GNU SETL Library Reference

Edition 5.8.13, for GNU SETL Version 5.8.13
Updated 8 April 2024

by dB

Table of Contents

About this Library Reference

This document is about the library of functions, operators, etc. (collectively the *intrinsics*) built into SETL. As part of the GNU SETL documentation, it also makes occasional reference to that implementation of SETL.

For the primary document in the present set, see the *GNU SETL Om*.

The Library

SETL has no pointers, and is therefore governed by *value semantics*, even for arbitrarily nested sets and tuples. Call arguments are passed by value, result, or value-result assignment, according as the formal parameter is `rd` (the default), `wr`, or `rw`.

SETL has no system of datatype declarations currently, but the function and operator signatures are shown here as if it were possible to declare the types of their formal parameters and return values. Where an argument can be of any type, the keyword `var` is used.

Those signatures also employ fake typenames for some common cases: `stream` means an `integer` or `string` or 2-`tuple` that refers to an I/O stream; `pattern` means a `string` or 2-`tuple` used as a pattern argument to a string matching function such as `mark`; and `proc_name` and `proc_ref` are respectively the fake typenames for what `routine` expects and returns.

A further liberty is taken in cases where the return type depends upon the values of the arguments and not just upon their types. For example, the type of the value returned by `val` depends upon what is in the input string, and this is reflected in a pair of signatures for `val`, one returning `integer` and the other returning `real`. Similarly, exponentiation (`**`) on `integer` arguments returns a `real` when the second argument is negative, but otherwise an `integer`, and again this is reflected in a pair of signatures. In cases where a wider range of return types is possible, the designation `var` is used. This applies to the *query* operator (`?`), `arb`, `call`, `random`, and `unstr`.

Finally, when the form of a `set` or `tuple` is restricted, it is often depicted that way. For example, an ordered pair of integers may be shown as `[integer, integer]`, and a set of 0 or more strings as `{string, ...}`.

If no return type is shown for a given intrinsic, it may be assumed to return `om` if it returns at all. Also, `om` may be a possible return value even though the only types shown are other than `om`. Such cases are identified in the accompanying descriptions.

Where error cases are described with no specific response such as setting `last_error`, but rather with vague and sinister language like "erroneous" or "It is an error ...", the most portable assumption to make is that unspecified disasters may follow. Nevertheless, a "checkout" SETL implementation such as GNU SETL will catch many of the easy cases such as nonsensical or out-of-bounds arguments passed to an intrinsic, and bring the program down with a diagnostic for them.

The finer details of many intrinsics are governed by the semantics of POSIX functions. The general intention in this spec is that support for IEEE Std 1003.1-2001 or any newer POSIX standard should suffice. See for example The Open Group Base Specifications Issue 7 (2018 edition) - IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008) (`http://www.opengroup.org/onlinepubs/9699919799/`). Linux and virtually all variants of Unix satisfy the system needs of the SETL library, as does Cygwin (mostly).

sharp (#) - size of set; length of string or tuple

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

The size (cardinality) of a set is the number of elements. The length of a tuple is the index of its last non-`om` element.

star (*) - 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
```

When one operand is `real` and the other `integer`, the `integer` value is converted as if by `float` before the numbers are multiplied.

In a bounded floating-point implementation such as 64-bit IEEE 754, multiplication can overflow to a floating-point infinity, or underflow to a subnormal number or a zero. Multiplying an infinity by 0 can produce a NaN (*Not a Number*). Multiplying a NaN by anything gives another NaN.

For a pair of `set` operands, this operator gives the set intersection, i.e., for sets `a` and `b`, `a*b` = `{x in a | x in b}`. (The first `in` iterates over `a`, the vertical bar means *such that*, and the second `in` tests for membership in `b`.)

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

power (**) - exponentiation

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

The first operand is the base, and the second is the power to raise it to.

When both operands are of type `integer`, the return type is `integer` unless the power is negative, in which case the operands are first converted as if by `float`, as is the `integer` operand in the mixed cases.

When the power is 0, the result is 1 regardless of the base, in the type appropriate for the operands.

In a bounded floating-point implementation such as 64-bit IEEE 754, this operator works like POSIX `pow()` for all cases that return a `real`.

It is right-associative, which is an incompatible change from CIMS SETL but agrees with SETL2 and with the Fortran operator it resembles. It has higher precedence than all other binary operators, just below all non-predicate unary operators. See [Operator Precedence], page 84.

plus (+) - addition; set union; 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, var) : string
op + (var, string) : string
```

The unary forms simply check that the operand is a number and return that number.

When one operand is `real` and the other `integer`, the `integer` value is converted as if by `float` before the numbers are added.

In a bounded floating-point implementation such as 64-bit IEEE 754, addition can overflow to a floating-point infinity, or underflow to a subnormal number. Addition of opposite infinities can produce a NaN (*Not a Number*). Adding a NaN to anything gives another NaN.

For a pair of `set` operands, this operator gives the set union, i.e., all elements from both sets. See also `with`.

The cases in which just one operand is a `string` are treated as if `str` is first applied to the non-string operand (shown here as `var`), for string concatenation.

The assigning `+:=` operator, when its left-hand operand is initially `om`, substitutes the additive identity (if any) of the type of its right-hand operand. So for example

```
sum +:= num
```

does not require *sum* to be pre-initialized before *num* is added to it. Here are the additive identities:

`integer`	0
`real`	0.0
`string`	the empty string, `""`
`set`	the empty set, `{}`
`tuple`	the empty tuple, `[]`

It is an error for the left-hand side of `+:=` to be `om` if the type of the right-hand side has no additive identity.

Note that when *x* is `om`,

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

is slightly different from

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

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 an error (trying to add om and a number).

Similarly, trying to accumulate a set or tuple into an uninitialized variable using plain + rather than the assigning form +:= is an error.

minus (-) - 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
```

When one operand is real and the other integer, the integer value is converted as if by float before the numbers are subtracted.

In a bounded floating-point implementation such as 64-bit IEEE 754, subtraction can overflow to a floating-point infinity, or underflow to a subnormal number. Subtraction of equal infinities can produce a NaN (*Not a Number*), and a NaN operand gives a NaN result.

For a pair of set operands, this operator gives the set difference, i.e., for sets a and b, a - b = {x in a | x notin b}.

See also less, and the symmetric set difference operator mod.

slash (/) - division

```
op / (integer, integer) : real     -- when intslash is false
op / (integer, integer) : integer  -- when intslash is true
op / (real, real) : real
op / (real, integer) : real
op / (integer, real) : real
```

By default, all integer operands are converted as if by float, and the result is real. You would normally use div to get integer quotients.

But you can force this *slash* operator to work like div for the integer / integer case by setting intslash := true, or equivalently by calling set_intslash (true). This makes the slash operator mimic SETL2, which entails some peril. Consider a program that reads pairs of numbers and computes their quotients. Unless it takes care to ensure that at least one operand is real, say by applying float, such a program will sometimes truncate quotients and sometimes not, depending on which input numbers happen to have decimal points (or exponents) in them.

Division of reals, if the floating-point implementation is 64-bit IEEE 754, can produce an infinity by means of overflow or a divisor of 0, or can underflow to a subnormal number or a zero. Division of infinities gives a NaN (*Not a Number*), and a NaN operand gives a NaN result.

equalities (=, /=) - equal, not equal

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

In most cases, two values of different types are considered unequal. But in the special case where an `integer` and `real` are numerically equal, they are considered equal despite their differing types.

A floating-point NaN (*Not a Number*) never equals anything, not even another bit-identical NaN.

comparatives (<, >, <=, >=) - 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
```

Strings are compared character by character, as if using the codes arising from `ichar`. Tuple comparisons are recursive. If one string or tuple is a prefix of the other, the shorter one is considered smaller.

Where one operand is `integer` and the other is `real`, the `integer` is converted as if by `float` before comparison.

Comparison of a floating-point NaN (*Not a Number*) with anything yields `false`.

See also `max` and `min`.

query (?) - short-circuiting `om` test

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

The expression

```
x ? y
```

is equivalent to the expression

```
if (temp := x) /= om then temp else y end
```

(or simply

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

if `x` has no side-effects).

Thus the query operator is *short-circuited* like `and` and `or`.

`abs` - absolute value; integer value of character; Euclidean norm

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

For a number, `abs` returns the magnitude.

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

For a `tuple t` of numbers,

```
abs t = sqrt (0 +/ [x**2 : x in t])
```

Thus for a complex number represented by `[x, y]`, its magnitude is `abs [x, y]`. Its phase is `y atan2 x`.

`accept` - accept connection on server socket

```
proc accept (stream) : integer
```

The argument must be a TCP or Unix-domain server socket opened by `open`, or denote a host and TCP port to auto-open in `"tcp-server"` mode (see [Automatic opening], page 47).

The `accept` function waits for a client to connect, and then returns a new stream (over a new socket fd) for that peer connection.

The `select` function can be used on a server socket to test whether an `accept` would block on it.

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` sets `last_error` and returns `om`.

See also `peer_address`, `peer_name`, `peer_port`, `peer_sockaddr`, `sockaddr`, and the `"unix-server"` mode of `open`.

`acos` - arc cosine

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

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

and - logical conjunction

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

The expression

```
x and y
```

is equivalent to the expression

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

which is to say that the **and** operator is *short-circuited*: it only evaluates *y* if *x* is `true`. 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) = "/" then
    ...
  end if;
end if;
```

can be replaced by

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

See also `or` and the *query* operator (`?`), which are likewise short-circuited, and the bitwise operators such as `bit_and`, which aren't.

The **and** operator has a rather low precedence, above `or` but below `not`.

any - extract leading character using character set

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

If the first character of *s* appears anywhere in *p* (treating *p* as a set of characters), that first character is removed from *s* and returned. Otherwise, nothing happens to *s*, and the empty string (`""`) is returned.

See also the other SNOBOL-inspired intrinsics, namely `break`, `len`, `match`, `notany`, `span`, `rany`, `rbreak`, `rlen`, `rmatch`, `rnotany`, and `rspan`. The inspiration is pretty much confined to the reuse of these names, as they are rather clumsy in their present form. But SETL currently has neither the syntactic nor semantic support for SNOBOL-like chained pattern matching with backtracking and easy extraction of intermediate substrings and positions into variables.

arb - arbitrary element of set

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

An arbitrary (not *random*, but dealer's choice) element of the set is returned. If the set is empty, `om` is returned.

See also `from` and `random`.

asin - arc sine

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

The operand must be in the range -1 to +1, and the result is in radians. See also sin.

atan - arc tangent

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

The result is in radians. See also atan2 and tan.

atan2 - 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)
```

but when *x* is 0, only the atan2 form can be used, and returns a floating-point approximation of $\pi/2$ or $-\pi/2$ depending on the sign of *y* (or 0 if *y* is 0).

SETL does not currently have *first class* support for complex numbers, but for such a number with real part *x* and imaginary part *y*, its phase is *y* atan2 *x* and its magnitude is abs [*x*, *y*].

bit_and, bit_not, bit_or, bit_xor - bitwise logical ops

```
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 integers as if they were expressed in 2's complement with an infinite sequence of leading 0 or 1 digits. For example, bit_not 1 = -2.

See also and, or, and not.

break - extract leading substring using character set

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

An initial substring of *s* up to but not including the first character that is found in *p* (treating *p* as a set of characters) is removed (*broken off*) from *s* and returned. If no character from *p* appears in *s*, the return value is the initial value of *s*, and *s* is reduced to the empty string ("").

See also the other SNOBOL-inspired intrinsics, namely any, len, match, notany, span, rany, rbreak, rlen, rmatch, rnotany, and rspan.

call - indirect call

```
proc call (proc_ref, var args(*)) : var
```

All of the 0 or more *args* 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 easily be returned, e.g.:

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

callout - call C function

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

This is a SETL2 compatibility feature. It calls a C function having the signature

```
char *setl2_callout (int service, int argc, char *const *argv)
```

by first converting *service* to a C int and *arglist* (which must contain only strings) to the pair of callout arguments *argc* (the number of strings) and *argv* (an array of pointers to C character strings).

The result of the call, a C character string, is then converted to a SETL string and returned by callout.

GNU SETL comes with a "stub" version of setl2_callout that simply reports details of the call on stderr and then returns a NULL pointer, causing callout to return om.

ceil - ceiling (least 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 operand.

For example, ceil -5.9 = -5.

Floating-point infinities and NaN (*Not a Number*) values give om.

See also floor, round, fix, and float.

char - character encoding of small integer

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

This operator makes a one-byte string rather directly from the operand, which must be an integer in the range 0 to 255. For example, char 32 = "\x20".

In SETL, NUL characters may occur within strings, so char 0, or equivalently "\x00", is valid.

See also ichar.

chdir - change directory

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

The current working directory ("folder") is changed to *s* if given and valid. Otherwise, if the HOME environment variable is defined and names a valid directory, the working directory becomes that.

It is an error for chdir to be called with no arg if HOME is not defined.

If *s* is given but not valid, or if *s* is not given and HOME is defined but not valid, then the current working directory is not changed, and last_error tells why. For example, the reason might be a nonexistent directory, a file that is not a directory, or insufficient permission to open the directory, as ruled by POSIX chdir().

See also getwd.

clear_error - clear system error indicator

```
proc clear_error
```

Sets last_error to no_error.

For example, a doubtful chdir call could be guarded thus:

```
clear_error;
chdir ("somewhere");
if last_error = no_error then
  -- success ...
else  -- the string last_error explains
  -- failure ...
end if;
```

clock - elapsed time in milliseconds

```
proc clock : integer
```

This is the total amount of wall-clock ("real") time, in milliseconds, that has elapsed since the current process began. It is a monotonic clock, and does not depend on changes to the time within the epoch.

See also time and tod.

close - close stream

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

If *f* is a stream, i.e., if is_open *f* is true, then it is flushed as if by flush and destroyed. Any stream associated with *f* by tie is also flushed, and the association is dissolved. Output failures in the flushing are *not* reflected in last_error.

The underlying file descriptor (*f* itself or the associated fd) is passed to POSIX close() if it is in the range of valid file descriptors at the POSIX (system) level. That range is 0 to some maximum. Most Unix shells have a built-in ulimit or limit command to report

or set the limit. For example, in Bourne-compatible shells, '`ulimit -n`' gives the current maximum fd plus 1.

It is permitted to call `close` on `stdin`, `stdout`, and `stderr`.

You can also call `close` on a fd *`f`* that is not open at the SETL level but is open at the POSIX level, e.g. as inherited by the process, or as arising from a call to one of the low-level functions `dup`, `dup2`, `socketpair`, `pipe`, or `recv_fd`. This just calls POSIX `close()` on the fd.

Closing a stream of type `"signal"`, `"ignore"`, `"default"` (see [Signal streams], page 44), or `"real-ms"` (see [Timer streams], page 46) can change the disposition of a signal. The fd is a *pseudo-fd* for these stream cases, outside the range of valid POSIX file descriptors.

As a trivial and dubious convenience, `close(om)` is a no-op.

The *`how`* argument to `close`, if present, must be one of the constants `close_await` (default), `close_autoreap`, or `close_zombie`. This second argument is meaningful for pipe, pump, and tty-pump streams (when the stream is the original one created by the caller, not one obtained from a duplicated fd), 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 available in `status`, except in the unusual case that the status has already been reaped, e.g. by an explicit and probably misdirected `waitpid` before the `close`. In that case, `close` sets `status` to `om`.
- 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 it does so.
- 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 before the parent, it will become a *zombie* (as defined by POSIX) until a `waitpid` (or equivalent) successfully reaps its status. You can `open` a SIGCHLD signal stream to get notified of child terminations.

Failure of the POSIX-level `close()` causes `last_error` to be set. In the case of *`how`* = `close_await` on a pipe, pump, or tty-pump stream, the POSIX `waitpid()` call after that can also set `last_error`, in the circumstances that set `status` to `om` described above.

When a Unix-domain server socket (`open` mode `"unix-server"` or `"unix-datagram-server"`) is closed, its associated pathname is removed if it still exists.

Streams that were automatically opened are automatically closed when appropriate. See [Automatic opening], page 47.

All streams are automatically flushed and drained, and then all closed, before the program exits. See [Buffering], page 47.

See `shutdown` about shutting down one or both directions of a bidirectional communications stream without closing the stream itself.

close_await, close_autoreap, close_zombie - constants for use with close

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

See close.

command_line - command-line arguments

```
command_line : tuple
```

This is a tuple of strings giving the command-line arguments that were passed to the SETL program when it was launched.

See also command_name.

command_name - command name

```
command_name : string
```

This is an implementation-defined name for the SETL program.

For GNU SETL, if the SETL program comes from an file or command, command_name is the name of that file, or the command string, including the leading vertical bar in the case of a command. Otherwise (i.e., when the program is read from a file descriptor, from standard input, or from the command-line argument itself), command_name is the name of the language processor command, normally setl.

For example, in POSIX, if this script is stored in the text file /tmp/help and made executable, it should print 'I'm /tmp/help' and a newline when invoked:

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

Note the use of the '#!' escape (see Section "#! invocation" in the *GNU SETL User Guide*).

See also command_line.

cos - cosine

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

The operand is in radians. See also acos.

cosh - hyperbolic cosine

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

Hyperbolic cosine. Popular with catenarians.

date - date and time of day

```
proc date : string
```

Equivalent to fdate (tod, "%c").

See also clock and time.

denotype - type of denotation in 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 instead of taking exception to it as unstr would.

See also val and str.

div - integer division

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

SETL's div operator truncates fractional results towards zero.

It is an error for the second operand (denominator) to be zero.

See also *slash* (/), and for remainders see mod and rem.

domain - domain of map

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

The operand must be a set of ordered pairs, that is, a set of 2-tuples in which no element is om. The result is the set of all first members of those pairs.

See also range and lessf.

dup, dup2 - duplicate a 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 want low-level control over system-level file descriptors, e.g. to play games with socketpair (or pipe) and fork and exec.

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 on it subsequently (see [Automatic opening], page 47).

In dup2, there is an implicit POSIX close() of *fd2* before the duplication of *fd1* occurs. If *fd2* is already open at the SETL level, any buffer structure it has remains intact (see [Buffering], page 47). Thus if *fd2* has an output buffer, it is usually best to flush it if necessary before the dup2 call, so that the redirection to a new sink (*fd1*) only applies to new output. This is similar to the case of a buffered fd across a fork.

Example:

