Proposals For EuLisp

Streams, Filenames, and Macros

©Copyright 1991, 1992, 1993 by ILOG,
2, avenue Galliéni
B.P. 85
94250 Gentilly Cedex
FRANCE
All Rights Reserved

This document and the software described in this document are furnished under a license or non-disclosure agreement, and may be used or copied only in accordance with the terms of such license or non-disclosure agreement.

December 2, 1993

Contents

1

Macros and Literals

TALK supports the notion of symbolic literal constants. These are defined by the defining form `defliteral` (page 3), and are referenced by name, just like lexical variables. Named literal constants are treated by the preprocessor rather than directly by the compiler, and an error is signaled by the preprocessor when there is an ambiguity between a lexical variable and a literal constant with the same name. Therefore, you should be careful about naming literal constants to avoid conflict with lexical variables. Some convention should be adopted, such as placing `%` characters around the constant's name.

Two named literal constants are sufficiently prominent in TALK to deserve special mention. These are `()` (pronounced *nil*) and `t`. The distinguished object `()` is the only instance of the class `<null>` (page ??), and represents the canonical false value and the empty list. The symbol `t` is a canonical value for true, although any object which is not `()` can also be used as true. Two famous dialects of LISP were called NIL [1] and T [2].

The TALK module analysis process is carried out in several phases. The first of these is the preprocessor. The preprocessor expands macros and replaces literal constants with their values[1]. All definitions used by the preprocessor — macros and literal constants — must be in the compilation environment of the module to be processed. To avoid evaluating forms as they are analyzed, a module is never in its own compilation environment. Therefore, you cannot define a macro or constant in the same module in which it is used.

Macros allow the substitution of a form by another (typically more complicated) form, thus letting you extend the syntax of the language itself. Symbolic constants allow efficient and convenient access to simple values which rarely but occasionally change over the life of a program.

Macros and literal constants are referenced just like normally called functions and lexical variables. One important difference is that macros, not being funcallable objects, cannot be used with the special form `function`. In other words, a form such as `(funcall #'when condition form)` is illegal; `when`, implemented as a macro, can only occur in a function call position where the preprocessor can correctly expand the call.

An implication of this style of reference, which is very convenient in most cases, is that a macro which has the same name as a function has priority over the function of the same name. TALK does not consider this situation an error to allow optimizing macros with the same name as a function. You should be careful when doing this; if the macro expansion does not have the same semantics as the function, the result can be very difficult to debug.

[1] The form `setf` requires that `setf` methods referenced in a module also exist in the module's compilation environment. See the description of the defining form `defsetf` for more information.

On the other hand, an error is signaled by the analyzer when a lexical variable has the same name as a defined literal constant. In this case, the lexical variable would be completely hidden by the constant, so it is preferable to explicitly disallow such conflicts.

While a macro itself is not a function, it does have an associated "expansion function". The expansion function is a TALK function whose arguments are TALK forms. It returns a new TALK form which is evaluated in place of the original macro form. Macro forms are expanded from the outside in; this allows macro arguments to be non-evaluated syntactic markers such as keywords. If the expansion of a macro is another macro call, that call is itself expanded. At the end of this process, only special forms and function calls remain.

Because TALK macros have the full power of the TALK language at their disposal, they are much more powerful than macros in other languages, such as C, which provide for only a limited text substitution capability. With this power comes enormous flexibility: TALK macros can be used to create entire sublanguages within TALK, create new control abstractions such as the powerful `for` form, and create new abstractions such as `defstructure`. In other languages, any such facility must be supplied with the base language; in TALK, the programmer himself can extend the language.

`(defmacro` macro-name lambda-list *form*+`)` [*defining form*]

Summary Defines a macro.

Arguments	macro-name	`<symbol>`
	lambda-list	`<list>`
	form	`ptr`

This defining form defines a macro named *macro-name*, accepting arguments described by the lambda-list *lambda-list*. Macros allow syntactic extension of TALK. Macros must be in the compilation environment of modules which call them; they may be exported from their defining modules under the type `macro`.

This compilation environment requirement means that a macro cannot be defined in a module which uses that macro; instead, a separate module must be created. This is due to the fact that the ILOG TALK module analyzer/compiler does not evaluate forms as it analyzes them. While this requirement of separation may seem harsh, it has several benefits:

- It encourages programmers to write macros which are shared among several modules, thus reducing *ad hoc* module-specific mechanisms.
- The clear separation of compilation versus runtime requirements allows the creation of smaller final applications, which contain only their runtime components and not their compilation components.

We encourage programmers to place all of their project's macros into a small number of macro modules, which are placed in a library separate from their runtime library. We also encourage the distinction between internal macros and documented macros: Documented macros are those which your users will need, while internal macros are those used only to compile the application itself. These two different types of macros should be placed in separate modules.

When you write macros which introduce local variables, you must be careful that these new variables cannot interfere with the lexical environment of the macro call itself. This is best achieved by creating them with the function `gensym`. Temporary variables should be careful to preserve the order of evaluation of the arguments to the macro; otherwise, unusual errors may occur when these arguments perform side-effects.

Macros can accept keyword arguments. Since they are resolved statically during macroexpansion, keywords impose no runtime cost. While the lambda list for `defmacro` is essentially compatible with that for `defun` and `lambda`, it also allows one additional lambda list keyword: `&key`. The arguments following `&key` are keywords accepted by the macro, or lists of keywords and default values for the keywords. These keywords are then available in the macro-expansion function as variables named by the symbol equivalent to the keyword. The default values are executed in the lexical environment of the `defmacro` form augmented by the values of any preceding parameters. If such a list contains a third element, it must be a symbol which is bound to either `t` or `()` depending on whether the corresponding keyword was supplied in the macro call. This syntax is thus identical to `&optional`. If both `&optional` and `&key` appear, `&optional` must appear before `&key`, and any arguments beyond the required arguments are first assigned to the optional parameters. Any arguments still remaining are treated as keyword arguments.

```
? (defmacro when1 (condition &rest result)
    `(if ,condition (progn ,result)))
= when1
? (when1 (consp '(x y z)) "got a cons")
= got a cons
? (defmacro key-mac (&key (key1: 'default1) (key2: () key2p))
    `(progn
      (print "Key1 = " ',key1)
      ,@(when key2p
          `((print "Key2 = " ',key2)))
      ()))
= key-mac
? (key-mac key1: my-value)
Key1 = my-value
= ()
? (key-mac key2: val2 key1: val1)
Key1 = val1
Key2 = val2
= ()
? (key-mac key2: val2)
Key1 = default1
Key2 = val2
= ()
```

See also `ppme` (??) `gensym` (??)

(`defliteral` literal-name *form*) [*defining form*]

Summary Defines a literal constant.

Arguments literal-name `<symbol>`
form `<object>`

This definition form declares that *name* is to be substituted by the result of evaluating *value-form* in the lexical environment of the `defliteral` form. The constant value must be a legal literal. Modules which reference a literal constant must include the defining module in their compilation environment so that the constant's value is available when the reference is analyzed. There is no special accessor to literal constants; any occurrence to *name* in an evaluated position returns the binding value. Literal constants must be exported from their defining modules under the export type `literal`.

In the defining module:

```
? (defliteral %pi% 3.1415926536)
= %pi%
```

In the referencing module:

```
? (cos %pi%)
= -1.
```

Programmers often want literal constants to be calculated based on the value of other values, including other literal constants. If this were done straightforwardly in ILOG TALK, it would require a level of module for each such dependency, since literal constant definitions must be in the compilation environment of their references. An easy way to get around this restriction, at no runtime cost, is to define a global variable in the module which defines the literal constant. The literal constant values can then be calculated based on this global variable without penalty to their users. For example:

```
(defglobal :pi 3.1415926535)
(defliteral %pi% :pi)
(defliteral %pi%/2 (/ :pi 2.0))
```

It is an error to physically modify a literal constant value.

See also `defdynamic` (??) `defglobal` (??)

2

Streams and Filenames

2.1 Introduction

TALK manages I/O *streams* which represent data sources and sinks, such as files, terminals, strings, and so on. Streams are instances of subclasses of the class `<stream>`. Streams can be opened for input, output, or both, and can be passed to various I/O functions to direct the input and output of data. Three distinguished streams represent the default input stream, the default output stream, and the default error output stream; these three streams are accessible via the functions `stdin`, `stdout`, and `stderr`. Normally, the standard input and output streams are connected to the values of the corresponding system streams in the process from which TALK was started, but they may be modified.

Most often, streams are associated with files, which are named by filenames, which are instances of the class `<filename>`. The class of file streams is `<file-stream>`. In POSIX-compliant systems, almost all streamable entities are represented by files: text and binary files, terminals, devices, and so on. Because TALK is aimed primarily at POSIX-compliant platforms, anything which is a file in POSIX is associated with a file stream when it is used for I/O in TALK.

By relying on the widely-accepted POSIX standard, TALK guarantees both a low-cost and highly reliable stream system, and eases the transition of programmers accustomed to these functions into the TALK world. In addition, POSIX simplifies the treatment of terminals: In POSIX systems, a terminal is simply a file, and terminal streams in TALK are represented by instances of `<file-stream>` (page 16). You should consult the POSIX documentation for any details not made clear in this documentation.

