NAME

scons − a software construction tool

SYNOPSIS

scons [*options...*] [*name=val...*] [*targets...*]

DESCRIPTION

The **scons** utility builds software (or other files) by determining which component pieces must be rebuilt and executing the necessary commands to rebuild them.

By default, **scons** searches for a file named *SConstruct*, *Sconstruct*, or *sconstruct* (in that order) in the current directory and reads its configuration from the first file found. An alternate file name may be specified via the **-f** option.

The *SConstruct* file can specify subsidiary configuration files using the **SConscript()** function. By convention, these subsidiary files are named *SConscript*, although any name may be used. (Because of this naming convention, the term "SConscript files" is sometimes used to refer generically to all **scons** configuration files, regardless of actual file name.)

The configuration files specify the target files to be built, and (optionally) the rules to build those targets. Reasonable default rules exist for building common software components (executable programs, object files, libraries), so that for most software projects, only the target and input files need be specified.

Before reading the *SConstruct* file, **scons** adds looks for a dir named *site_scons* in the dir containing the *SConstruct* file; it adds that *site_scons* to sys.path, reads the file *site_scons/site_init.py*, and adds the directory *site_scons/site_tools* to the default toolpath, if those exist. See the *--no-site-dir* and *--site-dir* options for more details.

scons reads and executes the SConscript files as Python scripts, so you may use normal Python scripting capabilities (such as flow control, data manipulation, and imported Python libraries) to handle complicated build situations. **scons**, however, reads and executes all of the SConscript files *before* it begins building any targets. To make this obvious, **scons** prints the following messages about what it is doing:

```
$ scons foo.out
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
cp foo.in foo.out
scons: done building targets.
$
```

The status messages (everything except the line that reads "cp foo.in foo.out") may be suppressed using the **-Q** option.

scons does not automatically propagate the external environment used to execute **scons** to the commands used to build target files. This is so that builds will be guaranteed repeatable regardless of the environment variables set at the time **scons** is invoked. This also means that if the compiler or other commands that you want to use to build your target files are not in standard system locations, **scons** will not find them unless you explicitly set the PATH to include those locations. Whenever you create an **scons** construction environment, you can propagate the value of PATH from your external environment as follows:

```
import os
env = Environment(ENV = {'PATH' : os.environ['PATH']})
```

Similarly, if the commands use external environment variables like $PATH, $HOME, $JAVA_HOME, $LANG, $SHELL, $TERM, etc., these variables can also be explicitly propagated:

```
import os
env = Environment(ENV = {'PATH' : os.environ['PATH'],
                         'HOME' : os.environ['HOME']})
```

Or you may explicitly propagate the invoking user's complete external environment:

```
import os
env = Environment(ENV = os.environ)
```

This comes at the expense of making your build dependent on the user's environment being set correctly, but it may be more convenient for many configurations.

scons can scan known input files automatically for dependency information (for example, #include statements in C or C++ files) and will rebuild dependent files appropriately whenever any "included" input file changes. **scons** supports the ability to define new scanners for unknown input file types.

scons knows how to fetch files automatically from SCCS or RCS subdirectories using SCCS, RCS or BitKeeper.

scons is normally executed in a top-level directory containing a *SConstruct* file, optionally specifying as command-line arguments the target file or files to be built.

By default, the command

```
scons
```

will build all target files in or below the current directory. Explicit default targets (to be built when no targets are specified on the command line) may be defined the SConscript file(s) using the **Default()** function, described below.

Even when **Default()** targets are specified in the SConscript file(s), all target files in or below the current directory may be built by explicitly specifying the current directory (.) as a command-line target:

```
scons .
```

Building all target files, including any files outside of the current directory, may be specified by supplying a command-line target of the root directory (on POSIX systems):

```
scons /
```

or the path name(s) of the volume(s) in which all the targets should be built (on Windows systems):

```
scons C:\ D:\
```

To build only specific targets, supply them as command-line arguments:

```
scons foo bar
```

in which case only the specified targets will be built (along with any derived files on which they depend).

Specifying "cleanup" targets in SConscript files is not usually necessary. The **-c** flag removes all files necessary to build the specified target:

```
scons -c .
```

to remove all target files, or:

```
scons -c build export
```

to remove target files under build and export. Additional files or directories to remove can be specified using the **Clean()** function. Conversely, targets that would normally be removed by the **-c** invocation can be prevented from being removed by using the **NoClean**() function.

A subset of a hierarchical tree may be built by remaining at the top-level directory (where the *SConstruct* file lives) and specifying the subdirectory as the target to be built:

```
scons src/subdir
```

or by changing directory and invoking scons with the **-u** option, which traverses up the directory hierarchy until it finds the *SConstruct* file, and then builds targets relatively to the current subdirectory:

```
cd src/subdir
scons -u .
```

scons supports building multiple targets in parallel via a **-j** option that takes, as its argument, the number of simultaneous tasks that may be spawned:

```
scons -j 4
```

builds four targets in parallel, for example.

scons can maintain a cache of target (derived) files that can be shared between multiple builds. When caching is enabled in a SConscript file, any target files built by **scons** will be copied to the cache. If an up-to-date target file is found in the cache, it will be retrieved from the cache instead of being rebuilt locally. Caching behavior may be disabled and controlled in other ways by the **--cache-force**, **--cache-disable**, and **--cache-show** command-line options. The **--random** option is useful to prevent multiple builds from trying to update the cache simultaneously.

Values of variables to be passed to the SConscript file(s) may be specified on the command line:

```
scons debug=1 .
```

These variables are available in SConscript files through the ARGUMENTS dictionary, and can be used in the SConscript file(s) to modify the build in any way:

```
if ARGUMENTS.get('debug', 0):
   env = Environment(CCFLAGS = '-g')
else:
   env = Environment()
```

The command-line variable arguments are also available in the ARGLIST list, indexed by their order on the command line. This allows you to process them in order rather than by name, if necessary. ARGLIST[0] returns a tuple containing (argname, argvalue). A Python exception is thrown if you try to access a list member that does not exist.

scons requires Python version 1.5.2 or later. There should be no other dependencies or requirements to run **scons.**

By default, **scons** knows how to search for available programming tools on various systems. On Windows systems, **scons** searches in order for the Microsoft Visual C++ tools, the MinGW tool chain, the Intel compiler tools, and the PharLap ETS compiler. On OS/2 systems, **scons** searches in order for the OS/2 compiler, the GCC tool chain, and the Microsoft Visual C++ tools, On SGI IRIX, IBM AIX, Hewlett Packard HP-UX, and Sun Solaris systems, **scons** searches for the native compiler tools (MIPSpro, Visual Age, aCC, and Forte tools respectively) and the GCC tool chain. On all other platforms, including POSIX (Linux and UNIX) platforms, **scons** searches in order for the GCC tool chain, the Microsoft Visual C++ tools, and the Intel compiler tools. You may, of course, override these default values by appropriate configuration of Environment construction variables.

OPTIONS

In general, **scons** supports the same command-line options as GNU **make**, and many of those supported by **cons**.

-b
Ignored for compatibility with non-GNU versions of **make.**

-c, --clean, --remove
Clean up by removing all target files for which a construction command is specified. Also remove any files or directories associated to the construction command using the **Clean**() function. Will not remove any targets specified by the **NoClean**() function.

--cache-debug=*file*
Print debug information about the **CacheDir**() derived-file caching to the specified *file*. If *file* is **–** (a hyphen), the debug information are printed to the standard output. The printed messages describe what signature file names are being looked for in, retrieved from, or written to the **CacheDir**() directory tree.

--cache-disable, --no-cache
Disable the derived-file caching specified by **CacheDir**(). **scons** will neither retrieve files from the cache nor copy files to the cache.

--cache-force, --cache-populate
When using **CacheDir**(), populate a cache by copying any already-existing, up-to-date derived files to the cache, in addition to files built by this invocation. This is useful to populate a new cache with all the current derived files, or to add to the cache any derived files recently built with caching disabled via the **--cache-disable** option.

--cache-show
When using **CacheDir**() and retrieving a derived file from the cache, show the command that would have been executed to build the file, instead of the usual report, "Retrieved ‘file’ from cache." This will produce consistent output for build logs, regardless of whether a target file was rebuilt or retrieved from the cache.

--config=*mode*
This specifies how the **Configure** call should use or generate the results of configuration tests. The option should be specified from among the following choices:

--config=auto
scons will use its normal dependency mechanisms to decide if a test must be rebuilt or not. This saves time by not running the same configuration tests every time you invoke scons, but will overlook changes in system header files or external commands (such as compilers) if you don’t specify those dependecies explicitly. This is the default behavior.

--config=force
If this option is specified, all configuration tests will be re-run regardless of whether the cached results are out of date. This can be used to explicitly force the configuration tests to be updated in response to an otherwise unconfigured change in a system header file or compiler.

--config=cache
If this option is specified, no configuration tests will be rerun and all results will be taken from cache. Note that scons will still consider it an error if --config=cache is specified and a necessary test does not yet have any results in the cache.

-C *directory*, --directory=*directory*
Change to the specified *directory* before searching for the *SConstruct*, *Sconstruct*, or *sconstruct* file, or doing anything else. Multiple **-C** options are interpreted relative to the previous one, and the right-most **-C** option wins. (This option is nearly equivalent to **-f directory/SConstruct**, except that it will search for *SConstruct*, *Sconstruct*, or *sconstruct* in the specified directory.)

-D
Works exactly the same way as the **-u** option except for the way default targets are handled. When this option is used and no targets are specified on the command line, all default targets are built, whether or not they are below the current directory.

--debug=*type*
Debug the build process. *type* specifies what type of debugging:

--debug=count
Print how many objects are created of the various classes used internally by SCons before and after reading the SConscript files and before and after building targets. This is not supported when run under Python versions earlier than 2.1, when SCons is executed with the Python **-O** (optimized) option, or when the SCons modules have been compiled with optimization (that is, when executing from ***.pyo** files).

--debug=dtree
A synonym for the newer **--tree=derived** option. This will be deprecated in some future release and ultimately removed.

--debug=explain
Print an explanation of precisely why **scons** is deciding to (re-)build any targets. (Note: this does not print anything for targets that are *not* rebuilt.)

--debug=findlibs
Instruct the scanner that searches for libraries to print a message about each potential library name it is searching for, and about the actual libraries it finds.

--debug=includes
Print the include tree after each top-level target is built. This is generally used to find out what files are included by the sources of a given derived file:

```
$ scons --debug=includes foo.o
```

--debug=memoizer
Prints a summary of hits and misses using the Memoizer, an internal subsystem that counts how often SCons uses cached values in memory instead of recomputing them each time they're needed. Only available when using Python 2.2 or later.

--debug=memory
Prints how much memory SCons uses before and after reading the SConscript files and before and after building targets.

--debug=nomemoizer
A deprecated option preserved for backwards compatibility.

--debug=objects
Prints a list of the various objects of the various classes used internally by SCons. This only works when run under Python 2.1 or later.

--debug=pdb
Re-run SCons under the control of the pdb Python debugger.

--debug=presub
Print the raw command line used to build each target before the construction environment variables are substituted. Also shows which targets are being built by this command. Output looks something like this:

```
$ scons --debug=presub
Building myprog.o with action(s):
  $SHCC $SHCFLAGS $SHCCFLAGS $CPPFLAGS $_CPPINCFLAGS -c -o $TARGET $SOURCES
```

--debug=stacktrace
Prints an internal Python stack trace when encountering an otherwise unexplained error.

--debug=stree
A synonym for the newer **--tree=all,status** option. This will be deprecated in some future release and ultimately removed.

--debug=time
Prints various time profiling information: the time spent executing each individual build command; the total build time (time SCons ran from beginning to end); the total time spent reading and executing SConscript files; the total time spent SCons itself spend running (that is, not counting reading and executing SConscript files); and both the total time spent executing all build commands and the elapsed wall-clock time spent executing those build commands. (When **scons** is executed without the **-j** option, the elapsed wall-clock time will typically be slightly longer than the total time spent executing all the build commands, due to the SCons processing that takes place in between executing each command. When **scons** is executed *with* the **-j** option, and your build configuration allows good parallelization, the elapsed wall-clock time should be significantly smaller than the total time spent executing all the build commands, since multiple build commands and intervening SCons processing should take place in parallel.)

--debug=tree
A synonym for the newer **--tree=all** option. This will be deprecated in some future release and ultimately removed.

--diskcheck=*types*

Enable specific checks for whether or not there is a file on disk where the SCons configuration expects a directory (or vice versa), and whether or not RCS or SCCS sources exist when searching for source and include files. The *types* argument can be set to: **all**, to enable all checks explicitly (the default behavior); **none**, to disable all such checks; **match**, to check that files and directories on disk match SCons' expected configuration; **rcs**, to check for the existence of an RCS source for any missing source or include files; **sccs**, to check for the existence of an SCCS source for any missing source or include files. Multiple checks can be specified separated by commas; for example, **--diskcheck=sccs,rcs** would still check for SCCS and RCS sources, but disable the check for on-disk matches of files and directories. Disabling some or all of these checks can provide a performance boost for large configurations, or when the configuration will check for files and/or directories across networked or shared file systems, at the slight increased risk of an incorrect build or of not handling errors gracefully (if include files really should be found in SCCS or RCS, for example, or if a file really does exist where the SCons configuration expects a directory).

--duplicate=*ORDER*

There are three ways to duplicate files in a build tree: hard links, soft (symbolic) links and copies. The default behaviour of SCons is to prefer hard links to soft links to copies. You can specify different behaviours with this option. *ORDER* must be one of *hard-soft-copy* (the default), *soft-hard-copy*, *hard-copy*, *soft-copy* or *copy*. SCons will attempt to duplicate files using the mechanisms in the specified order.

-f *file*, --file=*file*, --makefile=*file*, --sconstruct=*file*

Use *file* as the initial SConscript file.

-h, --help

Print a local help message for this build, if one is defined in the SConscript file(s), plus a line that describes the **-H** option for command-line option help. If no local help message is defined, prints the standard help message about command-line options. Exits after displaying the appropriate message.

-H, --help-options

Print the standard help message about command-line options and exit.

-i, --ignore-errors

Ignore all errors from commands executed to rebuild files.

-I *directory*, --include-dir=*directory*

Specifies a *directory* to search for imported Python modules. If several **-I** options are used, the directories are searched in the order specified.

--implicit-cache

Cache implicit dependencies. This causes **scons** to use the implicit (scanned) dependencies from the last time it was run instead of scanning the files for implicit dependencies. This can significantly speed up SCons, but with the following limitations:

scons will not detect changes to implicit dependency search paths (e.g. **CPPPATH**, **LIBPATH**) that would ordinarily cause different versions of same-named files to be used.

scons will miss changes in the implicit dependencies in cases where a new implicit dependency is added earlier in the implicit dependency search path (e.g. **CPPPATH**, **LIBPATH**) than a current implicit dependency with the same name.

