream *f* that yielded at least one character, the call ungetc (*f*, c) "pushes back" the characters in *c* onto *f* so that *c* will appear as the next input. At least one character of pushback is guaranteed after at least one character has been fetched. If *c* has at least one character, the call to ungetc also clears the eof status to false.

If *f* is not already open, an attempt is made to open it automatically for reading.

See also ungetchar, peekc, and peekchar.

1.217 Push characters back on stdin

```
proc ungetchar (string c)
```

Equivalent to ungetc (stdin, *c*).

1.218 Convert from hexadecimal

```
op unhex (string) : string
```

This is the inverse of hex, but returns om if its string argument does not contain an even number of (case-insensitively recognized) hexadecimal characters.

1.219 Destroy file reference

```
proc unlink (string f)
```

Remove the name *f* from the filesystem. If *f* is the last reference (link) to the underlying file, including "invisible" references from running processes, the space for the file is freed.

See also link, symlink, readlink, fexists, and lexists.

1.220 Byte unpacking

```
op unpack_short (string) : integer
op unpack_unsigned_short (string) : integer
op unpack_int (string) : integer
op unpack_unsigned_int (string) : integer
op unpack_long (string) : integer
op unpack_unsigned_long (string) : integer
op unpack_integer (string) : integer
op unpack_double (string) : real
op unpack_real (string) : real
```

These are low-level, platform-dependent operators for interpreting strings as C representations of numbers.

Apart from unpack_integer and unpack_real, you must pass a string of the right length for the C type suggested by the operator name.

For unpack_integer and unpack_real, you must pass a string that was (or could have been) returned by pack_integer or pack_real respectively.

See also mem_fetch_..., mem_store_..., and reverse.

1.221 "Unprettify" string

```
op unpretty (string s) : string
```

The string *s* should be in "pretty" form, although the unpretty operator is somewhat liberal in what it accepts relative to what pretty produces.

However, *s* must still begin and end with an apostrophe (single quote, ') or begin and end with a double quote (").

Inside *s*, every character must be one of the 95 characters ASCII considers "printable", including blank. The unpretty operator makes the following interpretations in transforming *s* into an unrestricted string:

Backslash (\) followed by any of the 32 "glyphs" (that is, all of the ASCII "printable" characters apart from alphanumerics and blank) means just that glyph.

Backslash followed by up to 3 octal digits means a character having the bit pattern indicated by the digits, as in C.

Backslash followed by x and then 1 or 2 hexadecimal digits is an alternative to the octal escape.

Backslash followed by a, b, f, n, r, t, or v means the same thing as it does in C, i.e., audible alarm, backspace, formfeed, newline, carriage return, horizontal tab, or vertical tab, respectively.

Currently these are exactly the rules governing what can be in a literal character string in SETL source code, and the escape sequences have the same meanings.

See also `pretty`, `unstr`, and `str`.

1.222 Remove environment variable definition

```
op unsetenv (string name)
```

If the environment variable *name* was defined, undefine it. Note that this is not the same as setting it to the empty string ("").

See also `setenv`, `getenv`, and poor old `putenv`.

1.223 Read value from string

```
op unstr (string s) : var
```

Essentially, this is the inverse of `str`, but more liberal in what it accepts relative to what `str` produces (except that it cannot handle the string representations of procedure references and atoms).

In particular, quoted strings passed to `unstr` can use either the apostrophe (') or the double quote (") as the beginning and ending quote character. Whichever one is used is also the one that should be twinned internally to represent that character. As a convenience (to be consistent with what `reada` accepts), each backslash (\) followed by a true newline character (not the sequence '`\n`') is silently absorbed when it occurs in a quoted string. Apart from the interpretation of twinned quotes and the backslash-newline absorption, `unstr` is completely literal about how it interprets what is inside a quoted string.

Other types are recognized by their first non-whitespace character, except that numeric types don't necessarily get resolved to `real` or `integer` until they are fully scanned.

Radix-prefixed integer denotations (see `strad`) are allowed to have trailing sharp signs (redundantly), just as in SETL source code.

The `denotype` operator can be used to check whether a string is acceptable to `unstr`.

See also `val`, `reads`, and `unpretty`.

1.224 Read numeric value from string

```
op val (string) : integer
op val (string) : real
```

This is similar to `unstr` but expects a numeric denotation as an argument. Another difference from `unstr` is that `val` will return `om` instead of raising an exception if the argument string does not satisfy its syntactic requirements (which are the same as those for numeric literals in SETL source code).