TALK distinguishes between absolute and relative filenames. A relative filename is any which does not begin with a reference to the root directory — in the portable UNIX syntax, a slash. Absolute filenames do begin with a slash.

The hierarchy of stream classes and related classes is:

```
<object>
   <filename>
   <stream>
      <file-stream>
   <condition>
      <error>
         <posix-error>
            <posix-file-error>
      <io-condition>
         <end-of-stream>
         <fill-buffer>
         <flush-buffer>
```

File system operations such as file creation, checking for file accessibility, and so on are done using Talk bindings to the standard POSIX functions which perform these operations. For example, the `access` function (page 12) determines a file's existence and read/write-ability.

When an operating-system error occurs during a POSIX file operation, a condition of the class `<posix-file-error>` (page 20)is signaled. This condition can be handled by a user-defined handler providing greater flexibility in the case of an error compared with C: You no longer need to do tedious checking of error return codes at every call site. If the function returns, you know that it succeeded. The stream condition object contains pointers to the function and stream which generated the condition, as well as the POSIX-defined error symbol, and the operating-system dependent error number and error message. By default, Talk uses the normal error-handling mechanism (see page ??) when a `<posix-file-error>` is signaled, but you can handle the condition to retry the operation or provide other intelligent treatment.

File streams are opened using the function `fopen` (page 17 and are closed using `fclose` (page 18). The result of the stream opening functions is a stream object, which should be preserved as it is necessary to pass this object to `fclose` in order to properly close the stream. The second argument to `fopen` specifies whether a stream is to be opened for input, output, or update.

Three distinguished file streams are always available. These are the standard Talk input, standard Talk output, and standard Talk error output streams. They can be returned and modified using the functions `stdin`, `stdout`, and `stderr` respectively. The names of these functions were chosen to be similar to the C variables `stdin`, `stdout`, and `stderr`, but these functions do not modify those variables, although they are initialized from them. Therefore, when calling C code from Talk, do not assume that the C variables will contain appropriate file descriptors based on the values of these functions in Talk.

Stream operations are structured into two levels. POSIX provides a low-level means to read and write characters and strings to file streams using file descriptors; you can access these functions in C code from Talk streams using the external function argument converter `<file-stream>`, which converts a file stream to a file descriptor. Talk also provides high-level functionality to read and write Talk objects as well as individual characters and strings to file and string streams. At this level, conditions associated with streams are signaled in predefined situations, allowing you to extend the stream system with new types of streams by treating these conditions a stream-specific way. You can also write class-specific and stream-specific methods for printing individual Talk objects on a stream, providing another dimension of extensibility. This document describes the basic functions used to do I/O; please consult the *Wizard's Manual* for information on extending the stream system.

When printing Talk objects to a stream, parameters such as the left and right margins of streams can be set. Conditions are signaled at certain key moments in stream interaction, allowing you to modify stream behavior to provide advanced functionality. Formatted output is supported via the `printf` family of functions (page 25), which offers a rich set of options and formats to help you easily write beautiful reports. Talk extends the POSIX `printf` format directives with a new directive for printing Talk objects. Additional properties are associated with streams to enhance Talk object-level operations.

Usually, read and write functions are used to manipulate Talk objects. It is nevertheless sometimes necessary to read or write character by character or line by line. Then the position of the newline character is important. When reading a character at a time (using the `read-char` function, page 22), Talk delimits the end of a line with the usual newline characters `#\\n`. This is true for terminals as well as for files: They are both file streams.

When reading a line at a time (using the `read-line` function, page 22), the physical end-of-line is not included in the result.

When printing, the newline character is explicitly inserted by the user (by calling the `print` and `newline` functions, pages 24 and 28 respectively), and automatically by the system when the right margin of the output stream is reached.

The right margin can be disabled. In this case, the system does not automatically add the end-of-line mark (see the `stream-right-margin` function, page 30). Thus, the standard idiom in TALK for performing output without any right margin is:

```
(withf (((stream-right-margin (stdout)) ()))
  ... i/o code ...)
```

The most often-used I/O functions — such as `read` and `print` — read and write characters to and from I/O buffers. They do not physically access the destination of their stream at each call. When they need to update the internal buffer or flush it, they signal I/O conditions. The I/O condition classes are `<fill-buffer>`, `<flush-buffer>`, and `<end-of-stream>`. By setting up your own handlers for these conditions, you can partially control the behavior of the system buffering and extend the stream system. In general, for class of I/O condition, there is only one instance per stream which is resignaled every time it is needed. This limits the amount of object allocation in a typically rather speed-intensive area. The *Wizard's Manual* documents the use of these conditions for extending the stream system.

There are two major ways to produce output in Talk. One is to format output using functions in the `printf` family. This method is fine for making nice output for humans, or for other non-TALK programs. Another way is to call the object printing function `print`. This method produces output which is fairly readable for human and, most importantly, is the most convenient method to store data for another TALK program to read.

All the printing functions print their arguments into the output buffer. The expressions are not actually printed until the buffer is flushed. If you need to print an expression and have it immediately displayed without waiting for the end-of-line interrupt, use the function `prin` (page 24) followed by `flush` (page 28).

The output buffer can be flushed — and thus actually output — at different times:

- When the characters reach the right margin of the output stream.
- When the characters reach the end of the buffer.
- Upon calls to functions that empty the buffer after placing something into it (`print`, `flush`, `newline`).

In the first of these cases, a newline is inserted into the output stream. Then, in all cases, the function `flush` is called. The buffer is then simply emptied.

2.2 Files

2.2.1 Filenames

`<filename>` `<object>` [*class*]

Summary Class of filename objects.

Instances of this structure represent operating system-independent filenames.

(make-filename *name***)** [*function*]

Summary Creates a filename from a string.

Arguments *name* `<string>`

Result `<filename>`

Allocation `<filename>` 1

This function converts the string *name* corresponding to a filename into an object of class `<filename>`.

A filename can be specified either in UNIX syntax or in Windows syntax, that is to say that the directory delimiter can be either / or \, but it will always be printed with delimiter /. Any final delimiter is stripped unless the filename represents the root directory. An optional device, terminated with a : and containing no delimitors, can be prepended to the path.

```
? (make-filename "foo.c")
= #f"foo.c"
? (make-filename "/tmp/my-file")
= #f"/tmp/my-file"
? (make-filename "../foo.c")
= #f"../foo.c"
? (make-filename "/tmp/")
= #f"/tmp"
? (make-filename "/")
= #f"/"
? (make-filename "c:\\msdos\\dir.exe")
= #f"c:/msdos/dir.exe"
```

See also `#f` (8) `string` (??)

#f string [*reader syntax*]

Summary Reads a filename literal.

The reader syntax `#f` constructs a filename literal. It can be used to create an unquoted literal filename constant in programs.

```
? #f"foo.c"
= #f"foo.c"
? #f"/tmp/my-file"
= #f"/tmp/my-file"
? #f"../foo.c"
= #f"../foo.c"
? #f"/tmp/"
= #f"/tmp"
? #f"/"
= #f"/"
? #f"c:\\msdos\\dir.exe"
= #f"c:/msdos/dir.exe"
```

See also `make-filename` (8)

(filenamep *object***)** [*function*]

Summary Determines if an object is a filename.

Arguments *object* `<object>`

Result `boolean`

This predicate returns its argument if it is an instance of the class **`<filename>`**, and `()` otherwise.

```
? (filenamep #f"../foo.c")
= #f"../foo.c"
```

See also `instancep` (??)

(string `<filename>`) [*method*]

Summary Conversion of a filename to a string.

Result `<string>`

Allocation `<string>` if neccesary

This method returns a string representing the filename in the current operating system syntax. Note that, on UNIX, the device is not stripped.

Example on UNIX:

```
? (string #f"foo.c")
= "foo.c"
? (string #f"/tmp/my-file")
= "/tmp/my-file"
? (string #f"../foo.c")
= "../foo.c"
? (string #f"/tmp/")
= "/tmp"
? (string #f"/")
= "/"
? (string "c:\\msdos\\dir.exe")
= "c:/msdos/dir.exe"
```

Example on Windows:

```
? (string #f"foo.c")
= "foo.c"
? (string "/tmp/my-file")
= "\\tmp\\my-file"
? (string #f"../foo.c")
= "..\\foo.c"
? (string #f"/tmp/")
= "\\tmp"
? (string #f"/")
= "\\"
? (string "c:\\msdos\\dir.exe")
= "c:\\msdos\\dir.exe"
```

(basename *filename* &optional *extension*) [*function*]

Summary Returns the basename of a filename.