--implicit-deps-changed

Forces SCons to ignore the cached implicit dependencies. This causes the implicit dependencies to be rescanned and recached. This implies **--implicit-cache**.

--implicit-deps-unchanged

Force SCons to ignore changes in the implicit dependencies. This causes cached implicit dependencies to always be used. This implies **--implicit-cache**.

--interactive

Starts SCons in interactive mode. The SConscript files are read once and a **scons>>>** prompt is printed. Targets may now be rebuilt by typing commands at interactive prompt without having to re-read the SConscript files and re-initialize the dependency graph from scratch.

SCons interactive mode supports the following commands:

build*[OPTIONS] [TARGETS] ...*

Builds the specified *TARGETS* (and their dependencies) with the specified SCons command-line *OPTIONS*. **b** and **scons** are synonyms.

The following SCons command-line options affect the **build** command:

--cache-debug=FILE
--cache-disable, --no-cache
--cache-force, --cache-populate
--cache-show
--debug=TYPE
-i, --ignore-errors
-j N, --jobs=N
-k, --keep-going
-n, --no-exec, --just-print, --dry-run, --recon
-Q
-s, --silent, --quiet
-s, --silent, --quiet
--taskmastertrace=FILE
--tree=OPTIONS

Any other SCons command-line options that are specified do not cause errors but have no effect on the **build** command (mainly because they affect how the SConscript files are read, which only happens once at the beginning of interactive mode).

clean*[OPTIONS] [TARGETS] ...*

Cleans the specified *TARGETS* (and their dependencies) with the specified options. **c** is a synonym. This command is itself a synonym for **build --clean**

exit Exits SCons interactive mode. You can also exit by terminating input (CTRL+D on UNIX or Linux systems, CTRL+Z on Windows systems).

help*[COMMAND]*

Provides a help message about the commands available in SCons interactive mode. If *COMMAND* is specified, **h** and **?** are synonyms.

shell*[COMMANDLINE]*

Executes the specified *COMMANDLINE* in a subshell. If no *COMMANDLINE* is specified, executes the interactive command interpreter specified in the **SHELL** environment variable (on UNIX and Linux systems) or the **COMSPEC** environment variable (on Windows systems). **sh** and **!** are synonyms.

version

Prints SCons version information.

An empty line repeats the last typed command. Command-line editing can be used if the **readline** module is available.

```
$ scons --interactive
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons>>> build -n prog
scons>>> exit
```

-j *N*, --jobs=*N*

Specifies the number of jobs (commands) to run simultaneously. If there is more than one **-j** option, the last one is effective.

-k, --keep-going

Continue as much as possible after an error. The target that failed and those that depend on it will not be remade, but other targets specified on the command line will still be processed.

-m Ignored for compatibility with non-GNU versions of **make**.

--max-drift=*SECONDS*

Set the maximum expected drift in the modification time of files to *SECONDS*. This value determines how long a file must be unmodified before its cached content signature will be used instead of calculating a new content signature (MD5 checksum) of the file's contents. The default value is 2 days, which means a file must have a modification time of at least two days ago in order to have its cached content signature used. A negative value means to never cache the content signature and to ignore the cached value if there already is one. A value of 0 means to always use the cached signature, no matter how old the file is.

-n, --just-print, --dry-run, --recon

No execute. Print the commands that would be executed to build any out-of-date target files, but do not execute the commands.

--no-site-dir

Prevents the automatic addition of the standard *site_scons* dir to *sys.path*. Also prevents loading the *site_scons/site_init.py* module if it exists, and prevents adding *site_scons/site_tools* to the toolpath.

--profile=*file*
Run SCons under the Python profiler and save the results in the specified *file*. The results may be analyzed using the Python pstats module.

-q, --question
Do not run any commands, or print anything. Just return an exit status that is zero if the specified targets are already up to date, non-zero otherwise.

-Q
Quiets SCons status messages about reading SConscript files, building targets and entering directories. Commands that are executed to rebuild target files are still printed.

--random
Build dependencies in a random order. This is useful when building multiple trees simultaneously with caching enabled, to prevent multiple builds from simultaneously trying to build or retrieve the same target files.

-s, --silent, --quiet
Silent. Do not print commands that are executed to rebuild target files. Also suppresses SCons status messages.

-S, --no-keep-going, --stop
Ignored for compatibility with GNU **make**.

--site-dir=*dir*
Uses the named dir as the site dir rather than the default *site_scons* dir. This dir will get prepended to *sys.path*, the module *dir*/site_init.py will get loaded if it exists, and *dir*/site_tools will get added to the default toolpath.

--stack-size=*KILOBYTES*
Set the size stack used to run threads to *KILOBYTES*. This value determines the stack size of the threads used to run jobs. These are the threads that execute the actions of the builders for the nodes that are out-of-date. Note that this option has no effect unless the **num_jobs** option, which corresponds to -j and --jobs, is larger than one. Using a stack size that is too small may cause stack overflow errors. This usually shows up as segmentation faults that cause scons to abort before building anything. Using a stack size that is too large will cause scons to use more memory than required and may slow down the entire build process.

The default value is to use a stack size of 256 kilobytes, which should be appropriate for most uses. You should not need to increase this value unless you encounter stack overflow errors.

-t, --touch
Ignored for compatibility with GNU **make**. (Touching a file to make it appear up-to-date is unnecessary when using **scons**.)

--taskmastertrace=*file*
Prints trace information to the specified *file* about how the internal Taskmaster object evaluates and controls the order in which Nodes are built. A file name of **-** may be used to specify the standard output.

-tree=*options*
Prints a tree of the dependencies after each top-level target is built. This prints out some or all of the tree, in various formats, depending on the *options* specified:

--tree=all
Print the entire dependency tree after each top-level target is built. This prints out the complete dependency tree, including implicit dependencies and ignored dependencies.

--tree=derived
Restricts the tree output to only derived (target) files, not source files.

--tree=status
Prints status information for each displayed node.

--tree=prune
Prunes the tree to avoid repeating dependency information for nodes that have already been displayed. Any node that has already been displayed will have its name printed in **[square brackets]**, as an indication that the dependencies for that node can be found by searching for the relevant output higher up in the tree.

Multiple options may be specified, separated by commas:

```
# Prints only derived files, with status information:
scons --tree=derived,status

# Prints all dependencies of target, with status information
# and pruning dependencies of already-visited Nodes:
scons --tree=all,prune,status target
```

-u, --up, --search-up
Walks up the directory structure until an *SConstruct ,* *Sconstruct* or *sconstruct* file is found, and uses that as the top of the directory tree. If no targets are specified on the command line, only targets at or below the current directory will be built.

-U
Works exactly the same way as the **-u** option except for the way default targets are handled. When this option is used and no targets are specified on the command line, all default targets that are defined in the SConscript(s) in the current directory are built, regardless of what directory the resultant targets end up in.

-v, --version
Print the **scons** version, copyright information, list of authors, and any other relevant information. Then exit.

-w, --print-directory
Print a message containing the working directory before and after other processing.

--no-print-directory
Turn off -w, even if it was turned on implicitly.

--warn=*type*, --warn=no-*type*

Enable or disable warnings. *type* specifies the type of warnings to be enabled or disabled:

--warn=all, --warn=no-all

Enables or disables all warnings.

--warn=cache-write-error, --warn=no-cache-write-error

Enables or disables warnings about errors trying to write a copy of a built file to a specified **CacheDir**(). These warnings are disabled by default.

--warn=corrupt-sconsign, --warn=no-corrupt-sconsign

Enables or disables warnings about unfamiliar signature data in **.sconsign** files. These warnings are enabled by default.

--warn=dependency, --warn=no-dependency

Enables or disables warnings about dependencies. These warnings are disabled by default.

--warn=deprecated, --warn=no-deprecated

Enables or disables all warnings about use of deprecated features. These warnings are enabled by default. Warnings for some specific deprecated features may be enabled or disabled individually; see below.

--warn=deprecated-copy, --warn=no-deprecated-copy Enables or disables warnings about use of the deprecated **env.Copy()** method.

--warn=deprecated-source-signatures, --warn=no-deprecated-source-signatures Enables or disables warnings about use of the deprecated SourceSignatures() function or **env.SourceSignatures()** method.

--warn=deprecated-target-signatures, --warn=no-deprecated-target-signatures Enables or disables warnings about use of the deprecated TargetSignatures() function or **env.TargetSignatures()** method.

--warn=duplicate-environment, --warn=no-duplicate-environment

Enables or disables warnings about missing SConscript files.

--warn=misleading-keywords, --warn=no-misleading-keywords

Enables or disables warnings about use of the misspelled keywords **targets** and **sources** when calling Builders. (Note the last **s** characters, the correct spellings are **target** and **source.)** These warnings are enabled by default.

--warn=missing-sconscript, --warn=no-missing-sconscript

Enables or disables warnings about attempts to specify a build of a target with two different construction environments that use the same action. These warnings are enabled by default.

--warn=no-md5-module, --warn=no-no-md5-module

Enables or disables warnings about the version of Python not having an MD5 checksum module available. These warnings are enabled by default.

--warn=no-metaclass-support, --warn=no-no-metaclass-support
Enables or disables warnings about the version of Python not supporting metaclasses when the **--debug=memoizer** option is used. These warnings are enabled by default.

--warn=no-object-count, --warn=no-no-object-count
Enables or disables warnings about the **--debug=object** feature not working when **scons** is run with the python **−O** option or from optimized Python (.pyo) modules.

--warn=no-parallel-support, --warn=no-no-parallel-support
Enables or disables warnings about the version of Python not being able to support parallel builds when the **-j** option is used. These warnings are enabled by default.

--warn=python-version, --warn=no-python-version
Enables or disables the warning about running SCons with a deprecated version of Python. These warnings are enabled by default.

--warn=reserved-variable, --warn=no-reserved-variable
Enables or disables warnings about attempts to set the reserved construction variable names **TARGET**, **TARGETS**, **SOURCE** or **SOURCES**. These warnings are disabled by default.

--warn=stack-size, --warn=no-stack-size
Enables or disables warnings about requests to set the stack size that could not be honored. These warnings are enabled by default.

-Y *repository*, --repository=*repository*, --srcdir=*repository*
Search the specified repository for any input and target files not found in the local directory hierarchy. Multiple **-Y** options may be specified, in which case the repositories are searched in the order specified.

CONFIGURATION FILE REFERENCE

Construction Environments

A construction environment is the basic means by which the SConscript files communicate build information to **scons**. A new construction environment is created using the **Environment** function:

```
env = Environment()
```

Variables, called *construction variables*, may be set in a construction environment either by specifyng them as keywords when the object is created or by assigning them a value after the object is created:

```
env = Environment(FOO = 'foo')
env['BAR'] = 'bar'
```

As a convenience, construction variables may also be set or modified by the *parse_flags* keyword argument, which applies the **ParseFlags** method (described below) to the argument value after all other processing is completed. This is useful either if the exact content of the flags is unknown (for example, read from a control file) or if the flags are distributed to a number of construction variables.

```
env = Environment(parse_flags = '-Iinclude -DEBUG -lm')
```

This example adds 'include' to **CPPPATH**, ´EBUG' to **CPPDEFINES**, and 'm' to **LIBS**.

By default, a new construction environment is initialized with a set of builder methods and construction variables that are appropriate for the current platform. An optional platform keyword argument may be used to specify that an environment should be initialized for a different platform:

```
env = Environment(platform = 'cygwin')
env = Environment(platform = 'os2')
env = Environment(platform = 'posix')
env = Environment(platform = 'win32')
```

Specifying a platform initializes the appropriate construction variables in the environment to use and generate file names with prefixes and suffixes appropriate for the platform.

Note that the **win32** platform adds the **SYSTEMDRIVE** and **SYSTEMROOT** variables from the user's external environment to the construction environment's **ENV** dictionary. This is so that any executed commands that use sockets to connect with other systems (such as fetching source files from external CVS repository specifications like **:pserver:anonymous@cvs.sourceforge.net:/cvsroot/scons**) will work on Windows systems.

The platform argument may be function or callable object, in which case the Environment() method will call the specified argument to update the new construction environment:

```
def my_platform(env):
   env['VAR'] = 'xyzzy'

env = Environment(platform = my_platform)
```

Additionally, a specific set of tools with which to initialize the environment may be specified as an optional keyword argument:

```
env = Environment(tools = ['msvc', 'lex'])
```

Non-built-in tools may be specified using the toolpath argument:

```
env = Environment(tools = ['default', 'foo'], toolpath = ['tools'])
```

This looks for a tool specification in tools/foo.py (as well as using the ordinary default tools for the platform). foo.py should have two functions: generate(env, **kw) and exists(env). The **generate()** function modifies the passed-in environment to set up variables so that the tool can be executed; it may use any keyword arguments that the user supplies (see below) to vary its initialization. The **exists()** function should return a true value if the tool is available. Tools in the toolpath are used before any of the built-in ones. For example, adding gcc.py to the toolpath would override the built-in gcc tool. Also note that the toolpath is stored in the environment for use by later calls to **Clone**() and **Tool**() methods:

```
base = Environment(toolpath=['custom_path'])
derived = base.Clone(tools=['custom_tool'])
derived.CustomBuilder()
```

The elements of the tools list may also be functions or callable objects, in which case the Environment() method will call the specified elements to update the new construction environment:

```
def my_tool(env):
   env['XYZZY'] = 'xyzzy'

env = Environment(tools = [my_tool])
```

The individual elements of the tools list may also themselves be two-element lists of the form (*toolname*, *kw_dict*). SCons searches for the *toolname* specification file as described above, and passes *kw_dict*, which must be a dictionary, as keyword arguments to the tool's **generate** function. The **generate** function can use the arguments to modify the tool's behavior by setting up the environment in different ways or otherwise changing its initialization.

```
# in tools/my_tool.py:
def generate(env, **kw):
  # Sets MY_TOOL to the value of keyword argument 'arg1' or 1.
  env['MY_TOOL'] = kw.get('arg1', '1')
def exists(env):
  return 1

# in SConstruct:
env = Environment(tools = ['default', ('my_tool', {'arg1': 'abc'})],
          toolpath=['tools'])
```

The tool definition (i.e. my_tool()) can use the PLATFORM variable from the environment it receives to customize the tool for different platforms.

If no tool list is specified, then SCons will auto-detect the installed tools using the PATH variable in the ENV construction variable and the platform name when the Environment is constructed. Changing the PATH variable after the Environment is constructed will not cause the tools to be redetected.

SCons supports the following tool specifications out of the box:

```
386asm
aixc++
aixcc
aixf77
aixlink
ar
as
bcc32
c++
cc
cvf
dmd
dvipdf
dvips
f77
f90
f95
fortran
g++
g77
gas
gcc
gfortran
gnulink
gs
hpc++
hpcc
hplink
```

icc
icl
ifl
ifort
ilink
ilink32
intelc
jar
javac
javah
latex
lex
link
linkloc
m4
masm
midl
mingw
mslib
mslink
msvc
msvs
mwcc
mwld
nasm
pdflatex
pdftex
qt
rmic
rpcgen
sgiar
sgic++
sgicc
sgilink
sunar
sunc++
suncc
sunf77
sunf90
sunf95
sunlink
swig
tar
tex
tlib
yacc
zip

Additionally, there is a "tool" named **default** which configures the environment with a default set of tools for the current platform.