See also `denotype` and `strad`.

1.225 Wait for any child process to terminate

```
proc wait : integer
proc wait (boolean) : integer
```

Equivalent to `waitpid (-1, ...)`.

1.226 Wait for child process to terminate

```
proc waitpid (integer p): integer
proc waitpid (integer p, boolean waitflag) : integer
```

Tries to "reap" the status of a child process identified by *p*.

If successful, the process ID of a child process that has terminated is returned, and the termination status thereof is made available in status.

The rules for *p* follow those of POSIX waitpid():

If *p* is -1, status is requested for any child process (like wait).

If *p* is greater than 0, *p* specifies the process ID of a single child process for which status is requested.

If *p* is 0, status is requested for any child process whose process group ID is equal to that of the calling process.

If *p* is less than -1, status is requested for any child process whose process group ID is equal to the absolute value of *p*.

The optional *waitflag* argument, which defaults to true, specifies whether the calling process should block, waiting for a child process to terminate. If false, and the process has at least one child identified by *p*, and no such child process has yet terminated, waitpid immediately returns 0.

Note that a *waitflag* of false corresponds to the POSIX-level WNOHANG. No access is provided by the GNU SETL waitpid to the functionality of WUNTRACED or WCONTINUED (notification of a child process being stopped or continued).

If there are no child processes identified by *p*, waitpid immediately sets last_error and returns -1, regardless of the *waitflag* setting.

The use of waitpid is seldom necessary except after closing a pipe or pump stream using close with a second argument of close_zombie, or when programming at the rather primitive level of fork. In other close cases, and for filter and system, reaping is automatic.

If a child process is still a zombie when the SETL program terminates, it will be inherited by a POSIX system process that immediately reaps and discards its status.

See also pump, pipe_from_child, pipe_to_child, pid, pexists, kill, time, wait, and the open modes 'pump', 'tty-pump', 'pipe-from', and 'pipe-to'.

1.227 Format integer

```
proc whole (integer x, integer w) : string
proc whole (real x, integer w) : string
```

The number round *x* is converted to a string of length abs *w* or more.

If abs *w* is larger than necessary, the string is padded with blanks on the left (for *w* greater than 0) or on the right (for negative *w*).

If abs *w* is too small, a longer string is produced as necessary to accommodate the number, with no padding.

See also str, fixed, floating, and strad.

1.228 Set plus one element

```
op with (set s, var x) : set
```

Definition: `s with x = s + {x}`.

See the set union ("plus") operator (`+`), and also see `less`.

1.229 Write values to `stdout`

```
proc write (...)
```

Equivalent to `writea (stdout, ...)`.

1.230 Write values to output stream

```
proc writea (string f, ...)
proc writea (integer f, ...)
```

There can be 0 or more arguments after `f`, of any type. They are sent in sequence to the stream `f`, separated by single spaces and followed by a newline character. All of them are written as if they had been passed through `str` first, with no exception for strings (contrast `printa`).

Values written by `writea`, except for atoms (see `newat`) and procedure references (see `routine`), can be read by `reada`.

If `f` is not already open, an attempt is made to open it automatically for writing.

This procedure is functionally identical to `putb`.

Index

#

`#` (cardinality) 1

*

`*` (star) 1
`**` (exponentiation) 2

+

`+` (plus) 2
`+:=` (plus and assign) 4

-

`-` (minus) 2

/

`/` (slash) 3, 23, 53
`/=` (not equal) 3
`/bin/sh` 14, 31

<

`<` (less than) 3
`<=` (less than or equal to) 3

=

`=` (equal) 3

>

`>` (greater than) 3
`>=` (greater than or equal to) 3

?

`?` (query) 4

A

`abs` 5
absolute value 5
`accept` connection on socket 5
`acos` (arc cosine) 5
addition 2
all subsets (power set) 40
allocate machine memory 27
alternative `open` mode args 35
`and` (conjunction) 5
`any` (extract initial character) 6
`arb` (arbitrary element of set) 6
arbitrary *vs.* random 6, 16
arc cosine 5
arc sine 6
arc tangent 6
arc tangent of quotient 6
ASCII 8, 21, 40, 60
`asin` (arc sine) 6
asterisk 1
asterisks 2
`atan` (arc tangent) 6
`atan2` (arc tangent of quotient) 6
`atom` type 29
auto-open and auto-close 33