Arguments *filename* `<filename>`
extension `<string>` *Default* `""`

Result `<filename>`

This function returns the basename of a filename — that is, with the device and directory stripped off. If the string *extension* is supplied, it will be removed from the filename if the filename ends in the given string. This function is derived from the UNIX command `basename`.

```
? (basename #f"c:/tmp/foo.t.old")
= #f"foo.t.old"
? (basename #f"../foo.t")
= #f"foo.t"
? (basename #f"/tmp/foo.t.old" ".old")
= #f"foo.t"
? (basename #f"/tmp/foo.t.old" ".c")
= #f"foo.t.old"
? (basename #f"/")
= #f""
? (basename #f"foo.c" "foo.c")
= #f""
```

(`extension` *filename*) [*function*]

Summary	Returns the last extension of a filename.
Arguments	*filename* `<filename>`
Result	`<string>`
Allocation	`<string>` 1

This function returns the last extension of a filename as a string. The dot which begins the extension is also included. If the filename has no extension, an empty string is returned.

```
? (extension #f"/tmp/foo.t.old")
= ".old"
? (extension #f"foo.t")
= ".t"
? (extension #f"foo")
= ""
? (extension #f"foo.")
= "."
? (extension #f".talkrc")
= ".talkrc"
```

(`dirname` *filename*) [*function*]

Summary	Returns the device and directory part of a filename.
Arguments	*filename* `<filename>`
Result	`<filename>`
Allocation	`<filename>` 1

This function returns a filename representing the device and directory of *filename*. If *filename* names the root directory, it is simply returned. This function is derived from the UNIXcommand `dirname`.

```
? (dirname #f"c:/tmp/foo.t")
= #f"c:/tmp"
? (dirname #f"../foo.t")
= #f".."
? (dirname #f"c:dir/foo.t")
= #f"c:dir"
```

```
? (dirname #f"hux/bar/foo.t")
= #f"hux/bar"
? (dirname #f"foo.t")
= #f""
? (dirname #f"/")
= #f"/"
```

`(device` *filename*`)` [*function*]

Summary	Returns the device of a filename.
Arguments	*filename* `<filename>`
Result	`<string>`
Allocation	`<string>` 1

This function returns the device of the filename as a string. The colon which ends the device is also included. If *filename* has no device, an empty string is returned.

```
? (device #f"c:foo.t")
= "c:"
? (device #f"c:/tmp/foo.t")
= "c:"
? (device #f"foo")
= ""
```

`(merge-filenames` *file1* *file2* `&optional` *suffix*`)` [*function*]

Summary	Merges two filenames.
Arguments	*file1* `<filename>`
	file2 `<filename>`
	suffix `<string>` *Default* `""`
Result	`<filename>`
Allocation	`<filename>` 1

This function merges two filenames according to the following rules:

- No device conflicts are allowed between *file1* and *file2*. In other words, if either filename has a device, the other one must either have the same device or no device; otherwise, an error is signaled. If any device is specified, the result will have that device.
- If *file2*'s is absolute, it is taken as the result's directory and basename; otherwise, the result's directory and basename is made from concatenating *file1* to *file2*.
- If *suffix* is given, it is concatenated onto the result.

```
? (merge-filenames #f"c:/tmp/subdir" #f"foo.t")
= #f"c:/tmp/subdir/foo.t"
? (merge-filenames #f"hux" #f"foo" ".t")
= #f"hux/foo.t"
? (merge-filenames #f"hux" #f"foo.t" ".t")
= #f"hux/foo.t.t"
? (merge-filenames #f"/tmp/subdir" #f"/home/user/foo.t")
= #f"/home/user/foo.t"
? (merge-filenames #f"/home/user" #f".talkrc")
= #f"/home/user/.talkrc"
? (merge-filenames #f"/home/user" #f"../foo.t")
= #f"/home/user/../foo.t"
```

2.2.2 File Operations

`(umask)` [*function*]

`(setf (umask)` *new-value*`)` [*setf form*]

Summary Determines default protection mode.

Result `<list>`

This function determines the default protection mode. This mode will be combined with any given protection mode for functions which create files, such as **`mkdir`**. The initial value for this function is inherited from the process which starts up TALK. The protection mode is described by a list of symbols each representing one protection attribute which must be among the following values:

- `S_IRUSR`: Allow the owner to read the file.
- `S_IWUSR`: Allow the owner to write the file.
- `S_IXUSR`: Allow the owner to execute the file.
- `S_IRGRP`: Allow members of the file's group to read the file.
- `S_IWGRP`: Allow members of the file's group to write the file.
- `S_IXGRP`: Allow members of the file's group to execute the file.
- `S_IROTH`: Allow the owner to read the file.
- `S_IWOTH`: Allow the owner to write the file.
- `S_IXOTH`: Allow the owner to execute the file.

These values may be repeated in *new-value*; duplicates will be eliminated.

`(access` *file-name* `&optional` *modes*`)` [*function*]

Summary Tests for file accessibility.

Arguments *file-name* `<filename>`
modes `<symbol> or <list>` *Default* `F_OK`

Result `boolean`

This function checks the access rights for the file *file-name*. The argument *modes* is a symbol or a list of symbols which determine the access rights to be checked; the function returns true if all of the specified access rights are available for the file and `()` otherwise. The list *modes* may contain the following symbols:

- `R_OK`: Check read access.
- `W_OK`: Check write access.
- `X_OK`: Check execute access.
- `F_OK`: Check for file existence.

By default, `access` checks for file existence only.

`(getcwd)` [*function*]

`(setf (getcwd)` *new-value*`)` [*setf form*]

Summary Returns the name of the current working directory.

Result `<filename>`

This function returns the name of the current working directory as a filename.

See also `getcwd` (13)

`(chmod` *file-name modes*`)` [*function*]

Summary Change the access mode of a file.

Arguments *file-name* `<filename>`
modes `<list>`

Result `<null>`

This function attempts to change the access rights of the file *file-name* to be the union of those specified by the list of symbols *modes*. In POSIX, there are three types of users of a file and three access levels for each type. The user types are **`owner`**, **`group`**, and **`other`**; the access levels are **`read`**, **`write`**, and **`execute`**. The symbols allowed in *modes* are named according to the schema `S_I`*access-code user-code*, where *access-code* is either `R` (read), `W` (write) , or `X` (execute), and *user-code* is either `USR` (owner), `GRP` (group), or `OTH` (anyone).

The values allowed in the list *modes* are therefore:

- `S_IRUSR`: Allow the owner to read the file.
- `S_IWUSR`: Allow the owner to write the file.
- `S_IXUSR`: Allow the owner to execute the file.
- `S_IRGRP`: Allow members of the file's group to read the file.
- `S_IWGRP`: Allow members of the file's group to write the file.
- `S_IXGRP`: Allow members of the file's group to execute the file.
- `S_IROTH`: Allow the owner to read the file.
- `S_IWOTH`: Allow the owner to write the file.
- `S_IXOTH`: Allow the owner to execute the file.

`(mkdir` *filename* `&optional` *mode*`)` [*function*]

Summary Creates a new directory.

Arguments *filename* `<filename>`
mode `<list>` *Default* `(umask)`

Result `<null>`

This function creates a new directory named *filename* with permission bits set according to *mode*, which is a list of symbols. See the description of the function **`chmod`** (page 13) for more information about the mode list.

```
? (mkdir #f"guck")
 ** mkdir : guck is not a directory filename
= ()
? (mkdir #f"guck/")
= ()
? (mkdir #f"o/")
 ** mkdir : File exists : #f"o/"
= ()
```

`(rmdir` *filename*`)` [*function*]

Summary Removes a directory.

Arguments *filename* `<filename>`

Result `<null>`

This function removes the directory named *filename* and returns `()`.

```
? (mkdir #f"guck/")
= ()
? (rmdir #f"guck/file")
 ** rmdir : "guck/file" is not a directory filename
= ()
? (rmdir #f"guck/")
= ()
? (rmdir #f"guck/")
 ** rmdir : No such file or directory : #f"guck/"
= ()
```

`(rename` *old new*`)` [*function*]

Summary Renames a file.

Arguments *old* `<filename>`
new `<filename>`

Result `<null>`

This function renames the existing file *old* to be *new*. The file *old* must either name a regular file or a directory.

```
? (rename #f"old-file-name" #f"new-file-name")
= ()
? (rename #f"../a/relative/path/to/old-file-name"
          #f"../a/relative/path/to/new-file-name")
= ()
```

`(tmpnam)` [*function*]

Summary Returns the name of a non-existing temporary file.

Result `<filename>`

This function returns the name of a non-existing file which could be opened for reading and writing. It returns a different result each time it is called, something like **`gensym`**.

```
? (tmpnam)
= #f"/tmp/x33300"
```

`(unlink` *filename*`)` [*function*]

Summary Deletes a file.

Arguments *filename* `<filename>`

Result `<null>`

This function deletes the physical link named *filename*. This decrements the link count of the file's node and may, if the link count goes to zero, actually delete the node itself. On some systems, the link count is limited to one, and this operation may always remove the file.

```
? (unlink #f"an-existing-file")
= t
? (unlink #f"../a/relative/path/to/an-existing-file")
= t
```

`(file-mode` *filename*`)` [*function*]

`(file-ino` *filename*`)` [*function*]

`(file-dev` *filename*`)` [*function*]

`(file-nlink` *filename*`)` [*function*]

`(file-size` *filename*`)` [*function*]

`(file-atime` *filename*`)` [*function*]

`(file-mtime` *filename*`)` [*function*]

`(file-ctime` *filename*`)` [*function*]

`(file-type` *filename*`)` [*function*]

Summary Returns file stream information.