On posix and cygwin platforms the GNU tools (e.g. gcc) are preferred by SCons, on Windows the Microsoft tools (e.g. msvc) followed by MinGW are preferred by SCons, and in OS/2 the IBM tools (e.g. icc) are preferred by SCons.

Builder Methods

Build rules are specified by calling a construction environment's builder methods. The arguments to the builder methods are **target** (a list of targets to be built, usually file names) and **source** (a list of sources to be built, usually file names).

Because long lists of file names can lead to a lot of quoting, **scons** supplies a **Split()** global function and a same-named environment method that split a single string into a list, separated on strings of white-space characters. (These are similar to the string.split() method from the standard Python library, but work even if the input isn't a string.)

Like all Python arguments, the target and source arguments to a builder method can be specified either with or without the "target" and "source" keywords. When the keywords are omitted, the target is first, followed by the source. The following are equivalent examples of calling the Program builder method:

```
env.Program('bar', ['bar.c', 'foo.c'])
env.Program('bar', Split('bar.c foo.c'))
env.Program('bar', env.Split('bar.c foo.c'))
env.Program(source =  ['bar.c', 'foo.c'], target = 'bar')
env.Program(target = 'bar', Split('bar.c foo.c'))
env.Program(target = 'bar', env.Split('bar.c foo.c'))
env.Program('bar', source = string.split('bar.c foo.c'))
```

Target and source file names that are not absolute path names (that is, do not begin with **/** on POSIX systems or **on Windows systems,** with or without an optional drive letter) are interpreted relative to the directory containing the **SConscript** file being read. An initial **#** (hash mark) on a path name means that the rest of the file name is interpreted relative to the directory containing the top-level **SConstruct** file, even if the **#** is followed by a directory separator character (slash or backslash).

Examples:

```
# The comments describing the targets that will be built
# assume these calls are in a SConscript file in the
# a subdirectory named "subdir".

# Builds the program "subdir/foo" from "subdir/foo.c":
env.Program('foo', 'foo.c')

# Builds the program "/tmp/bar" from "subdir/bar.c":
env.Program('/tmp/bar', 'bar.c')

# An initial '#' or '#/' are equivalent; the following
# calls build the programs "foo" and "bar" (in the
# top-level SConstruct directory) from "subdir/foo.c" and
# "subdir/bar.c", respectively:
env.Program('#foo', 'foo.c')
env.Program('#/bar', 'bar.c')

# Builds the program "other/foo" (relative to the top-level
# SConstruct directory) from "subdir/foo.c":
env.Program('#other/foo', 'foo.c')
```

When the target shares the same base name as the source and only the suffix varies, and if the builder method has a suffix defined for the target file type, then the target argument may be omitted completely, and **scons** will deduce the target file name from the source file name. The following examples all build the executable program **bar** (on POSIX systems) or **bar.exe** (on Windows systems) from the bar.c source file:

```
env.Program(target = 'bar', source = 'bar.c')
env.Program('bar', source = 'bar.c')
env.Program(source = 'bar.c')
env.Program('bar.c')
```

As a convenience, a **srcdir** keyword argument may be specified when calling a Builder. When specified, all source file strings that are not absolute paths will be interpreted relative to the specified **srcdir**. The following example will build the **build/prog** (or **build/prog.exe** on Windows) program from the files **src/f1.c** and **src/f2.c**:

```
env.Program('build/prog', ['f1.c', 'f2.c'], srcdir='src')
```

It is possible to override or add construction variables when calling a builder method by passing additional keyword arguments. These overridden or added variables will only be in effect when building the target, so they will not affect other parts of the build. For example, if you want to add additional libraries for just one program:

```
env.Program('hello', 'hello.c', LIBS=['gl', 'glut'])
```

or generate a shared library with a non-standard suffix:

```
env.SharedLibrary('word', 'word.cpp',
          SHLIBSUFFIX='.ocx',
          LIBSUFFIXES=['.ocx'])
```

(Note that both the $SHLIBSUFFIX and $LIBSUFFIXES variables must be set if you want SCons to search automatically for dependencies on the non-standard library names; see the descriptions of these variables, below, for more information.)

It is also possible to use the *parse_flags* keyword argument in an override:

```
env = Program('hello', 'hello.c', parse_flags = '-Iinclude -DEBUG -lm')
```

This example adds 'include' to **CPPPATH**, ´EBUG' to **CPPDEFINES**, and 'm' to **LIBS**.

Although the builder methods defined by **scons** are, in fact, methods of a construction environment object, they may also be called without an explicit environment:

```
Program('hello', 'hello.c')
SharedLibrary('word', 'word.cpp')
```

In this case, the methods are called internally using a default construction environment that consists of the tools and values that **scons** has determined are appropriate for the local system.

Builder methods that can be called without an explicit environment may be called from custom Python modules that you import into an SConscript file by adding the following to the Python module:

```
from SCons.Script import *
```

All builder methods return a list-like object containing Nodes that represent the target or targets that will be built. A *Node* is an internal SCons object which represents build targets or sources.

The returned Node-list object can be passed to other builder methods as source(s) or passed to any SCons function or method where a filename would normally be accepted. For example, if it were necessary to add a specific **-D** flag when compiling one specific object file:

```
bar_obj_list = env.StaticObject('bar.c', CPPDEFINES='-DBAR')
env.Program(source = ['foo.c', bar_obj_list, 'main.c'])
```

Using a Node in this way makes for a more portable build by avoiding having to specify a platform-specific object suffix when calling the Program() builder method.

Note that Builder calls will automatically "flatten" the source and target file lists, so it's all right to have the bar_obj list return by the StaticObject() call in the middle of the source file list. If you need to manipulate a list of lists returned by Builders directly using Python, you can either build the list by hand:

```
foo = Object('foo.c')
bar = Object('bar.c')
objects = ['begin.o'] + foo + ['middle.o'] + bar + ['end.o']
for object in objects:
    print str(object)
```

Or you can use the **Flatten**() function supplied by scons to create a list containing just the Nodes, which may be more convenient:

```
foo = Object('foo.c')
bar = Object('bar.c')
objects = Flatten(['begin.o', foo, 'middle.o', bar, 'end.o'])
for object in objects:
    print str(object)
```

Note also that because Builder calls return a list-like object, not an actual Python list, you should *not* use the Python **+=** operator to append Builder results to a Python list. Because the list and the object are different types, Python will not update the original list in place, but will instead create a new Node-list object containing the concatenation of the list elements and the Builder results. This will cause problems for any other Python variables in your SCons configuration that still hold on to a reference to the original list. Instead, use the Python **.extend()** method to make sure the list is updated in-place. Example:

```
object_files = []

# Do NOT use += as follows:
#
#    object_files += Object('bar.c')
#
# It will not update the object_files list in place.
#
# Instead, use the .extend() method:
object_files.extend(Object('bar.c'))
```

The path name for a Node's file may be used by passing the Node to the Python-builtin **str()** function:

```
bar_obj_list = env.StaticObject('bar.c', CPPDEFINES='-DBAR')
print "The path to bar_obj is:", str(bar_obj_list[0])
```

Note again that because the Builder call returns a list, we have to access the first element in the list

(bar_obj_list[0]) to get at the Node that actually represents the object file.

Builder calls support a **chdir** keyword argument that specifies that the Builder's action(s) should be executed after changing directory. If the **chdir** argument is a string or a directory Node, scons will change to the specified directory. If the **chdir** is not a string or Node and is non-zero, then scons will change to the target file's directory.

```
# scons will change to the "sub" subdirectory
# before executing the "cp" command.
env.Command('sub/dir/foo.out', 'sub/dir/foo.in',
        "cp dir/foo.in dir/foo.out",
        chdir='sub')

# Because chdir is not a string, scons will change to the
# target's directory ("sub/dir") before executing the
# "cp" command.
env.Command('sub/dir/foo.out', 'sub/dir/foo.in',
        "cp foo.in foo.out",
        chdir=1)
```

Note that scons will *not* automatically modify its expansion of construction variables like **$TARGET** and **$SOURCE** when using the chdir keyword argument--that is, the expanded file names will still be relative to the top-level SConstruct directory, and consequently incorrect relative to the chdir directory. If you use the chdir keyword argument, you will typically need to supply a different command line using expansions like **${TARGET.file}** and **${SOURCE.file}** to use just the filename portion of the targets and source.

scons provides the following builder methods:

CFile()

env.CFile()

Builds a C source file given a lex (**.l**) or yacc (**.y**) input file. The suffix specified by the $CFILE-SUFFIX construction variable (**.c** by default) is automatically added to the target if it is not already present. Example:

```
# builds foo.c
env.CFile(target = 'foo.c', source = 'foo.l')
# builds bar.c
env.CFile(target = 'bar', source = 'bar.y')
```

CXXFile()

env.CXXFile()

Builds a C++ source file given a lex (**.ll**) or yacc (**.yy**) input file. The suffix specified by the $CXXFILESUFFIX construction variable (**.cc** by default) is automatically added to the target if it is not already present. Example:

```
# builds foo.cc
env.CXXFile(target = 'foo.cc', source = 'foo.ll')
# builds bar.cc
env.CXXFile(target = 'bar', source = 'bar.yy')
```

DVI()

env.DVI()

Builds a **.dvi** file from a **.tex**, **.ltx** or **.latex** input file. If the source file suffix is **.tex**, **scons** will examine the contents of the file; if the string **ocumentclass** or **ocumentstyle** is found, the file is assumed to be a LaTeX file and the target is built by invoking the $LATEXCOM command line; otherwise, the $TEXCOM command line is used. If the file is a LaTeX file, the **DVI**() builder method will also examine the contents of the **.aux** file and invoke the $BIBTEX command line if the string **bibdata** is found, start $MAKEINDEX to generate an index if a **.ind** file is found and will examine the contents **.log** file and re-run the $LATEXCOM command if the log file says it is necessary.

The suffix **.dvi** (hard-coded within TeX itself) is automatically added to the target if it is not already present. Examples:

```
# builds from aaa.tex
env.DVI(target = 'aaa.dvi', source = 'aaa.tex')
# builds bbb.dvi
env.DVI(target = 'bbb', source = 'bbb.ltx')
# builds from ccc.latex
env.DVI(target = 'ccc.dvi', source = 'ccc.latex')
```

Install()

env.Install()

Installs one or more source files or directories in the specified target, which must be a directory. The names of the specified source files or directories remain the same within the destination directory.

```
env.Install('/usr/local/bin', source = ['foo', 'bar'])
```

InstallAs()

env.InstallAs()

Installs one or more source files or directories to specific names, allowing changing a file or directory name as part of the installation. It is an error if the target and source arguments list different numbers of files or directories.

```
env.InstallAs(target = '/usr/local/bin/foo',
        source = 'foo_debug')
env.InstallAs(target = ['../lib/libfoo.a', '../lib/libbar.a'],
        source = ['libFOO.a', 'libBAR.a'])
```

Jar()

env.Jar()

Builds a Java archive (**.jar**) file from the specified list of sources. Any directories in the source list will be searched for **.class** files). Any **.java** files in the source list will be compiled to **.class** files by calling the **Java**() Builder.

If the $JARCHDIR value is set, the **jar** command will change to the specified directory using the **−C** option. If $JARCHDIR is not set explicitly, &SCons; will use the top of any subdirectory tree in which Java **.class** were built by the **Java**() Builder.

If the contents any of the source files begin with the string **Manifest-Version**, the file is assumed to be a manifest and is passed to the **jar** command with the **m** option set.

```
env.Jar(target = 'foo.jar', source = 'classes')
```

```
env.Jar(target = 'bar.jar',
        source = ['bar1.java', 'bar2.java'])
```

Java()

env.Java()

Builds one or more Java class files. The sources may be any combination of explicit **.java** files, or directory trees which will be scanned for **.java** files.

SCons will parse each source **.java** file to find the classes (including inner classes) defined within that file, and from that figure out the target **.class** files that will be created. The class files will be placed underneath the specified target directory.

SCons will also search each Java file for the Java package name, which it assumes can be found on a line beginning with the string **package** in the first column; the resulting **.class** files will be placed in a directory reflecting the specified package name. For example, the file **Foo.java** defining a single public *Foo* class and containing a package name of *sub.dir* will generate a corresponding **sub/dir/Foo.class** class file.

Example:

```
env.Java(target = 'classes', source = 'src')
env.Java(target = 'classes', source = ['src1', 'src2'])
env.Java(target = 'classes', source = ['File1.java', 'File2.java'])
```

JavaH()

env.JavaH()

Builds C header and source files for implementing Java native methods. The target can be either a directory in which the header files will be written, or a header file name which will contain all of the definitions. The source can be the names of **.class** files, the names of **.java** files to be compiled into **.class** files by calling the **Java**() builder method, or the objects returned from the **Java**() builder method.

If the construction variable $JAVACLASSDIR is set, either in the environment or in the call to the **JavaH**() builder method itself, then the value of the variable will be stripped from the beginning of any **.class** file names.

Examples:

```
# builds java_native.h
classes = env.Java(target = 'classdir', source = 'src')
env.JavaH(target = 'java_native.h', source = classes)

# builds include/package_foo.h and include/package_bar.h
env.JavaH(target = 'include',
          source = ['package/foo.class', 'package/bar.class'])

# builds export/foo.h and export/bar.h
env.JavaH(target = 'export',
          source = ['classes/foo.class', 'classes/bar.class'],
          JAVACLASSDIR = 'classes')
```

Library()

env.Library()
A synonym for the **StaticLibrary**() builder method.

LoadableModule()

env.LoadableModule()
On most systems, this is the same as **SharedLibrary**(). On Mac OS X (Darwin) platforms, this creates a loadable module bundle.

M4()

env.M4()
Builds an output file from an M4 input file. This uses a default $M4FLAGS value of **−E**, which considers all warnings to be fatal and stops on the first warning when using the GNU version of m4. Example:

```
env.M4(target = 'foo.c', source = 'foo.c.m4')
```

Moc()

env.Moc()
Builds an output file from a moc input file. Moc input files are either header files or cxx files. This builder is only available after using the tool 'qt'. See the $QTDIR variable for more information. Example:

```
env.Moc('foo.h') # generates moc_foo.cc
env.Moc('foo.cpp') # generates foo.moc
```

MSVSProject()

env.MSVSProject()
Builds a Microsoft Visual Studio project file, and by default builds a solution file as well.

This builds a Visual Studio project file, based on the version of Visual Studio that is configured (either the latest installed version, or the version specified by $MSVS_VERSION in the Environment constructor). For Visual Studio 6, it will generate a **.dsp** file. For Visual Studio 7 (.NET) and later versions, it will generate a **.vcproj** file.