```
dup2 (stdout, stderr)
```

redirects stderr to wherever stdout points, like the shell notation '2>&1'.

On failure, dup and dup2 set last_error and return om.

eof - end-of-file indicators

```
proc eof : boolean
proc eof (stream) : boolean
```

When a stream input operation fails to get any input, two `eof` indicators are set: a global one and a stream-specific one, accessed respectively by the nullary and unary forms of this intrinsic.

The `eof` indicators are cleared by sequential input intrinsics (and by `recv_fd` and `gets`) before they attempt input, and are only set when no input at all is received. Thus a `getb` that asks for 3 values but only gets 2 does *not* set the `eof` indicators, even though an end of file on the underlying medium has been reached. The stream's `eof` indicator is *pending* in that case, meaning that a subsequent input attempt on that stream will get nothing and will set the `eof` indicators. The setting will happen even if the attempt is for no input, such as for 0 values in the case of `getb`, 0 characters in the case of `getn` or `gets`, or 0 lines in the case of `geta`.

A vacuous input attempt like that is thus a way to convert a pending `eof` to a `true` result from the `eof` intrinsic, making it no longer pending. I haven't worked out whether that use case is plausible or more likely symptomatic. I have not wanted it yet.

A vacuous input attempt on a stream for which `eof` is *not* pending is an unconditional way of clearing the `eof` indicators. Again, this is not something I have felt moved to do in practice.

Besides ordinary end-of-file conditions such as reaching the end of a medium, the `eof` indicators may also be set by input errors. These can be distinguished from normal end of file using `last_error`. A stream may have a pending setting of `last_error` when it has a pending `eof`; the actual setting occurs when the `eof` indicators are set.

Setting the `eof` indicators also triggers auto-closing when appropriate (see [Automatic opening], page 47). Note that closing a stream invalidates the unary form of `eof` for that stream, as it has been destroyed by then, while the nullary form remains valid.

For some input sources, an `eof` indication of `true` does not necessarily mean that further attempts to read from that source will fail. For example, when the stream is connected to a terminal with the normal *cooked* line discipline settings, a `ctrl-D` instead of an input line typically makes `getline` yield `om` and `eof` yield `true`, but another `getline` after that will yield another string if another line is then entered at the terminal.

See `get`, `geta`, `getb`, `getc`, `getchar`, `getfile`, `getline`, `getn`, `gets`, `peekc`, `peekchar`, `read`, `reada`, and `recv_fd`; but do not see `recv`, `recvfrom`, `seek`, `sys_read`, nor `ungetc`, as they never set `eof`.

even - test for integer divisible by 2

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

Divisible by 2. Not `odd`. See also `mod`.

Note that the precedence of this operator is quite low, like that of other unary predicates, and well below that of unary non-predicates (see [Operator Precedence], page 84). Its relative precedence was higher in the original CIMS SETL.

exec - execute another program in place of current one

```
proc exec (string file)
proc exec (string file, tuple argv)
proc exec (string file, 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 *`file`* be a full pathname identifying a command.

If the *`envp`* argument is *not* given, then `execvp()` is used, so the `PATH` environment variable is searched for a directory containing an executable named *`file`* unless *`file`* contains a slash (/) character. The POSIX implementation defines what happens if `PATH` isn't defined, but a default search of `/bin` and `/usr/bin` is common.

If the second argument (*`argv`*) appears, it must be a tuple of strings giving the arguments that will be supplied to the command, beginning with the 0^{th} which conventionally is the name by which the command identifies itself. A missing *`argv`* defaults to the one-element tuple `[`*`file`*`]`.

If *`envp`* is present, it must be a tuple of strings defining the environment variables to be seen by the command. Each string is of the form `"`*`name=value`*`"`. Otherwise, the existing environment is inherited.

If `exec` is successful, it does not return; the current process is replaced in that its image in memory is replaced by that of the new command. The process ID does not change.

Signals that are being caught because of an open `"signal"` stream are set to their POSIX defaults in the new execution context. If any `"real-ms"` timer streams are open (which is *not* the case in a new child of `fork`), SIGALRM is also set to the POSIX default (`SIG_DFL`). Other signal dispositions are inherited as POSIX `SIG_DFL` or `SIG_IGN` as appropriate. See [Signal streams], page 44.

Compare `filter`, `system`, and the `open` modes `"pipe-from"`, `"pipe-to"`, `"pump"`, and `"tty-pump"`, all of which use `/bin/sh` to invoke a shell command. One of those may be able to achieve what you want more cleanly and directly than `exec` does.

If you want your SETL program to set up an environment and then replace itself with a shell command, this would do the latter:

```
exec ("/bin/sh", ["sh", "-c", "command and args"]);
```

See also `pipe_from_child`, `pipe_to_child`, `pump`, and `tty_pump` for some ways beyond `fork` to create a child process suitable for calling `exec` in.

exp - natural exponential (e raised to a power)

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

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

false - predefined boolean value

```
false : boolean
```

A starting point from which all conclusions are possible.

See also true.

fdate - 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 (936433255888) might be 'Sat Sep 4 04:20:55.888 EDT 1999' if invoked in the POSIX locale in the US Eastern time zone, and fdate (tod) renders the current calendar time.

The %-sign patterns in *fmt* are those defined for POSIX strftime() when applied to the result of applying POSIX localtime() to *ms* div 1000, together with one extension: "%s" expands to the low-order 3 decimal digits of *ms*. (Note: this meaning of "%s" differs from a GNU extension to strftime(), where it means the number of seconds since the beginning of 1970, a number that can be obtained in SETL as tod div 1000.)

See also date, which is equivalent to fdate (tod, "%c").

fexists - test for existence of file

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

Returns true iff POSIX stat() returns 0 on the given pathname.

Note that stat() *does* follow symbolic links, so fexists will only return true if an existing file is found after following all links (and provided the caller has sufficient access to all pathname components in reaching it).

Thus fexists is stricter than lexists, which uses POSIX lstat(). Both provide mere snapshots, not automatically synchronized with actions by other processes. See link and symlink for some ways to use files as mutual exclusion (mutex) locks.

See also fsize, readlink, and unlink.

Note the low precedence of this operator relative to unary *non*-predicates (see [Operator Precedence], page 84). This differs from its precedence rank in SETL2.

filename - name of stream

```
op filename (stream) : integer
op filename (stream) : string
op filename (stream) : [integer, integer]
op filename (stream) : [string, string]
```

This is some form of the "name" under which the given stream was opened, as follows.

For an ordinary file, it is the string name passed to open or used to auto-open the file (see [Automatic opening], page 47), or the name assigned by mkstemp.

For a Unix-domain socket, it is the pathname used at `open` time. See [Unix-domain sockets], page 43.

Similarly, for a subprocess started by `open` mode `"pipe-from"`, `"pipe-to"`, `"pump"`, or `"tty-pump"`, it is the command string that launched that subprocess. See [Connected subprocesses], page 44.

For a signal-catching, -ignoring, or -defaulting stream, the name is returned in the same case and spelling (i.e., with or without the SIG prefix) as was originally used to `open` the stream. See [Signal streams], page 44.

For a `"real-ms"` stream, `filename` returns a pair of numbers `[`*`initial`*`, `*`interval`*`]`, each representing milliseconds. See [Timer streams], page 46.

For a TCP or UDP socket stream, it returns a pair `[`*`node`*`, `*`service`*`]`, where *`node`* is a `string` host name or Internet address (IPv4 dotted or IPv6 colon-rich), or is `om` to signify an unspecified address (as is common for servers); and *`service`* is a `string` service name or port number, converted as if by `str` if the original spec was an `integer`. See [TCP and UDP sockets], page 42.

Finally, for a stream created by `accept`, `pipe_from_child`, `pipe_to_child`, `pump`, or `tty_pump`, or a stream over a fd that was already open at the system (POSIX) level, the "name" is simply the `integer` fd.

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

See also `is_open`, `fileno`, `port`, `sockaddr`, `peer_sockaddr`, `peer_name`, `ip_names`, and `ip_addresses`.

`fileno` - file descriptor of stream

```
op fileno (stream f) : integer
```

If *`f`* exists as a stream according to `is_open`, `fileno` returns its underlying POSIX file descriptor (*fd*) or, for a signal-related stream (see [Signal streams], page 44) or timer stream (see [Timer streams], page 46), its pseudo-fd.

It is an error to apply `fileno` to anything else. The fact that GNU SETL kindly stops everything and issues a diagnostic leads to the following non-portable idiom in programs that would rather crash immediately than continue with a bad result from `open`:

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

See also `filename`.

`filepos` - current file position or #bytes transferred

```
op filepos (stream f) : integer
```

Similar to `seek (`*`f`*`, 0, seek_cur)`, but does not flush or drain the stream before giving its result. Thus while the result of `seek` always matches the OS-level file position, `filepos` takes into account any buffered output bytes that have not been written out yet at the system level and any buffered input that has not yet been consumed by the SETL program; it gives the offset that would obtain *as if* the flushing or draining had occurred.

Also, `filepos` is allowed even on non-seekable streams, where it returns the number of bytes that have been read and/or written since the stream was opened.

Finally (again in contrast to seek), filepos does not attempt to auto-open *f*, but requires that *f* already be open (see open).

filter - filter string through external command

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

This feeds the string *input* into an external command and returns its output.

The *cmd* argument specifies a shell command, which is performed as if by exec ("/bin/sh", ["sh", "-c", *cmd*]) in a child process spawned as if by pump.

The command may read from its standard input and/or write to its standard output. The *input* arg (default "") is fed to the child's standard input while the content of its standard output is being captured.

When all of *input* has been fed to the child, the end of the pipe that does the feeding is closed, causing an end of file condition to be presented to the child's standard input.

Meanwhile, when an end of file condition is seen by the SETL program on the capturing end of the pipe, signifying that the child's standard output has been closed, POSIX waitpid() is called in an attempt to get the child's exit status, and the captured output is returned by filter.

Just as with system, the signals SIGINT and SIGQUIT are temporarily ignored in the parent while it waits for the child to complete. Thus a terminal-generated signal (typically ctrl-C for SIGINT and ctrl-\ for SIGQUIT) that goes to the process group will be seen by the child but not the parent, which remains to handle the child termination. Also as with system, SIGCHLD is not blocked during the filter call.

The termination status of the /bin/sh invocation is placed in status. By convention, 0 means a successful command execution. In the event that the final POSIX waitpid() call to get that status fails, status is set to om and the reason for the failure appears in last_error, just as in the similar scenario of close with the close_await parameter.

See also fork, pipe_from_child, pipe_to_child, tty_pump, socketpair, pipe, dup, and dup2; and the open modes "pipe-from", "pipe-to", "pump", and "tty-pump".

fix - truncate real number to integer

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

Truncation of real operands is towards zero; integer operands are simply returned.

Example: fix -5.6 = -5.

Floating-point infinities and NaN (*Not a Number*) values give om.

See also ceil, floor, round, and float.

fixed - format number with optional decimal point

```
proc fixed (real x, integer wid, integer aft) : string
proc fixed (integer x, integer wid, integer aft) : string
```

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

If *`aft`* is zero, there is no decimal point, and you might as well use `whole` instead of `fixed`. Negative *`aft`* is an error.

If `abs` *`wid`* is larger than necessary, the string is padded with blanks on the left (for positive *`wid`*) or on the right (for negative *`wid`*).

If `abs` *`wid`* is too small, a longer string is produced as necessary to accommodate the number.

It is possible for the conversion to result in the string `"nan"`, `"inf"`, or `"infinity"`, with or without a minus sign in front.

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

`float` - convert number to `real`

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

If `integer`s are unbounded, and `real`s are not, it is possible for this conversion to produce a floating-point infinity. On a 64-bit IEEE 754 implementation, this will happen for any integer of magnitude `2**1024` or more, a 309-digit integer in decimal.

Also, loss of precision can occur for integers too big to fit in the mantissa (*significand*) of a bounded floating-point representation. In the 64-bit IEEE 754 case, this means that integers of magnitude at most `2**53` will be reproduced with absolute fidelity by `float`. Beyond that, the gaps in coverage begin, starting with the odd numbers.

Applied to a `real`, `float` simply returns it.

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

`floating` - format number in scientific notation

```
proc floating (real x, integer wid, integer aft) : string
proc floating (integer x, integer wid, integer aft) : string
```

The number `float` *`x`* is converted to a string of length `abs` *`wid`* or more in *scientific* notation, with one digit before the decimal point, *`aft`* digits after it, and the string `"e+`*`dd`*`"` or `"e-`*`dd`*`"` after that, where the latter stands for "times 10 to the power of *`dd`* (or *`-dd`*)", and *`dd`* has at least 2 digits.

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

If `abs` *`wid`* is larger than necessary, the string is padded with blanks on the left (for positive *`wid`*) or on the right (for negative *`wid`*).

If `abs` *`wid`* is too small, a longer string is produced as necessary to accommodate the number.

It is possible for the conversion to result in the string `"nan"`, `"inf"`, or `"infinity"`, with or without a minus sign in front.

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

floor - 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 operand.

For example, `floor -5.1 = -6`.

Floating-point infinities and NaN (*Not a Number*) values give `om`.

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

flush - flush output buffer

```
proc flush (stream)
```

All buffered output for the given stream is written out using POSIX `write()`, blocking the program if necessary until the entire buffer has been written.

If POSIX `write()` fails, `last_error` is set, and the number of bytes written is then indeterminate.

Applying `flush` to a stream with no pending output (which is always the case for a read-only or datagram stream) has no effect, not even on `last_error`.

For most use cases, flushing is done automatically when it needs to be. Some auto-flushing can also be arranged using `tie`. See [Buffering], page 47.

See also `open`, `close`, and `is_open`.

fork - fork into parent and child process

```
proc fork : integer
```

This is an interface to POSIX `fork()` with accommodations for SETL.

In the parent process, `fork` returns an integer representing the process ID of the child.

In the child, `fork` returns 0.

All output buffers are flushed as if by `flush` before the spawning attempt, possibly causing some blocking. Output errors in the flushing are suppressed, and not reflected in `last_error`.

In the child, before `fork` returns, all unread input on signal streams is drained (discarded), all timer streams (`open` mode `"real-ms"`) are closed (see [Signal streams], page 44), and the time base for elapsed time (see `clock`) is reset to 0.

If the system cannot spawn a new process, `fork` sets `last_error` and returns `om`, in contrast to all other intrinsics, which consider spawning failures errors.

In many cases, simply using `system`, `filter`, `pipe_from_child`, `pipe_to_child`, `pump`, `tty_pump`, or one of the `open` modes `"pipe-from"`, `"pipe-to"`, `"pump"`, or `"tty-pump"` will be easier than dancing with `fork`, `exec`, `socketpair` (or `pipe`), `dup2`, `close`, and `waitpid`.

See also `getpid`, `pexists`, and `kill`.

from - take arbitrary element from set

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

An element of the set *s* is chosen arbitrarily (but probably not *randomly*), removed from *s*, and assigned to *x*.

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

Currently, from is a statement form, not actually an operator.

See also arb, fromb, frome, less, lessf, and the *minus* operator (-) as applied to sets.

fromb - 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 one-character string if *s* is a string), and that element is assigned to *x*.

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

Currently, fromb is a statement form, not actually an operator.

See also from and frome.

frome - 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 one-character string if *s* is a string), and that element is assigned to *x*.

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

Currently, frome is a statement form, not actually an operator.

See also from and fromb.

fsize - size of file in bytes

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

If the operand is a stream, fsize returns the size of the thing that is open (see is_open). More precisely, it returns the value of the st_size field in the POSIX struct stat, which is only required to be meaningful for files, though particular implementations of POSIX fstat() may extend its meaning to other things.

If the operand is not a stream, POSIX stat() is tried on it to get a size, again from the st_size field.

Errors in fstat() or stat() cause fsize to set last_error and return om.

Like fexists, fsize gives just a snapshot, not automatically synchronized with updates by other processes.

See also open and ftrunc.

ftrunc - set size of file in bytes

```
op ftrunc (stream f, integer length)
op ftrunc (string pathname, integer length)
```

The file associated with the writable stream *f* or, if *f* is not a stream, the writable file named by *pathname*, is resized to the given *length*, truncating the file or extending it as necessary. When extended, it is padded with NUL characters (\0), possibly in a way that is optimized by the underlying file system.

For the stream case, *f* is first flushed as if by flush (but ignoring output errors) and drained. See [Buffering], page 47.

The underlying POSIX call is then ftruncate() for *f* and truncate() for *pathname*. Errors are reflected in last_error.

See also open and fsize.

get - read lines from stdin

```
proc get (wr string args(*))
```

Equivalent to geta (stdin, *args*(*)).

This signature for get follows that of SETL2, while geta is patterned after the old CIMS SETL **get**. This makes the signatures of get and geta consistent with those of read and reada.

geta - read lines from stream

```
proc geta (stream f, wr string args(*))
```

Zero or more lines are read from the stream *f* and assigned to the succeeding *args* in order, as strings. If an end of input (end of file or error) is reached before all those arguments have been assigned to, trailing arguments are set to om. If it is reached before *any* have been assigned to, the eof indicators are set.

If *f* is not already open, an attempt is made to auto-open it, for reading or for bidirectional I/O depending on the form of *f*. See [Automatic opening], page 47.

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

There is no distinction between "text" and "binary" files, nor any special processing of carriage return (\r).

The rules on auto-flushing *f*'s output associations, on setting last_error, and on auto-closing are as for getc.

The operator-form getline in place of geta may often be stylistically preferable.