Arguments *filename* `<filename>`

Result `<object>`

These functions return information about a file; they correspond to most of the information available via the POSIX function `stat`. Most of these functions return a number. While on most systems and for most fields this number is an integer, it may also be a float on some systems for some fields. Other functions in this group return a symbol or boolean. In more detail:

- `file-mode`: This function returns a list of symbols describing the access mode for the file. See the function `chmod` (page 13) for more information about these symbols.
- `file-ino`: This function returns an integer uniquely identifying the file on its file system. However, this number may not be unique; you must use it in combination with the device number returned by `file-dev` to determine a file's true identity.
- `file-nlink`: This function returns the number of physical links pointing to the file, a number. This number is *not* the number of symbolic links to the file.

- `file-size`: This function returns the file's size in bytes, a number. This function is only defined for regular files, not pipes, directories or other strange types of files.
- `file-atime`: This function returns a `<time>` object representing the last time the file was accessed. This time should be used only to compare with other access times for files on the same server.
- `file-ctime`: This function returns a `<time>` object representing the last time the file was changed. By "change", we mean information *about* the file, such as its access modes, rather than its contents. This number should only be used to compare with other change times for files on the same server.
- `file-mtime`: This function returns a `<time>` object representing the last time the file's contents were modified.
- `file-type`: This function returns a symbol indicating the type of file attached to *filename*; it provides the TALK binding for POSIX macros `S_ISDIR`, `S_ISCHR`, `S_ISBLK`, `S_ISREG`, and `S_ISFIFO`. The symbol returned by this function may be one of the following values:
  - `S_ISDIR`: Indicates that the file is a directory.
  - `S_ISCHR`: Indicates that the file is a character-special file.
  - `S_ISBLK`: Indicates that the file is a block-special file.
  - `S_ISREG`: Indicates that the file is a regular file.
  - `S_ISFIFO`: Indicates that the file is a FIFO file.
  - `S_ISLNK`: For systems that support symbolic links, this value will be returned for symbolic links.
  - `S_ISSOCK`: For systems that support sockets, this value will be returned for sockets.

2.3 Streams

`<stream>` `<object>` [*class*]

Summary The superclass of stream classes.

This class is the superclass of all stream classes.

2.3.1 File Streams

`<file-stream>` `<stream>` [*class*]

Summary The class of streams associated with POSIX files.

This is the class of all streams associated with files. These streams are created using the function `fopen` and closed using `fclose`.

See also `fopen` (17) `fclose` (18) `prin` (24) `print` (24) `read` (22) `fprintf` (24) `printf` (25)

(`file-stream-filename` *file-stream*) [*function*]

Summary Returns the filename associated with a file stream.

Arguments *file-stream* `<file-stream>`

Result `<filename>`

This function returns the filename associated with the file stream *file-stream*. This will be equivalent to the filename passed to **fopen** to create the file stream.

`(stdin)` [*function*]

`(setf (stdin)` *new-value*`)` [*setf form*]

`(stdout)` [*function*]

`(setf (stdout)` *new-value*`)` [*setf form*]

`(stderr)` [*function*]

`(setf (stderr)` *new-value*`)` [*setf form*]

Summary Determines the current standard input, output, or error stream.

Result `<file-stream>`

These functions determine the current standard input, output, or error stream. By default, this is the standard input stream in effect when TALK is started up; generally, the terminal or window in which the command **italk** was typed. However, you can also pipe into and from TALK, in which case the pipe is taken as the standard stream.

If this value is modified, the mode of the new stream must be compatible with the mode appropriate for the function:

- `stdin`: `r`, `r+`, `w+`, or `a+`
- `stdout`: `w`, `a`, `r+`, `w+`, or `a+`
- `stderr`: `w`, `a`, `r+`, `w+`, or `a+`

You should be careful to keep track of the original value so that it can be restored later.

```
? (stdin)
= #<file-stream "/dev/pts/8" r+ #@00853742>
? (withf ((stdin (fopen (which 'pretty 't)))) (read))
= (deffeature pretty)
? (stdout)
= #<file-stream "/dev/pts/8" r+ #@00853742>
? (stderr)
= #<file-stream "/dev/pts/8" r+ #@00853742>
```

`(fopen` *file-name* `&optional` *mode*`)` [*function*]

Summary Opens a file stream.

Arguments *file-name* `<filename>`
mode `<symbol>` *Default* `'r`

Result `<file-stream>`

This function opens and returns a file stream for the file named *file-name*. The mode of the stream is a symbol specifying how the file is opened. It should be one of the following symbols:

- `r`: Opens the file for reading.
- `w`: Opens the file for writing, creating it if it does not exist.
- `a`: Opens the file for append output, creating it if it does not exist. If the file exists, the stream position points to the end of the file, so any output adds to the file.
- `r+`: Opens the file for update. Both input and output can be performed on the stream. Any existing data in the file is preserved.
- `w+`: Opens the file for update. Both input and output can be performed on the stream. Any existing data in the file is eliminated; the file is re-created if it already exists.
- `a+`: Opens the file for update. The file can be read at any location, but any output to the file occurs starting from its current end.

```
? (fopen #f"an-existing-readable-file" 'r)
= #<file-stream "an-existing-readable-file" r #@0047AA83>
? (fopen #f"a-writable-file" 'w)
= #<file-stream "a-writable-file" w #@00332211>
? (fopen #f"/absolute/path/to/a-non-existing-file" 'r)
 ** fopen : No such file or directory :
 #f"/absolute/path/to/a-non-existing-file"
= ()
? (fopen #f"/absolute/path/to/an-existing-writable-file" 'r+)
= #<file-stream "/absolute/path/to/an-existing-writable-file" r+ #@00123123>
? (fopen #f"../relative/path/to/a-non-existing-writable-file" 'w+)
=  #<file-stream "../relative/path/to/a-non-existing-writable-file" w+ #@00321321>
? (fopen #f"/bin/an-unwritable-file" 'w+)
 ** fopen : Permission denied : #f"/bin/an-unwritable-file"
= ()
? (fopen "/bin/an-unwritable-file" 'a+)
 ** fopen : Bad file number : #f"/bin/an-unwritable-file"
= ()
```

See also `fclose` (18)

(`fclose` *stream*) [*function*]

Summary Closes a file stream.

Arguments *stream* `<file-stream>`

Result `<null>`

This function closes the stream *stream* and returns `()`. No further I/O operations can be performed on *stream*. Any pending output is flushed to the file before it is closed.

```
? (fclose (fopen #f"/tmp/foo" 'w))
= ()
```

Here is an example of file handling:

```
(defun my-copyfile (in-file out-file)
  ;; This function copies the entire contents of the
  ;; input file IN-FILE to the output file OUT-FILE, compacting
  ;; at the same time the representations of Talk expressions
  ;; in the latter.
  (withf* (((stdin) (fopen in-file))
```

```
            ((stdout) (fopen out-file 'w))
            ((stream-left-margin (stdout)) 0)
            ((stream-right-margin (stdout)) 70))
      (let ((eos (cons () ())))
        (for ((item (read (stdin) () eos)))
          (when (eq item eos) (finish))
          (print item))
        (fclose (stdout))
        out-file))

? (withf ((stdout (fopen "foo.t" 'w)))
    (print "(defun foo (x)")
    (print "  (+ x x))"))
= (+ x x))
? (my-copyfile #f"foo.t" #f"bar.t")
= #f"bar.t"
? !cat foo.t
(defun foo (x)
  (+ x x))
= ()
? !cat bar.t
(defun foo (x) (+ x x))
= ()
```

(`fcntl` *stream* `&optional` *command* *command-values*) [*function*]

Summary	Returns or sets information about a file stream.		
Arguments	*stream*	`<file-stream>`	
	command	`<symbol>`	*Default* `F_GETFL`
	command-values	`<list>`	*Default*
Result	`<object>`		

This function returns or sets information about an open file stream. The symbol *command* determines exactly what action this function will take. In some cases, the third argument *command-values* must also be given. The *command* can be one of the following symbols (which are a subset of those accepted by POSIX):

- `F_GETFL`: This is the default value. In this case, `fcntl` returns a list of symbols or integers, in no particular order, describing the stream's attributes. These values can be among the following:
  - `O_APPEND`: The file is opened in append mode.
  - `O_NONBLOCK`: Waiting for data does not block the system.
  - `O_RDONLY`: The file is opened for read only.
  - `O_RDWR`: The file is opened for reading and writing.
  - `O_WRONLY`: The file is opened for writing only.
  - Other system-specific integers may also be implemented.
- `F_SETFL`: In this case, the third argument *command-values* must be a list of attributes from the set described above for `F_GETFL`. However, according to POSIX, only the values `O_APPEND` and `O_NONBLOCK` can be modified.

  The proper way to set the values is to use a get/modify cycle as follows:

  ```
  (defun set-file-stream-values (stream new-values)
    (fcntl stream 'F_SETFL (union (fcntl stream) new-values)))
  ```

Future evolution : Other POSIX options may be added in the future.

`(open-file-streams)` [*function*]

Summary Returns a list of all open file streams.

Result `<list>`

This function returns a list of all open file streams. This is not a POSIX feature, but a convenience provided by TALK to help track down lost stream references.

2.3.2 Terminal Files

POSIX specifies a number of functions which handle terminal characteristics to facilitate this special mode of I/O. Terminals in POSIX-conformant systems are just special kinds of files, but because of their interactive nature, they require a few extra operations.

`(ctermid)` [*function*]

Summary Returns a filename corresponding to the terminal.