By default, this also generates a solution file for the specified project, a **.dsw** file for Visual Studio 6 or a **.sln** file for Visual Studio 7 (.NET). This behavior may be disabled by specifying **auto_build_solution=0** when you call **MSVSProject**(), in which case you presumably want to build the solution file(s) by calling the **MSVSSolution**() Builder (see below).

It takes several lists of filenames to be placed into the project file. These are currently limited to **srcs**, **incs**, **localincs**, **resources**, and **misc**. These are pretty self-explanatory, but it should be noted that these lists are added to the $SOURCES construction variable as strings, NOT as SCons File Nodes. This is because they represent file names to be added to the project file, not the source files used to build the project file.

The above filename lists are all optional, although at least one must be specified for the resulting project file to be non-empty.

In addition to the above lists of values, the following values may be specified:

target: The name of the target **.dsp** or **.vcproj** file. The correct suffix for the version of Visual Studio must be used, but the $MSVSPROJECTSUFFIX construction variable will be defined to the correct value (see example below).

variant: The name of this particular variant. For Visual Studio 7 projects, this can also be a list of variant names. These are typically things like "Debug" or "Release", but really can be anything you want. For Visual Studio 7 projects, they may also specify a target platform separated from the variant name by a **|** (vertical pipe) character: **Debug|Xbox**. The default target platform is Win32. Multiple calls to **MSVSProject**() with different variants are allowed; all variants will be added to the project file with their appropriate build targets and sources.

buildtarget: An optional string, node, or list of strings or nodes (one per build variant), to tell the Visual Studio debugger what output target to use in what build variant. The number of **buildtarget** entries must match the number of **variant** entries.

runfile: The name of the file that Visual Studio 7 and later will run and debug. This appears as the value of the **Output** field in the resutling Visual Studio project file. If this is not specified, the default is the same as the specified **buildtarget** value.

Example usage:

```
barsrcs = ['bar.cpp'],
barincs = ['bar.h'],
barlocalincs = ['StdAfx.h']
barresources = ['bar.rc','resource.h']
barmisc = ['bar_readme.txt']

dll = env.SharedLibrary(target = 'bar.dll',
                source = barsrcs)

env.MSVSProject(target = 'Bar' + env['MSVSPROJECTSUFFIX'],
            srcs = barsrcs,
            incs = barincs,
            localincs = barlocalincs,
            resources = barresources,
            misc = barmisc,
            buildtarget = dll,
            variant = 'Release')
```

MSVSSolution()

env.MSVSSolution()

Builds a Microsoft Visual Studio solution file.

This builds a Visual Studio solution file, based on the version of Visual Studio that is configured (either the latest installed version, or the version specified by $MSVS_VERSION in the construction environment). For Visual Studio 6, it will generate a **.dsw** file. For Visual Studio 7 (.NET), it will generate a **.sln** file.

The following values must be specified:

target: The name of the target .dsw or .sln file. The correct suffix for the version of Visual Studio must be used, but the value $MSVSSOLUTIONSUFFIX will be defined to the correct value (see example below).

variant: The name of this particular variant, or a list of variant names (the latter is only supported for MSVS 7 solutions). These are typically things like "Debug" or "Release", but really can be anything you want. For MSVS 7 they may also specify target platform, like this "Debug|Xbox". Default platform is Win32.

projects: A list of project file names, or Project nodes returned by calls to the **MSVSProject**() Builder, to be placed into the solution file. It should be noted that these file names are NOT added to the $SOURCES environment variable in form of files, but rather as strings. This is because they represent file names to be added to the solution file, not the source files used to build the solution file.

(NOTE: Currently only one project is supported per solution.)

Example Usage:

```
env.MSVSSolution(target = 'Bar' + env['MSVSSOLUTIONSUFFIX'],
         projects = ['bar' + env['MSVSPROJECTSUFFIX']],
         variant = 'Release')
```

Object()

env.Object()

A synonym for the **StaticObject**() builder method.

Package()

env.Package()

Builds software distribution packages. Packages consist of files to install and packaging information. The former may be specified with the &source; parameter and may be left out, in which case the &FindInstalledFiles; function will collect all files that have an **Install**()or**InstallAs**()**Builder**attached.**If**the**⌖,**is not specified it will be deduced from additional information given to this Builder.

The packaging information is specified with the help of construction variables documented below. This information is called a tag to stress that some of them can also be attached to files with the &Tag; function. The mandatory ones will complain if they were not specified. They vary depending on chosen target packager.

The target packager may be selected with the "PACKAGETYPE" command line option or with the $PACKAGETYPE construction variable. Currently the following packagers available:

```
 * msi - Microsoft Installer
 * rpm - Redhat Package Manger
 * ipkg - Itsy Package Management System
 * tarbz2 - compressed tar
 * targz - compressed tar
 * zip - zip file
 * src_tarbz2 - compressed tar source
 * src_targz - compressed tar source
 * src_zip - zip file source
```

An updated list is always available under the "package_type" option when running "scons --help" on a project that has packaging activated.

```
env = Environment(tools=['default', 'packaging'])
env.Install('/bin/', 'my_program')
env.Package( NAME       = 'foo',
       VERSION     = '1.2.3',
```

```
        PACKAGEVERSION = 0,
        PACKAGETYPE    = 'rpm',
        LICENSE        = 'gpl',
        SUMMARY        = 'balalalalal',
        DESCRIPTION    = 'this should be really really long',
        X_RPM_GROUP    = 'Application/fu',
        SOURCE_URL     = 'http://foo.org/foo-1.2.3.tar.gz'
    )
```

PCH()

env.PCH()

Builds a Microsoft Visual C++ precompiled header. Calling this builder method returns a list of two targets: the PCH as the first element, and the object file as the second element. Normally the object file is ignored. This builder method is only provided when Microsoft Visual C++ is being used as the compiler. The PCH builder method is generally used in conjuction with the PCH construction variable to force object files to use the precompiled header:

```
env['PCH'] = env.PCH('StdAfx.cpp')[0]
```

PDF()

env.PDF()

Builds a **.pdf** file from a **.dvi** input file (or, by extension, a **.tex**, **.ltx**, or **.latex** input file). The suffix specified by the $PDFSUFFIX construction variable (**.pdf** by default) is added automatically to the target if it is not already present. Example:

```
# builds from aaa.tex
env.PDF(target = 'aaa.pdf', source = 'aaa.tex')
# builds bbb.pdf from bbb.dvi
env.PDF(target = 'bbb', source = 'bbb.dvi')
```

PostScript()

env.PostScript()

Builds a **.ps** file from a **.dvi** input file (or, by extension, a **.tex**, **.ltx**, or **.latex** input file). The suffix specified by the $PSSUFFIX construction variable (**.ps** by default) is added automatically to the target if it is not already present. Example:

```
# builds from aaa.tex
env.PostScript(target = 'aaa.ps', source = 'aaa.tex')
# builds bbb.ps from bbb.dvi
env.PostScript(target = 'bbb', source = 'bbb.dvi')
```

Program()

env.Program()

Builds an executable given one or more object files or C, C++, D, or Fortran source files. If any C, C++, D or Fortran source files are specified, then they will be automatically compiled to object files using the **Object**() builder method; see that builder method's description for a list of legal source file suffixes and how they are interpreted. The target executable file prefix (specified by the $PROGPREFIX construction variable; nothing by default) and suffix (specified by the $PROGSUFFIX construction variable; by default, **.exe** on Windows systems, nothing on POSIX systems) are automatically added to the target if not already present. Example:

```
env.Program(target = 'foo', source = ['foo.o', 'bar.c', 'baz.f'])
```

RES()

env.RES()

Builds a Microsoft Visual C++ resource file. This builder method is only provided when Microsoft Visual C++ or MinGW is being used as the compiler. The **.res** (or **.o** for MinGW) suffix is added to the target name if no other suffix is given. The source file is scanned for implicit dependencies as though it were a C file. Example:

```
env.RES('resource.rc')
```

RMIC()

env.RMIC()

Builds stub and skeleton class files for remote objects from Java **.class** files. The target is a directory relative to which the stub and skeleton class files will be written. The source can be the names of **.class** files, or the objects return from the **Java**() builder method.

If the construction variable $JAVACLASSDIR is set, either in the environment or in the call to the **RMIC**() builder method itself, then the value of the variable will be stripped from the beginning of any **.class** file names.

```
classes = env.Java(target = 'classdir', source = 'src')
env.RMIC(target = 'outdir1', source = classes)

env.RMIC(target = 'outdir2',
        source = ['package/foo.class', 'package/bar.class'])

env.RMIC(target = 'outdir3',
        source = ['classes/foo.class', 'classes/bar.class'],
        JAVACLASSDIR = 'classes')
```

RPCGenClient()

env.RPCGenClient()

Generates an RPC client stub (**_clnt.c**) file from a specified RPC (**.x**) source file. Because rpcgen only builds output files in the local directory, the command will be executed in the source file's directory by default.

```
# Builds src/rpcif_clnt.c
env.RPCGenClient('src/rpcif.x')
```

RPCGenHeader()

env.RPCGenHeader()

Generates an RPC header (**.h**) file from a specified RPC (**.x**) source file. Because rpcgen only builds output files in the local directory, the command will be executed in the source file's directory by default.

```
# Builds src/rpcif.h
env.RPCGenHeader('src/rpcif.x')
```

RPCGenService()

env.RPCGenService()

Generates an RPC server-skeleton (**_svc.c**) file from a specified RPC (**.x**) source file. Because rpcgen only builds output files in the local directory, the command will be executed in the source file's directory by default.

```
# Builds src/rpcif_svc.c
env.RPCGenClient('src/rpcif.x')
```

RPCGenXDR()

env.RPCGenXDR()

Generates an RPC XDR routine (**_xdr.c**) file from a specified RPC (**.x**) source file. Because rpcgen only builds output files in the local directory, the command will be executed in the source file's directory by default.

```
# Builds src/rpcif_xdr.c
env.RPCGenClient('src/rpcif.x')
```

SharedLibrary()

env.SharedLibrary()

Builds a shared library (**.so** on a POSIX system, **.dll** on Windows) given one or more object files or C, C++, D or Fortran source files. If any source files are given, then they will be automatically compiled to object files. The static library prefix and suffix (if any) are automatically added to the target. The target library file prefix (specified by the $SHLIBPREFIX construction variable; by default, **lib** on POSIX systems, nothing on Windows systems) and suffix (specified by the $SHLIBSUFFIX construction variable; by default, **.dll** on Windows systems, **.so** on POSIX systems) are automatically added to the target if not already present. Example:

```
env.SharedLibrary(target = 'bar', source = ['bar.c', 'foo.o'])
```

On Windows systems, the **SharedLibrary**() builder method will always build an import (**.lib**) library in addition to the shared (**.dll**) library, adding a **.lib** library with the same basename if there is not already a **.lib** file explicitly listed in the targets.

Any object files listed in the **source** must have been built for a shared library (that is, using the **SharedObject**() builder method). **scons** will raise an error if there is any mismatch.

On Windows systems, specifying **register=1** will cause the **.dll** to be registered after it is built using REGSVR32. The command that is run ("regsvr32" by default) is determined by $REGSVR construction variable, and the flags passed are determined by $REGSVRFLAGS. By default, $REGSVRFLAGS includes the **/s** option, to prevent dialogs from popping up and requiring user attention when it is run. If you change $REGSVRFLAGS, be sure to include the **/s** option. For example,

```
env.SharedLibrary(target = 'bar',
        source = ['bar.cxx', 'foo.obj'],
        register=1)
```

will register **bar.dll** as a COM object when it is done linking it.

SharedObject()

env.SharedObject()

Builds an object file for inclusion in a shared library. Source files must have one of the same set of extensions specified above for the **StaticObject**() builder method. On some platforms building a shared object requires additional compiler option (e.g. **−fPIC** for gcc) in addition to those needed to build a normal (static) object, but on some platforms there is no difference between a shared object and a normal (static) one. When there is a difference, SCons will only allow shared objects to be linked into a shared library, and will use a different suffix for shared objects. On platforms where there is no difference, SCons will allow both normal (static) and shared objects to be linked into a shared library, and will use the same suffix for shared and normal (static) objects. The target object file prefix (specified by the $SHOBJPREFIX construction variable; by default, the same as $OBJPREFIX) and suffix (specified by the $SHOBJSUFFIX construction variable) are automatically added to the target if not already present. Examples:

```
env.SharedObject(target = 'ddd', source = 'ddd.c')
env.SharedObject(target = 'eee.o', source = 'eee.cpp')
env.SharedObject(target = 'fff.obj', source = 'fff.for')
```

Note that the source files will be scanned according to the suffix mappings in the **SourceFileScanner** object. See the section "Scanner Objects," below, for a more information.

StaticLibrary()

env.StaticLibrary()

Builds a static library given one or more object files or C, C++, D or Fortran source files. If any source files are given, then they will be automatically compiled to object files. The static library prefix and suffix (if any) are automatically added to the target. The target library file prefix (specified by the $LIBPREFIX construction variable; by default, **lib** on POSIX systems, nothing on Windows systems) and suffix (specified by the $LIBSUFFIX construction variable; by default, **.lib** on Windows systems, **.a** on POSIX systems) are automatically added to the target if not already present. Example:

```
env.StaticLibrary(target = 'bar', source = ['bar.c', 'foo.o'])
```

Any object files listed in the **source** must have been built for a static library (that is, using the **StaticObject**() builder method). **scons** will raise an error if there is any mismatch.

StaticObject()

env.StaticObject()

Builds a static object file from one or more C, C++, D, or Fortran source files. Source files must have one of the following extensions:

```
  .asm    assembly language file
  .ASM    assembly language file
  .c      C file
  .C      Windows:  C file
          POSIX:  C++ file
  .cc     C++ file
  .cpp    C++ file
  .cxx    C++ file
  .cxx    C++ file
  .c++    C++ file
  .C++    C++ file
  .d      D file
  .f      Fortran file
  .F      Windows:  Fortran file
          POSIX:  Fortran file + C pre-processor
  .for    Fortran file
  .FOR    Fortran file
  .fpp    Fortran file + C pre-processor
  .FPP    Fortran file + C pre-processor
  .m      Object C file
  .mm     Object C++ file
  .s      assembly language file
  .S      Windows:  assembly language file
```

POSIX: assembly language file + C pre-processor
.spp assembly language file + C pre-processor
.SPP assembly language file + C pre-processor

The target object file prefix (specified by the $OBJPREFIX construction variable; nothing by default) and suffix (specified by the $OBJSUFFIX construction variable; **.obj** on Windows systems, **.o** on POSIX systems) are automatically added to the target if not already present. Examples:

```
env.StaticObject(target = 'aaa', source = 'aaa.c')
env.StaticObject(target = 'bbb.o', source = 'bbb.c++')
env.StaticObject(target = 'ccc.obj', source = 'ccc.f')
```

Note that the source files will be scanned according to the suffix mappings in **SourceFileScanner** object. See the section "Scanner Objects," below, for a more information.