See also open, get, getb, getn, gets, peekc, reada, puta, and printa.

getb - read values from stream

```
proc getb (stream f, wr var args(*))
```

Zero or more values are read from the stream *f* and assigned to the succeeding *args* in order. If an end of input (end of file or error) is reached before all those arguments have

been assigned to, trailing arguments are set to `om`. If it is reached before *any* have been assigned to, the `eof` indicators are set.

If *f* is not already open, an attempt is made to auto-open it. See [Automatic opening], page 47.

Values written by `putb` or `writea`, except for atoms (see `newat`) and procedure references (see `routine`), are readable by `getb`. Tokens denoting values are separated by whitespace (ERE `"[ \f\n\r\t\v]+"`) and converted as if by `unstr`.

There is a difference between `getb` and `reada` in that after reading the requested number of values, `reada` continues reading characters until it either absorbs a newline (`\n`) or encounters an end of input, whereas `getb` stops right after the end of the last value read.

The rules on auto-flushing *f*'s output associations, on setting `last_error`, and on auto-closing are as for `getc`.

See also `geta`, `getline`, `getfile`, `getn`, and `val`.

`getc` - read character from stream

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

One character is read from the stream *f* and returned as a string of length 1. If an end of input (end of file or error) is reached instead, `getc` sets the `eof` indicators and possibly `last_error`, and returns `om`.

If *f* is not already open, an attempt is made to auto-open it. See [Automatic opening], page 47.

If `eof` is set by `getc` for an auto-opened sequential stream, the stream is auto-closed, leaving the nullary `eof` true and the unary `eof(f)` invalid.

The `getc` intrinsic, like all input intrinsics, automatically flushes any output buffered for *f* and for any stream associated with *f* by `tie` before attempting input. It does not set `last_error` on output failures in the flushing, but you can `flush` explicitly before the input operation if details on such failures are of interest.

See also `getchar`, `getfile`, `getline`, `getn`, `gets`, `geta`, `getb`, `peekc`, `reada`, `ungetc`, and `putc`.

`getchar` - read character from `stdin`

```
proc getchar : string
```

Equivalent to `getc (stdin)`.

`getegid` - 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).

getenv - get value of environment variable

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

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

See also setenv and unsetenv.

geteuid - 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.

getfile - read stream up to the end

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

Zero or more characters are read from the stream *f* until an end of input (end of file or error) is reached, and returned as a string.

If *f* is not already open, an attempt is made to auto-open it. See [Automatic opening], page 47. As usual, if the file was automatically opened, it is automatically closed on end of input.

The `getfile` intrinsic is unique in that if it fails on the auto-open attempt, it returns om rather than considering the failure erroneous. This helps to discourage racy code like

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

when

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

is race-free and serves a common use case. The latter is more or less equivalent to

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

The eof indicators are always set by `getfile`. This does not matter to users, but allows auto-closing to be defined as something that only happens upon the setting of the eof indicators, as is the case for all other input intrinsics. On the other hand, this makes `getfile` the only intrinsic that sets eof even when it succeeds in getting more than 0 input items.

The rules on auto-flushing *f*'s output associations, on setting last_error, and on auto-closing are as for getc.

See also getline, getn, gets, getb, and putfile.

getgid - get real group ID

```
proc getgid : integer
```

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

Note that group IDs have no relation to *process* group IDs (see `getpgrp`).

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

getline - read line from stream

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

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

Characters through the next newline (`\n`) if any are read from *f* and returned as a string, without the newline. Thus the trailing newline is optional on the last line of a file except when needed to signify an empty line there.

If no characters can be read, `getline` sets the `eof` indicators and returns `om`.

If the stream is not already open, an attempt is made to auto-open it. See [Automatic opening], page 47.

The rules on auto-flushing *f*'s output associations, on setting `last_error`, and on auto-closing are as for `getc`.

If respectability isn't your thing, the expression

```
[getline f : until eof]
```

is a convenient way to read all the lines of a file into a tuple. If *f* is the name of a file that was not previously open, this leaves the file closed after the tuple is accumulated, by the usual rules of auto-opening and auto-closing.

The above expression only works because the `om` coming from the final `getline` call in the loop is trimmed from the tuple.

See also `open`, `close`, `putline`, `getfile`, `getb`, `getn`, `gets`, `reada`, and `printa`.

getn - read fixed number of characters from stream

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

Up to *n* characters are read from the stream *f* and returned as a string. If an end of input (end of file or error) is reached before *n* characters have been read, a shorter string is returned. If it is reached before *any* characters have been read, the `eof` indicators are set and the empty string (`""`) is returned.

If *f* is not already open, an attempt is made to auto-open it. See [Automatic opening], page 47.

The rules on auto-flushing *f*'s output associations, on setting `last_error`, and on auto-closing are as for `getc`.

See also `getfile`, `getline`, `gets`, and `putc`.

getpgrp - get process group ID

```
proc getpgrp : integer
```

Gets the process group ID of the calling process, or in other words the process ID of the process group leader.

See also setpgid, getpid, getppid, getsid, pexists, kill, and waitpid.

getpid - get process ID

```
proc getpid : integer
```

Gets the process ID, in the POSIX sense, of the calling process.

See also pid, getppid, getpgrp, getsid, pexists, kill, and waitpid.

getppid - get parent process ID

```
proc getppid : integer
```

Gets the process ID of the parent of the calling process.

See also getpid.

gets - direct-access read

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

The file under the readable, seekable stream *f* (open mode "r", "r+", "w+", "n+", or "a+") is viewed as a string, where *start* specifies the index (1 or higher) of the first character to read.

The gets intrinsic reads up to *n* characters and returns them as a string through the *x* arg. If an end of input (end of file or error) is reached before *n* characters have been read, a shorter string is returned. If it is reached before *any* characters have been read, the eof indicators are set and the empty string ("") is returned.

If *f* is not already open, an attempt is made to auto-open it in "r+" mode, which allows seeking, reading, and writing. See [Automatic opening], page 47.

If *f* was auto-opened in "r" (not "r+") mode, it will be auto-closed when gets sets eof.

As with seek, *f* is flushed of output (ignoring errors) and drained of input before the file is repositioned. See [Buffering], page 47.

See also getfile, getn, puts, seek, and mkstemp.

getsid - get session ID

```
proc getsid : integer
proc getsid (integer p) : integer
```

Gets the POSIX session ID for process ID *p*, which is to say the process group ID of *p*'s session leader. If *p* is 0 or omitted, the session ID of the calling process is returned.

On error, getsid sets last_error and returns om.

Sessions are used in job control—see setsid.

getuid - 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`.

getwd - current working directory

```
proc getwd : string
```

Current working directory of the process.

See also `chdir`.

glob - pathname wildcard expansion

```
op glob (string) : [string, ...]
```

Using the POSIX `glob()` function with no flags and no error callback, `glob` expands the pathname pattern given in the string operand to produce a tuple of strings. The empty tuple is produced if there is no match to an accessible filename.

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

```
glob "*.h"
```

might equal the tuple

```
["bar.h", "foo.h", "mumble.h"]
```

See also `getwd` and `chdir`.

gmark - find all occurrences of pattern in string

```
proc gmark (string s, pattern p) : [[integer, integer], ...]
```

The locations of all non-overlapping occurrences of the pattern *p* in the string *s* are returned in left-to-right order as a tuple of pairs of integers `[i, j]`, where each matched substring can be addressed as *s*`(i..j)`. If *p* does not occur in *s*, the empty tuple is returned.

For example:

- `gmark ("banana", "an")` is `[[2,3], [4,5]]`
- `gmark ("banana", "ana")` is `[[2,4]]`, not `[[2,4], [4,6]]`

The pattern *p* is subject to the setting of `magic`, and can be a string or a 2-tuple, as detailed under `mark`.

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

gsub - replace patterns in string

```
proc gsub (rw string s, pattern p) : [string, ...]
proc gsub (rw string s, pattern p, string r) : [string, ...]
```

All non-overlapping occurrences in *s* of the pattern *p* are replaced by *r*, which defaults to the empty string (`""`). The substrings of *s* that were matched by *p* are returned as a tuple of strings in left-to-right order.

The pattern *p* is subject to the setting of magic, and can be a string or a 2-tuple, as detailed under mark.

When magic is true, ampersands and backslash-digit sequences in the replacement pattern *r* are expanded as in sub.

Example:

```
s := "abcd aabbccdd";
print (gsub (s, "a([bc]*)d", "&/<\\1>"));  -- prints [abcd abbccd]
print (s);  -- prints abcd/<bc> aabbccd/<bbcc>d
```

See also gmark and split.

hex - convert string to hexadecimal

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

Hexadecimal string representation of *s*. For example, `hex "\011\xCf" = "09CF"`, and `hex char 16#dB = "DB"`.

In general, `#hex s = 2 * #s`.

See also unhex.

hostaddr - current host address

```
proc hostaddr : string
```

This is some plausible Internet address for the current host system, defined as the first `AF_INET` or `AF_INET6` address on the list returned by POSIX `getaddrinfo()` for the host name obtained by POSIX `gethostname()`, in dotted IPv4 or colon-delimited IPv6 notation.

If `gethostname()` or `getaddrinfo()` fails, `hostaddr` sets last_error and returns om.

If `getaddrinfo()` succeeds but yields no addresses in the family `AF_INET` or `AF_INET6`, the value of `hostaddr` is om but last_error is *not* set.

Uses for this function seem limited.

See also hostname, ip_addresses, and ip_names.

hostname - current host name

```
proc hostname : string
```

This is the "standard" name for the current host system as given by POSIX `gethostname()`.

In the unlikely case that `gethostname()` fails, `hostname` sets last_error and returns om.

See also hostaddr, peer_name, ip_names, and ip_addresses.

ichar - integer code for character

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

This operator interprets the one byte in the operand as an integer in the range 0 to 255. For example, `ichar "\x20" = 32`, the code for an ASCII blank.

See also char.

impl - implication

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

Here is the *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 isn't. That is to say, both sides of impl are always evaluated.

This is no great loss, however, because it is usually more natural to write the short-circuiting expression

```
q or not p
```

or

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

than the propositional

```
p impl q
```

especially in the context of if or while tests.

The impl operator has the lowest precedence of any operator (see [Operator Precedence], page 84).

in - membership test; iterator form

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

The keyword in plays a dual role in SETL. Depending on context, it is either the boolean-valued membership test operator whose signature is given above, or the basis of a common iterator form that occurs in loop headers, quantifiers, and set and tuple formers.

Here are two examples of its use in iterators, where *x* acts like a bound variable in each iteration in that it is assigned successive members of a set or tuple *s*, or characters of a string *s*:

```
for x in s loop
  ...
end loop;
squares := {x*x : x in s};  -- set former
```

In its other role, as a binary operator,

```
x in s
```

it is a type-dependent membership test:

- For a set *s*, it tells whether *x* occurs in *s*; om is never considered to be a set member.
- For a tuple *s*, it likewise seeks an occurrence of *x* in *s*, perhaps searching linearly; om is considered present if the tuple has at least one *hole*, i.e., non-trailing om member.
- For a string *s*, it indicates whether *x* is a substring of *s*.

See also arb, from, and notin.

incs - subset test

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

Returns `true` when every member of *ss* is also in *s*. Thus

```
s incs ss
```

has the same truth value as

```
ss subset s
```

Its precedence is quite low, like that of other binary predicates. See [Operator Precedence], page 84.

intslash - integer quotient type switch

```
intslash : boolean
```

By default, the result of dividing two `integer` values in SETL is `real`, as in Pascal and the Algol family. This default corresponds to `intslash = false`. See the discussion of the *slash* operator (`/`) for why it is best to leave it this way if possible.

See also `set_intslash`.

ip_addresses - internet host addresses

```
proc ip_addresses (string host) : {string, ...}
```

Returns a set of Internet addresses as strings in IPv4 dotted or IPv6 colon-separated notation, for the host name or Internet address *host*.

POSIX `getaddrinfo()` is used to obtain the addresses.

For example,

```
ip_addresses ("uccs.edu")
```

might produce the set

```
{"128.198.1.50", "128.198.1.71", "128.198.4.52"}
```

If `getaddrinfo()` fails, `last_error` is set and the empty set is returned. The empty set can be returned with *no* setting of `last_error` if `getaddrinfo()` succeeds but doesn't find any addresses in the `AF_INET` or `AF_INET6` family for the given *host*.

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

ip_names - internet host names

```
proc ip_names (string host) : {string, ...}
```

Returns a set of Internet host names for the host name or IPv4/IPv6 address *host*.

It is like `ip_addresses` but with each address translated to a name using POSIX `getnameinfo()` if possible or omitted if not. For example,

```
ip_names ("uccs.edu")
```

might give the set

```
{"federation.uccs.edu", "klingon.uccs.edu", "warp.uccs.edu"}
```

and

```
ip_names ("::1")
```

might be

```
{"ip6-localhost"}
```

Failure of POSIX `getaddrinfo()` is treated the same as by `ip_addresses`, including the setting of `last_error`. Failure of `getnameinfo()` on an address found by `getaddrinfo()` is *not* reflected in `last_error`, but leaves a name out of the return set.

See also `hostname` and `peer_name`.

`is_type` - 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 operand is a set consisting entirely of ordered pairs (tuples of length 2), none of which has `om` as its first member.

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

The operator `is_atom` tests for a value created by `newat`, and `is_routine` tests for a value created by `routine`.

See also `type` and `denotype`.

The type-testing operators have rather low precedence, like other unary predicates (see [Operator Precedence], page 84).

`is_open` - test for being a stream

```
op is_open (stream f) : boolean
```

Tests whether *`f`* is one of the pre-opened streams `stdin`, `stdout`, or `stderr`; a stream returned by `open`, `accept`, `pipe_from_child`, `pipe_to_child`, `pump`, `tty_pump`, or `mkstemp`; or an automatically opened stream (see [Automatic opening], page 47).

Being a stream is the same as being open at the SETL level; a stream ceases to exist when it is closed.

Although SETL provides no intrinsic specifically for testing whether a given plausible file descriptor *`fd`* is open at the underlying *POSIX* level (an uncommon use case), `dup2(`*`fd`*`,`*`fd`*`)` returns *`fd`* if it is, or sets `last_error` and returns `om` otherwise.

See also `close`.

Being a predicate, `is_open` has rather low precedence (see [Operator Precedence], page 84).

join - concatenate tuple of strings, with delimiter

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

All elements of the tuple *t* must be strings, and they are concatenated together, separated by the delimiter string *glue*.

In general,

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

Thus if *t* is the empty tuple ([]), the result is the empty string (""). Note that *glue* is not used when the number of elements #*t* is 0 or 1, but must still be a string.

See also split.

kill - send signal to process

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

Calls POSIX kill(). Among processes that the caller of kill has permission to send a signal to, *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 allowed process except for an unspecified set of system processes. The signal may or may not be sent to the calling process. POSIX doesn't address this, but Linux excludes the caller in the -1 case.

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

If *signal* is omitted, it defaults to "TERM", or equivalently "SIGTERM". Signals may be given 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. A few other common signals such as PWR and WINCH may also be defined.

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 also pid, pexists, getpgrp, fork, pipe_from_child, pipe_to_child, pump, tty_pump, system, filter, and the open modes "pipe-from", "pipe-to", "pump" and "tty-pump".

last_error - last error message from system function

```
last_error : string
```

After a clear_error call, last_error has the value no_error, and is referred to as *not set*. Otherwise, it has the most recent setting by an intrinsic documented as being able to set last_error.

More precisely, if an intrinsic does set it, it is to

- the error message returned by POSIX `strerror()` for the most recent setting of POSIX `errno`, or
- the error message returned by POSIX `gai_strerror()` for the most recent failed POSIX `getaddrinfo()` or `getnameinfo()` call.

len - extract leading substring by length

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

An initial substring of length *n* `min` `#`*s* is removed from *s* and returned. It is an error for *n* to be less than 0.

See also the other SNOBOL-inspired intrinsics, namely `any`, `break`, `match`, `notany`, `span`, `rany`, `rbreak`, `rlen`, `rmatch`, `rnotany`, and `rspan`.

less - set less given element

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

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

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

lessf - map less given 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 first (i.e., domain) element are removed.

See also `less` and `from`.

lexists - test for existence of file or symlink

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

Returns `true` iff POSIX `lstat()` returns 0 on the given pathname.

Note that `lstat()` does *not* follow symbolic links, so `lexists` returns `true` for any existing pathname (provided the caller has sufficient access to all pathname components in reaching it), even a *dangling* symlink (one that refers to a file that doesn't exist).

Thus `lexists` is less strict than `fexists`, which uses POSIX `stat()`. Both provide mere snapshots, not automatically synchronized with actions by other processes. See `link` and `symlink` for some ways to use files as mutual exclusion (mutex) locks.

See also `readlink` and `unlink`.

Like `fexists`, this predicate has rather low precedence (see [Operator Precedence], page 84).

link - create hard link

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

Creates a link (*hard link*) *new* to the existing file *f* using POSIX `link()`, if *new* does not exist before the call.

If `link` succeeds, *`new`* and *`f`* then refer to the same file. Otherwise, such as when *`new`* already exists, `last_error` is set.

Given stable directory structures above *`f`* and *`new`*, `link` behaves atomically, and can be used as a test-and-set mechanism for inter-process synchronization: the mutex (lock file) *`new`* is acquired if and when `link` succeeds, and releasing it is done atomically by `unlink`.

See also `fexists`, `symlink`, and `rename`, and the `open` modes `"n"` and `"n+"`.

`log` - natural logarithm

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

The operand must be greater than 0.

See also `exp`.

`lpad` - pad string on left with blanks

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

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

It is an error for *`n`* to be less than 0.

See also `rpad`, which *left-justifies* by padding on the right.

`magic` - regular expression recognition switch

```
magic : boolean
```

This is a global modal switch.

By default, `magic` is `true`, meaning that subscripting and slicing of subject strings like say *`s`* by pattern strings *`p`*, *`p1`*, and *`p2`* in expressions like *`s`*`(`*`p`*`)` and *`s`*`(`*`p1`*`..`*`p2`*`)` interprets the pattern strings as POSIX *extended regular expressions* (EREs). This also affects `sub`, `gsub`, `mark`, `gmark`, and `split`.