Result `<filename> or <null>`

This function returns a filename which corresponds to the current terminal. This function can be useful in conjunction with **`fopen`** to regenerate reasonable values for **`(stdin)`**, **`(stdout)`**, or **`(stderr)`** in extreme cases, but this capability is not guaranteed. If no such filename can be determined, `()` is returned.

`(isatty` *stream*`)` [*function*]

Summary Tests if a file stream is attached to a terminal.

Arguments *stream* `<file-stream>`

Result `boolean`

This function returns `t` if the file stream *stream* is attached to an interactive terminal, and `()` otherwise.

2.3.3 POSIX Errors

`<posix-error>` `<error>` [*class*]

Summary Abstract superclass of all POSIX errors.

This abstract class is the superclass of all POSIX error classes, including `<posix-file-error>`.

`<posix-file-error>` `<posix-error>` [*class*]

Summary Condition signaled when an error occurs during stream operations.

A condition of this class is signaled when an operating-system error occurs during a stream operation. Associated with the condition is the name of the function and the stream which generated the error, as well as the POSIX-defined error code associated with the error, and the operating-system dependent error number and textual message.

See also `condition-caller` (??) `posix-error-code` (21) `posix-error-string` (21) `posix-error-stream` (??)

(`posix-error-type` *condition*) [*function*]

Summary Returns the POSIX error type of a stream condition.

Arguments *condition* `<posix-error>`

Result `<symbol> or <null>`

(`posix-error-code` *condition*) [*function*]

Summary Returns the operating system dependent numeric error code of a stream condition.

Arguments *condition* `<posix-error>`

Result `<integer> or <symbol>`

(`posix-error-string` *condition*) [*function*]

Summary Returns the operating system dependent error string of a stream condition.

Arguments *condition* `<posix-error>`

Result `<string> or <null>`

(`posix-file-error-stream` *condition*) [*function*]

Summary Returns the stream which generated a stream condition.

Arguments *condition* `<posix-error>`

Result `<file-stream>`

These functions provide access to the context surrounding a signaled stream condition. The condition type is a symbol; the set of symbols which might be signaled for each stream function is determined by POSIX. Please consult the POSIX documentation for a description of the various error types for each function. The condition code is an integer which depends on the operating system. This code may be important because operating systems are free to signal errors which are not defined in the POSIX specification. In this case, the condition type will be `()`, and the condition code will be the unique identifier of the exact error type. You should consult your operating system documentation for a complete description of the various codes which might be signaled. The condition string is an operating system dependent string which usually contains a more or less informative message describing the error. The default handling of stream conditions prints this message along with the function which generated the error (retrieved using the function `condition-caller`, page ??), and the stream itself. Finally, the stream which was responsible for the error can be retrieved from the condition via the accessor `posix-file-error-stream`.

2.4 Input

`(read &optional` *stream eos-error-p eos-value*`)` [*function*]

Summary	Reads one expression from a stream			
Arguments	*stream*	`<stream>`	*Default*	`(stdin)`
	eos-error-p	`boolean`	*Default*	`t`
	eos-value	`ptr`	*Default*	`(eos-default-value)`
Result	`<object>`			

This function reads the next expression from the stream *stream* and returns it. Its behavior is described in the introduction to I/O, section 2.1, page 5. The stream *stream* must be opened for input or update.

The optional argument *eos-error-p* is a flag which determines the behavior if an end-of-file is encountered during the reading. If it is `()`, no error will be signaled and the value of the argument *eos-value* will be returned. If it is true, an error will be signaled.

`(read-line &optional` *stream eos-error-p eos-value*`)` [*function*]

Summary	Reads one line as a string in a stream			
Arguments	*stream*	`<input-stream>`	*Default*	`(stdin)`
	eos-error-p	`boolean`	*Default*	`t`
	eos-value	`<object>`	*Default*	`(eos-default-value)`
Result	`<string>`			
Allocation	`<string>`			

This function reads the next line from the input stream and returns it as a character string. This string does not contain the end-of-line delimiters that might have been present in the input. The `read-line` function provides a way to do portable reads at the input line level. It is used to build efficient and easily-implemented interfaces between programs and users.

This function reads characters from the current position on the current input line up to the next end of line marker. Any characters already read, either via `read-char` or `read`, will not appear in the string.

The optional argument *eos-error-p* is a flag which determines the behavior if an end-of-file is encountered during the reading. If it is `()`, no error will be signaled and the value of the argument *eos-value* will be returned. If it is true, an error will be signaled.

`(read-char &optional` *stream eos-error-p eos-value*`)` [*function*]

Summary	Reads one character in a stream			
Arguments	*stream*	`<input-stream>`	*Default*	`(stdin)`
	eos-error-p	`boolean`	*Default*	`t`
	eos-value	`<object>`	*Default*	`(eos-default-value)`
Result	`<character>`			

This function reads and returns a character from the input stream. If there are no more characters to read from the current line, this function calls the generic function `bol` on the stream, which, by default, signals the condition `<bol>` to get a new line from the input stream.

The optional argument *eos-error-p* is a flag which determines the behavior if an end-of-file is encountered during the reading. If it is `()`, no error will be signaled and the value of the argument *eos-value* will be returned. If it is true, an error will be signaled.

(peek-next-char &optional *stream eos-error-p eos-value*) [*function*]

Summary	Looks ahead one character in a stream			
Arguments	*stream*	`<input-stream>`	*Default*	`(stdin)`
	eos-error-p	`boolean`	*Default*	`t`
	eos-value	`<object>`	*Default*	`(eos-default-value)`
Result	`<character>`			

This function returns the next character in the input stream in the same way that the **read-char** function does, except that the character is not pulled off the input stream. It is merely examined. As a result, consecutive calls to **peek-next-char** with no intervening read always return the same result. **peek-next-char** is used for look-ahead operations on the input stream.

The optional argument *eos-error-p* is a flag which determines the behavior if an end-of-file is encountered during the reading. If it is `()`, no error will be signaled and the value of the argument *eos-value* will be returned. If it is true, an error will be signaled.

<end-of-stream> <io-condition> [*class*]

Summary Signaled when reading past the end of a stream

An instance of this class is signaled when an attempt is made to read past the end of a stream by any of the input functions described here.

(eos-default-value) [*function*]

Summary The default value returned at the end of a stream

Result `<object>`

This function returns the unique object which indicates that the end of a stream has been reached. Its class is not specified, but calling this function always returns the same object.

```
? (eq (eos-default-value) (eos-default-value))
= t
? !more foo
(hi)
(mom)
= ()
? (let ((stream (fopen #f"foo")))
    (for ((expr (read stream ())))
      (when (eq expr (eos-default-value)) (finish))
      (print expr)))
(hi)
(mom)
= ()
```

(sread *string* &optional *eos-error-p eos-value*) [*function*]

Summary Reads an expression from a string

Arguments	*string*	`<string>`	
	eos-error-p	`boolean`	*Default* `t`
	eos-value	`ptr`	*Default* `()`
Result	`<object>`		

This function reads one TALK object from a string *string*.

The optional argument *eos-error-p* is a flag which determines the behavior if the string does not contain a complete expression. If it is `()`, no error will be signaled and the value of the argument *eos-value* will be returned. If it is true, an error will be signaled.

```
? (sread "(+ 1 2)")
= (+ 1 2) ; 3-element list
? (sread "hello")
= hello    ; Talk symbol
? (sread "\"abr\"")
= "abr"    ; character string
? (sread "(list (a b)")
 ** sread : incomplete expression "(list (a b)"
= () ; incomplete expression
? (sread "(list (a b)") () 'oops)
= oops
? (sread "(list) (a b)")
= (list) ; several expressions
```

See also `read` (22)

2.5 Output

(`prin` *object* `&optional` *stream*) [*function*]

(`print` *object* `&optional` *stream*) [*function*]

Summary	Prints an object on a stream.		
Arguments	*object*	`ptr`	
	stream	`<stream>`	*Default* `(stdout)`
Result	`ptr`		

These functions print the written representation of *object* on the stream *stream* and return *object*. The function `prin` does not print a newline and the expressions will not be actually printed until the stream is flushed, either explicitly by calling `flush` or `eol`, by printing past the stream's right margin, or by printing a newline on the stream.

While the `prin` and `print` functions make every effort to print objects rereadably, it cannot be guaranteed that all objects have such a rereadable print representation. However, for lists, vectors, numbers, strings, and ids, this is always possible if they only contain rereadable elements.

The function `print` is similar to `prin`, but it flushes the stream immediately after printing the object, and begins a new line by calling the function `eol`.

(`fprintf` *stream format-string* `&rest` *args*) [*function*]

(`efprintf` *stream format-string* `&rest` *args*) [*function*]

Summary Formatted output to a stream.

Arguments *stream* `<stream>`
format-string `<string>`
args *list of* `<object>`

Result `<integer>`

`(printf` *format-string* `&rest` *args*`)` [*function*]

Summary Formatted output to the standard output.

Arguments *format-string* `<string>`
args *list of* `<object>`

Result `<integer>`

`(sprintf` *stream format-string* `&rest` *args*`)` [*function*]

Summary Formatted output to a string.