B

backslash escapes 40, 60
big-endian 37, 60
`bit_and` (bitwise "and") 7
`bit_not` (bitwise "not") 7
`bit_or` (bitwise "or") 7
`bit_xor` (bitwise "exclusive or") 7
bitwise logical operators 7
blanks, padding string with 26, 48
boolean 13
`break` (extract initial substring) 7
buffering output 11, 13, 15, 32

C

calendar time in milliseconds 59
`call` (indirect call) 7
callbacks 49
`callout` 7
cardinality 1
`ceil` (lowest integer upper bound) 8
change directory 8
`char` (character representation of small integer) .. 8
character code value 5, 21
`chdir` (change directory) 8
checking type 23
child process 9, 16, 39, 40, 41, 55, 58, 61, 62
choice, set element 6, 16
CIMS SETL 17, 32, 41
`clear_error` (clear system error indicator) 8
client socket 45, 51
`clock` (elapsed time in ms.) 9
`close` (close stream) 9
`close_await`, `close_autoreap`, `close_zombie` .. 10
command, execute from within program 57
`command_line` (command-line arguments) 10
`command_name` (command name) 10
comparison operators 3

complex numbers ... 6
concatenate tuple of strings, with delimiter ... 24
concatenation, string or tuple ... 1, 2
conjunction ... 5
conversion to string, implicit ... 2
convert arbitrary value to string ... 55
convert character to integer ... 5, 21
convert from hexadecimal ... 59
convert from string ... 45, 61
convert number to string ... 14, 15, 56, 62
convert small integer to character ... 8
convert string to lowercase ... 58
convert string to number ... 61
convert string to uppercase ... 58
convert to floating-point ... 15
convert to integer ... 8, 14, 15, 48
copy machine memory ... 28
`cos` (cosine) ... 10
`cosh` (hyperbolic cosine) ... 10
CPU time ... 58
current working directory ... 8, 20

D

datagram ... 45, 46, 51
datatype ... 11, 23, 59
date ... 13, 59
`date` (date and time of day) ... 11
decimal point ... 14
delimiter ... 7, 24, 44, 54
`denotype` (type of denotation in string) ... 11
destroy file reference ... 60
direct (random) access ... 31
direct access ... 20, 43, 47, 48
directory, change ... 8
disjunction ... 37
`div` (integer division) ... 11
divisible by 2 ... 12, 30
division, floating-point ... 3, 23, 53
division, integer ... 3, 11, 23, 53
DNS ... 21, 23, 38
`domain` (map domain) ... 11
`dup`, `dup2` (duplicate file descriptor) ... 11
dynamic type ... 23

E

effective group ID ... 18, 51, 52
effective user ID ... 18, 52
elapsed CPU time ... 58
elapsed time ... 9
element of domain ... 25
element of set ... 6, 16, 25, 63
element of set, string, tuple ... 22, 30
end of file ... 12
endianness ... 37, 60
environment variable ... 18, 42, 51, 61
`eof` (test for end of file) ... 12
epoch ... 13
equality operators ... 3
`errno` ... 8, 25, 29
`even` (test for integer divisible by 2) ... 12
event-driven programming ... 49
exceptional I/O conditions ... 49
`exec` (replace current process) ... 12
`exp` (natural exponential) ... 12
exponentiation ... 2
extract arbitrary element from set ... 16
extract final character using character set ... 43, 44
extract final substring by exact match ... 44
extract final substring by length ... 44
extract final substring using character set ... 44
extract first element from string or tuple ... 16
extract initial character using character set ... 6, 29
extract initial substring by exact match ... 27
extract initial substring by length ... 25
extract initial substring using character set ... 7, 54
extract last element from string or tuple ... 16

F

`false` (truth value) ... 13
`fdate` (format date and time) ... 13
`fexists` (test for file existence) ... 13
file creation mode mask ... 59
file descriptor (fd) ... 11, 13, 31, 39, 46, 49, 51, 55
file size ... 17
`filename` (filename of open stream) ... 13
`fileno` (fd of open stream) ... 13, 32
`filter` (filter string through command) ... 14
first element from string or tuple, extract ... 16
`fix` (truncate floating-point to integer) ... 14
`fixed` (format with optional decimal point) ... 14
`float` (convert to floating-point) ... 15
`floating` (format in scientific notation) ... 15
floating-point division ... 3, 23, 53
floating-point, convert from integer ... 15
floating-point, convert to integer ... 8, 14, 15, 48
floating-point, convert to string ... 14, 15
`floor` (greatest integer lower bound) ... 15
`flush` (flush output buffer) ... 15, 32, 58
`fork` (create child process) ... 16
format date and time as string ... 13
format number as string ... 14, 15, 56, 62
free machine memory ... 27
`from` (take arbitrary element from set) ... 16
`fromb` (take from beginning of string or tuple) ... 16
`frome` (take from end of string or tuple) ... 16
`fsize` (file size) ... 17
function reference ... 7, 48