You can assign `magic := false`, or call `set_magic (false)`, to cause pattern strings to be interpreted literally, i.e., as strings to be matched exactly somewhere in *`s`*.

`mark` - find first occurrence of pattern in string

```
proc mark (string s, pattern p) : [integer, integer]
```

The location of the first (leftmost) occurrence of the pattern *`p`* in the string *`s`* is returned as a pair of integers `[`*`i`*`, `*`j`*`]` which index the first and last characters in the substring matched by *`p`*, i.e., *`s`*`(`*`i`*`..`*`j`*`)`.

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

With `magic` left at its default setting of `true`, a string-valued *`p`* is interpreted as a POSIX extended regular expression (ERE).

If `magic` is `false`, a string-valued *`p`* is interpreted literally as the substring of *`s`* to match.

If *`p`* is a pair `[`*`p1`*`, `*`p2`*`]` of strings, it represents a pattern that begins with *`p1`* and ends with the first subsequent occurrence of *`p2`*. The setting of `magic` applies to each of *`p1`* and *`p2`*.

There can be any characters between *p1* and *p2*, regardless of `magic`—sort of like the ERE `".*"`, but not so greedy.

As with the SETL expression *s*`(`*p1*`..`*p2*`)`, the substring of *s* matched by a pattern pair begins with the first character of the first substring matching *p1* and ends with the last character of the first substring matching *p2* after that.

Also in line with SETL *s*`(`*p1*`..`*p2*`)` expressions, either or both of *p1* and *p2* may be an integer index rather than a pattern-bearing string, again irrespective of `magic`. If *p1* is an integer, it simply becomes the *i* in the returned `[`*i*`, `*j*`]`. If *p2* is an integer, *j* is the greater of *p2* and *i*`-1`.

Like in string subscripting expressions, *p* can be an `integer`, in which case `mark` returns `[`*p*`, `*p*`]` if *p* `<=` `#`*s*, or `[`*p*`, `*p*`-1]` otherwise.

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

`match` - extract leading substring by exact match

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

If *p* is an initial substring of *s*, i.e., if *s*`(1..#`*p*`) = `*p*, it is removed from *s* and returned. Otherwise, nothing happens to *s* and the empty string (`""`) is returned.

See also the other SNOBOL-inspired intrinsics, namely `any`, `break`, `len`, `notany`, `span`, `rany`, `rbreak`, `rlen`, `rmatch`, `rnotany`, and `rspan`.

`max` - maximum

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

Strings are compared character by character, as if using the codes arising from `ichar`. Tuple comparisons are element by element, and recursive. If one string or tuple is a prefix of the other, the longer one is considered larger.

For mixed numeric modes, the `integer` is converted to `real` before comparison, but the result has the type of the larger, or of the first operand in case of a tie.

If either operand is a floating-point NaN (*Not a Number*), the result is a NaN. Contrast POSIX `fmax()`, which returns the non-NaN arg when there is just one.

Examples:

```
1 max 2 = 2  -- max is a binary operator
max/ [1, 2, 3] = 3  -- max/ t gives max over set or tuple t
x max/ [] = x  -- but unary max/ [] is erroneous
```

See also the order-based *comparatives*, and `min`.

min - minimum

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

Strings are compared character by character, as if using the codes arising from `ichar`. Tuple comparisons are element by element, and recursive. If one string or tuple is a prefix of the other, the shorter one is considered smaller.

For mixed numeric modes, the `integer` is converted to `real` before comparison, but the result has the type of the smaller, or of the first operand in case of a tie.

If either operand is a floating-point NaN (*Not a Number*), the result is a NaN. Contrast POSIX `fmin()`, which returns the non-NaN arg when there is just one.

This operator is commonly used in the combining form, `min/`, over a set or tuple. See `max` examples.

mkstemp - create and open temporary file

```
proc mkstemp (rw string template) : integer
```

The *`template`* must end in the characters `"XXXXXX"`, which will be overwritten by characters that make the resulting string contain the name of a file that does not currently exist.

Then a file with that name is created with read/write permissions for the owner and none for others, using POSIX `mkstemp()`. A SETL `open` in `"w+"` mode is effectively performed over that, and the resulting file descriptor is returned.

On failure, `last_error` is set and `om` is returned. The *`template`* is not modified in that case.

The funky signature, with its read/write template arg, resembles that of the underlying POSIX function.

See also `seek`, `rewind`, `puts`, `gets`, and `filename`.

mod - integer modulus; symmetric set difference

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

SETL yields 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
```

See also `rem` and `div`.

The set-theoretic symmetric difference operator `mod` is analogous to the logical *exclusive or*, and is likewise associative and commutative (unlike `mod` over integers, which is neither). Two sets `s` and `t` can be swapped without an intermediate temporary variable thus:

```
s mod:= t;  -- add the info in t to s
t mod:= s;  -- take out the t, leaving old s
s mod:= t;  -- take out the old s, leaving old t
```

See also the regular set difference (*minus*) operator (`-`).

nargs - number of arguments given by caller

```
nargs : integer
```

For procedures that take a variable number of arguments (i.e., have the token sequence `(*)` after the final formal parameter, which the procedure sees as a tuple), `nargs` is the total number of arguments supplied by the caller to the currently active procedure.

newat - create new atom

```
proc newat : atom
```

This creates a unique `atom`, whose salient property is merely that it is different from all other `atom`s created by the current process. Atoms are like opaque pointers, and cannot be meaningfully exchanged between programs. They are sometimes used to highlight the domain independence of an abstract algorithm, but are otherwise rather useless. Real applications tend to be expressed over concrete domains with fitting rules and conventions.

See also `is_atom`.

no_error - non-error message

```
no_error : string
```

This is the value of `last_error` immediately after a call to `clear_error`. It is typically some locale-dependent version of `"No error"` or `"Success"`.

not - logical negation

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

The unary predicate `not` has a precedence above `and`, `or`, and `impl`, but below that of all the other binary operators and non-predicates. See [Operator Precedence], page 84. Still, generous use of parentheses is recommended for readability and for ease of transliteration to other languages.

See also the bitwise operators such as `bit_not`.

notany - extract leading character using character set

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

If the first character of `s` does not occur in `p` (treating `p` as a set of characters), that first character is removed from `s` and returned. Otherwise, nothing happens to `s`, and the empty string (`""`) is returned.

See also the other SNOBOL-inspired intrinsics, namely any, break, len, match, span, rany, rbreak, rlen, rmatch, rnotany, and rspan.

notin - 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)`.

npow - all subsets of 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 subsets of *s* that have *n* members, or the empty set if *n* exceeds #*s*. It is an error for *n* to be negative.

nprint - print to stdout with no trailing newline

```
proc nprint (var args(*))
```

Equivalent to `nprinta (stdout, args(*))`.

See also print and write.

nprinta - print to stream with no trailing newline

```
proc nprinta (stream f, var args(*))
```

The 0 or more *args* are written in sequence to the stream *f*, separated by single spaces. String arguments are written directly; all others are converted as if by str first.

If *f* is not already open, an attempt is made to auto-open it. See [Automatic opening], page 47.

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'.

On output error, output may be incomplete and last_error may be set.

See also nprint, printa, and writea.

odd - test for integer not divisible by 2

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

Not even, though it shares the rather low precedence of that predicate.

om - 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 routines that do not return anything else, and the default result of many operations when they fail in ways that are not held to be errors.

Ideal if nothing is what you want. Sounds nice when said slowly. Depicted as a capital omega in the ancient texts. Could stand for *omitted* in some places.

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

open - open a stream

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

Tries to create a stream for *f*, where *f* may be

- a string such as a filename, other pathname, command, or signal name,
- a tuple representing an interval timer or Internet service location, or
- an integer fd that is already open at the POSIX (operating system) level but not at the SETL level (see below).

How *f* is interpreted depends on the mode argument, *how*. See [Arguments to open], page 40.

On success, the returned fd (or *pseudo-fd*, for a signal or timer stream) serves as a stream handle for use in SETL I/O. The fd is then "open at the SETL level". The stream incorporates relevant state and buffer structure. See [Buffering], page 47.

On failure due to external factors such as a missing file, `open` sets `last_error` and returns `om`.

The `open` described here is almost upwardly compatible with the old CIMS SETL **open**. It is completely compatible for programs which ignored the return value, because all I/O intrinsics in the present SETL accept as a stream specifier either the argument that was originally passed to a successful `open` call (if the arg is unique) 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.

Arguments to open

Valid values of the case-insensitive I/O mode argument *how*, and their meanings, are:

mode	**meaning**
`"r"`	sequential and direct access input
`"w"`	sequential and direct access output
`"n"`	like `"w"`, but new file only
`"a"`	sequential output, append to file
`"r+"`	direct access r/w, existing file
`"w+"`	direct access r/w, empty file first
`"n+"`	like `"w+"`, but new file only
`"a+"`	direct access read, write at end
`"rw"`	sequential bidirectional I/O
`"pipe-from"`	input from shell command
`"pipe-to"`	output to shell command

`"pump"`	I/O to and from shell command
`"tty-pump"`	I/O to and from pty-wrapped command
`"tcp-client"`	TCP client socket
`"tcp-server"`	TCP server socket
`"tcp-peer"`	connected TCP socket
`"udp-client"`	UDP client socket
`"udp-server"`	UDP server socket
`"unix-client"`	Unix-domain stream client socket
`"unix-server"`	Unix-domain stream server socket
`"unix-peer"`	connected Unix-domain stream socket
`"unix-datagram-client"`	Unix-domain datagram client socket
`"unix-datagram-server"`	Unix-domain datagram server socket
`"signal"`	one input line per catch
`"ignore"`	signal to be ignored
`"default"`	signal to be given default effect
`"real-ms"`	one input line per timer expiry

GNU SETL also has many synonyms for these I/O modes, perhaps best not used in new code (see [Alternative *how* arguments to `open`], page 49).

For all of modes `"r"` through `"unix-datagram-server"` above, *`f`* may be a file descriptor (fd) that is already open at the POSIX level but not at the SETL level. More on that below.

Modes `"r"` through `"rw"` cause a POSIX `open()` call with the following flags:

mode	**POSIX `open()` flags**	**seekable?**
`"r"`	`O_RDONLY`	yes
`"w"`	`O_WRONLY | O_CREAT | O_TRUNC`	yes
`"n"`	`O_WRONLY | O_CREAT | O_EXCL`	yes
`"a"`	`O_WRONLY | O_CREAT | O_APPEND`	no
`"r+"`	`O_RDWR`	yes
`"w+"`	`O_RDWR | O_CREAT | O_TRUNC`	yes
`"n+"`	`O_RDWR | O_CREAT | O_EXCL`	yes
`"a+"`	`O_RDWR | O_CREAT | O_APPEND`	yes
`"rw"`	`O_RDWR`	no

A stream termed *seekable* is not actually known at the time of `open` to be on a file supporting direct access. Where not, a `seek`, `rewind`, `gets`, or `puts` call will generally fail on the underlying POSIX `lseek()` attempt, which is then considered erroneous.

Prior to that seeking attempt, any buffered output is flushed (written out to the file, ignoring any output errors that may occur), and any buffered input is drained (discarded). See [Buffering], page 47.

The `"pipe-from"`, `"pipe-to"`, `"pump"`, and `"tty-pump"` modes cause an external program, given as a shell command string, to be run as a subprocess whose standard input and/or standard output is connected to the fd returned by `open`. See [Connected subprocesses], page 44.

Modes `"tcp-client"`, `"tcp-server"`, `"udp-client"`, and `"udp-server"` create network socket streams, with *`f`* specifying an Internet host and service, except of course where *`f`* is the fd of a socket that already exists at the POSIX level. The mode `"tcp-peer"` opens a stream over the fd of a connected TCP socket. See [TCP and UDP sockets], page 42.

Modes `"unix-client"` through `"unix-datagram-server"` create the corresponding Unix-domain socket streams, with *f* specifying a pathname for the socket in the local filesystem space. For symmetry with `"tcp-peer"`, the mode `"unix-peer"` can be used when *f* is the fd of a connected non-datagram Unix-domain socket. See [Unix-domain sockets], page 43.

The term *stream* for datagram sockets is a bit of a stretch, as they are distinguished by being *not* "stream-oriented". Likewise, `"tcp-server"` and `"unix-server"` sockets do not stream data, but are only used to `accept` new clients.

When *f* is a fd, the assumption is that the mode you provide is compatible with how the fd is open at the POSIX level. So, for example, `"rw"` is a good mode to use to `open` the fd of an inherited bidirectional stream (such as a connected socket, coprocess, or input/output device); and `"r"` or `"r+"` might be appropriate for a direct-access file. In some cases, you might want to provide a more specific mode, like the abovementioned `"tcp-peer"` to indicate that you intend to do operations requiring a connected TCP socket.

For modes `"signal"`, `"ignore"`, and `"default"`, *f* must be a signal name. See [Signal streams], page 44.

For mode `"real-ms"`, *f* must be an ordered pair (2-tuple) of integers `[`*initial*`, `*interval*`]` where *initial* is the number of milliseconds before the first desired timer expiry and *interval* the period after that. See [Timer streams], page 46.

TCP and UDP sockets

For modes `"tcp-client"`, `"tcp-server"`, `"udp-client"`, and `"udp-server"`, the first argument to `open` should be (unless a fd) a 2-tuple `[`*h*`, `*p*`]` where *h* identifies an Internet host by name or by address in IPv4 dotted or IPv6 colon-rich notation, and *p* is a service name or a port number given as an integer or string of decimal digits.

For the server modes, *h* may be `"0.0.0.0"` to request that connections or datagrams be accepted on any IPv4 interface, or `"::"` for any IPv6 interface. It is also possible to let the system choose which kind of interface to use for the server socket, by letting *h* be `om` or the empty string (`""`). The wise client would then try both IPv4 and IPv6. A more accommodating server might listen (or take datagrams) on both `"0.0.0.0"` and `"::"`.

If the *p* in `[`*h*`, `*p*`]` is zero or `om` (omitted), the system chooses an available port number which can be retrieved using `port` or `sockaddr`.

For example, given