Arguments *stream* `<string>` or `<null>`
format-string `<string>`
args *list of* `<object>`

Result `<integer>` or `<string>`

These functions provide formatted output. The function **fprintf** writes to a file stream, while **printf** always writes to (**stdout**). The function **efprintf** is like **fprintf**, but never signals an error. Instead, if there is an error during the processing of the format string or printing, it makes its best attempt to produce useful output, but may print anything or nothing.

The function **sprintf** writes to a string — the string may be provided; otherwise, it is created and returned. If the string is provided, **sprintf** returns the number of characters written into the string. An error is signaled if the number of characters written to the string exceeds the length of the string.

For the numeric directives, all TALK integers are converted to signed **long**s. TALK objects can be printed using the new **%a** directive. Any TALK object to be printed using **%a** directive will first be printed to a string using the function **sprin**. Then the strings thus created are printed as if with the directive **%s**. These new directives allow the use of the **-** and *width* modifiers; these are applied to the string produced for final formatting.

For the functions **printf**, **sprintf**, and **fprintf**, type checking is done to avoid the kinds of errors typical of the use of this function in C. An error is signaled if there is a type mismatch between the directive and the actual argument, or between the number of directives and the number of arguments. On the other hand, **efprintf** is used specifically in situations where such error checking is to be avoided — typically, within the output produced by the **error** function itself. It treats all directives as **%a** and does no type or argument number checking. Any missing arguments will have a strange, ugly object printed instead. Any extra arguments will simply be printed after the format string with spaces between them.

Each format directive begins with a percent character **%**. Following the percent character, other directive-dependent flags and modifiers may appear. They must appear in the following order:

- This modifier will cause the result of the directive be left-justified in its field. By default, the result is right-justified.

+ This modifier will cause the result of any numeric fields to begin with a sign, even positive numbers. By default, numbers are only printed with a sign if they are negative. This modifier is illegal for non-numeric directives.

space This modifier is identical to `+` except that a space is used for positive numbers instead of a plus sign. If both `+` and the space are used, the plus sign is printed.

`#` The meaning of this binary flag is different for various directives. Please consult the table below for details.

width After the above flags, a field width may be specified. The exact meaning of the field width depends on the directive. See the table below for details.

`.`*precision* A precision may next be specified, preceded by a dot to separate it from a possible width specification. The precision modifier may only be used with the floating point directives `e`, `E`, `f`, `g`, and `G`.

format Finally, the directive type is specified as a single letter. The directives accepted are summarized in the table below.

Directive	Description	Meaning of *width*	Meaning of #
`i` or `d`	Prints a number converted to an int in decimal format.	Specifies the minimum size of the field. If the actual size is smaller, the field is padded according to the `-` modifier. The default width is 1. If 0 is specified, nothing is printed.	Undefined.
`o`	Prints a number converted to an int in octal format.	Same as `d`.	The first digit is forced to 0 for rereadable output.
`u`	Prints a number converted to an int as an unsigned decimal.	Same as `d`.	Undefined.
`x`	Prints a number converted to an int in hexadecimal format. Small letters from `a` to `f` are used.	Same as `d`.	The result is prefixed with `0x`.
`X`	Identical to `x` except capital letters from `A` to `F` are used.	Same as `d`.	Same as `x`.
`f`	Prints a number converted to a float in the format `[-]ddd.ddd`.	The minimum number of characters to be printed. The *precision* modifier may be used. If a decimal point is printed, there will always be at least one digit printed to its left.	Prints a decimal point even if the fractional part is 0.
`e`	Prints a number converted to a float in exponential notation `[-]ddd.ddde dd`. The exponent will always contain at least two digits.	Same as `f`.	Same as `f`.
`E`	Same as `e` except that `E` is used to introduce the exponent instead of `e`.	Same as `f`.	Same as `f`.
`g`	If the exponent is less than -4 or greater than the value of the *precision* modifier, uses `e`. Otherwise, uses `f`.	Same as `f`.	Same as `f`.
`G`	Just like `g`, except `E` is used instead of `e`.	Same as `f`.	Same as `f`.
`c`	The value is converted to a character which is written. The value can either be an int or a character.	Undefined	Undefined
`s`	The value is converted to a string using the function `sprin`; this string is written.	Specifies the maximum number of characters to be written.	Undefined.
`p`	The value is taken to be a pointer which is printed in an implementation-dependent way. Usually the address of the object is printed in hexadecimal format.	Undefined	Undefined
`a`	The value, which must be a TALK object, is printed out as if with `print`.	Same as `s`.	Undefined

(flush &optional *stream*) [*function*]

Summary Flushes a stream to its target.

Arguments *stream* `<stream>` *Default* `(stdout)`

Result `<null>`

This function flushes any pending output to the *stream*. No newline is printed.

(newline &optional *stream*) [*function*]

Summary Flushes a stream to its target and prints a newline.

Arguments *stream* `<stream>` *Default* `(stdout)`

Result `<null>`

This function flushes any pending output to the *stream* and then prints a newline.

(prin-char *character-num* &optional *stream times*) [*function*]

Summary Prints a single character from its code.

Arguments	*character-num*	`<character>`	
	stream	`<stream>`	*Default* `(stdout)`
	times	`<integer>`	*Default* `1`
Result	`<integer>`		

This function prints the *character-num* character code *times* times into the output stream *stream*. If *times* is omitted, the code *character-num* is only printed once. This function returns *times* as its value.

```
(defun drawfn (fn start stop inc &optional (stream (stdout)))
  (for ((i from: start to: stop by: inc))
    (fprintf stream "%d:" i)
    (setf (stream-output-buffer-position stream) 8)
    (prin-char #\* stream (integer (funcall fn i)))
    (newline stream)))

? (drawfn (lambda (f) (f* (sin (float (/ f 10))) 25.0)) 1 31 1)
1:      **
2:      ****
3:      *******
4:      *********
5:      ***********
6:      **************
7:      ****************
8:      *****************
9:      *******************
10:     *********************
11:     **********************
12:     ***********************
13:     ************************
14:     ************************
15:     ************************
16:     ************************
17:     ************************
18:     ************************
19:     ***********************
20:     **********************
21:     *********************
22:     ********************
23:     ******************
```

```
24:    ****************
25:    **************
26:    ************
27:    **********
28:    ********
29:    *****
30:    ***
31:    *
```

(sprin *form***)** [*function*]

Summary Prints an expression to a string.

Arguments *form* ptr

Result <string>

Allocation <string>

This function prints *form* into a new string and returns it. This is a convenient way to write a single object into a string, without going through the whole process of creating a string stream, printing to it, and retrieving its string. Strings printed in this way have no left or right margins; the string output is created on one long line.

```
? (sprin '(+ 1 2))
= "(+ 1 2)"
? (sprin 'hello)
= "hello"
? (sprin "abr")
= "\"abr\""
```

See also prin (24)

(defprinter class-name (ovar {svar | (svar sclass)}) *form****)** [*defining form*]

Summary Defines a print method for a type.

Arguments

class-name	<symbol>
var	<symbol>
svar	<symbol>
sclass	<symbol>
form	ptr

It is possible to redefine the way in which objects are printed, for the standard print functions all search for methods named `prin` associated with the type of the object to be printed. If such a method exists, it is called in lieu of the standard print function. If it does not exist, the generic function `prin-object` is used to print the object. Methods on `prin-object` are best defined using the defining form `defprinter`.

This defining form defines a new print method on the generic function `prin-object` for the class named *class-name*, a symbol naming an existing class. The lambda-list must be a list with two arguments, the first of which will be an instance of the given type, and the second of which will be stream onto which the object will be printed. The second argument may be optionally specialized by a stream class, in which case the method is only applicable for instances of the given stream class or its subclasses. The function is called whenever an instance of the type is to be printed on a particular stream.

It is recommended that a print method which prints an un-rereadable object prefix the string printed with the characters `#<` and terminate with the character `>`.

See also `prin` (24) `prin-object` (30) `defstructure` (??)

(`prin-object` *object stream*) [*generic function*]

Summary Invokes a class-specific print method for an object.

Arguments *object* `<object>`
stream `<stream>`

Result `<object>`

This generic function is invoked when the printer prints an object *object* to the stream *stream* when the object is not an instance of one of the predefined TALK classes. New methods can be written for this generic function using either `defprinter` or `defmethod`; `defprinter` is preferred. This function should not be called explicitly; use one of the standard printing functions instead.

(`stream-right-margin` `&optional` *stream*) [*function*]

(`setf` (`stream-right-margin` `&optional` *stream*) *new-value*) [*setf form*]

(`stream-left-margin` `&optional` *stream*) [*function*]

(`setf` (`stream-left-margin` `&optional` *stream*) *new-value*) [*setf form*]

Summary Determines the right and left margins for a stream.

Arguments *stream* `<stream>` *Default* `(stdout)`

Result `<integer> or <null>`

These functions are used by the high-level stream operations to determine the left and right margins used when printing TALK objects to the stream *stream* using the functions `prin` and `print`. They are ignored when using the `printf` family of functions, which provide their own formatting. By default, when a stream is created its left margin is 0 and its right margin is 78.

For `stream-right-margin`, the value may also be `()`, indicating that the stream has no right margin and a newline will never be printed unless placed explicitly using `newline` or `print`.

An error is signaled if *new-value* is negative or greater than current output buffer width.