G

`get` (get lines from `stdin`) ... 17
get number from string ... 61
get process ID ... 39

get value(s) from string 61
get values from string 45
geta (get lines from stream) 17
getb (get values from stream) 17
getc (get character from stream) 18
getchar (get character from stdin) 18
getegid (get effective group ID) 18
getenv (get value of environment variable) 18
geteuid (get effective user ID) 18
getfile (get file contents as string) 18
getgid (get group ID) 19
getline (get line from stream) 19
getn (get *n* characters from stream) 19
getpgrp (get process group ID) 20
gets (direct-access read) 20
getuid (get real user ID) 20
getwd (current working directory) 20
glob (pathname wildcard expansion) 20
gmark (find occurrences of pattern) 21
greater than 3
greater than or equal to 3
greatest integer below 15
group ID 18, 19, 51, 52
gsub (replace patterns in string) 21

H

half-close 53
hard link 26
hex (convert string to hexadecimal) 21
hexadecimal string 21, 59
host address 23, 38
hostaddr (local host address) 21
hostname 23, 38
hostname (local hostname) 21
hyperbolic cosine 10
hyperbolic sine 54
hyperbolic tangent 58

I

ichar (integer code for character) 21
IEEE 754 37
impl (implies) 22
implicit conversion to string 2
in (membership test) 22
incs (subset test) 23, 56
indirect call 7, 48
inequality 3
integer division 3, 11, 23, 53
integer modulus 29
integer remainder 47
integer, convert from floating-point ... 8, 14, 15, 48
integer, convert to floating-point 15
Internet address 21, 23, 38
Internet name 21, 23, 38
intersection of sets 1
intslash (integer quotient type switch) 23, 53
IP address 21, 23, 38, 46, 51
ip_addresses (Internet host addresses) 23
ip_names (Internet hostnames) 23
is_atom (type test) 23
is_boolean (type test) 23
is_integer (type test) 23
is_map (type test) 23
is_mmap (type test) 23
is_numeric (type test) 23
is_om (type test) 23
is_open (stream is open) 24, 33
is_real (type test) 23
is_routine (type test) 23
is_set (type test) 23
is_smap (type test) 23
is_string (type test) 23
is_tuple (type test) 23

J

join (delimited concatenation) 24

K

kill (send signal to process) 24

L

last element from string or tuple, extract 16
last_error (error message from system call) ... 25
len (extract initial substring) 25
length, string or tuple 1
less (set less one element) 25
less than 3
less than or equal to 3
lessf (map less one domain element) 25
lexists (existence of file or symlink) 26
line buffering 33
link (create hard link) 26
little-endian 37, 60
log (natural logarithm) 26
logical operators 5, 29, 37
logical operators, bitwise 7
lowercase, convert to 58
lowest integer above 8
lpad (pad string on left with blanks) 26

M

magic (recognize regular expressions) 26, 53
magnitude 5
map domain 11, 25
map range 43
mark (find first occurrence of pattern) 27
match (extract initial substring) 27
max (numeric maximum) 27
mem_alloc (allocate machine memory) 27

`mem_copy` (copy machine memory) 28
`mem_fetch_...` (machine memory read) 28
`mem_free` (free machine memory) 27
`mem_realloc` (reallocate machine memory) 28
`mem_store_...` (machine memory write) 28
membership test 22, 30
memory, reading 28
memory, writing 28
milliseconds 9, 13
`min` (numeric minimum) 28
minus 2
`mod` (modulus; symmetric set difference) 29
mode, `open` 31
multiplication 1
mutex 26, 57