```
fd := open ([om, 0], "tcp-server")
```

the value of `sockaddr` *fd* might be something like `["0.0.0.0", 42113]` or `["::", 53622]`.

For mode `"tcp-server"`, the POSIX-level socket option `SO_REUSEADDR` is set on the listening socket.

A call to `open` a TCP client connection can block for an unspecified length of time, as can a call to an intrinsic that attempts to auto-open one (see [Automatic opening], page 47).

If the first arg to `open` is the fd of an already connected TCP socket, such as might have arisen from `accept`, the mode `"tcp-peer"` can be used to open a SETL stream over it, to indicate intent to call things like `peer_sockaddr` that require a connected TCP socket. Such a socket is indistinguishable at the programming level from a client socket, so mode `"tcp-client"` would work just as well but be misleading in the case of an accepted client. The designation `"tcp-peer"` is also a good fit to the case where you don't care which role

(client or server) the socket plays. (If you don't even care that it's a socket, the more generic mode `"rw"` might serve better still.)

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`. Conversely, `recv` can only be used on UDP client sockets (not on Unix-domain datagram client sockets, which have no names and thus cannot be sent to). But `send` can be used on either kind of datagram client socket, and `recvfrom` and `sendto` can be used on either kind of datagram server socket.

For legacy support, the canonical `[h, p]` tuple for identifying a host and port may be given as a string of the form `"h:p"`.

Failure of `open` for a network socket mode gives `om` instead of a fd, and sets `last_error` in accordance with the failing POSIX `socket()`, `connect()`, `bind()` or `listen()` call. A fd passed to `open` that is not open at the POSIX level sets `last_error` to a locale-dependent version of "Bad file descriptor".

See also `peer_address`, `peer_name`, and `peer_port`.

Unix-domain sockets

For modes `"unix-client"`, `"unix-server"`, `"unix-datagram-client"`, and `"unix-datagram-server"`, the first arg to `open` must be (unless a fd) a pathname *f* for the socket. A Unix-domain (local) socket is created by POSIX `socket()`, and its file descriptor is returned by `open` if one of these cases also succeeds:

- For a client mode, POSIX `connect()` connects the socket to the server at *f*. No actual connection is made in the case of `"unix-datagram-client"`, but a later `send` will use the remembered pathname.
- For a server mode, any existing pathname *f* is removed by POSIX `unlink()`, and then created and bound to the socket by POSIX `bind()`. The `unlink()` is meant to improve the chances of success of the `bind()`, but the sequence of calls is not performed as an atomic unit. The pathname has all permissions enabled, minus those turned off by the current file mode creation mask (see `umask`). For mode `"unix-server"`, POSIX `listen()` is then called.

Errors in any of the above POSIX calls cause `open` to return `om` and set `last_error`. The expected ENOENT from `unlink()` does not count as an error.

If the first arg to `open` is the fd of an already connected Unix-domain stream socket, such as might have arisen from `accept`, the mode `"unix-peer"` can be used to open a SETL stream over it, for symmetry with `"tcp-peer"` mode.

Upon `close` of a server socket, the pathname is again removed. Errors from `unlink()` are ignored in this case, as the pathname may legitimately have been removed already.

File descriptors may be passed on non-datagram Unix-domain sockets using `send_fd` and `recv_fd`.

Unix-domain sockets do not support network-oriented queries such as `sockaddr` and `peer_name`. Unix-domain datagram client sockets do not support `recv`, as they are anonymous and thus cannot be sent to.

In other respects, Unix-domain sockets act much like their network counterparts (see [TCP and UDP sockets], page 42):

Unix-domain	network
"unix-client"	"tcp-client"
"unix-server"	"tcp-server"
"unix-peer"	"tcp-peer"
"unix-datagram-client"	"udp-client"
"unix-datagram-server"	"udp-server"

They do have some particular advantages over network sockets though, such as the absence of lingering connection state when the server closes first, and the immediate *broken pipe* error seen by the sender on a write when the receiver has done a shut_rd.

Connected subprocesses

For a "pipe-from" stream, the standard output of the child process is connected to the (readable) fd returned by open in the parent process.

For a "pipe-to" stream, the child's standard input is connected to the parent's (writable) fd.

For a "pump" stream, the standard input and output of the child process are both connected to the parent's (bidirectional) fd.

The connection between parent and child in all three of these cases is a Unix-domain socketpair, and therefore supports send_fd and recv_fd.

A "tty-pump" stream resembles a "pump" stream, but the child's standard input and output are instead connected to the slave side of a pseudo-terminal (pty) in *raw* mode, while the open caller gets the fd of the master. The child's terminal-like environment lets the parent direct a program intended for interactive use. Also, since POSIX programs usually line buffer their standard output instead of block buffering it when stdout appears to be connected to a terminal, the "tty-pump" mode lets you use an off-the-shelf program such as sed or awk as a coprocess in line-by-line message exchange fashion, without response lines getting stuck in the child's output buffer and never being seen by the parent.

A string first argument *f* to open gives a command to be run by the standard shell, as if by exec ("/bin/sh", ["sh", "-c", *f*]). The command is run in a subprocess created as if by pipe_from_child, pipe_to_child, pump, or tty_pump.

For all these subprocess streams, the second argument to the close that balances the open is significant. The default of close_await is usually appropriate, and sets status to the child's exit status. Unclosed pipe/pump streams are closed automatically upon program termination, but *not* using close_await.

The signal dispositions in the child are as for a fork, followed in the above open modes by an exec.

See also pid, kill, filter, system, flush, shutdown, and socketpair.

Signal streams

When the *how* argument to open is "signal", "ignore", or "default", the *f* argument may 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; dump core	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
"SIGALRM"	terminate process	timer expiry
"SIGTERM"	terminate process	software termination request
"SIGCHLD"	ignore	child status change
"SIGCONT"	ignore	continue after stoppage
"SIGTSTP"	stop process	terminal stop signal
"SIGTTIN"	stop process	background process attempting read
"SIGTTOU"	stop process	background process attempting write
"SIGXCPU"	terminate; dump core	soft CPU limit exceeded
"SIGXFSZ"	terminate; dump core	soft filesize limit exceeded
"SIGPWR"	ignore	low battery, or power failure imminent
"SIGWINCH"	ignore	terminal window size change

The default actions shown here are the POSIX defaults, corresponding to a disposition of SIG_DFL at the POSIX sigaction() level. The actual disposition for most may be inherited by the SETL program as if by SIG_IGN (ignore the signal).

SIGPWR and SIGWINCH are not specified by POSIX, but are widely available.

There are many signals that can be sent by kill but cannot be caught or ignored, such as SIGKILL.

The *file descriptor* returned by open for any of these stream types is a *pseudo-fd*, meaning a small integer like a POSIX fd but phony in that it lies outside the normal fd range (which is limited by the host operating system; see your shell's ulimit or limit command for the maximum number of open files, which is one more than the highest possible system-level fd.

The pecking order for signal handling is as follows:

- Whenever a signal is caught because there is at least one "signal" stream open on that signal name, an empty line is delivered to every such stream.

  Applications are encouraged to anticipate information content in extensions of the "signal" stream type by reading an arbitrary line, e.g. by using getline or geta, or reada with only the stream arg.
- Otherwise (no "signal" streams open for the given signal name), when a signal is received and there is at least one stream of mode "ignore" open on that signal name, the signal is ignored as if by SIG_IGN at the POSIX sigaction() level.
- Otherwise (neither of the above), when a signal is received and there is at least one stream of mode "default" open on that signal name, the signal is defaulted as if by POSIX SIG_DFL to the default action in the above table.
- Otherwise, with no streams open on a given signal name, the signal disposition is given by the program environment, which is typically (but not necessarily) the default listed in the above table.

Note that "ignore" and "default" streams are created only for their effects on signal dispositions. The only thing you can do with one is close it, which may cause its disposition to revert according to the above pecking order (catch, ignore, default, inherit).

To allow the SETL implementation to use SIGCHLD in support of close_autoreap, opening SIGCHLD in "ignore" or "default" mode does not respectively prevent or enable zombies as a POSIX-level SIG_IGN or SIG_DFL disposition would.

A "signal" stream opened on SIGCHLD gets a line each time a child process terminates, stops (suspends), or continues (resumes). This type of signal does not necessarily queue, and applications may do well to loop over some non-blocking waitpid and status checks each time a line is received from a SIGCHLD stream.

To allow the SETL implementation to use SIGALRM in support of the "real-ms" mode (see [Timer streams], page 46), SIGALRM cannot be opened as a "signal" stream, despite its appearance in the above table. It can, however, be opened in "ignore" or "default" mode, in order to control the initial disposition of SIGALRM in child processes, as all timer streams are closed initially in the child, letting the POSIX-level SIG_IGN or SIG_DFL take effect according to the rules given above. If no SIGALRM stream is open, its disposition in the child is what the parent began with.

Signal streams can be used with select.

Timer streams

For mode "real-ms", the first arg to open should be a pair of integers [*initial, interval*] giving the number of milliseconds before the first timer expiry and between subsequent expiries.

The degenerate form [*interval*], meaning [*interval, interval*], may also be used, and for back-compatibility, the interval may be given as a decimal string rather than in a tuple.

As with a signal stream, the fd returned by open is a pseudo-fd.

All timer streams are initially closed in a child process, which allows its initial SIGALRM disposition to be controlled by the parent (see [Signal streams], page 44).

Timer streams can be used with select.

Predefined streams

There are three predefined streams with the following case-insensitive aliases:

name	fd	aliases	meaning
stdin	0	"", "-", "stdin", "input"	standard input
stdout	1	"", "-", "stdout", "output"	standard output
stderr	2	"stderr", "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.

Automatic opening (and closing) of streams

The intrinsics `geta`, `getb`, `getc`, `getfile`, `getline`, `getn`, `peekc`, and `reada` attempt to open a stream `f` automatically if it is not already open at the SETL level. This also applies when `f` is a member of the *readable* set passed to `select`.

If `f` is a tuple of up to 2 elements, these intrinsics try to open a bidirectional TCP client connection or a (read-only) interval timer, depending on the type of `f(1)`: an integer indicates a timer (see [Timer streams], page 46).

If `f` is a string or integer, they try to `open` it in sequential reading (`"r"`) mode.

Similarly, `nprinta`, `printa`, `puta`, `putb`, `putc`, `putfile`, `putline`, and `writea` (and `select` when `f` appears in the *writable* set) attempt to auto-open `f` as a bidirectional TCP client socket if it is a 2-tuple, or as a sequential output stream otherwise (`"w"` mode).

The `accept` intrinsic attempts to auto-open a TCP server socket for any argument that would satisfy `open` in `"tcp-server"` mode.

The intrinsics `gets`, `puts`, `rewind`, and `seek` attempt to auto-open a stream in `"r+"` mode (read/write direct access to an existing file) for any string or plausible fd first argument.

The intrinsics `send` and `recv` attempt to auto-open a UDP client socket if `f` is not already a stream, and `sendto` and `recvfrom` try to auto-open a UDP server socket. This applies when `f` is a tuple, string, or integer.

Failure of auto-open is considered erroneous except in the case of `getfile`, which sets `last_error` and returns `om`.

When the `eof` indicators are being set, a stream that has been auto-opened will be auto-closed as if by `close` (which may cause `last_error` to be set), except that a stream auto-opened in `"r+"` mode is never auto-closed.

The only *output* intrinsic that auto-closes a stream is `putfile`, and only on an auto-opening call.

No intrinsic ever attempts to open a stream in mode `"n"`, `"a"`, `"w+"`, `"n+"`, `"a+"`, `"rw"`, `"pipe-from"`, `"pipe-to"`, `"pump"`, `"tty-pump"`, `"unix-client"`, `"unix-server"`, `"unix-datagram-client"`, `"unix-datagram-server"`, `"signal"`, `"ignore"`, or `"default"`.

Buffering

An important difference between file descriptors that are open at the SETL level and the POSIX file descriptors that underlie them at the system level is that at the SETL level, a fd returned by `open` implicitly has an attached buffer structure for SETL-level I/O state.

This is also true for a fd returned by `accept`, `mkstemp`, `pipe_from_child`, `pipe_to_child`, `pump`, or `tty_pump`, but *not* for any fd returned by the low-level intrinsics `dup`, `dup2`, `socketpair`, `pipe`, or `recv_fd`.

Thus the POSIX fd, a small non-negative integer, serves as a handle for the SETL stream. Contrast this with C, where buffered I/O is usually done by the *stdio* layer using a separate `FILE` object that contains the fd and the buffer structure.

Data written by intrinsics such as `putc` accumulates in the stream's output buffer until the stream is *flushed* by writing the data out using POSIX `write()` or equivalent. Flushing

is done automatically according to the buffering policy associated with the stream, and whenever `flush` is called:

- In the *block* buffering policy, the output buffer is flushed whenever it becomes full. The implementation-defined capacity of the output buffer is typically on the order of a few thousand bytes.
- The *line* buffering policy is the same as block except that if a newline character (`\n`) is written by the SETL program, the buffer is flushed even if it is not yet full.
- In the *byte* buffering policy, characters are written out at the system level as soon as they are written by the SETL program.

The buffering policy is set when the stream is created. Most streams are block buffered, but if a sequential data stream is opened on what appears to be a tty-like device according to POSIX `isatty()`, it is line buffered, except in the case of `stderr` (fd 2), which is byte buffered by default (though see Section "`SETL_LINEBUF_STDERR`" in the *GNU SETL User Guide*).

One implication of output buffering is that most programs that exchange messages with other programs should always flush output before attempting input, so that messages are fully sent before replies are awaited.

The SETL I/O system takes care of most such flushing automatically. In particular, a bidirectional stream (such as for a connected socket or pump or direct-access file) is implicitly flushed whenever input is initiated on that stream.

More generally, automatic flushing of the output buffer occurs on a data stream `f`, and on the stream if any that is linked to `f` by `tie`, when any of these occur:

- SETL input (except via `gets`) is attempted on `f`;
- a `select` call includes `f` in the *readable* set;
- `recv_fd` is called on `f`.

Thus the call `tie (stdin, stdout)` can be used to ensure that `stdout` is auto-flushed before any attempt is made by the program to read from `stdin` or await its readability in a `select`-based event loop.

Auto-flushing of `f` (but not of any tie) is done on calls to:

- `seek`, `rewind`, `gets`, or `puts`;
- `close`, or `shutdown` with a second arg of `shut_wr` or `shut_rdwr`;
- `send_fd`;
- `ftrunc` when `f` is a stream arg (rather than a pathname that is not open).

Also, whenever `fork` is called (at least effectively, as for example in an intrinsic like `pump` which is described as creating a child process "as if by `fork`"), all streams are flushed automatically before the spawning attempt.

Furthermore, when the program is finalizing in preparation for exit, all streams are flushed and then all closed.

Although an explicit `flush` call sets `last_error` in the event of a POSIX `write()` error, auto-flushing never does, with the sole exception of `putfile` in the auto-open/auto-close case (see [Automatic opening], page 47), which acts as if `flush` were called explicitly.

Input buffering is largely invisible at the SETL level. The SETL implementation is expected to request up to some number of bytes from the system (again, typically on the order of a few thousand bytes) whenever input is requested by the SETL program in the presence of an empty input buffer. That system call, typically POSIX `read()` or equivalent, after waiting as long as necessary for at least one byte or an end of file, receives some bytes into the buffer or indicates an end-of-file or error condition.

Although input buffering is mostly just silently efficient, there are places where it can be noticed. One such case is `ungetc`, as the SETL implementation is only required to support *at least* one character of pushback after a `getc`. It *may* allow more.

More significantly, input is *drained* (discarded) upon initiation of any of these operations on *f*:

- SETL output;
- a seeking operation (`seek`, `rewind`, `gets`, `puts`);
- `ftrunc` when *f* is a stream (rather than a pathname that is not open).

The draining of input on every output attempt on *f*, along with the automatic flushing of output on every input attempt, is ideal for the common case of a bidirectional stream used in a message-and-reply or immediate-handshake regime. But for truly full-duplex cases where messages in each direction are to be overlapped, the use of subprocesses, each with its own copy of the fd, is generally the best approach in SETL. The full-duplex operation is then effectively relegated to the so-called "file description" level (the buffer structure of the POSIX kernel or equivalent; a file description is shared by a fd and all its duplicates, whether arising from the likes of POSIX `dup()` or from process duplication *à la* `fork()`).

Buffering can also be bypassed entirely by using the "non-SETL I/O" `sys_read` and `sys_write` intrinsics.

Datagrams, sent by `send` and `sendto`, are not buffered. Nor are file descriptors, sent by `send_fd`.

Alternative *how* arguments to `open`

The following synonyms for the *how* argument also exist in GNU SETL, the apparent result of over-building on some back-compatibility union of historical opening mode names.

If the mode names in the left column may be taken as standard, the aliases on the right should perhaps be deprecated.

Despite the presentation, they are all case-insensitive:

`"a"`	`"AB"`
`"a"`	`"APPEND"`
`"a"`	`"BINARY-APPEND"`
`"a"`	`"CODED-APPEND"`
`"a"`	`"OUTPUT-APPEND"`
`"a"`	`"PRINT-APPEND"`
`"a"`	`"TEXT-APPEND"`
`"a+"`	`"A+B"`
`"a+"`	`"AB+"`
`"default"`	`"DEFAULT-SIGNAL"`
`"default"`	`"SIGNAL-DEFAULT"`

```
"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"
"signal"                "SIGNAL-IN"
"tcp-client"            "TCP-CLIENT-SOCKET"
"tcp-client"            "CLIENT-SOCKET"
"tcp-client"            "SOCKET"
"tcp-server"            "TCP-SERVER-SOCKET"
"tcp-server"            "SERVER-SOCKET"
"tty-pump"              "LINE-PUMP"
"udp-client"            "UDP-CLIENT-SOCKET"
"udp-server"            "UDP-SERVER-SOCKET"
"unix-client"           "UNIX-STREAM-CLIENT"
"unix-client"           "UNIX-CLIENT-SOCKET"
"unix-server"           "UNIX-STREAM-SERVER"
"unix-server"           "UNIX-SERVER-SOCKET"
"w"                     "BINARY-OUT"
"w"                     "CODED-OUT"
"w"                     "OUTPUT"
"w"                     "PRINT"
"w"                     "TEXT-OUT"
"w"                     "WB"
"w+"                    "W+B"
"w+"                    "WB+"
```

or - logical disjunction

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

The expression

```
x or y
```

is equivalent to the expression

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

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

Contrast the bitwise operators such as `bit_or`.

The `or` operator has a very low precedence, above `impl` but below `and`. See [Operator Precedence], page 84.

peekc - peek at next character in stream

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

The next available character, if any, in the stream `f` is returned as a string of length 1, as if by `getc`. However, the character also remains in the input as if it were *pushed back* by `ungetc`. In all other respects, including auto-opening, the flushing of any output associations `f` may have, the setting of `eof` and `last_error`, and auto-closing, `peekc` behaves like `getc`.

See also peekchar, ungetc, and ungetchar.

peekchar - peek at next character in stdin

```
proc peekchar : string
```

Equivalent to peekc (stdin).

peer_address - peer host address

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

If the stream *f* is a connected TCP or UDP socket, peer_address returns the Internet address of the peer in either IPv4 dotted or IPv6 colon-delimited notation. It is permissible for *f* to be a file descriptor that is open only at the POSIX level.

Note that for a UDP client socket, there is no actual connection, just a record made of the peer address when it was opened.

On failure of the underlying POSIX getpeername() to find a peer for *f* in address family AF_INET or AF_INET6, peer_address sets last_error and returns om.

See also open, filename, peer_name, peer_port, peer_sockaddr, ip_addresses, ip_names, and hostaddr.

peer_name - peer host name

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

If the stream *f* is a connected TCP or UDP socket, peer_name returns an Internet host name for the peer. It is permissible for *f* to be a file descriptor that is open only at the POSIX level.

If the underlying POSIX getpeername() fails to find a peer address for *f* in address family AF_INET or AF_INET6, or if a corresponding name cannot be found by POSIX getnameinfo(), peer_name sets last_error and returns om.

See also open, filename, peer_address, peer_port, peer_sockaddr, ip_names, and hostname.

peer_port - peer port number

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

If the stream *f* is a connected TCP or UDP socket, peer_port returns the port number of the peer. It is permissible for *f* to be a file descriptor that is open only at the POSIX level.

On failure of the underlying POSIX getpeername() to find a peer for *f* in address family AF_INET or AF_INET6, peer_port sets last_error and returns om.

See also open, filename, port, peer_address, peer_name, and peer_sockaddr.

peer_sockaddr - peer address and port number

```
proc peer_sockaddr (stream f) : [string, integer]
```

If the stream *f* is a connected TCP or UDP socket, peer_sockaddr returns a 2-tuple [peer_address *f*, peer_port *f*]. It is permissible for *f* to be a file descriptor that is open only at the POSIX level.

On failure of the underlying POSIX `getpeername()` to find a peer for *f* in address family `AF_INET` or `AF_INET6`, `peer_sockaddr` sets `last_error` and returns `om`.

See also `accept`, `open`, `filename`, `sockaddr`, `peer_name`, `ip_addresses`, and `ip_names`.

pexists - test for existence of processes

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

Tests whether the process or set of processes identified by pid *p* exists, according to the same rules as for the *p* argument to `kill`. Unlike `kill`, however, `pexists` does not set `last_error`.

If *p* exists, `pexists` *p* may return `true` even if the caller has no permission to send a signal to *p*. Permission can be checked by calling `kill` with a signal number of 0 and then examining `last_error`.

Both of those operations give but a transient snapshot of system state.

See also `pid` and `getpid`.

The precedence of `pexists` is quite low, like that of other unary predicates (see [Operator Precedence], page 84).

pid - process ID of connected child