(`stream-max-characters` `&optional` *stream*) [*function*]

(`setf` (`stream-max-characters` `&optional` *stream*) *new-value*) [*setf form*]

Summary Determines maximum number of characters for printing an expression.

Arguments *stream* `<stream>` *Default* `(stdout)`

Result `<integer>`

This function determines the maximum number of characters which will be printed by the function `print` and its friends on a stream *stream* for a single expression. By default this value is 500000. If an expression requires more than this number of characters, the last line will appear with an unreadable marker at the end. In order to deactivate this limit, call `stream-max-characters` with 0 as its argument. The function returns the current maximum number of print characters allowed as its result.

2.6 Extending the Stream System

2.6.1 Output buffer management

All the print functions in Talk work using buffers. Print functions write characters in the output buffer.

This buffer is filled by the various printing functions and emptied by the system by signaling the stream's `<flush-buffer>` condition, whose default handler calls the generic function `flush-buffer`.

A character string, accessed by the function `stream-output-buffer`, holds the contents of the current output buffer for a string. An index into the string, `stream-output-buffer-position`, marks the current write position (the position in the buffer where the next character should be written). Two indices, `stream-left-margin` and `stream-right-margin`, mark the left and right margins, respectively, of the output buffer.

If the value of `stream-output-buffer-position` for a given stream becomes equal to or greater than `stream-right-margin` during a write, this means that a program is trying to write a character beyond the right margin of the output buffer. In this case, a newline is inserted in the output buffer and the system signals `<flush-buffer>`, which calls `flush-buffer`. Methods on this generic function must empty the output buffer onto the stream's target object, fill the output buffer with spaces, and set the `stream-output-buffer-position` index to the left margin (the value of `stream-left-margin`).

If the value of `stream-output-buffer-position` for a given stream reaches the end of the buffer during a write, this means that a program is trying to write a line longer than the buffer. This is possible when the right margin (`stream-right-margin`) is `()`. In this case, the `<flush-buffer>` condition for the stream is signaled without inserting a newline in the output buffer. The default handler method calls the generic function `flush-buffer` on the stream. Its method must empty the output buffer onto the given stream, fill the output buffer with *space* characters, and set the `stream-output-buffer-position` index to zero. The `stream-output-buffer-position` index is set to zero *even if output continues on the same physical line.* This is because the logical output buffer has changed, and `stream-output-buffer-position` is an index into this string rather than the physical output line. This means that calling `flush` explicitly in the middle of printing a line may produce counter-intuitive results if you also use `stream-output-buffer-position`: After an explicit flush, the output buffer is emptied and `stream-output-buffer-position` is reset to 0, even though the position on the physical line is greater than 0.

`(stream-end-of-stream` *stream*`)` [*function*]

`(setf (stream-end-of-stream` *stream*`)` *new-value*`)` [*setf form*]

`(stream-flush-buffer` *stream*`)` [*function*]

`(setf (stream-flush-buffer` *stream*`)` *new-value*`)` [*setf form*]

`(stream-fill-buffer` *stream*`)` [*function*]

`(setf (stream-fill-buffer` *stream*`)` *new-value*`)` [*setf form*]

Summary Determines the various `<io-condition>` conditions associated with a stream.

Arguments *stream* `<stream>`

Result `<io-condition>`

These functions return the various I/O conditions associated with a stream. Each stream has at most one instance of each such condition, which is signaled each time the appropriate situation is generated while reading from or writing to the stream. The condition object may not be instantiated until the first time it is called for. The various conditions are:

- `<end-of-stream>`: Signaled when reading past the end of a stream.
- `<flush-buffer>`: Signaled when buffered output is flushed to the stream. Only signaled when doing high-level stream operations.
- `<fill-buffer>`: Signaled when a new buffered input line is needed for the stream. Only signaled when doing high-level stream operations.

2.6.2 I/O Conditions

`<io-condition>` `<condition>` [*class*]

Summary Superclass of all I/O condition classes.

This class is the superclass of all the I/O condition classes which are signaled at defined moments to allow user intervention in buffering actions. Each I/O condition has a stream associated with it; each stream has at most one instance of each of the condition classes associated with it.

`(io-condition-stream` *condition*`)` [*function*]

Summary Returns the stream associated with an I/O condition.

Arguments *condition* `<io-condition>`

Result `<stream>`

This function returns the stream associated with an I/O condition.

`<end-of-stream>` `<io-condition>` [*class*]

Summary Signaled at end-of-file.

When an attempt is made to read past the end of a stream, the system signals the stream's `<end-of-stream>` condition. The default handler method for this condition calls the generic function `end-of-stream`. The method invoked for this generic function must in general close the stream by a call to the `fclose` function, and throw to the tag `end-of-stream`. This allows the use of the common idiom `(catch 'end-of-stream (for ... (read) ...))` to read from a file until its end.

You can write your own handler for this conditoin to provide various extended functionality. The following code, for example, provides a way to load a sequence of files named `file`.1 .. `file`.n all at once. Forms can straddle two files.

```
(defdynamic :*n* ())        ; the current extension
(defdynamic :*file* ())     ; the filename

(defhandler sequence-handler)

(define-handler-method sequence-handler ((c <end-of-stream>))
  (let ((next-file (:make-seq-path (dynamic :*file*) (i+ 1 (dynamic :*n*)))))
    (if (not (access next-file 'F_OK))
        ;; no more files:
        ;; standard end-of-file processing
        (call-next-handler)
       (progn
        ;; go on to the next file in the sequence
        (printf "Closing %d, opening %d.\n"
                (dynamic :*n*) (i+ 1 (dyanmic :*n*)))
        (fclose (io-condition-stream c))
        (incf (dynamic :*n*))
        (setf (stdin) (fopen next-file 'r))))))

(defun :make-seq-path (file number)
  (make-filename (catenate (string file) (string number))))

(defun load-in-sequence (file)
  (dynamic-let ((:*n* 1)
                (:*file* file))
    (with-handler sequence-handler
      (load-file (:make-seq-path file 1) t))))
```

(`end-of-stream` *stream*) [*generic function*]

Summary Called when the end of file of a stream is read.

Arguments *stream* `<stream>`

Result `<null>`

This generic function is called when an attempt is made to read past the end of a stream. By default, it throws to the tag `end-of-stream`.

(`end-of-stream <stream>`) [*method*]

Summary Normal handling of end-of-stream.

Result `<null>`

The default method for `end-of-stream` throws to the tag `end-of-stream`. If an end-of-file appears while a Talk object is being read, an error is signaled.

(`end-of-stream <file-stream>`) [*method*]

Summary Handling end-of-stream for file streams.

Result `<null>`

For file streams, in addition to the normal end-of-stream handling, the file stream is first closed by calling `fclose`.

2.6.3 Terminal File Streams

(`tcgetattr` `&optional` *stream*) [*function*]

Summary Retrieves terminal characteristics.

Arguments *stream* `<file-stream>` *Default* `(stdin)`

Result `<list>`

This function returns a list of terminal characteristics for the file stream *stream*, which must be attached to a terminal. The characteristics are a subset of the values in the `c_lflag` member of the `termios` structure defined by POSIX for this function. If the returned list contains one of the values listed below, that characteristic is true of the terminal attached to *stream*. If the list does not contain a value, the characteristic is false. The possible values in the returned list are:

- `ECHO`: Echo is activated.
- `ICANON`: Canonical I/O is activated. Canonical I/O is line-based. If this value is missing, it means the terminal processes input character-by-character.

(`tcsetattr` `&optional` *stream when attributes*) [*function*]

Summary Changes terminal characteristics.

Arguments *stream*	`<file-stream>`	*Default*	`(stdin)`
when	`<symbol>`	*Default*	`TCSANOW`
attributes	`<list>`	*Default*	`()`

Result `<null>`

This function changes the current terminal characteristics of the file stream *stream*, which must be attached to a terminal. The new characteristics are described by the list of symbols *attributes*, which can contain symbols as described for the function `tcgetattr`. The symbol *when* describes when the new attributes will come into effect. It can take on one of the following values:

- `TCSANOW`: The changes are effectuated immediately.
- `TCSADRAIN`: The changes are effectuated after all pending output has been transmitted.
- `TCSAFLUSH`: The changes are effectuated after all pending output has been transmitted. In addition, all pending input is flushed without waiting. This is useful, for example, for resetting the terminal to accept passwords or whenever typeahead should be ignored.

2.6.4 Advanced Use of the Reader

The reader presented earlier explains the semantics of the TALK reader, but does not discuss its extension or dynamic modification. In this section, we present the features which allow you to parameterize the reader and input streams.

(`flush-input` `&optional` *stream*) [*function*]

Summary Flushes the rest of the current line.

Arguments *stream* `<input-stream>` *Default*

Result `<object>`

This function flushes all the characters waiting in the input buffer of the stream *stream*. The first read immediately following a call to this function will signal the `<fill-buffer>` condition associated with *stream*, generally calling the generic function `fill-buffer`. This function is particularly useful for flushing the separation characters (delimiters) present in the reader's internal buffers just after the action of the `read` function.

```
? (defun foo ()
    (printf "read->%a\n" (read))
    (printf "read-line->%s\n" (read-line))
    t)
= foo
? (foo)
? hello
read->hello
read-line->
= t
? (defun bar ()
    (printf "read->%a\n" (read))
    (flush-input)
    (printf "read-line->%s\n" (read-line))
    t)
= bar
? (bar)
? hello
read->hello
? hello world
read-line->hello world
= t
```

(`fill-buffer` *stream*) [*generic function*]

Summary Fills a stream's input buffer and returns number of characters.