N

`nargs` (number of arguments passed by caller) .. 29
natural exponential 12
natural logarithm 26
negation 2
`newat` (create new atom) 29
`no_error` 29
nondeterministic choice 6, 16
`not` (logical negation) 29
`notany` (extract initial character) 29
`notin` (membership test) 30
`npow` (all subsets having a given size) 30
`nprint` (print to `stdout` *sans* newline) 30
`nprinta` (print *sans* newline) 30
number of arguments passed by caller 29
number, format as string 14, 15, 56, 62
numeric affirmation 2
numeric conversion 8, 14, 15, 48
numeric difference 2
numeric maximum 27
numeric minimum 28
numeric power 2

O

`odd` (test for integer not divisible by 2) 30
`om` (the "undefined" value) 31
`open` (open a stream) 31
`open` compatibility 32
`open` results 32
open stream 24
`or` (disjunction) 37
overlay process image 12

P

`pack_...` (byte packing) 37
pathname accessibility and existence 13, 26
pathname wildcard expansion 20
pattern matching, regexp-based 21, 26, 27, 53, 54, 56
pattern matching, SNOBOL-inspired 6, 7, 25, 27, 29, 43, 44, 54
`peekc` (peek at next input character) 37
`peekchar` (peek at next character in `stdin`) 38
`peer_address` (peer host address) 38
`peer_name` (peer hostname) 38
`peer_port` (peer port number) 38
`pexists` (test for existence of processes) 38
phase, `atan2` operator 6
`pid` (process ID) 39
`pipe` (create primitive pipe) 39
`pipe_from_child` (pipe from child process) 39
`pipe_to_child` (pipe to child process) 40
piping to and from programs 31
plus 2
`port` (retrieve port number) 40
port number 38, 40
`pow` (power set) 40
power set members of a given size 30
`pretty` ("prettify" string) 40
`print` (print to `stdout`) 40
print lines 30
`printa` (print to stream) 40
printable characters 40, 60
procedure reference 7, 48
process group ID 20, 24, 38, 52
process ID 16, 20, 24, 38, 52, 61, 62
process, existence 38
process, replace (overlay) 12
product 1
program arguments 10
program name 10
pseudo-random 43, 52
pump 31, 33
`pump` (bidirectional stream to child process) 41
push characters back on input stream 59
`put` (put lines on `stdout`) 41
`puta` (put lines on stream) 41
`putb` (put values on stream) 41
`putc` (put characters on stream) 42
`putchar` (put characters on `stdout`) 42
`putenv` (set environment variable) 42
`putfile` (put characters on stream) 42
`putline` (put lines on stream) 42
`puts` 43

Q

query 4
quotient 3, 11, 23, 53

R

radix-prefixed number 56
raise number to power 2
`random` (numbers and selections) 43
random access 20, 43, 47, 48
`range` (map range) 43

`rany` (extract final character) 43
`rbreak` (extract final substring) 44
`read` (get values from `stdin`) 44
read character 18
read characters 18
read line 19
read lines 17
read *n* characters 19
read number from string 61
read value from string 61
read values 17, 44, 45
read, low-level 57
`reada` (get values from stream) 45
reading from memory 28
`readlink` (symbolic link referent) 45
`reads` (get values from string) 45
`real` (floating-point) 3, 8, 14, 15, 23, 48, 53
real group ID 19, 52
real time 9, 31, 34
real user ID 20, 52
reallocate machine memory 28
reap termination status 9, 61, 62
`recv` (receive on UDP client socket) 33, 45
`recv_fd` (receive file descriptor) 46
`recvfrom` (receive on UDP server socket) ... 33, 46
regular expressions 21, 26, 27, 53, 56
`rem` (integer remainder) 47
remove arbitrary element from set 16
remove first element from string or tuple 16
remove last element from string or tuple 16
replace process 12
replication, string or tuple 1
`reverse` (reverse string) 47
`rewind` (rewind direct-access stream) 47
`rlen` (extract final substring) 44
`rmatch` (extract final substring) 44
`rnotany` (extract final character) 44
`round` (round to nearest integer) 48
`routine` (create procedure reference) 48
`rpad` (pad string on right with blanks) 48
`rspan` (extract final substring) 44