```
proc pid (stream) : integer
```

If the argument is a pipe, pump, or tty-pump stream connected to a child process, `pid` returns the child's POSIX process ID. No other stream types are allowed.

If the stream is of an acceptable type but came from an `open` call over a fd rather than from a child-creating call, -1 is returned.

See also `getpid`, `getppid`, `getpgrp`, `pipe_from_child`, `pipe_to_child`, `pump`, `tty_pump`, `pexists`, `kill`, and the `open` modes `"pipe-from"`, `"pipe-to"`, `"pump"`, and `"tty-pump"`.

pipe - create primitive pipe

```
proc pipe : [integer, integer]
```

This is a synonym for `socketpair`, except that the first fd of the returned pair may only be open for reading, and the second only for writing, as in POSIX `pipe()`.

pipe_from_child - pipe from child process

```
proc pipe_from_child : integer
```

The `pipe_from_child` intrinsic is a unidirectional form of `pump`.

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

See also `close`, `filter`, `system`, `pid`, `pipe_to_child`, and the `open` mode `"pipe-from"`.

pipe_to_child - pipe to child process

```
proc pipe_to_child : integer
```

The `pipe_to_child` intrinsic is a unidirectional form of `pump`.

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

See also `close`, `filter`, `system`, `pid`, `pipe_from_child` and the `open` mode `"pipe-to"`.

port - Internet port number

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

Local ("this side") port number of the TCP or UDP stream *f*. It is permissible for *f* to be a file descriptor that is open only at the POSIX level.

On failure of the underlying POSIX `getsockname()` to find an address for *f* in family `AF_INET` or `AF_INET6`, `port` sets `last_error` and returns `om`.

See also `open`, `filename`, `sockaddr`, `peer_port`, and `peer_sockaddr`.

pow - power set

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

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

See also `npow`.

pretty - printable ASCII rendering of string

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

If the operand is not already a `string`, the `pretty` operator first converts it to one as if by `str`. It then returns a copy of that string in which the 95 characters that ASCII considers *printable* are left unchanged, 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 non-printable characters, 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 characters are converted to `\`*ooo* form (backslash followd by 3 octal digits). For example,

```
print (pretty +/[char i : i in [0..255]]);
```

exhibits this printable encoding for all the characters in ASCII. It really is not all that pretty, but at least it won't do horrid things to your terminal.

See also `unpretty` and `unstr`.

print - print to stdout

```
proc print (var args(*))
```

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

See also `nprint` and `write`.

printa - print to stream

```
proc printa (stream f, var args(*))
```

The 0 or more *args* are written in sequence to the stream *f*, separated by single spaces. String arguments are written directly; all others are converted as if by str first. A newline character (\n) then follows.

If *f* is not already open, an attempt is made to auto-open it. See [Automatic opening], page 47.

On output error, output may be incomplete and last_error may be set.

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

pump - bidirectional stream to child process

```
proc pump : integer
```

The pump intrinsic creates a child process as if by fork, and returns in the parent a bidirectional stream that is connected to the child's standard input and output.

In the child process, pump returns -1, an unfortunate convention motivated by wanting an integer outside the range of all possible file descriptors.

Failure to create the child or the bidirectional stream, generally indicating resource exhaustion, is considered erroneous. (You need to call fork directly in order to detect and recover from spawning failure.)

The child in this kind of arrangement is sometimes called a *coprocess*. See [Connected subprocesses], page 44.

The call tie (stdin, stdout) can be useful in the child in the case of a straightforward message-and-response application-level protocol over the stream. See [Buffering], page 47.

The bidirectional channel between parent and child is opened over Unix-domain sockets created as if by socketpair, and therefore supports the passing of file descriptors using send_fd and recv_fd.

See also open (particularly the "pump", "tty-pump", "pipe-from", and "pipe-to" modes), tty_pump, pipe_from_child, pipe_to_child, pid, filter, system, flush, close, and shutdown.

put - write lines to stdout

```
proc put (string args(*))
```

Equivalent to puta (stdout, *args*(*)).

This signature for put follows that of SETL2, while puta is patterned after the old CIMS SETL **put**. This makes the signatures of put and puta consistent with those of print and printa.

puta - write lines to stream

```
proc puta (stream f, string args(*))
```

The 0 or more *args* are written in sequence to the stream *f*, with a newline character (\n) after each string.

If *f* is not already open, an attempt is made to auto-open it. See [Automatic opening], page 47.

On output error, output may be incomplete and `last_error` may be set.

A synonym for `puta` is `putline`.

See also `printa`.

`putb` - write values to stream

```
proc putb (stream f, var args(*))
```

The 0 or more *args* are written in sequence to the stream *f*, separated by single spaces and followed by a newline character (`\n`). All of them are converted as if by `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 auto-open it. See [Automatic opening], page 47.

On output error, output may be incomplete and `last_error` may be set.

A synonym for `putb` is `writea`.

See also `puta`.

`putc` - write characters to stream

```
proc putc (stream f, string s)
```

The 0 or more characters in *s* are written to the stream *f*.

If *f* is not already open, an attempt is made to auto-open it. See [Automatic opening], page 47.

On output error, output may be incomplete and `last_error` may be set.

See also `putfile`.

`putchar` - write characters to `stdout`

```
proc putchar (string s)
```

The call `putchar (s)` is the same as `putc (stdout, s)`.

`putfile` - write characters to stream

```
proc putfile (stream f, string s)
```

The 0 or more characters in *s* are written to the stream *f*.

If *f* is not already open, an attempt is made to auto-open it. See [Automatic opening], page 47.

The `putfile` intrinsic is almost equivalent to `putc`, except that if `putfile` auto-opens *f*, it also makes sure all output has been written as if by `flush` and then auto-closes *f*.

It is the only output intrinsic that auto-closes, making it a convenient one-stop way of writing a string to a file or network destination, e.g.:

```
putfile ('timestamp', date);  -- put date into timestamp file
```

On error, output may be incomplete and `last_error` may be set by POSIX `write()` and/or POSIX `close()`.

See also `getfile`.

`putline` - write lines to stream

```
proc putline (stream f, string args(*))
```

The 0 or more *`args`* are written in sequence to the stream *`f`*, with a newline character (`\n`) after each string.

If *`f`* is not already open, an attempt is made to auto-open it. See [Automatic opening], page 47.

On output error, output may be incomplete and `last_error` may be set.

A synonym for `putline` is `puta`.

See also `printa`.

`puts` - direct-access write

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

The file under the direct-access (seekable) stream *`f`* (`open` mode `"w"`, `"n"`, `"r+"`, `"w+"`, `"n+"`, or `"a+"`) is treated as a string, where *`start`* specifies the index (1 or higher) of the first character to write. In `"a+"` mode, *`start`* is ignored, and the writing will be at the end of the file.

The `puts` intrinsic writes *`n`* characters, increasing the size of the file as necessary. Writing to a position beyond the current end of the file is allowed, and the gap is filled with NUL characters (`\0`), possibly in a way that is optimized by the underlying file system. Again, this does not apply to `"a+"` mode, where *`start`* is ignored.

If *`f`* is not already open, an attempt is made to auto-open it in `"r+"` mode, which allows seeking, reading, and writing. See [Automatic opening], page 47.

As with `seek`, *`f`* is flushed of output (ignoring errors), and drained of input, before the repositioning of the file. See [Buffering], page 47.

On output error, output may be incomplete and `last_error` may be set.

See also `gets`, `putc`, `putfile`, and `mkstemp`.

`random` - 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 uniformly distributed pseudo-random integer in the range `0` through *`i`*. For *`i`* `<= 0`, the return value is in the range *`i`* through `0`.

The closed interval range in the integer case is a regrettable SETL idiosyncrasy: `random` *`i`* can return any one of *`i`*`+1` values, including *`i`* itself. It is perhaps best buried in a wrapper with a more conventional convention:

```
-- Uniformly chosen random number in [0..i-1]
op my_random (i);
  return random (i-1);
end op;
```

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]. Paradoxically, `random 0.0` is `0.0`.

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`.

There is a theoretical possibility that `random` applied to real *r* sets `last_error` to something like `"Numerical result out of range"`, but most extremely small values of *r* will map to `0.0` or `-0.0` depending on *r*'s sign, and most extremely large values to positive or negative floating-point infinity, without setting `last_error`. In a 64-bit IEEE 754 implementation, a magnitude of *r* between `1e-288` and `1e+288` will keep clear of these extremes.

See also `setrandom`, which sets the *random seed*.

range - range of map

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

The operand must be a set of ordered pairs, that is, a set of 2-tuples in which no element is `om`. The result is the set of all second members of those pairs.

See also `domain`.

rany - extract trailing character using character set

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

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

See also the other SNOBOL-inspired intrinsics, namely `any`, `break`, `len`, `match`, `notany`, `span`, `rbreak`, `rlen`, `rmatch`, `rnotany`, and `rspan`.

rbreak - extract trailing 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* (treating *p* as a set of characters), the substring of *s* after that character is removed from the tail of *s* and returned. If no character from *p* appears in *s*, the return value is the initial value of *s*, and *s* is reduced to the empty string (`""`).

See also the other SNOBOL-inspired intrinsics, namely `any`, `break`, `len`, `match`, `notany`, `span`, `rany`, `rlen`, `rmatch`, `rnotany`, and `rspan`.

rlen - extract trailing substring by length

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

A substring of length *n* min #*s* is removed from the tail of *s* and returned. It is an error for *n* to be less than 0.

See also the other SNOBOL-inspired intrinsics, namely any, break, len, match, notany, span, rany, rbreak, rmatch, rnotany, and rspan.

rmatch - extract trailing substring by exact match

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

If *p* is equal to the rightmost substring of *s*, it is removed from *s* and returned. Otherwise, nothing happens to *s* and the empty string ("") is returned.

See also the other SNOBOL-inspired intrinsics, namely any, break, len, match, notany, span, rany, rbreak, rlen, rnotany, and rspan.

rnotany - extract trailing character using character set

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

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

See also the other SNOBOL-inspired intrinsics, namely any, break, len, match, notany, span, rany, rbreak, rlen, rmatch, and rspan.

rspan - extract trailing substring using character set

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

The longest trailing substring of *s* consisting of characters that are in *p* (treating *p* as a set of characters) is removed from *s* and returned. If there is no such substring, nothing happens to *s*, and the empty string ("") is returned.

See also the other SNOBOL-inspired intrinsics, namely any, break, len, match, notany, span, rany, rbreak, rlen, rmatch, and rnotany.

read - read values from one or more lines of stdin

```
proc read (wr var args(*))
```

Equivalent to reada (stdin, *args*(*)).

reada - read values from one or more lines of stream

```
proc reada (stream f, wr var args(*))
```

Zero or more values are read from the stream *f* and assigned to the succeeding *args* in order. If an end of input (end of file or error) is reached before all those arguments have been assigned to, trailing arguments are set to om. Otherwise, characters through the next newline (\n)) if any are read and discarded. If the end of input is reached before any arguments have been assigned to, the eof indicators are set.

If *f* is not already open, an attempt is made to auto-open it. See [Automatic opening], page 47.

Values written by `writea`, except for atoms (see `newat`) and procedure references (see `routine`), are readable by `reada`. Tokens denoting values are separated by whitespace (ERE `"[ \f\n\r\t\v]+"`) and converted as if by `unstr`. Since newline is a whitespace character, input values may be distributed over more than one line.

There is a difference between `reada` and `getb` in that after reading the requested number of values, `reada` continues reading characters until it either absorbs a newline (`\n`) or encounters an end of input, whereas `getb` stops right after the end of the last value read. Thus `reada (f)` is a way of reading a line from *f* but ignoring its content.

The rules on auto-flushing *f*'s output associations, on setting `last_error`, and on auto-closing are as for `getc`.

See also `read`, `reads`, `geta`, `getline`, `getfile`, `getn`, `printa`, `val`, and `unpretty`.

`readlink` - symbolic link referent

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

When *f* names a file that is a symbolic link (*symlink*), `readlink` returns the string associated with *f*, i.e., the symlink. The 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 symlink to a file that doesn't exist.

If *f* is not a symlink, `readlink` returns `om` and sets `last_error` according to the rules for POSIX `readlink()`.

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

`reads` - read values from a string

```
proc reads (stream s, wr var args(*))
```

Zero or more values are "read" from the string *s* and assigned to the succeeding *args* in order. If the end of the string is reached before all those arguments have been assigned to, trailing arguments get `om`. The rules for value recognition and conversion are the same as for `reada` and `unstr`.

See also `val`.

`recv` - receive datagram on UDP client socket

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

A datagram is read from the socket *f* and returned as a string. The stream *f* must be of `open` mode `"udp-client"`—`recv` is not supported for `"unix-datagram-client"` streams, which can only `send`.

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

If the underlying POSIX `recv()` fails, `recv` sets `last_error` and returns `om`.

The `eof` indicators are not set by `recv`, not even when an empty datagram (`""`) is received.

If *f* is not already open at the SETL level, an attempt is made to auto-open it in `"udp-client"` mode. See [Automatic opening], page 47.

See also `recvfrom`, `sendto`, and `sockaddr`.

recvfrom - receive datagram on server socket

```
op recvfrom (stream f) : [[string, integer], string]
op recvfrom (stream f) : [string, string]
```

A datagram is read from the socket *f*, and information about its source is returned along with it.

If *f* is of open mode "udp-server", the sender's Internet address and port number are sensed, and bundled together with the datagram as the return value [[*address*, *portnum*], *datagram*], where *address* is a string in IPv4 or IPv6 notation, *portnum* is an integer, and *datagram* is a string.

If *f* is of mode "unix-datagram-server", the return value is [*pathname*, *datagram*], where *pathname* is the socket name of the sender if it has one, otherwise the empty string (""). (Unix-domain clients, mode "unix-datagram-client", do not bind names, so the *pathname* is only useful when receiving from other Unix-domain servers.)

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

If the underlying POSIX recvfrom() fails, recvfrom sets last_error and returns om.

The eof indicators are not set by recvfrom, not even when an empty datagram ("") is received.

If *f* is not already open at the SETL level, an attempt is made to auto-open it in "udp-server" mode. See [Automatic opening], page 47.

See also sendto, recv, send, and sockaddr.

recv_fd - receive file descriptor

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

A file descriptor (fd) sent by a process executing a send_fd is returned and left open at the system level. It is not immediately opened at the SETL level, but can be opened as a stream by open like any other fd, or auto-opened (see [Automatic opening], page 47).

The stream *f* should be a Unix-domain socket, such as is created by pipe_from_child, pipe_to_child, or pump; or by open mode "pipe-from", "pipe-to", "pump", "unix-client", or "unix-server". Or by socketpair, as *f* is itself allowed to be a fd not open at the SETL level.

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

The integer value of the returned fd is chosen by recv_fd as the next available integer, in the manner of dup, and refers to the same POSIX-level *open file description* object as the fd that the other process passed to send_fd does, even though it is numerically independent.

If recv_fd fails to receive a fd, it sets the eof indicators and returns om. An error in the underlying POSIX recvmsg() call is reflected in last_error.

No attempt to auto-open *f* is made by recv_fd, but it does follow the getc rules on initial flushing of output associations and on auto-close upon eof.

rem - 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). Rearrangement of terms in the above definition of rem allows *n* to be rather directly reconstructed from its div and rem given the divisor *d*.

rename - rename file

```
proc rename (string old, string new)
```

Changes the name of a file if possible. On failure of the underlying POSIX rename() call, last_error is set.

See also link, symlink, unlink, and system.

reverse - reverse string

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

Characters in reverse order.

rewind - rewind direct-access stream

```
proc rewind (stream f)
```

Equivalent to seek (*f*, 0).

round - round to nearest integer

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

Numbers ending in .5 are rounded away from zero. All others are rounded to the nearest integer.

For example, round -5.5 = -6, and round -5.4 = -5.

Floating-point infinities and NaN (*Not a Number*) values give om.

See also floor, ceil, fix, and float.

routine - 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 it returns a handle which you can later pass to `call`.

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

rpad - pad string on right with blanks

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

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

It is an error for *n* to be less than 0.

See also `lpad`, which *right-justifies* by padding on the left.

seek - reposition direct-access stream

```
proc seek (stream f, integer offset) : integer
proc seek (stream f, integer offset, integer whence) : integer
```

The file under the direct-access (seekable) stream *f* (`open` mode `"r"`, `"w"`, `"n"`, `"r+"`, `"w+"`, `"n+"`, or `"a+"`) is repositioned so that the next sequential read or write operation will start at *offset* bytes relative to the point indicated by *whence* (except that for `"a+"`, writes are always at the end of the file). The possibilities for *whence* are

- `seek_set` (the default), meaning an absolute offset from the beginning of the file;
- `seek_cur`, meaning relative to the current read/write position;
- `seek_end`, meaning from the end of the file.

The *offset* convention has the beginning of the file at position 0, consistent with the conventions of POSIX `lseek()`. (By contrast, the *start* argument of `gets` and `puts` follows the convention of SETL strings, where the first character has index 1.)

If *f* is not already open, an attempt is made to auto-open it in `"r+"` mode, which allows seeking, reading, and writing. See [Automatic opening], page 47.

The return value is the new read/write position (offset from beginning of file).

Note that `seek` does *not* affect `eof`. Seeking beyond the end of the file and writing is allowed, as it is for `puts`, and the gap is filled with NUL characters (`\0`), possibly in a way that is optimized by the underlying file system.

The stream *f* is flushed of output (ignoring output errors) and drained of input, before any seeking attempt, even one that does not change the current position. See [Buffering], page 47. Consider using `filepos` if you just want the current position without the flushing and draining.

See also `rewind` and `mkstemp`.

seek_set, seek_cur, seek_end - constants for use with seek