Arguments *stream* `<stream>`

Result `<integer>`

When there are no more characters to read in the input buffer, the system signals a condition of the class `<fill-buffer>`, which, by default, calls the generic function `fill-buffer` on the stream . The default method for this function fills the input buffer with characters read from the current input stream. It is possible to manage this condition yourself by writing your own method on `fill-buffer` for a new class of streams, or by writing a handler which treats `<fill-buffer>`. If this handler does not call `call-next-handler`, you must handle any buffering issues yourself. This generic function returns the number of characters actually read.

Methods for this generic function must either the current value of `stream-input-buffer` with characters from the stream and return the number of characters in the buffer. If no more characters can be read from the stream, the stream's `<end-of-stream>` condition should be signaled.

`<fill-buffer>` `<condition>` [*class*]

Summary Class of beginning-of-line condition.

This condition is signaled when the input buffer has no more characters to read, and one is asked for. There is only one instance of this class per input stream, which is signaled every time that stream needs a new line. The default handler method calls the generic function `fill-buffer`.

Here is an example `<fill-buffer>` handler that counts the number of lines the user types into the terminal. It changes the prompt to display this value.

```
? (defglobal :user-lines 0)
= :user-lines
? (defhandler numbering-handler)
= numbering-handler
? (define-handler-method numbering-handler ((c <fill-buffer>))
    (when (isatty (io-condition-stream c))
      (setf (stream-prompt (io-condition-stream c))
            (catenate (string (incf :user-lines)) "? "))
    (call-next-handler))
= numbering-handler <fill-buffer>
? (add-handler numbering-handler)
= ()
1?
2? 123
= 123
3? ()
= ()
4? (defun foo (x)
5?   (+ x x))
= foo
6? (foo 12)
= 24
7?
```

(fill-buffer <file-stream>) [*method*]

Summary Default buffer filling method.

Result `<string>`

This method reads the next line from the terminal or from a file and stores it in the stream's input buffer. It also sets the number of characters read in the stream's input buffer. If no more characters can be read from the stream, the method signals the stream's `<end-of-stream>` condition.

(stream-input-buffer *stream*) [*function*]

(setf (stream-input-buffer *stream*) *new-value*) [*setf form*]

Summary Determines a stream's input buffer.

Arguments *stream* `<stream>`

Result `<string>`

This function returns or sets a stream *stream*'s input buffer. This function is primarily used by methods on `fill-buffer`; they can either replace the buffer or fill the existing buffer with new characters.

(stream-input-buffer-size *stream*) [*function*]

(setf (stream-input-buffer-size *stream*) *new-value*) [*setf form*]

Summary Determines the number of characters read into a stream's input buffer.

Arguments *stream* `<stream>`

Result `<integer>`

This function returns or sets the number of characters currently read into *stream*'s input buffer. This is not necessarily the length of the string returned by **`stream-input-buffer`**; in particular, the last time the stream's buffer is filled, the number of characters read may be less than the buffer size.

2.6.5 Example Input Stream Extension

This section presents a sample stream extension. We want to create a new kind of stream which is attached to vectors of strings, each string in the vector representing one line of input to be read. Tokens must fit within one string, but expressions can cross over several strings. Our strategy is to write a new method for **`fill-buffer`** which selects successive strings of the vector into the stream's input buffer slot. This example is based on the code used by Talk to read a module's literal string vector which contains descriptions of each module's literals.

```
(defstructure <vector-stream> <stream>
  (:vector (:index 0)))

(defmethod initialize-instance ((object <vector-stream>) init-list)
  (let ((vector (plref init-list vector:)))
    (assert (and (vectorp vector)
                 (for ((elt in-vector: vector))
                   (collect-and (stringp elt)))))
    (call-next-method)))

(defmethod fill-input-stream-buffer ((stream <vector-stream>))
  ;; Replace STREAM's input buffer with the next vector string and
  ;; increment the vector index.  If there are no more strings in the
  ;; vector, signal <end-of-stream>.
  (let ((vector (vector-stream-vector stream))
        (index (vector-stream-index stream)))
    (if (i= index (vlen vector))
        (signal (stream-end-of-stream stream))
      (let ((next-buffer (vref vector index)))
        (call-next-method)
        (setf (stream-input-buffer stream) next-buffer)
        (setf (stream-input-buffer-size stream) (slen next-buffer))
        (incf (vector-stream-index stream))
        ()))))

;;; Using it:

? (defglobal vstream
    (make-vector-stream vector: #("(a" "b" "c) other-symbol-buffer")))
= vstream
? (read vstream)
= (a b c)
? (read vstream)
= other-symbol-buffer
? (read vstream)
 ** read : end of stream on #<vstream #@00AB34CD>
= ()
```

2.6.6 Advanced Use of the Printer

(`stream-output-buffer-position` `&optional` *stream*) [*function*] `(setf (stream-output-buffer-position &optional` *stream*`)` *new-value*`)` [*setf form*]

Summary Sets or returns the current output buffer position.

Arguments *stream* `<stream>` *Default* `(stdout)`

Result `<integer>`

This function gets or sets the current output buffer position of *stream*. When a new value is supplied, the current index is set to *new-value*. This index is always treated as the first free character position in the buffer. The function `stream-output-buffer-position` returns its current value. At the beginning of each line, the output buffer is filled with spaces. Therefore, you can use `stream-output-buffer-position` as a way to move directly to a particular output column without fear of contamination from previous buffer contents or printing strange characters. On the other hand, you must be careful of flushes on *stream*, which always reset `stream-output-buffer-position` to 0 without doing a carriage return.

An error is signaled if *new-value* is negative or greater than current output buffer width.

`(stream-output-buffer &optional` *stream*`)` [*function*]

Summary The current output buffer and its contents.

Arguments *stream* `<stream>` *Default* `(stdout)`

Result `<string>`

This function returns the character string representing the current contents of the output buffer of the stream *stream*. This string can be manipulated by the usual string-handling functions.

`<flush-buffer>` `<io-condition>` [*class*]

Summary Class of conditions signaled when flushing the output buffer of a stream.

Instances of this condition are signaled by the various print functions when they need to flush their buffer. The default handler calls the generic function `generic-flush`. You can write new handlers which have methods on this condition to provide dynamic hooks for all stream flushes.

`(flush-buffer` *stream*`)` [*generic function*]

Summary Flushes a stream to its target.

Arguments *stream* `<stream>`

Result `ptr`

This generic function is called by the default handler for the condition `<flush-buffer>`, which is signaled by the various print functions when the current print position of the stream *stream*, `stream-output-buffer-position`, reaches the end of the output buffer, or when the right margin is passed, or when a newline is printed.

The applicable method must perform the following operations:

- empty the output buffer, from position 0 through position `stream-output-buffer-position`, onto the stream's target;
- fill the output buffer with the *space* character;
- set the `stream-output-buffer-position` index to 0;

2.6.7 Advanced Filename Operations

(pathconf *file name*) [*function*]

Summary Returns a configuration limit for a path.

Arguments *file* `<filename>`
name `<symbol>`

Result `<integer> or boolean`

This function returns a configuration limit for the filename *file*. This value is either an integer or a boolean flag. In some cases, the value `t` might be returned for an integer limit; this indicates that the system has no preset limit for the given parameter. The exact value returned depends on the value of *name*, which can be one of the following symbols:

- `_PC_LINK_MAX`: The maximum value of the file's link count is returned. If *file* is a directory, this value refers to the entire directory.
- `_PC_MAX_CANON`: The maximum length of a formatted input line is returned. In this case, *file* must refer to a terminal.
- `_PC_MAX_INPUT`: The maximum length of an input line is returned. In this case, *file* must refer to a terminal.
- `_PC_NAME_MAX`: The stream *file* must refer to a directory. The maximum length of a filename in this directory is returned.
- `_PC_PATH_MAX`: The stream *file* must refer to a directory. The maximum length of a relative path name when *file* is the current working directory is returned. This is not the global maximum filename length.
- `_PC_PIPE_BUF`: The stream *file* must refer to a pipe or FIFO. The size of the pipe buffer is returned.
- `_PC_CHOWN_RESTRICTED`: A boolean is returned which indicates whether `chown` may be called on the file *file*. If *stream* is a directory, this applies to all files in the directory.
- `_PC_NO_TRUNC`: The stream *file* must refer to a directory. A boolean is returned which indicates if an error of type `ENAMETOOLONG` would be signaled if an attempt is made to create a file in the directory *stream* and the file name would be truncated.
- `_PC_VDISABLE`: The stream *file* must refer to a terminal. A boolean is returned which indicates if special character processing can be disabled in the terminal.

Bibliography

[1] Glenn Burke and George Carrette. NIL Notes for Release 0. Technical report, Massachusetts Institute of Technology Laboratory for Computer Science, 1982.

[2] Stephen Slade. *The T Programming Language.* Prentice-Hall, Inc., 1987.

Table of Entities

Index

G

I

L

M

N

O

P

R

S

T

U

V

Y