Tar()

env.Tar()

Builds a tar archive of the specified files and/or directories. Unlike most builder methods, the **Tar**() builder method may be called multiple times for a given target; each additional call adds to the list of entries that will be built into the archive. Any source directories will be scanned for changes to any on-disk files, regardless of whether or not **scons** knows about them from other Builder or function calls.

```
env.Tar('src.tar', 'src')

# Create the stuff.tar file.
env.Tar('stuff', ['subdir1', 'subdir2'])
# Also add "another" to the stuff.tar file.
env.Tar('stuff', 'another')

# Set TARFLAGS to create a gzip-filtered archive.
env = Environment(TARFLAGS = '-c -z')
env.Tar('foo.tar.gz', 'foo')

# Also set the suffix to .tgz.
env = Environment(TARFLAGS = '-c -z',
                  TARSUFFIX = '.tgz')
env.Tar('foo')
```

TypeLibrary()

env.TypeLibrary()

Builds a Windows type library (**.tlb**) file from an input IDL file (**.idl**). In addition, it will build the associated inteface stub and proxy source files, naming them according to the base name of the **.idl** file. For example,

```
env.TypeLibrary(source="foo.idl")
```

Will create **foo.tlb**, **foo.h**, **foo_i.c**, **foo_p.c** and **foo_data.c** files.

Uic()

env.Uic()

Builds a header file, an implementation file and a moc file from an ui file. and returns the corresponding nodes in the above order. This builder is only available after using the tool 'qt'. Note:

you can specify **.ui** files directly as source files to the **Program**(), **Library**()and**SharedLibrary**()**builders** without using this builder. Using this builder lets you override the standard naming conventions (be careful: prefixes are always prepended to names of built files; if you don't want prefixes, you may set them to ``). See the $QTDIR variable for more information. Example:

```
env.Uic('foo.ui') # -> ['foo.h', 'uic_foo.cc', 'moc_foo.cc']
env.Uic(target = Split('include/foo.h gen/uicfoo.cc gen/mocfoo.cc'),
    source = 'foo.ui') # -> ['include/foo.h', 'gen/uicfoo.cc', 'gen/mocfoo.cc']
```

Zip()

env.Zip()

Builds a zip archive of the specified files and/or directories. Unlike most builder methods, the **Zip**() builder method may be called multiple times for a given target; each additional call adds to the list of entries that will be built into the archive. Any source directories will be scanned for changes to any on-disk files, regardless of whether or not **scons** knows about them from other Builder or function calls.

```
env.Zip('src.zip', 'src')

# Create the stuff.zip file.
env.Zip('stuff', ['subdir1', 'subdir2'])
# Also add "another" to the stuff.tar file.
env.Zip('stuff', 'another')
```

All targets of builder methods automatically depend on their sources. An explicit dependency can be specified using the **Depends** method of a construction environment (see below).

In addition, **scons** automatically scans source files for various programming languages, so the dependencies do not need to be specified explicitly. By default, SCons can C source files, C++ source files, Fortran source files with **.F** (POSIX systems only), **.fpp,** or **.FPP** file extensions, and assembly language files with **.S** (POSIX systems only), **.spp,** or **.SPP** files extensions for C preprocessor dependencies. SCons also has default support for scanning D source files, You can also write your own Scanners to add support for additional source file types. These can be added to the default Scanner object used by the **Object**() **StaticObject**() and **SharedObject**() Builders by adding them to the **SourceFileScanner** object as follows:

See the section "Scanner Objects," below, for a more information about defining your own Scanner objects.

Methods and Functions to Do Things

In addition to Builder methods, **scons** provides a number of other construction environment methods and global functions to manipulate the build configuration.

Usually, a construction environment method and global function with the same name both exist so that you don't have to remember whether to a specific bit of functionality must be called with or without a construction environment. In the following list, if you call something as a global function it looks like:

Function(*arguments*)

and if you call something through a construction environment it looks like:

env.Function(*arguments*)

If you can call the functionality in both ways, then both forms are listed.

Global functions may be called from custom Python modules that you import into an SConscript file by adding the following to the Python module:

```
from SCons.Script import *
```

Except where otherwise noted, the same-named construction environment method and global function provide the exact same functionality. The only difference is that, where appropriate, calling the functionality through a construction environment will substitute construction variables into any supplied strings. For example:

```
env = Environment(FOO = 'foo')
Default('$FOO')
env.Default('$FOO')
```

In the above example, the first call to the global **Default()** function will actually add a target named **$FOO** to the list of default targets, while the second call to the **env.Default()** construction environment method will expand the value and add a target named **foo** to the list of default targets. For more on construction variable expansion, see the next section on construction variables.

Construction environment methods and global functions supported by **scons** include:

Action(*action*, [*strfunction*, *varlist*])

env.Action(*action*, [*strfunction*, *varlist*])

Creates an Action object for the specified *action*. See the section "Action Objects," below, for a complete explanation of the arguments and behavior.

Note that the **env.Action**() form of the invocation will expand construction variables in any arguments strings, including the *action* argument, at the time it is called using the construction variables in the **env** construction environment through which **env.Action**() was called. The **Action**() form delays all variable expansion until the Action object is actually used.

AddMethod(*object,*function*, [*name*])*

env.AddMethod(*function*, [*name*])

When called with the **AddMethod**() form, adds the specified *function* to the specified *object* as the specified method *name*. When called with the **env.AddMethod**() form, adds the specified *function* to the construction environment *env* as the specified method *name*. In both cases, if *name* is omitted or **None**, the name of the specified *function* itself is used for the method name.

Examples:

```
# Note that the first argument to the function to
# be attached as a method must be the object through
# which the method will be called; the Python
# convention is to call it 'self'.
def my_method(self, arg):
    print "my_method() got", arg

# Use the global AddMethod() function to add a method
# to the Environment class.  This
AddMethod(Environment, my_method)
env = Environment()
env.my_method('arg')

# Add the function as a method, using the function
# name for the method call.
env = Environment()
env.AddMethod(my_method, 'other_method_name')
env.other_method_name('another arg')
```

AddOption(*arguments*)

This function adds a new command-line option to be recognized. The specified *arguments* are the same as supported by the standard Python **optparse.add_option**() method (with a few additional capabilities noted below); see the documentation for **optparse** for a thorough discussion of its option-processing capabities. (Note that although the **optparse** module was not a standard module until Python 2.3, **scons** contains a compatible version of the module that is used to provide identical functionality when run by earlier Python versions.)

In addition to the arguments and values supported by the **optparse.add_option ()** method, the SCons **AddOption**() function allows you to set the **nargs** keyword value to **'?'** (a string with just the question mark) to indicate that the specified long option(s) take(s) an *optional* argument. When **nargs = '?'** is passed to the **AddOption**() function, the **const** keyword argument may be used to supply the "default" value that should be used when the option is specified on the command line without an explicit argument.

If no **default=** keyword argument is supplied when calling **AddOption**(), the option will have a default value of **None**.

Once a new command-line option has been added with **AddOption**(), the option value may be accessed using **GetOption**() or **env.GetOption**(). The value may also be set, using **SetOption**() or **env.SetOption**(), if conditions in a **SConscript** require overriding any default value. Note, however, that a value specified on the command line will *always* override a value set by any SConscript file.

Any specified **help=** strings for the new option(s) will be displayed by the **-H** or **-h** options (the latter only if no other help text is specified in the SConscript files). The help text for the local options specified by **AddOption**() will appear below the SCons options themselves, under a separate **Local Options** heading. The options will appear in the help text in the order in which the **AddOption**() calls occur.

Example:

```
AddOption('--prefix',
        dest='prefix',
        nargs=1, type='string',
        action='store',
        metavar='DIR',
        help='installation prefix')
env = Environment(PREFIX = GetOption('prefix'))
```

AddPostAction(*target*, *action*)

env.AddPostAction(*target*, *action*)

Arranges for the specified *action* to be performed after the specified *target* has been built. The specified action(s) may be an Action object, or anything that can be converted into an Action object (see below).

AddPreAction(*target*, *action*)

env.AddPreAction(*target*, *action*)

Arranges for the specified *action* to be performed before the specified *target* is built. The specified action(s) may be an Action object, or anything that can be converted into an Action object (see below).

Alias(*alias*, [*targets*, [*action*]])

env.Alias(*alias*, [*targets*, [*action*]])

Creates one or more phony targets that expand to one or more other targets. An optional *action* (command) or list of actions can be specified that will be executed whenever the any of the alias targets are out-of-date. Returns the Node object representing the alias, which exists outside of any file system. This Node object, or the alias name, may be used as a dependency of any other target, including another alias. **Alias** can be called multiple times for the same alias to add additional targets to the alias, or additional actions to the list for this alias.

Examples:

```
Alias('install')
Alias('install', '/usr/bin')
Alias(['install', 'install-lib'], '/usr/local/lib')

env.Alias('install', ['/usr/local/bin', '/usr/local/lib'])
env.Alias('install', ['/usr/local/man'])

env.Alias('update', ['file1', 'file2'], "update_database $SOURCES")
```

AllowSubstExceptions([*exception*, ...])

Specifies the exceptions that will be allowed when expanding construction variables. By default, any construction variable expansions that generate a **NameError** or **IndexError** exception will expand to a **''** (a null string) and not cause scons to fail. All exceptions not in the specified list will generate an error message and terminate processing.

If **AllowSubstExceptions** is called multiple times, each call completely overwrites the previous list of allowed exceptions.

Example:

```
# Requires that all construction variable names exist.
# (You may wish to do this if you want to enforce strictly
# that all construction variables must be defined before use.)
AllowSubstExceptions()

# Also allow a string containing a zero-division expansion
# like '${1 / 0}' to evalute to ''.
AllowSubstExceptions(IndexError, NameError, ZeroDivisionError)
```

AlwaysBuild(*target*, ...)

env.AlwaysBuild(*target*, ...)

Marks each given *target* so that it is always assumed to be out of date, and will always be rebuilt if needed. Note, however, that **AlwaysBuild**() does not add its target(s) to the default target list, so the targets will only be built if they are specified on the command line, or are a dependent of a target specified on the command line--but they will *always* be built if so specified. Multiple targets can be passed in to a single call to **AlwaysBuild**().

env.Append(*key*=*val*, [...])

Appends the specified keyword arguments to the end of construction variables in the environment. If the Environment does not have the specified construction variable, it is simply added to the environment. If the values of the construction variable and the keyword argument are the same type, then the two values will be simply added together. Otherwise, the construction variable and the value of the keyword argument are both coerced to lists, and the lists are added together. (See also the Prepend method, below.)

Example:

```
env.Append(CCFLAGS = ' -g', FOO = ['foo.yyy'])
```

env.AppendENVPath(*name*, *newpath*, [*envname*, *sep*])
This appends new path elements to the given path in the specified external environment (**ENV** by default). This will only add any particular path once (leaving the last one it encounters and ignoring the rest, to preserve path order), and to help assure this, will normalize all paths (using **os.path.normpath** and **os.path.normcase**). This can also handle the case where the given old path variable is a list instead of a string, in which case a list will be returned instead of a string.

Example:

```
print 'before:',env['ENV']['INCLUDE']
include_path = '/foo/bar:/foo'
env.AppendENVPath('INCLUDE', include_path)
print 'after:',env['ENV']['INCLUDE']

yields:
before: /foo:/biz
after: /biz:/foo/bar:/foo
```

env.AppendUnique(*key*=*val*, [...])
Appends the specified keyword arguments to the end of construction variables in the environment. If the Environment does not have the specified construction variable, it is simply added to the environment. If the construction variable being appended to is a list, then any value(s) that already exist in the construction variable will *not* be added again to the list.

Example:

```
env.AppendUnique(CCFLAGS = '-g', FOO = ['foo.yyy'])
```

env.BitKeeper()
A factory function that returns a Builder object to be used to fetch source files using BitKeeper. The returned Builder is intended to be passed to the **SourceCode** function.

Example:

```
env.SourceCode('.', env.BitKeeper())
```

BuildDir(*build_dir*, *src_dir*, [*duplicate*])

env.BuildDir(*build_dir*, *src_dir*, [*duplicate*])
Synonyms for **VariantDir**() and **env.VariantDir**(). The *build_dir* argument becomes the *variant_dir* argument of **VariantDir**() or **env.VariantDir**(). (This will be officially deprecated some day.)

Builder(*action*, [*arguments*])

env.Builder(*action*, [*arguments*])

Creates a Builder object for the specified *action*. See the section "Builder Objects," below, for a complete explanation of the arguments and behavior.

Note that the **env.Builder**() form of the invocation will expand construction variables in any arguments strings, including the *action* argument, at the time it is called using the construction variables in the **env** construction environment through which **env.Builder**() was called. The **Builder**() form delays all variable expansion until after the Builder object is actually called.

CacheDir(*cache_dir*)

env.CacheDir(*cache_dir*)

Specifies that **scons** will maintain a cache of derived files in *cache_dir* . The derived files in the cache will be shared among all the builds using the same **CacheDir**() call. Specifying a *cache_dir* of **None** disables derived file caching.

Calling **env.CacheDir**() will only affect targets built through the specified construction environment. Calling **CacheDir**() sets a global default that will be used by all targets built through construction environments that do *not* have an **env.CacheDir**() specified.

When a **CacheDir**() is being used and **scons** finds a derived file that needs to be rebuilt, it will first look in the cache to see if a derived file has already been built from identical input files and an identical build action (as incorporated into the MD5 build signature). If so, **scons** will retrieve the file from the cache. If the derived file is not present in the cache, **scons** will rebuild it and then place a copy of the built file in the cache (identified by its MD5 build signature), so that it may be retrieved by other builds that need to build the same derived file from identical inputs.

Use of a specified **CacheDir()** may be disabled for any invocation by using the **--cache-disable** option.

If the **--cache-force** option is used, **scons** will place a copy of *all* derived files in the cache, even if they already existed and were not built by this invocation. This is useful to populate a cache the first time **CacheDir**() is added to a build, or after using the **--cache-disable** option.

When using **CacheDir**(), **scons** will report, "Retrieved ‘file’ from cache," unless the **--cache-show** option is being used. When the **--cache-show** option is used, **scons** will print the action that *would* have been used to build the file, without any indication that the file was actually retrieved from the cache. This is useful to generate build logs that are equivalent regardless of whether a given derived file has been built in-place or retrieved from the cache.

The **NoCache**() method can be used to disable caching of specific files. This can be useful if inputs and/or outputs of some tool are impossible to predict or prohibitively large.

Clean(*targets*, *files_or_dirs*)

env.Clean(*targets*, *files_or_dirs*)

This specifies a list of files or directories which should be removed whenever the targets are specified with the **-c** command line option. The specified targets may be a list or an individual target. Multiple calls to **Clean**() are legal, and create new targets or add files and directories to the clean list for the specified targets.