```
seek_set : integer
seek_cur : integer
seek_end : integer
```

See seek.

select - wait for event or timeout

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

Waits for I/O events on sets of streams, with optional timeout (0 to poll).

Because interprocess communication, signals, interval timers, and sockets are all wrapped as I/O streams in SETL, and SETL is resolutely single-threaded, `select` is the fundamental intrinsic for event-driven programming in SETL.

The *stream_sets* argument gives up to 3 sets of streams:

- *stream_sets*(1) - streams that may produce input. The meaning of this is actually extended to include TCP and Unix-domain server sockets that are ready to accept without blocking, UDP client sockets that have datagrams ready for receipt by recv, UDP or Unix-domain server sockets ready for a recvfrom, and Unix-domain sockets on which recv_fd can be called without blocking.
- *stream_sets*(2) - streams that take output, including UDP and Unix-domain datagram sockets ready for send or sendto operations, and Unix-domain sockets ready for send_fd.
- *stream_sets*(3) - streams that can indicate exceptional conditions. No such conditions are currently defined in SETL.

An empty *stream_sets* tuple can be given as [] or om.

The *ms* argument, if present, specifies how many milliseconds `select` should wait until a stream in *stream_sets* becomes ready. If *ms* is 0, the streams are tested without waiting. If *ms* is absent, `select` waits indefinitely.

The return value from `select` is a 3-tuple of stream sets having input, output, and exceptional readiness respectively. In the case of a timeout, all 3 sets will be empty.

Historically, there could be any number of streams in each of these return sets, and some of the consequences of that rather direct interface to the underlying POSIX `select()` or `pselect()` are discussed in Section 5.3.8.3, "The Event Loop", of SETL for Internet Data Processing (https://cs.nyu.edu/media/publications/bacon_david.pdf), p. 163 ff. In brief, an unprocessed stream in a ready set can be closed during the processing of an event associated with another stream, and then reopened in another way before being processed on that burst of events as if it were the original stream. POSIX promotes fd reuse by picking the lowest unused number, making such aliasing quite possible unless the application takes special care to prevent it.

To remove that subtle and unnecessary hazard, `select` now returns at most one stream. So at least 2 of the 3 ready sets will be empty, and the other at most a singleton. Implementations are encouraged to cache results from an underlying POSIX `pselect()` call, doling a stream out from the cache in preference to calling `pselect()`, and removing a stream from the

cache when it is closed. But an implementation could instead skip that optimization and simply pick one stream from what `pselect()` gives, ignoring the rest in the faith that they will still be there on the next `pselect()` call if they should be.

Streams in the input and output sets in the *`stream_sets`* argument to `select` are subject to auto-opening (see [Automatic opening], page 47).

Also, appearance in the input set causes auto-flushing of a stream's output associations in the manner of `getc`.

Since a *`stream_sets`* argument of `om` means no streams, the statement

```
select (om, ms);
```

is pretty much a standard way to delay program execution for *`ms`* milliseconds, and

```
select (om);
```

suspends the program until it is killed by external forces.

Mostly, `select` is used to check and wait for the availability of input, not for the possibility of output. Even when `select` tells you a stream is ready to take output, you do not in general know how much. Moreover, it's the pipe that is ready, not necessarily the receiver. Such notifications could conceivably be of use in some high-performance multiplexing application, but if output blockage is a concern in any given instance, it is usually best to interpose a child process that can afford to block on its own output, and notifies the parent when it is ready for more. The parent-child communication goes at local speed.

For a SETL stream, which is buffered, "ready for output" means that a POSIX `write()` of some number of bytes will not block. It does not merely mean that there is room in the SETL buffer, which is always true, but rather that the stream is *flushable* without blocking (unless the buffer is too full for what the pipe can currently accommodate). It also means that `sys_write` can be called on the stream without blocking, provided the request is not too big. It is seldom necessary to test for output readiness. For datagram sockets, it is never even useful.

Using `routine`, a *callback* scheme for input events can be realized with `select`:

- Global map from stream handle to unary callback routine:

  ```
  var callbacks := {};
  ```

- Register callback procedure *`cb`* to be called when stream *`f`* has input available:

  ```
  callbacks(f) := routine cb;
  ```

- Unregister stream *`f`*:

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

- Input event dispatcher loop:

  ```
  loop  -- indefinitely
    [ready] := select ([domain callbacks]);
    for f in ready loop  -- at most 1 iteration nowadays
      call (callbacks(f), f);  -- f is map key and callback arg
    end loop;
  end loop;
  ```

- A callback *`cb`* for input available on stream *`f`*:

```
proc cb (f);
  reada (f, cmd);  -- read a request from f
  -- act on it ...
end cb;
```

send - send datagram on client socket

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

The string *datagram* is sent via the client socket *f* (open mode "udp-client" or "unix-datagram-client"). Datagram output is not buffered (see [Buffering], page 47).

Failure of the underlying POSIX send() is reflected in last_error.

If *f* is not already open at the SETL level, an attempt is made to auto-open it in "udp-client" mode. See [Automatic opening], page 47.

See also sendto, recv, recvfrom, and sockaddr.

sendto - send datagram on server socket

```
proc sendto (stream f, tuple dest, string datagram)
proc sendto (stream f, string pathname, string datagram)
```

The *datagram* is sent via the server socket *f* to the destination *dest* or *pathname*.

If *f* is of open mode "udp-server", *dest* must be a 2-tuple [*host*, *port*], where *host* is a string containing an Internet host name or an address in IPv4 or IPv6 notation, and *port* is an integer client port number or string port name.

If *host* is om or the empty string (""), it is taken to mean a loopback interface such as "127.0.0.1" or "::1".

If POSIX getaddrinfo() fails on the given *dest*, or all POSIX sendto() attempts on the Internet address(es) found for it fail, last_error is set according to the last of those failures.

If *f* is of open mode "unix-datagram-server", *pathname* is the Unix-domain socket of another server. Errors in POSIX sendto() are reflected in last_error.

If *f* is not already open at the SETL level, an attempt is made to auto-open it in "udp-server" mode. See [Automatic opening], page 47.

See the open mode "udp-server", and see also recvfrom, send, recv, and select.

send_fd - send file descriptor

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

The stream *f* is first flushed as if by flush (but without setting last_error), and then the file descriptor *fd* (which may or may not be open at the SETL level) is sent via *f* to a process that is expected to call recv_fd.

The stream *f* should be a non-datagram Unix-domain socket, such as is created by pipe_from_child, pipe_to_child, or pump; or by open mode "pipe-from", "pipe-to", "pump", "unix-client", or "unix-server". Or by socketpair, as *f* is itself allowed to be a fd not open at the SETL level.

The select function can be used to check or wait for *f* to be ready to take a file descriptor without blocking. This will never be necessary in practice, though the POSIX sendmsg() specification makes it theoretically possible that send_fd could block on that call.

The more significant possibility is that `send_fd` blocks while waiting for the receiver to acknowledge receipt of the fd. There are POSIX platforms such as QNX where the sender cannot afford to close the fd before the receiver is known to have received it, or the *open file description* may have been destroyed in the meantime. Although this responsibility could have been left to applications to deal with, the design decision was made to insist that the SETL implementation take care of it. This imposes a different and perfectly normal responsibility on applications, namely to ensure that the blocking by `send_fd` during this rendezvous, while it awaits confirmation by the peer `recv_fd`, is kept brief or benign as appropriate.

If the POSIX `sendmsg()` underlying `send_fd` fails, `last_error` is set.

`setctty` - acquire controlling terminal

```
proc setctty (stream f)
```

The terminal or pseudo-terminal connected to stream `f` is made (in POSIX terms) the *controlling terminal* of the calling process if it is a session leader (see `setsid`) and does not already have a controlling terminal. This is used in job control.

POSIX does not define a standard way of acquiring a controlling terminal, but the BSD `ioctl()` TIOCSCTTY is widely supported. For older SystemV-based Unix systems that do not have this capability but do support the notion of a controlling terminal, the SETL implementation is expected to fall back to acquiring it when a session leader without one opens a terminal device (real or virtual). Conversely, if a mechanism like TIOCSCTTY is supported by the host system, the SETL implementation is expected to use that as the only way of setting the controlling terminal (specifically, by including the O_NOCTTY flag on all POSIX tty `open()` and `posix_openpt()` calls so that the controlling terminal isn't automatically assigned).

When the controlling terminal is acquired, the foreground process group (see `tcgetpgrp` and `tcsetpgrp`) is set to be that of the session leader, which becomes the *controlling process* for that terminal.

Failure of `setctty`, such as when the calling process is not a session leader, is reflected in `last_error`. If the host system does not have a mechanism like TIOCSCTTY (or like the `tcsetsid()` that appears in FreeBSD and QNX), `last_error` corresponds to a POSIX `errno` value of ENOSYS.

See also `unsetctty`, `getpgrp`, `setpgid`, and `tty_pump`.

`setegid` - set effective group ID

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

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

Sets `last_error` on failure.

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

setenv - 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 passing the empty string (`""`).

Note that `setenv` is an interface to the POSIX `setenv()`, and uses some system memory, so should be used sparingly.

See also `getenv` and `unsetenv`.

seteuid - set effective user ID

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

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

Sets `last_error` on failure.

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

setgid - set group ID

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

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

Sets `last_error` on failure.

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

setpgid - set process group ID

```
proc setpgid (integer p, integer pg)
```

Sets the process group ID of the process with process ID *p* to *pg*, using POSIX `setpgid()`. If *p* and *pg* match, a new process group is created within the session. Otherwise, *pg* must identify an existing process group within the session. The pid *p* must be that of the caller or of a direct child that has not yet called `exec`.

The process group ID of a session leader (see `setsid`) cannot be changed.

If *p* is 0, `getpid` is substituted. If *pg* is 0, it is taken to be *p* (after substitution).

On failure, `last_error` is set.

The main intended role of this intrinsic is in job control. When a new process is spawned, it needs to be put into the right process group. In order to ensure that this happens before either the parent starts sending signals intended for the process group or the child does an `exec`, both the parent and the child should call `setpgid` on the child's pid, putting it in the desired (new or existing) process group right after the spawning, so that it does not matter which attempt succeeds first.

See also `getpgrp`, `setctty`, `unsetctty`, `tcgetpgrp`, and `tcsetpgrp`.

setrandom - set random seed

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

Establishes a starting point for pseudo-random number generation. If *seed* is 0, the starting point is as unpredictable as the SETL implementation can make it with reasonable effort, given available sources of entropy such as /dev/urandom.

See also random.

setsid - create new session

```
proc setsid
```

Sessions are fundamental to job control in POSIX, as each job is a process group, and a session is a set of process groups.

If the caller is not a process group leader, that is, its process ID is not the process group ID of any process, setsid establishes a new session: the session ID and process group ID are set to the process ID of the caller, thus making the caller the new process group leader and the new session leader. After a successful call to setsid, there is no controlling terminal (but see setctty).

On failure (such as when the caller is already a process group leader), last_error is set.

See also getsid, getpgrp, setpgid, tcgetpgrp, tcsetpgrp, and unsetctty.

setuid - set user ID

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

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

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

If a user (called the *real* user) other than the owner of such a file runs it, the *effective* user ID and *saved* user ID are initially defined to be those of the file's owner, rather than the real user ID. The program runs 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 "superuser", setuid sets all 3 IDs and is in that respect like a one-way door.)

For example, suppose you are a DYI 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 setuid, however, it is easy to set up what you want: simply have your submit program, when run by a student, use the student's privileges to fetch the student's files, and your privileges to store them in a private area of yours.

The situation for group IDs and *setgid executables* is exactly analogous to that for user IDs.

If setuid fails, last_error is set.

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

set_intslash - muck with integer division semantics

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

A call to `set_intslash` returns the current value of `intslash` and sets `intslash` to the value given by the `boolean` argument.

set_magic - regular expression recognition

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

A call to `set_magic` returns the current value of `magic` and sets it to the value given by the `boolean` argument.

shutdown - disable I/O in one or both directions

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

If *f* is a TCP or Unix-domain socket stream, `shutdown` disables the 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()`.

If *how* is `shut_wr` or `shut_rdwr`, the stream is first flushed as if by `flush`, but without reflecting output errors in `last_error`.

With a `shut_wr` or `shut_rd` argument, `shutdown` performs a *half-close*. For example, `shut_wr` can be used to tell a peer that you have finished sending data (thus making the peer see an end-of-file condition) but that you would still like to receive a reply on the same (still *half-open*) connection.

Unlike `close`, which does nothing to the communications channel unless the stream's fd is the last reference to the underlying file description, `shutdown` actually closes the direction(s) of communication that it is asked to.

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

Failure of the underlying POSIX `shutdown()` is reflected in `last_error`.

See also `open` and `pump`.

shut_rd, shut_wr, shut_rdwr - constants for use with shutdown

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

See `shutdown`.

sign - 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.

sin - sine

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

The operand is in radians. See also asin.

sinh - hyperbolic sine

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

Engorged nasal passage.

sockaddr - Internet address and port number

```
proc sockaddr (stream f): [string, integer]
```

If stream *f* is a TCP or UDP socket, sockaddr returns a 2-tuple whose first member is its local Internet address, in either IPv4 dotted or IPv6 colon hexadecimal notation, and whose second member is the TCP or UDP port number. It is permissible for *f* to be a file descriptor that is open only at the POSIX level. In the case of a listening TCP server socket or a UDP server socket, the address may indicate *any available* interface, e.g., "0.0.0.0" or "::".

On failure of the underlying POSIX getsockname() to find an address for *f* in family AF_INET or AF_INET6, sockaddr sets last_error and returns om.

See also open, filename, port, hostaddr, ip_addresses, ip_names, peer_name, and peer_sockaddr.

socketpair - create bidirectional local channel

```
proc socketpair : [integer, integer]
```

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

It returns the file descriptors of a pair of connected, unnamed Unix-domain sockets such that output to each socket is presented as input to the other.

This is chiefly of use when you are (perhaps as a classroom or prototyping exercise) managing processes at the level of fork, dup2, close, exec, and waitpid. Otherwise, one of the higher-level intrinsics {filter, system, pipe_from_child, pipe_to_child, pump, tty_pump} or open modes {"pipe-from", "pipe-to", "pump", "tty-pump"} might cover your needs more conveniently.

That said, there may be times when you want an extra channel between the program and a child process, or between offspring; and socketpair creates such a channel before the spawning. Each process would then keep one fd and close the other, possibly opening a SETL stream over the kept fd in mode "r", "w", or "rw", either explicitly with open or (for "r" or "w") implicitly (see [Automatic opening], page 47).

Being a Unix-domain socket, each fd in the pair returned by socketpair supports the passing of other file descriptors with send_fd and recv_fd.

On failure, socketpair sets last_error and returns om.

See also pipe, which can (but need not be) used for unidirectional use cases.

span - extract leading substring using character set

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

The longest initial substring of *s* consisting of characters that are in *p* (treating *p* as a set of characters) is removed from *s* and returned. If there is no such substring, nothing happens to *s*, and the empty string (`""`) is returned.

See also the other SNOBOL-inspired intrinsics, namely any, break, len, match, notany, rany, rbreak, rlen, rmatch, rnotany, and rspan.

split - split string into tuple

```
proc split (string s, pattern p) : [string, ...]
proc split (string s) : [string, ...]
```

Substrings of *s* are returned as a tuple, where the extended regular expression (ERE) *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 (":ab::c", ":")` is `["", "ab", "", "c"]` while `split (":ab::c", ":+")` is `["ab", "c"]`.

The pattern *p* is subject to the setting of magic, and can be a string or a 2-tuple, as detailed under mark. When *p* is omitted, the whitespace ERE is used regardless of the setting of magic.

Splitting the empty string yields the empty tuple; i.e., `split ("", p) = []` for any pattern *p*;

See also join and gsub.

sqrt - square root

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

Where squares come from.

status - child process status