S

saved user ID 52
scientific notation 15
seed, pseudo-random number generation 52
`seek` (reposition direct-access stream) 48
seeking (direct access) 20, 43, 47, 48
`select` (wait for I/O event or timeout) 49
`send` (send on UDP client socket) 33, 51
`send_fd` (send file descriptor) 51
`sendto` (send on UDP server socket) 33, 51
server socket 5, 31, 46, 51
set difference 2
set element choice 6, 16
set intersection 1
set less one element 25
set membership 22, 30
set plus one element 63
set union 2
set, number of members 1
`set_intslash` (integer quotient type) 23, 53
`set_magic` (recognize regular expressions) 53
`setegid` (set effective group ID) 51
`setenv` (set environment variable) 51
`seteuid` (set effective user ID) 52
`setgid` (set group ID) 52
`setl` command 10
SETL2 ..... 7, 17, 23, 32, 41
`setpgrp` (set process group ID) 52
`setrandom` (set random seed) 52
`setuid` (set user ID) 52
sharp sign 1
short-circuiting operators 4, 5, 37
`shut_rd`, `shut_wr`, `shut_rdwr` 54
`shutdown` (close I/O in one or both directions) .. 53
`sign` (numeric sign) 54
signal 24
signal stream 31, 34
`sin` (sine) 54
`sinh` (hyperbolic sine) 54
size of file 17
size of set 1
SNOBOL-inspired pattern matching .. 6, 7, 25, 27, 29, 43, 44, 54
socket 5, 31, 38, 40, 45, 46, 51
`span` (extract initial substring) 54
`split` (split string into tuple) 54
`sqrt` (square root) 55
star 1
stars 2
`status` (child process termination status) 55
status, child process 9, 61, 62
`stderr` (standard error output) 35, 55
`stdin` (standard input) 35, 55
`stdout` (standard output) 35, 55
`str` (string representation of value) 55
`strad` (integer as radix-prefixed string) 56
stream buffering 11, 13, 15, 32
stream handle 32
stream, closing 9, 53
stream, end of file 12
stream, opening 31
string concatenation 1, 2
string length 1
string membership 22, 30
string padding 26, 48
string replication 1
string splitting 54
string substitution 21, 56
string, convert number to 14, 15, 56, 62
string, reverse 47
`sub` (replace pattern in string) 56
`subset` (subset test) 23, 56
subtraction 2

sum 2
symbolic link 13, 26, 45, 57
`symlink` (create symbolic link) 57
symmetric set difference 29
`sys_read` (low-level read) 57
`sys_write` (low-level write) 57
`system` (execute command in subshell) 57
system call errors 8, 25, 29

T

take arbitrary element from set 16
take first element from string or tuple 16
take last element from string or tuple 16
`tan` (tangent) 57
`tanh` (hyperbolic tangent) 58
TCP 5, 31, 33, 38, 40, 53
temporary filename, unique 58
termination status, child process .. 9, 55, 57, 61, 62
test for open stream 24, 33
testing type 23
`tie` (flush output upon input elsewhere) 32, 58
`time` (elapsed CPU time in milliseconds) 58
time of day 11, 13, 59
time, real 9
timer stream 31, 34
`tmpnam` (unique temporary filename) 58
`to_lower` (convert to lowercase) 58
`to_upper` (convert to uppercase) 58
`tod` (calendar time in milliseconds) 59
`true` 59
truncation, numeric 14
truth table 22
tty-pump 31, 33
tuple concatenation 1, 2
tuple membership 22, 30
tuple replication 1
tuple, highest defined index 1
two stars (asterisks) 2
`type` (type of SETL value) 59
type of value that will be returned by `unstr` ... 11
type, checking 23

U

UDP 31, 33, 38, 40, 45, 46, 51
`umask` (set file creation mode mask) 59
undefined 31
unequal 3
`ungetc` (push characters back on input) 59
`ungetchar` (push characters back on `stdin`) 59
`unhex` (convert from hexadecimal) 59
uninitialized SETL variable 31
union of sets 2
unique temporary filename 58
Unix-domain socket 46, 51
`unlink` (destroy file reference) 60
`unpack_...` (byte unpacking) 60
`unpretty` ("unprettify" string) 60
`unsetenv` (remove environment variable) 61
`unstr` (read value from string) 61
uppercase, convert to 58
user ID 18, 20, 52
UTC 13

V

`val` (read numeric value from string) 61
variable number of arguments 29

W

`wait` (wait for any child process to terminate) .. 61
`waitpid` (wait for child process to terminate) ... 62
whitespace 54
`whole` (format integer) 62
wildcard expansion, pathname 20
`with` (set plus one element) 63
working directory, current 8, 20
`write` (write values to `stdout`) 63
write characters 42
write lines 41, 42
write values 41, 63
write, low-level 57
`writea` (write values to stream) 63
writing to memory 28

Z

zombie process 9, 61, 62