Multiple files or directories should be specified either as separate arguments to the **Clean**() method, or as a list. **Clean**() will also accept the return value of any of the construction environment Builder methods. Examples:

The related **NoClean**() function overrides calling **Clean**() for the same target, and any targets passed to both functions will *not* be removed by the **-c** option.

Examples:

```
Clean('foo', ['bar', 'baz'])
Clean('dist', env.Program('hello', 'hello.c'))
Clean(['foo', 'bar'], 'something_else_to_clean')
```

Command(*target*, *source*, *action*, [*key*=*val*, ...])

env.Command(*target*, *source*, *action*, [*key*=*val*, ...])

Executes a specific action (or list of actions) to build a target file or files. This is more convenient than defining a separate Builder object for a single special-case build.

As a special case, the **source_scanner** keyword argument can be used to specify a Scanner object that will be used to scan the sources. (The global **DirScanner** object can be used if any of the sources will be directories that must be scanned on-disk for changes to files that aren't already specified in other Builder of function calls.)

Any other keyword arguments specified override any same-named existing construction variables.

An action can be an external command, specified as a string, or a callable Python object; see "Action Objects," below, for more complete information. Also note that a string specifying an external command may be preceded by an @ (at-sign) to suppress printing the command in question, or by a – (hyphen) to ignore the exit status of the external command.

Examples:

```
env.Command('foo.out', 'foo.in',
        "$FOO_BUILD < $SOURCES > $TARGET")

env.Command('bar.out', 'bar.in',
        ["rm -f $TARGET",
         "$BAR_BUILD < $SOURCES > $TARGET"],
        ENV = {'PATH' : '/usr/local/bin/'})

def rename(env, target, source):
    import os
    os.rename('.tmp', str(target[0]))

env.Command('baz.out', 'baz.in',
        ["$BAZ_BUILD < $SOURCES > .tmp",
            rename ])
```

Note that the **Command**() function will usually assume, by default, that the specified targets and/or sources are Files, if no other part of the configuration identifies what type of entry it is. If necessary, you can explicitly specify that targets or source nodes should be treated as directoriese by using the **Dir**() or **env.Dir**() functions.

Examples:

```
env.Command('ddd.list', Dir('ddd'), 'ls -l $SOURCE > $TARGET')

env['DISTDIR'] = 'destination/directory'
env.Command(env.Dir('$DISTDIR')), None, make_distdir)
```

(Also note that SCons will usually automatically create any directory necessary to hold a target file, so you normally don't need to create directories by hand.)

Configure(*env*, [*custom_tests*, *conf_dir*, *log_file*, *config_h*])

env.Configure([*custom_tests*, *conf_dir*, *log_file*, *config_h*])
Creates a Configure object for integrated functionality similar to GNU autoconf. See the section "Configure Contexts," below, for a complete explanation of the arguments and behavior.

env.Clone([*key*=*val*, ...])
Return a separate copy of a construction environment. If there are any keyword arguments specified, they are added to the returned copy, overwriting any existing values for the keywords.

Example:

```
env2 = env.Clone()
env3 = env.Clone(CCFLAGS = '-g')
```

Additionally, a list of tools and a toolpath may be specified, as in the Environment constructor:

```
def MyTool(env): env['FOO'] = 'bar'
env4 = env.Clone(tools = ['msvc', MyTool])
```

The *parse_flags* keyword argument is also recognized:

```
# create an environment for compiling programs that use wxWidgets
wx_env = env.Clone(parse_flags = '!wx-config --cflags --cxxflags')
```

env.Copy([*key*=*val*, ...])
A now-deprecated synonym for **env.Clone()**.

env.CVS(*repository*, *module*)
A factory function that returns a Builder object to be used to fetch source files from the specified CVS *repository*. The returned Builder is intended to be passed to the **SourceCode** function.

The optional specified *module* will be added to the beginning of all repository path names; this can be used, in essence, to strip initial directory names from the repository path names, so that you only have to replicate part of the repository directory hierarchy in your local build directory.

Examples:

```
# Will fetch foo/bar/src.c
# from /usr/local/CVSROOT/foo/bar/src.c.
env.SourceCode('.', env.CVS('/usr/local/CVSROOT'))

# Will fetch bar/src.c
# from /usr/local/CVSROOT/foo/bar/src.c.
env.SourceCode('.', env.CVS('/usr/local/CVSROOT', 'foo'))
```

```
# Will fetch src.c
# from /usr/local/CVSROOT/foo/bar/src.c.
env.SourceCode('.', env.CVS('/usr/local/CVSROOT', 'foo/bar'))
```

Decider(*function*)

env.Decider(*function*)

Specifies that all up-to-date decisions for targets built through this construction environment will be handled by the specified *function*. The *function* can be one of the following strings that specify the type of decision function to be performed:

timestamp-newer Specifies that a target shall be considered out of date and rebuilt if the dependency's timestamp is newer than the target file's timestamp. This is the behavior of the classic Make utility, and **make** can be used a synonym for **timestamp-newer**.

timestamp-match Specifies that a target shall be considered out of date and rebuilt if the dependency's timestamp is different than the timestamp recorded the last time the target was built. This provides behavior very similar to the classic Make utility (in particular, files are not opened up so that their contents can be checksummed) except that the target will also be rebuilt if a dependency file has been restored to a version with an *earlier* timestamp, such as can happen when restoring files from backup archives.

MD5 Specifies that a target shall be considered out of date and rebuilt if the dependency's content has changed sine the last time the target was built, as determined be performing an MD5 checksum on the dependency's contents and comparing it to the checksum recorded the last time the target was built. **content** can be used as a synonym for **MD5**.

MD5-timestamp Specifies that a target shall be considered out of date and rebuilt if the dependency's content has changed sine the last time the target was built, except that dependencies with a timestamp that matches the last time the target was rebuilt will be assumed to be up-to-date and *not* rebuilt. This provides behavior very similar to the **MD5** behavior of always checksumming file contents, with an optimization of not checking the contents of files whose timestamps haven't changed. The drawback is that SCons will *not* detect if a file's content has changed but its timestamp is the same, as might happen in an automated script that runs a build, updates a file, and runs the build again, all within a single second.

Examples:

```
# Use exact timestamp matches by default.
Decider('timestamp-match')

# Use MD5 content signatures for any targets built
# with the attached construction environment.
env.Decider('content')
```

In addition to the above already-available functions, the *function* argument may be an actual Python function that takes the following three arguments:

dependency

The Node (file) which should cause the *target* to be rebuilt if it has "changed" since the last tme *target was built.*

target The Node (file) being built. In the normal case, this is what should get rebuilt if the *dependency* has "changed."

prev_ni Stored information about the state of the *dependency* the last time the *target* was built. This can be consulted to match various file characteristics such as the timestamp, size, or content signature.

The *function* should return a **True** (non-zero) value if the *dependency* has "changed" since the last time the *target* was built (indicating that the target *should* be rebuilt), and **False** (zero) otherwise (indicating that the target should *not* be rebuilt). Note that the decision can be made using whatever criteria are appopriate. Ignoring some or all of the function arguments is perfectly normal.

Example:

```
def my_decider(dependency, target, prev_ni):
    return not os.path.exists(str(target))

env.Decider(my_decider)
```

Default(*targets*)

env.Default(*targets*)

This specifies a list of default targets, which will be built by **scons** if no explicit targets are given on the command line. Multiple calls to **Default**() are legal, and add to the list of default targets.

Multiple targets should be specified as separate arguments to the **Default**() method, or as a list. **Default**() will also accept the Node returned by any of a construction environment's builder methods.

Examples:

```
Default('foo', 'bar', 'baz')
env.Default(['a', 'b', 'c'])
hello = env.Program('hello', 'hello.c')
env.Default(hello)
```

An argument to **Default**() of **None** will clear all default targets. Later calls to **Default**() will add to the (now empty) default-target list like normal.

The current list of targets added using the **Default**() function or method is available in the **DEFAULT_TARGETS** list; see below.

DefaultEnvironment([*args*])

Creates and returns a default construction environment object. This construction environment is used internally by SCons in order to execute many of the global functions in this list, and to fetch source files transparently from source code management systems.

Depends(*target*, *dependency*)

env.Depends(*target*, *dependency*)

Specifies an explicit dependency; the target file(s) will be rebuilt whenever the dependency file(s) has changed. This should only be necessary for cases where the dependency is not caught by a Scanner for the file.

Example:

```
env.Depends('foo', 'other-input-file-for-foo')
```

env.Dictionary([*vars*])

Returns a dictionary object containing copies of all of the construction variables in the environment. If there are any variable names specified, only the specified construction variables are returned in the dictionary.

Example:

```
dict = env.Dictionary()
cc_dict = env.Dictionary('CC', 'CCFLAGS', 'CCCOM')
```

Dir(*name*, [*directory*])

env.Dir(*name*, [*directory*])

This returns a Directory Node, an object that represents the specified directory *name*. *name* can be a relative or absolute path. *directory* is an optional directory that will be used as the parent directory. If no *directory* is specified, the current script's directory is used as the parent.

If *name* is a list, SCons returns a list of Dir nodes. Construction variables are expanded in *name*.

Directory Nodes can be used anywhere you would supply a string as a directory name to a Builder method or function. Directory Nodes have attributes and methods that are useful in many situations; see "File and Directory Nodes," below.

env.Dump([*key*])

Returns a pretty printable representation of the environment. *key*, if not *None*, should be a string containing the name of the variable of interest.

This SConstruct:

```
env=Environment()
print env.Dump('CCCOM')
```

will print:

```
env=Environment()
print env.Dump()
```

will print:

```
{ 'AR': 'ar',
  'ARCOM': '$AR $ARFLAGS $TARGET $SOURCES0RANLIB $RANLIBFLAGS $TARGET',
  'ARFLAGS': ['r'],
  'AS': 'as',
  'ASCOM': '$AS $ASFLAGS -o $TARGET $SOURCES',
  'ASFLAGS': [],
  ...
```

EnsurePythonVersion(*major*, *minor*)

env.EnsurePythonVersion(*major*, *minor*)

Ensure that the Python version is at least *major*.*minor*. This function will print out an error message and exit SCons with a non-zero exit code if the actual Python version is not late enough.

Example:

```
EnsurePythonVersion(2,2)
```

EnsureSConsVersion(*major*, *minor*, [*revision*])

env.EnsureSConsVersion(*major*, *minor*, [*revision*])

Ensure that the SCons version is at least *major.minor*, or *major.minor.revision*. if *revision* is specified. This function will print out an error message and exit SCons with a non-zero exit code if the actual SCons version is not late enough.

Examples:

```
EnsureSConsVersion(0,14)

EnsureSConsVersion(0,96,90)
```

Environment([*key*=*value*, ...])

env.Environment([*key*=*value*, ...])

Return a new construction environment initialized with the specified *key*=*value* pairs.

Execute(*action*, [*strfunction*, *varlist*])

env.Execute(*action*, [*strfunction*, *varlist*])

Executes an Action object. The specified *action* may be an Action object (see the section "Action Objects," below, for a complete explanation of the arguments and behavior), or it may be a command-line string, list of commands, or executable Python function, each of which will be converted into an Action object and then executed. The exit value of the command or return value of the Python function will be returned.

Exit([*value*])

env.Exit([*value*])

This tells **scons** to exit immediately with the specified *value*. A default exit value of **0** (zero) is used if no value is specified.

Export(*vars*)

env.Export(*vars*)

This tells **scons** to export a list of variables from the current SConscript file to all other SConscript files. The exported variables are kept in a global collection, so subsequent calls to **Export**() will over-write previous exports that have the same name. Multiple variable names can be passed to **Export**() as separate arguments or as a list. A dictionary can be used to map variables to a different name when exported. Both local variables and global variables can be exported.

Examples:

```
env = Environment()
# Make env available for all SConscript files to Import().
Export("env")

package = 'my_name'
# Make env and package available for all SConscript files:.
Export("env", "package")
```

```
# Make env and package available for all SConscript files:
Export(["env", "package"])

# Make env available using the name debug:.
Export({"debug":env})
```

Note that the **SConscript**() function supports an *exports* argument that makes it easier to to export a variable or set of variables to a single SConscript file. See the description of the **SConscript**() function, below.

File(*name*, [*directory*])

env.File(*name*, [*directory*])

This returns a File Node, an object that represents the specified file *name*. *name* can be a relative or absolute path. *directory* is an optional directory that will be used as the parent directory.

If *name* is a list, SCons returns a list of File nodes. Construction variables are expanded in *name*.

File Nodes can be used anywhere you would supply a string as a file name to a Builder method or function. File Nodes have attributes and methods that are useful in many situations; see "File and Directory Nodes," below.

FindFile(*file*, *dirs*)

env.FindFile(*file*, *dirs*)

Search for *file* in the path specified by *dirs*. *file* may be a list of file names or a single file name. In addition to searching for files that exist in the filesytem, this function also searches for derived files that have not yet been built.

Example:

```
foo = env.FindFile('foo', ['dir1', 'dir2'])
```

FindInstalledFiles()

env.FindInstalledFiles()

Returns the list of targets set up by the **Install()** or **InstallAs()** builders.

This function serves as a convenient method to select the contents of a binary package.

Example:

```
Install( '/bin', [ 'executable_a', 'executable_b' ] )

# will return the file node list
# [ '/bin/executable_a', '/bin/executable_b' ]
FindInstalledFiles()

Install( '/lib', [ 'some_library' ] )

# will return the file node list
# [ '/bin/executable_a', '/bin/executable_b', '/lib/some_library' ]
FindInstalledFiles()
```

FindSourceFiles(*node*='"."')

env.FindSourceFiles(*node*='"."')

Returns the list of nodes which serve as the source of the built files. It does so by inspecting the dependency tree starting at the optional argument **node** which defaults to the '"."'-node. It will then return all leaves of **node.** These are all children which have no further children.

This function is a convenient method to select the contents of a Source Package.

Example:

```
Program( 'src/main_a.c' )
Program( 'src/main_b.c' )
Program( 'main_c.c' )

# returns ['main_c.c', 'src/main_a.c', 'SConstruct', 'src/main_b.c']
FindSourceFiles()

# returns ['src/main_b.c', 'src/main_a.c' ]
FindSourceFiles( 'src' )
```

As you can see build support files (SConstruct in the above example) will also be returned by this function.

FindPathDirs(*variable*)

Returns a function (actually a callable Python object) intended to be used as the **path_function** of a Scanner object. The returned object will look up the specified *variable* in a construction environment and treat the construction variable's value as a list of directory paths that should be searched (like **CPPPATH**, **LIBPATH**, etc.).

Note that use of **FindPathDirs**() is generally preferable to writing your own **path_function** for the following reasons: 1) The returned list will contain all appropriate directories found in source trees (when **VariantDir**() is used) or in code repositories (when **Repository**() or the **−Y** option are used). 2) scons will identify expansions of *variable* that evaluate to the same list of directories as, in fact, the same list, and avoid re-scanning the directories for files, when possible.