```
status : integer
```

Status of the child process that was last successfully waited for by filter, system, or waitpid; or by close as applied to a pipe, pump, or tty-pump stream in close_await mode (the default).

The initial value of `status`, when no child status has yet been reaped, is om. It is also set to om when any of the above calls is unsuccessful, or when waitpid returns 0.

If a successfully waited-for child made a normal exit, e.g. via POSIX `exit()` or `_exit()`, return from C `main()` program, or SETL `stop` statement, `status` is the exit status given by the `exit()` or `_exit()` argument, `main()` return value, or `stop` operand, modulo 256. In POSIX conventions, an exit status of 0 indicates *success*.

If the child was terminated by signal *sig* (see kill), `status` is `-128 + sig`. This allows the SETL program to distinguish normal exits from termination by signal, but also dovetails

with shell practice (which uses 128 plus the signal number). For example, given a SIGINT value of 2, the `status` of a waited-for subprocess killed by SIGINT is -126 (-128 + 2), which if used in turn as the operand of a `stop` statement gives an exit status of 130 when taken modulo 256, just as for a program the shell terminates with SIGINT, e.g. on a ctrl-C from the keyboard.

Additionally, if the child is waited for by `waitpid` (rather than by `close`, `filter`, or `system`), *stopped* and *continued* events are reflected in a `status` of `-32` for a continued (resumed) child, and `-32 + stopsig` for a stopped (suspended) child, where `stopsig` is the number of the SIGSTOP, SIGTSTP, SIGTTIN, or SIGTTOU signal that caused the stoppage. These 4 signal numbers lie within the range 17 to 28 on all known POSIX implementations, putting the corresponding `status` values in an otherwise unused range of small negative integers.

See also `pipe_from_child`, `pipe_to_child`, `pump`, `tty_pump`, and the `open` modes `"pipe-from"`, `"pipe-to"`, `"pump"`, and `"tty-pump"`.

stdin, stdout, stderr - predefined streams

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

The constant fd `stdin` is `0`, `stdout` is `1`, and `stderr` is `2`. See [Predefined streams], page 46.

str - string representation of value

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

The operand *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 `real` *x*, and the decimal point may be omitted if the fraction in *x* is 0. These hazards can be offset by the use of `fixed` or `floating` in place of `str`. Bizarre forms such as `"nan"` and `"-inf"` are also possible in extreme cases.

Procedure references (see `routine`) are converted to `"<ROUTINE>"`, and `atom`s are converted to strings of the form `"#`*digits*`"`. Those two forms cannot be converted back to values with `unstr`.

The value `om` renders as an asterisk (`*`).

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 *x* *does* have the form of a SETL identifier, `str` *x* `=` `unstr` *x* `=` *x*.

For a `set` *x*, `str` *x* begins with a left brace (`{`) and ends with a right one (`}`). Likewise brackets (`[`,`]`) surround a `tuple`. Sets and tuples nest to arbitrary degree, and the converted values within them are separated by single spaces.

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

strad - 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 prefix in the result denotation "*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`. The string is prefixed with a hyphen if *x* is negative. 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`.

`sub` - replace pattern in string

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

The leftmost occurrence in *s* of the pattern *p* is replaced by *r*, which defaults to the empty string (`""`). The matched substring of *s* replaced by this operation is returned. If *p* is not found, *s* is left unmodified and `om` is returned.

The pattern *p* is subject to the setting of `magic`, and can be a string or a 2-tuple, as detailed under `mark`.

In the replacement pattern *r*, when `magic` is `true`, the ampersand character (`&`) is special, as is a backslash (\) followed by a decimal digit. The ampersand refers to the entire matched substring, and that is also the meaning of a backslash followed by zero (`0`). Backslash followed by another digit *k* refers to the substring matching the *k*th parenthesized subexpression in the pattern *p*. When *p* is a tuple `[`*p1*`, `*p2*`]`, parenthesized subexpressions are counted starting in *p1* and continuing in *p2*.

If `magic` is set to `false`, *r* is taken literally as the string to replace the matched substring, with no ampersand or backslash interpretation.

Notice that for string *p*, the expression

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

is similar to

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

except that the value of the latter is not simply the substring that was replaced as it is in the `sub` case, but rather the new value of *s*`(`*p*`)`—a rematch!

If *p* is the tuple `[`*p1*`, `*p2*`]`, the corresponding assignment is:

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

See also `gsub` and `gmark`.

`subset` - subset test

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

Returns `true` if every member of *ss* is also in *s*.

Being a predicate, it has quite a low precedence. See [Operator Precedence], page 84.

See also `incs`.

symlink - create symbolic link

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

Creates a symbolic link (*soft link*) to *s* under the filename *new* using POSIX `symlink()`, if *new* does not exist before the call.

For symbolic links, unlike *hard* links (see `link`), it does not matter whether *s* refers to an existing file in order for the `symlink` call to succeed. However, if *s* does exist as a file and the `symlink` call does succeed, subsequent attempts to access the contents of the file *new* will be redirected to the file *s*.

If `symlink` fails, `last_error` is set.

Given a stable directory structure above *new*, `symlink` behaves atomically, and can be used as a test-and-set mechanism for inter-process synchronization: the lock (mutex) *new* is acquired if and when `symlink` succeeds, and releasing it is done atomically by `unlink`.

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

system - run command in subshell

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

The *cmd* argument specifies a shell command, which is performed as if by `exec ("/bin/sh", ["sh", "-c", cmd])` in a child process spawned as if by `fork`.

Just as with `filter`, the signals SIGINT and SIGQUIT are temporarily ignored in the parent while it waits for the child to complete. Thus a terminal-generated signal (typically ctrl-C for SIGINT and ctrl-\ for SIGQUIT) that goes to the process group will be seen by the child but not the parent, which thus remains to handle the child termination.

SIGCHLD is not blocked during the `system` call, as it would be during a POSIX `system()` call. The SETL implementation is expected to do all handling of SIGCHLD, and has to be free to reap child processes detached via `close_autoreap` even while it is waiting for the *cmd* child to complete. Effectively, this means that it has to be in no danger of competing with a user-supplied handler that intercepts SIGCHLD. Since that signal, like others, is reflected up to the SETL program when a `"signal"` stream over it is open, there is no need in a SETL implementation, and really no room in the semantics of SETL, for unrestricted asynchronous signal handlers.

The parent uses POSIX `waitpid()` to get the exit status of the command, which is placed in `status` and returned by `system`. If `waitpid()` fails, `status` is set to `om` and the reason for the failure appears in `last_error`, just as in the similar scenario of `close` with the `close_await` parameter.

See also `pipe_from_child`, `pipe_to_child`, `pump`, `tty_pump`, `socketpair`, `pipe`, `dup`, `dup2`, and `waitpid`; and the `open` modes `"pipe-from"`, `"pipe-to"`, `"pump"`, and `"tty-pump"`.

sys_read - 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 [Buffering], page 47).

Up to *n* bytes are read from the *fd* and returned as a string.

End of file is *not* indicated by a change in `eof`, only by an empty string result (`""`).

If an error occurs at the POSIX `read()` level, `sys_read` sets `last_error` and returns `om`.

See also `sys_write`.

`sys_write` - 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 [Buffering], page 47).

An attempt is made to write all `#`*`s`* bytes of *`s`*. The actual number of bytes written is returned, unless there is an error at the POSIX `write()` level, in which case -1 is returned and `last_error` gives details.

See also `sys_read`.

`tan` - trigonometric tangent

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

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

`tanh` - hyperbolic tangent

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

An extreme diversion.

`tcgetpgrp` - get foreground process group ID

```
proc tcgetpgrp (stream f) : integer
```

Uses POSIX `tcgetpgrp` to get the process group ID of the foreground process group for which the terminal given by stream *`f`* is the caller's controlling terminal. If there is no foreground process group, the return value will be greater than 1 and not match the ID of any existing process group.

On failure, `last_error` is set and `om` is returned.

See also `tcsetpgrp`.

`tcsetpgrp` - put process group into foreground

```
proc tcsetpgrp (stream f, integer p)
```

For use in job control, `tcsetpgrp` uses POSIX `tcsetpgrp` to set the foreground process group ID associated with the controlling terminal (see `setctty`) given by stream *`f`* to *`p`*, which must match the process group ID of a process in the same session as the calling process. The terminal driver sends keyboard-generated signals (SIGINT and SIGQUIT) to the foreground process group (job).

If `tcsetpgrp` is called by a member of a background process group in its session, and the caller is not blocking or ignoring SIGTTOU, a SIGTTOU signal is sent to the background

process group (as it is for attempted output to the controlling terminal by a background process when POSIX '`stty tostop`' is in effect). If SIGTTOU is being blocked or ignored, the `tcsetpgrp` operation is allowed. Interactive session leaders use it to assign foreground status to one job or another.

On failure, `tcsetpgrp` sets `last_error`.

See also `tcgetpgrp`, `getpgrp`, `setpgid`, and `tty_pump`.

`tie` - auto-flush stream upon input from other stream

```
proc tie (stream, stream)
```

After a call to `tie`, whenever an input operation such as `reada` or `geta` is requested on one of the two given streams, all output buffered for the other is written out as if by `flush` first but without setting `last_error`. Flushing a read-only stream, or a stream with no pending output, is a no-op.

There are a few other operations besides attempted sequential stream input that trigger the auto-flushing of a tied stream. See [Buffering], page 47.

A stream can only be tied to one other stream at a time. Previous associations are dissolved by `tie` when it forms a new one, and by `untie`.

`time` - 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, in milliseconds. This includes both *user* time and time spent by the operating system on behalf of the user.

See also `clock`, `tod`, `close` (on streams connected to child processes), `filter`, `system`, and `waitpid`.

`to_lower` - convert string to lowercase

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

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

See also `to_upper`.

`to_upper` - convert string to uppercase

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

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

See also `to_lower`.

tod - 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.

See also clock, time, date, and fdate.

true - predefined boolean value

```
true : boolean
```

Information is not knowledge.
Knowledge is not wisdom.
Wisdom is not truth.
Truth is not beauty.
Beauty is not love.
Love is not music.
Music is the best.

– *Frank Zappa, 1979*

See also false.

tty_pump - master end of child stream over pseudo-terminal

This is the same as pump, but instead of a socketpair, the child's standard input and output are connected to the slave side of a pseudo-terminal (pty) in *raw* mode; `tty_pump` returns the fd of the master side.

The open mode `"tty-pump"` is like `tty_pump` but spawns an external program rather than a local coprocess.

See [Connected subprocesses], page 44.

type - 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` *type-testing* operators, and denotype.

umask - set file mode creation 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 file mode creation 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.

In restricted mode (see [Restricted Mode], page 85), `bit_or` is applied between the mask argument and the mask with which the SETL program began. This allows the environment to prevent a restricted program from creating files with excessive permissions. For example, starting the restricted program with a umask of octal 137 would make sure files were created with no execute access by the owner, at most read access by the group, and no access by the rest of the world.

ungetc - push characters back into stream

```
proc ungetc (stream f, string s)
```

The 0 or more characters of *s* are *pushed back* into the stream *f*, so that they will appear as the next input. The SETL implementation may insist that *s* match the `#s` characters that were most recently read from the stream.

At least one character of pushback is allowed if at least one character has been read since the input buffer was last empty, such as after it was last drained (see [Buffering], page 47).

A successful `ungetc` does *not* clear `eof`, but the next input attempt on *f* will begin by doing so. Pending `eof` and error indicators remain pending, and any characters pushed back can be read again before `eof` and possibly `last_error` are actually set.

See also `ungetchar`, `peekc`, and `peekchar`.

ungetchar - push characters back into stdin

```
proc ungetchar (string s)
```

Equivalent to `ungetc (stdin, s)`.

unhex - convert from hexadecimal

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

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

unlink - destroy file reference

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

Remove the pathname *f* from the filesystem, using POSIX `unlink()`. If *f* is a symbolic link, the pathname is removed. Otherwise, if *f* is the last link to the object in the filesystem, and the last of any file descriptors open on it have closed, the space if any occupied by the object is freed.

If `unlink` fails, `last_error` is set.

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

unpretty - convert string from *pretty* form

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

The string *s* should be in *pretty* form, though 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*, which includes blank. The `unpretty` operator makes the following interpretations in transforming *s* into an unrestricted string, where backslash (\) is the *escape* character that introduces a special sequence:

- Escape followed by any of the 33 non-alphanumeric printable characters, including backslash, means just that character.
- Escape followed by up to 3 octal digits starting with the digit 0, 1, 2, or 3 means a character having the bit pattern indicated by the digits, as in C.
- Escape followed by `x` and then 1 or 2 hexadecimal digits is an alternative to the octal escape sequence. Thus `"\xdB"` means the same as `"\333"` to `unpretty`.
- Escape followed by these letters means what it does in C:

letter	equivalent	ASCII meaning
a	x07	alert
b	x08	backspace
f	x0c	formfeed
n	x0a	newline
r	x0d	carriage return
t	x09	horizontal tab
v	x0b	vertical tab

- Escape followed by anything else is invalid.

Other characters represent themselves, except that if the string begins with an apostrophe, an internal apostrophe may be indicated by a pair of successive apostrophes instead of `\'`. Likewise, if the string begins with a double quote, an internal double quote may be indicated by a pair of successive double quotes rather than `\"`.

Those are also the rules and interpretations for literal strings in SETL source code.

For any string *s*, `unpretty pretty s = s`.

See also `unstr` and `str`.

unsetctty - relinquish controlling terminal

```
proc unsetctty (stream f)
```

If the terminal or pseudo-terminal connected to stream *f* is the controlling terminal for the calling process (see `setctty`), it is given up, provided that a mechanism at the host system level such as the BSD-rooted `ioctl()` TIOCNOTTY is available. On some systems, such as Linux, this will also cause signals SIGHUP and SIGCONT to be sent to the foreground process group if the caller is session leader.

Failure of `unsetctty` is reflected in `last_error`.

unsetenv - 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 and getenv.

unstr - read value from string

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

The unstr operator converts any string produced by str (a *denotation*) back to its original type, except that it does not accept the string representations of procedure references or atoms, interprets "nan" and "inf" as strings, treats "-nan" and "-inf" as erroneous, and classifies numbers as real or integer according to their appearance (so the type of unstr str 2.0 is integer). Also, since str may do some rounding when rendering a real, you don't necessarily get back exactly the number you put in for reals when making this round trip.

Quoted strings within *s* 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 can be twinned internally to represent that character. Apart from reducing those twins, unstr is completely literal about how it interprets what is inside a quoted string. Even backslashes are not special, in contrast to string literals in SETL source code or equivalently strings interpreted by unpretty.

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

Arbitrary whitespace, and/or a comma, is allowed wherever str would put a single blank between elements of sets and tuples. Whitespace is permitted around the overall denotation in *s*.

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

See also reada, reads, and val.

untie - dissolve stream association made by tie

```
proc untie (stream, stream)
```

For symmetry with tie.

val - read number from string

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

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

See also denotype and strad.

wait - wait for any child process status change

```
proc wait : integer
proc wait (boolean waitflag) : integer
```

Equivalent to waitpid (-1) or waitpid (-1, *waitflag*).

waitpid - wait for child process status change

```
proc waitpid (integer p): integer
proc waitpid (integer p, boolean waitflag) : integer
```

Tries to obtain the status of a child process identified by *p*.

If successful, it returns the process ID (pid) of a child process that has terminated, stopped (suspended), or continued (resumed). The child status appears 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 pid 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, stop, or continue. If false, meaning don't block, and the caller has at least one child identified by *p*, and no child in that set has a state change to report, waitpid sets status to om and returns 0. Otherwise, one such child is selected, status gets the state change details, and the pid is returned, just as in the blocking case.

In the case of termination, the child process and its exit status are said to be *reaped*, and the system clears its record of both.

If there are no child processes identified by *p*, waitpid sets status to om, sets last_error, and returns -1.

The use of waitpid is seldom necessary except after applying close in mode close_zombie to a pipe or pump stream, or when programming at the more foundational level of fork. Otherwise, you might reap away the status of a child created by pipe_from_child, pipe_to_child, pump, tty_pump, or one of the open modes "pipe-from", "pipe-to", "pump", or "tty-pump", before close gets a chance to.

Child processes that have not been waited for when the SETL program terminates are inherited by a POSIX system process that reaps and discards their statuses when they terminate, if ever.

See also system, filter, kill, pid, pexists, and time.

whole - format integer

```
proc whole (integer x, integer wid) : string
proc whole (real x, integer wid) : string
```

The number round *x* is converted to a string of length abs *wid* or more.

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

If abs *wid* is too small, a longer string is produced as necessary to accommodate the number.

Although this operator is intended primarily for rendering `integer` values, the automatic rounding in the case of `real` can result in the string `"nan"`, `"inf"`, or `"infinity"`, with or without a minus sign in front.

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

`with` - set plus one element

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

Definition: *s* `with` *x* `=` *s* `+ {`*x*`}`.

See the set union (*plus*) operator (`+`), and see also `less`.

`write` - write values to `stdout`

```
proc write (var args(*))
```

Equivalent to `writea (stdout, `*args*`(*))`.

See also `print` and `nprint`.

`writea` - write values to stream

```
proc writea (stream f, var args(*))
```

The 0 or more *args* are written in sequence to the stream *f*, separated by single spaces and followed by a newline character (`\n`). All of them are converted as if by `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` and `getb`.

If *f* is not already open, an attempt is made to auto-open it. See [Automatic opening], page 47.

On output error, output may be incomplete and `last_error` may be set.

A synonym for `writea` is `putb`.

See also `write` and `nprinta`.

Operator Precedence

From highest precedence (most tightly binding) to lowest:

operators	**associativity**
unary non-predicates, user-declared unary operators	right
power (`**`)	right
star (`*`), *slash* (`/`), `atan2`, `div`, `mod`, `rem`, `bit_and`	left
binary *plus* (`+`), binary *minus* (`-`), `max`, `min`, `bit_or`, `bit_xor`	left
`less`, `lessf`, `with`	left
`npow`	left
query (`?`)	left
user-declared binary operators	left
binary predicates: *equalities* (`=`, `/=`), *comparatives* (`<`, `>`, `<=`, `>=`), `in`, `notin`, `subset`, `incs`	none
unary predicates: `not`, `even`, `odd`, `fexists`, `lexists`, `pexists`, `is_open`, *type-testing* operators (`is_`*type*)	right
`and`	left
`or`	left
`impl`	left

Predicates are intrinsic operators that return `boolean`, and have lower precedence than all non-predicate operators.

Unary combining forms such as `+/ [1..9]`, where a left-associative operator is followed by slash, have the highest precedence, like other non-predicate unary forms. Binary combining forms such as `0 +/ [1..9]`, where the left-associative operator is followed by a slash but also preceded by an operand, have the same precedence as the binary operator.

The associativity of *power* (`**`) is changed from the original CIMS SETL to agree with SETL2 and Fortran. To avoid confusion, it is not allowed in the combining forms, which use a left-to-right chaining rule.

Restricted Mode

It is an error to call the following intrinsics in *restricted* mode (`--restricted` command-line option):

`callout`,

`chdir`,

`close` (except on streams that were opened by the SETL program),

`dup`, `dup2`,

`exec`,

`fexists`,

`filter` (except where permitted by an `--allow-filter` command-line option),

`fork`,

`fsize`,

`getegid`,

`getenv`,

`geteuid`,

`getgid`,

`getsid` with a non-zero argument,

`getuid`,

`getwd`,

`glob`,

`kill` (except on processes spawned by `pipe_from_child`, `pipe_to_child`, `pump`, or `tty_pump`),

`lexists`,

`link`,

`mkstemp` (except where permitted by an `--allow-mkstemp` command-line option),

`open` (except on a timer or where permitted by an `--allow-open` or `--allow-fd-open` command-line option),

`pexists`,

`pipe`,

`rename`,

`setctty`,

`setegid`,

`setenv`,

`seteuid`,

`setgid`,

`setpgid`,

`setsid`,

`setuid`,

`socketpair`,
`symlink`,
`system` (except where permitted by an `--allow-system` command-line option),
`sys_read`, `sys_write`,
`tcgetpgrp`, `tcsetpgrp`,
`unlink`,
`unsetctty`, and
`unsetenv`.

Also, `umask` in restricted mode ORs its argument with the mask that was in effect when the SETL program was started.

Concept Index

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z