Example:

```
def my_scan(node, env, path, arg):
   # Code to scan file contents goes here...
   return include_files

scanner = Scanner(name = 'myscanner',
              function = my_scan,
              path_function = FindPathDirs('MYPATH'))
```

Flatten(*sequence*)

env.Flatten(*sequence*)

Takes a sequence (that is, a Python list or tuple) that may contain nested sequences and returns a flattened list containing all of the individual elements in any sequence. This can be helpful for collecting the lists returned by calls to Builders; other Builders will automatically flatten lists specified as input, but direct Python manipulation of these lists does not.

Examples:

```
foo = Object('foo.c')
bar = Object('bar.c')

# Because 'foo' and 'bar' are lists returned by the Object() Builder,
# 'objects' will be a list containing nested lists:
objects = ['f1.o', foo, 'f2.o', bar, 'f3.o']

# Passing such a list to another Builder is all right because
# the Builder will flatten the list automatically:
Program(source = objects)

# If you need to manipulate the list directly using Python, you need to
# call Flatten() yourself, or otherwise handle nested lists:
for object in Flatten(objects):
    print str(object)
```

GetBuildFailures()

Returns a list of exceptions for the actions that failed while attempting to build targets. Each element in the returned list is a **BuildError** object with the following attributes that record various aspects of the build failure:

.node The node that was being built when the build failure occurred.

.status The numeric exit status returned by the command or Python function that failed when trying to build the specified Node.

.errstr The SCons error string describing the build failure. (This is often a generic message like "Error 2" to indicate that an executed command exited with a status of 2.)

.filename The name of the file or directory that actually caused the failure. This may be different from the **.node** attribute. For example, if an attempt to build a target named **sub/dir/target** fails because the **sub/dir** directory could not be created, then the **.node** attribute will be **sub/dir/target** but the **.filename** attribute will be **sub/dir**.

.executor The SCons Executor object for the target Node being built. This can be used to retrieve the construction environment used for the failed action.

.action The actual SCons Action object that failed. This will be one specific action out of the possible list of actions that would have been executed to build the target.

.command The actual expanded command that was executed and failed, after expansion of **$TARGET**, **$SOURCE**, and other construction variables.

Note that the **GetBuildFailures**() function will always return an empty list until any build failure has occurred, which means that **GetBuildFailures**() will always return an empty list while the **SConscript** files are being read. Its primary intended use is for functions that will be executed before SCons exits by passing them to the standard Python **atexit.register**() function. Example:

```
import atexit

def print_build_failures():
    from SCons.Script import GetBuildFailures
```

```
    for bf in GetBuildFailures():
        print "%s failed: %s" % (bf.node, bf.errstr)

atexit.register(print_build_failures)
```

GetBuildPath(*file*, [...])

env.GetBuildPath(*file*, [...])

Returns the **scons** path name (or names) for the specified *file* (or files). The specified *file* or files may be **scons** Nodes or strings representing path names.

GetLaunchDir()

env.GetLaunchDir()

Returns the absolute path name of the directory from which **scons** was initially invoked. This can be useful when using the **−u**, **−U** or **−D** options, which internally change to the directory in which the **SConstruct** file is found.

GetOption(*name*)

env.GetOption(*name*)

This function provides a way to query the value of SCons options set on scons command line (or set using the *SetOption*() function). The options supported are:

cache_debug
which corresponds to --cache-debug;

cache_disable
which corresponds to --cache-disable;

cache_force
which corresponds to --cache-force;

cache_show
which corresponds to --cache-show;

clean which corresponds to -c, --clean and --remove;

config
which corresponds to --config;

directory
which corresponds to -C and --directory;

diskcheck
which corresponds to --diskcheck

duplicate
which corresponds to --duplicate;

file which corresponds to -f, --file, --makefile and --sconstruct;

help which corresponds to -h and --help;

ignore_errors
which corresponds to --ignore-errors;

implicit_cache
which corresponds to --implicit-cache;

implicit_deps_changed
which corresponds to --implicit-deps-changed;

implicit_deps_unchanged
which corresponds to --implicit-deps-unchanged;

interactive
which corresponds to --interact and --interactive;

keep_going
which corresponds to -k and --keep-going;

max_drift
which corresponds to --max-drift;

no_exec
which corresponds to -n, --no-exec, --just-print, --dry-run and --recon;

no_site_dir
which corresponds to --no-site-dir;

num_jobs
which corresponds to -j and --jobs;

profile_file
which corresponds to --profile;

question
which corresponds to -q and --question;

random
which corresponds to --random;

repository
which corresponds to -Y, --repository and --srcdir;

silent which corresponds to -s, --silent and --quiet;

site_dir
which corresponds to --site-dir;

stack_size
which corresponds to --stack-size;

taskmastertrace_file
which corresponds to --taskmastertrace; and

warn which corresponds to --warn and --warning.

See the documentation for the corresponding command line object for information about each specific option.

Glob(*pattern*, [*ondisk*, *source*, *strings*])

env.Glob(*pattern*, [*ondisk*, *source*, *strings*])
Returns Nodes (or strings) that match the specified *pattern*, relative to the directory of the current **SConscript** file. The **env.Glob**() form performs string substition on *pattern* and returns whatever matches the resulting expanded pattern.

The specified *pattern* uses Unix shell style metacharacters for matching:

```
  *       matches everything
  ?       matches any single character
  [seq]   matches any character in seq
```

[!seq] matches any char not in seq

Character matches do *not* span directory separators.

The **Glob**() knows about repositories (see the **Repository**() function) and source directories (see the **VariantDir**() function) and returns a Node (or string, if so configured) in the local (SConscript) directory if matching Node is found anywhere in a corresponding repository or source directory.

The **ondisk** argument may be set to **False** (or any other non-true value) to disable the search for matches on disk, thereby only returning matches among already-configured File or Dir Nodes. The default behavior is to return corresponding Nodes for any on-disk matches found.

The **source** argument may be set to **True** (or any equivalent value) to specify that, when the local directory is a **VariantDir**(), the returned Nodes should be from the corresponding source directory, not the local directory.

The **strings** argument may be set to **True** (or any equivalent value) to have the **Glob**() function return strings, not Nodes, that represent the matched files or directories. The returned strings will be relative to the local (SConscript) directory. (Note that This may make it easier to perform arbitrary manipulation of file names, but if the returned strings are passed to a different **SConscript** file, any Node translation will be relative to the other **SConscript** directory, not the original **SConscript** directory.)

Example:

```
Program('foo', Glob('*.c'))
```

Help(*text*)

env.Help(*text*)

This specifies help text to be printed if the **-h** argument is given to **scons**. If **Help** is called multiple times, the text is appended together in the order that **Help** is called.

Ignore(*target*, *dependency*)

env.Ignore(*target*, *dependency*)

The specified dependency file(s) will be ignored when deciding if the target file(s) need to be rebuilt.

Examples:

```
env.Ignore('foo', 'foo.c')
env.Ignore('bar', ['bar1.h', 'bar2.h'])
```

Import(*vars*)

env.Import(*vars*)

This tells **scons** to import a list of variables into the current SConscript file. This will import variables that were exported with **Export**() or in the *exports* argument to **SConscript**(). Variables exported by **SConscript**() have precedence. Multiple variable names can be passed to **Import**() as separate arguments or as a list. The variable "*" can be used to import all variables.

Examples:

```
Import("env")
Import("env", "variable")
Import(["env", "variable"])
Import("*")
```

Literal(*string*)

env.Literal(*string*)

The specified *string* will be preserved as-is and not have construction variables expanded.

Local(*targets*)

env.Local(*targets*)

The specified *targets* will have copies made in the local tree, even if an already up-to-date copy exists in a repository. Returns a list of the target Node or Nodes.

env.MergeFlags(*arg*, [*unique*])

Merges the specified *arg* values to the construction envrionment's construction variables. If the *arg* argument is not a dictionary, it is converted to one by calling **env.ParseFlags()** on the argument before the values are merged. Note that *arg* must be a single value, so multiple strings must be passed in as a list, not as separate arguments to **env.MergeFlags**().

By default, duplicate values are eliminated; you can, however, specify **unique=0** to allow duplicate values to be added. When eliminating duplicate values, any construction variables that end with the string **PATH** keep the left-most unique value. All other construction variables keep the right-most unique value.

Examples:

```
# Add an optimization flag to $CCFLAGS.
env.MergeFlags('-O3')

# Combine the flags returned from running pkg-config with an optimization
# flag and merge the result into the construction variables.
env.MergeFlags(['!pkg-config gtk+-2.0 --cflags', '-O3'])

# Combine an optimization flag with the flags returned from running pkg-config
# twice and merge the result into the construction variables.
env.MergeFlags(['-O3',
         '!pkg-config gtk+-2.0 --cflags --libs',
         '!pkg-config libpng12 --cflags --libs'])
```

NoCache(*target*, ...)

env.NoCache(*target*, ...)

Specifies a list of files which should *not* be cached whenever the **CacheDir**() method has been activated. The specified targets may be a list or an individual target.

Multiple files should be specified either as separate arguments to the **NoCache**() method, or as a list. **NoCache**() will also accept the return value of any of the construction environment Builder methods.

Calling **NoCache**() on directories and other non-File Node types has no effect because only File Nodes are cached.

Examples:

```
NoCache('foo.elf')
NoCache(env.Program('hello', 'hello.c'))
```

NoClean(*target*, ...)

env.NoClean(*target*, ...)

Specifies a list of files or directories which should *not* be removed whenever the targets (or their dependencies) are specified with the **-c** command line option. The specified targets may be a list or an individual target. Multiple calls to **NoClean**() are legal, and prevent each specified target from being removed by calls to the **-c** option.

Multiple files or directories should be specified either as separate arguments to the **NoClean**() method, or as a list. **NoClean**() will also accept the return value of any of the construction environment Builder methods.

Calling **NoClean**() for a target overrides calling **Clean**() for the same target, and any targets passed to both functions will *not* be removed by the **-c** option.

Examples:

```
NoClean('foo.elf')
NoClean(env.Program('hello', 'hello.c'))
```

env.ParseConfig(*command*, [*function*, *unique*])

Calls the specified *function* to modify the environment as specified by the output of *command* . The default *function* is **env.MergeFlags**(), which expects the output of a typical **-config command* (for example, **gtk-config**) and adds the options to the appropriate construction variables. By default, duplicate values are not added to any construction variables; you can specify **unique=0** to allow duplicate values to be added.

Interpreted options and the construction variables they affect are as specified for the **env.ParseFlags**() method (which thie method calls). See that method's description, below, for a table of options and construction variables.

ParseDepends(*filename*, [*must_exist*, *only_one*])

env.ParseDepends(*filename*, [*must_exist*, *only_one*])

Parses the contents of the specified *filename* as a list of dependencies in the style of **Make** or **mkdep**, and explicitly establishes all of the listed dependencies.

By default, it is not an error if the specified *filename* does not exist. The optional *must_exist* argument may be set to a non-zero value to have scons throw an exception and generate an error if the file does not exist, or is otherwise inaccessible.

The optional *only_one* argument may be set to a non-zero value to have scons thrown an exception and generate an error if the file contains dependency information for more than one target. This can provide a small sanity check for files intended to be generated by, for example, the **gcc -M** flag, which should typically only write dependency information for one output file into a corresponding **.d** file.

The *filename* and all of the files listed therein will be interpreted relative to the directory of the *SConscript* file which calls the **ParseDepends** function.

env.ParseFlags(*flags*, ...)

Parses one or more strings containing typical command-line flags for GCC tool chains and returns a dictionary with the flag values separated into the appropriate SCons construction variables. This is intended as a companion to the **env.MergeFlags**() method, but allows for the values in the returned dictionary to be modified, if necessary, before merging them into the construction environment. (Note that **env.MergeFlags**() will call this method if its argument is not a dictionary, so it is usually not necessary to call **env.ParseFlags**() directly unless you want to manipulate the values.)

If the first character in any string is an exclamation mark (!), the rest of the string is executed as a command, and the output from the command is parsed as GCC tool chain command-line flags and added to the resulting dictionary.

Flag values are translated accordig to the prefix found, and added to the following construction variables:

```
-arch             CCFLAGS, LINKFLAGS
-D                CPPDEFINES
-framework        FRAMEWORKS
-frameworkdir=    FRAMEWORKPATH
-include          CCFLAGS
-isysroot         CCFLAGS, LINKFLAGS
-I                CPPPATH
-l                LIBS
-L                LIBPATH
-mno-cygwin       CCFLAGS, LINKFLAGS
-mwindows         LINKFLAGS
-pthread          CCFLAGS, LINKFLAGS
-std=             CFLAGS
-Wa,              ASFLAGS, CCFLAGS
-Wl,-rpath=       RPATH
-Wl,-R,           RPATH
-Wl,-R            RPATH
-Wl,              LINKFLAGS
-Wp,              CPPFLAGS
-                 CCFLAGS
+                 CCFLAGS, LINKFLAGS
```

Any other strings not associated with options are assumed to be the names of libraries and added to the **LIBS** construction variable.

Examples (all of which produce the same result):

```
dict = env.ParseFlags('-O2 -Dfoo -Dbar=1')
dict = env.ParseFlags('-O2', '-Dfoo', '-Dbar=1')
dict = env.ParseFlags(['-O2', '-Dfoo -Dbar=1'])
dict = env.ParseFlags('-O2', '!echo -Dfoo -Dbar=1')
```

env.Perforce()

A factory function that returns a Builder object to be used to fetch source files from the Perforce source code management system. The returned Builder is intended to be passed to the **SourceCode** function.

Example:

```
env.SourceCode('.', env.Perforce())
```

Perforce uses a number of external environment variables for its operation. Consequently, this function adds the following variables from the user's external environment to the construction environment's ENV dictionary: P4CHARSET, P4CLIENT, P4LANGUAGE, P4PASSWD, P4PORT, P4USER, SYSTEMROOT, USER, and USERNAME.

Platform(*string*)

Returns a callable object that can be used to initialize a construction environment using the platform keyword of the Environment() method.

Example:

```
env = Environment(platform = Platform('win32'))
```

env.Platform(*string*)

Applies the callable object for the specified platform *string* to the environment through which the method was called.

```
env.Platform('posix')
```

Note that the **win32** platform adds the **SYSTEMDRIVE** and **SYSTEMROOT** variables from the user's external environment to the construction environment's **ENV** dictionary. This is so that any executed commands that use sockets to connect with other systems (such as fetching source files from external CVS repository specifications like **:pserver:anonymous@cvs.sourceforge.net:/cvsroot/scons**) will work on Windows systems.

Progress(*callable*, [*interval*])

Progress(*string*, [*interval*, *file*, *overwrite*])

Progress(*list_of_strings*, [*interval*, *file*, *overwrite*])

Allows SCons to show progress made during the build by displaying a string or calling a function while evaluating Nodes (e.g. files).

If the first specified argument is a Python callable (a function or an object that has a **__call__**() method), the function will be called once every *interval* times a Node is evaluated. The callable will be passed the evaluated Node as its only argument. (For future compatibility, it's a good idea to also add ***args** and ****kw** as arguments to your function or method. This will prevent the code from breaking if SCons ever changes the interface to call the function with additional arguments in the future.)

An example of a simple custom progress function that prints a string containing the Node name every 10 Nodes:

```
def my_progress_function(node, *args, **kw):
    print 'Evaluating node %s!' % node
Progress(my_progress_function, interval=10)
```

A more complicated example of a custom progress display object that prints a string containing a count every 100 evaluated Nodes. Note the use of **\r** (a carriage return) at the end so that the string will overwrite itself on a display:

```
import sys
class ProgressCounter:
    count = 0
    def __call__(self, node, *args, **kw):
```

```
        self.count += 100
        sys.stderr.write('Evaluated %s nodes\r' % self.count)
Progress(ProgressCounter(), interval=100)
```

If the first argument **Progress**() is a string, the string will be displayed every *interval* evaluated Nodes. The default is to print the string on standard output; an alternate output stream may be specified with the **file=** argument. The following will print a series of dots on the error output, one dot for every 100 evaluated Nodes:

```
import sys
Progress('.', interval=100, file=sys.stderr)
```

If the string contains the verbatim substring **$TARGET,** it will be replaced with the Node. Note that, for performance reasons, this is *not* a regular SCons variable substition, so you can not use other variables or use curly braces. The following example will print the name of every evaluated Node, using a **\r** (carriage return) to cause each line to overwritten by the next line, and the **overwrite=** keyword argument to make sure the previously-printed file name is overwritten with blank spaces:

```
import sys
Progress('$TARGET\r', overwrite=True)
```

If the first argument to **Progress**() is a list of strings, then each string in the list will be displayed in rotating fashion every *interval* evaluated Nodes. This can be used to implement a "spinner" on the user's screen as follows:

```
Progress(['-\r', '\\\r', '|\r', '/\r'], interval=5)
```

Precious(*target*, ...)

env.Precious(*target*, ...)

Marks each given *target* as precious so it is not deleted before it is rebuilt. Normally **scons** deletes a target before building it. Multiple targets can be passed in to a single call to **Precious**().

env.Prepend(*key*=*val*, [...])

Appends the specified keyword arguments to the beginning of construction variables in the environment. If the Environment does not have the specified construction variable, it is simply added to the environment. If the values of the construction variable and the keyword argument are the same type, then the two values will be simply added together. Otherwise, the construction variable and the value of the keyword argument are both coerced to lists, and the lists are added together. (See also the Append method, above.)

Example:

```
env.Prepend(CCFLAGS = '-g ', FOO = ['foo.yyy'])
```

env.PrependENVPath(*name*, *newpath*, [*envname*, *sep*])

This appends new path elements to the given path in the specified external environment (**ENV** by default). This will only add any particular path once (leaving the first one it encounters and ignoring the rest, to preserve path order), and to help assure this, will normalize all paths (using **os.path.normpath** and **os.path.normcase**). This can also handle the case where the given old path variable is a list instead of a string, in which case a list will be returned instead of a string.

Example:

```
print 'before:',env['ENV']['INCLUDE']
```

```
include_path = '/foo/bar:/foo'
env.PrependENVPath('INCLUDE', include_path)
print 'after:',env['ENV']['INCLUDE']
```

The above exmaple will print:

```
before: /biz:/foo
after: /foo/bar:/foo:/biz
```

env.PrependUnique(*key*=*val*, [...])

Appends the specified keyword arguments to the beginning of construction variables in the environment. If the Environment does not have the specified construction variable, it is simply added to the environment. If the construction variable being appended to is a list, then any value(s) that already exist in the construction variable will *not* be added again to the list.

Example:

```
env.PrependUnique(CCFLAGS = '-g', FOO = ['foo.yyy'])
```

env.RCS()

A factory function that returns a Builder object to be used to fetch source files from RCS. The returned Builder is intended to be passed to the **SourceCode** function:

Examples:

```
env.SourceCode('.', env.RCS())
```

Note that **scons** will fetch source files from RCS subdirectories automatically, so configuring RCS as demonstrated in the above example should only be necessary if you are fetching from RCS,v files in the same directory as the source files, or if you need to explicitly specify RCS for a specific subdirectory.

env.Replace(*key*=*val*, [...])

Replaces construction variables in the Environment with the specified keyword arguments.

Example:

```
env.Replace(CCFLAGS = '-g', FOO = 'foo.xxx')
```

Repository(*directory*)

env.Repository(*directory*)

Specifies that *directory* is a repository to be searched for files. Multiple calls to **Repository**() are legal, and each one adds to the list of repositories that will be searched.

To **scons**, a repository is a copy of the source tree, from the top-level directory on down, which may contain both source files and derived files that can be used to build targets in the local source tree. The canonical example would be an official source tree maintained by an integrator. If the repository contains derived files, then the derived files should have been built using **scons**, so that the repository contains the necessary signature information to allow **scons** to figure out when it is appropriate to use the repository copy of a derived file, instead of building one locally.

Note that if an up-to-date derived file already exists in a repository, **scons** will *not* make a copy in the local directory tree. In order to guarantee that a local copy will be made, use the **Local()**

method.

Requires(*target*, *prerequisite*)

env.Requires(*target*, *prerequisite*)

Specifies an order-only relationship between the specified target file(s) and the specified prerequisite file(s). The prerequisite file(s) will be (re)built, if necessary, *before* the target file(s), but the target file(s) do not actually depend on the prerequisites and will not be rebuilt simply because the prerequisite file(s) change.

Example:

```
env.Requires('foo', 'file-that-must-be-built-before-foo')
```

Return([*vars*... , *stop*=])

By default, this stops processing the current SConscript file and returns to the calling SConscript file the values of the variables named in the *vars* string arguments. Multiple strings contaning variable names may be passed to **Return**(). Any strings that contain white space

The optional **stop=** keyword argument may be set to a false value to continue processing the rest of the SConscript file after the **Return**() call. This was the default behavior prior to SCons 0.98. However, the values returned are still the values of the variables in the named *vars* at the point **Return**() is called.

Examples:

```
# Returns without returning a value.
Return()

# Returns the value of the 'foo' Python variable.
Return("foo")

# Returns the values of the Python variables 'foo' and 'bar'.
Return("foo", "bar")

# Returns the values of Python variables 'val1' and 'val2'.
Return('val1 val2')
```

Scanner(*function*, [*argument*, *keys*, *path_function*, *node_class*, *node_factory*, *scan_check*, *recursive*])

env.Scanner(*function*, [*argument*, *keys*, *path_function*, *node_class*, *node_factory*, *scan_check*, *recursive*])

Creates a Scanner object for the specified *function*. See the section "Scanner Objects," below, for a complete explanation of the arguments and behavior.

env.SCCS()

A factory function that returns a Builder object to be used to fetch source files from SCCS. The returned Builder is intended to be passed to the **SourceCode** function.

Example:

```
env.SourceCode('.', env.SCCS())
```

Note that **scons** will fetch source files from SCCS subdirectories automatically, so configuring SCCS as demonstrated in the above example should only be necessary if you are fetching from *s.SCCS* files in the same directory as the source files, or if you need to explicitly specify SCCS for a specific subdirectory.

SConscript(*scripts*, [*exports*, *variant_dir*, *src_dir*, *duplicate*])

env.SConscript(*scripts*, [*exports*, *variant_dir*, *src_dir*, *duplicate*])

SConscript(dirs=*subdirs*, [name=*script*, *exports*, *variant_dir*, *src_dir*, *duplicate*])

env.SConscript(dirs=*subdirs*, [name=*script*, *exports*, *variant_dir*, *src_dir*, *duplicate*])

This tells **scons** to execute one or more subsidiary SConscript (configuration) files. There are two ways to call the **SConscript**() function.

The first way you can call **SConscript**() is to explicitly specify one or more *scripts* as the first argument. A single script may be specified as a string; multiple scripts must be specified as a list (either explicitly or as created by a function like **Split**()).

The second way you can call **SConscript**() is to specify a list of (sub)directory names as a dirs=*subdirs* keyword argument. In this case, **scons** will, by default, execute a subsidiary configuration file named **SConscript** in each of the specified directories. You may specify a name other than **SConscript** by supplying an optional name=*script* keyword argument.

The optional *exports* argument provides a list of variable names or a dictionary of named values to export to the *script(s)*. These variables are locally exported only to the specified *script(s)*, and do not affect the global pool of variables used by the **Export**() function. The subsidiary *script(s)* must use the **Import**() function to import the variables.

In effect, the optional *variant_dir* argument causes the files (and subdirectories) in the directory where *script* resides to be copied to *variant_dir* and the build performed in *variant_dir*. Thus, all of the targets (for example, object files and executables) that would normally be built in (or underneath) the directory containing *script* would actually be built in (or underneath) *variant_dir*. See the description of the **VariantDir**() function below for the details and restrictions. *variant_dir* is interpreted relative to the directory of the calling SConscript file.

Normally, the source for the variant build is the directory containing *script*. If the sources are not in *script*'s directory, the optional *src_dir* argument provides the location of the sources. *src_dir* is interpreted relative to the directory of the calling SConscript file.

By default, **scons** will link or copy (depending on the platform) all the source files into the variant directory tree. This behavior may be disabled by setting the optional *duplicate* argument to 0 (it is set to 1 by default), in which case **scons** will refer directly to the source files in their source directory when building target files. See the description for **VariantDir**() below for the details and restrictions.

Any variables returned by *script* using **Return**() will be returned by the call to **SConscript**().

Examples:

```
SConscript('subdir/SConscript')
foo = SConscript('sub/SConscript', exports='env')
SConscript('dir/SConscript', exports=['env', 'variable'])
SConscript('src/SConscript', variant_dir='build', duplicate=0)
SConscript('bld/SConscript', src_dir='src', exports='env variable')
SConscript(dirs=['sub1', 'sub2'])
SConscript(dirs=['sub3', 'sub4'], name='MySConscript')
```

SConscriptChdir(*value*)

env.SConscriptChdir(*value*)

By default, **scons** changes its working directory to the directory in which each subsidiary SConscript file lives. This behavior may be disabled by specifying either:

```
SConscriptChdir(0)
env.SConscriptChdir(0)
```

in which case **scons** will stay in the top-level directory while reading all SConscript files. (This may be necessary when building from repositories, when all the directories in which SConscript files may be found don't necessarily exist locally.) You may enable and disable this ability by calling SConscriptChdir() multiple times.

Example:

```
env = Environment()
SConscriptChdir(0)
SConscript('foo/SConscript')       # will not chdir to foo
env.SConscriptChdir(1)
SConscript('bar/SConscript')       # will chdir to bar
```

SConsignFile([*file*,*dbm_module*])

env.SConsignFile([*file*,*dbm_module*])

This tells **scons** to store all file signatures in the specified database *file*. If the *file* name is omitted, **.sconsign** is used by default. (The actual file name(s) stored on disk may have an appropriated suffix appended by the *dbm_module*.) If *file* is not an absolute path name, the file is placed in the same directory as the top-level **SConstruct** file.

If *file* is **None**, then **scons** will store file signatures in a separate **.sconsign** file in each directory, not in one global database file. (This was the default behavior prior to SCons 0.96.91 and 0.97.)

The optional *dbm_module* argument can be used to specify which Python database module The default is to use a custom **SCons.dblite** module that uses pickled Python data structures, and which works on all Python versions from 1.5.2 on.

Examples:

```
# Explicitly stores signatures in ".sconsign.dblite"
# in the top-level SConstruct directory (the
# default behavior).
SConsignFile()

# Stores signatures in the file "etc/scons-signatures"
# relative to the top-level SConstruct directory.
SConsignFile("etc/scons-signatures")

# Stores signatures in the specified absolute file name.
SConsignFile("/home/me/SCons/signatures")

# Stores signatures in a separate .sconsign file
# in each directory.
SConsignFile(None)
```

env.SetDefault(key=val, *[...])*

Sets construction variables to default values specified with the keyword arguments if (and only if) the variables are not already set. The following statements are equivalent:

```
env.SetDefault(FOO = 'foo')

if not env.has_key('FOO'): env['FOO'] = 'foo'
```

SetOption(*name*, *value*)

env.SetOption(*name*, *value*)

This function provides a way to set a select subset of the scons command line options from a SConscript file. The options supported are:

clean which corresponds to -c, --clean and --remove;

duplicate
which corresponds to --duplicate;

help which corresponds to -h and --help;

implicit_cache
which corresponds to --implicit-cache;

max_drift
which corresponds to --max-drift;

no_exec
which corresponds to -n, --no-exec, --just-print, --dry-run and --recon;

num_jobs
which corresponds to -j and --jobs;

random
which corresponds to --random; and

stack_size
which corresponds to --stack-size.

See the documentation for the corresponding command line object for information about each specific option.

Example:

```
SetOption('max_drift', 1)
```

SideEffect(*side_effect*, *target*)

env.SideEffect(*side_effect*, *target*)

Declares *side_effect* as a side effect of building *target*. Both *side_effect* and *target* can be a list, a file name, or a node. A side effect is a target that is created as a side effect of building other targets. For example, a Windows PDB file is created as a side effect of building the .obj files for a static library. If a target is a side effect of multiple build commands, **scons** will ensure that only one set of commands is executed at a time. Consequently, you only need to use this method for side-effect targets that are built as a result of multiple build commands.

SourceCode(*entries*, *builder*)

env.SourceCode(*entries*, *builder*)

Arrange for non-existent source files to be fetched from a source code management system using the specified *builder*. The specified *entries* may be a Node, string or list of both, and may represent either individual source files or directories in which source files can be found.

For any non-existent source files, **scons** will search up the directory tree and use the first **SourceCode** builder it finds. The specified *builder* may be **None**, in which case **scons** will not use a builder to fetch source files for the specified *entries*, even if a **SourceCode** builder has been specified for a directory higher up the tree.

scons will, by default, fetch files from SCCS or RCS subdirectories without explicit configuration. This takes some extra processing time to search for the necessary source code management files on disk. You can avoid these extra searches and speed up your build a little by disabling these searches as follows:

```
env.SourceCode('.', None)
```

Note that if the specified *builder* is one you create by hand, it must have an associated construction environment to use when fetching a source file.

scons provides a set of canned factory functions that return appropriate Builders for various popular source code management systems. Canonical examples of invocation include:

```
env.SourceCode('.', env.BitKeeper('/usr/local/BKsources'))
env.SourceCode('src', env.CVS('/usr/local/CVSROOT'))
env.SourceCode('/', env.RCS())
env.SourceCode(['f1.c', 'f2.c'], env.SCCS())
env.SourceCode('no_source.c', None)
```

env.subst(*input*, [*raw*, *target*, *source*, *conv*])

Performs construction variable interpolation on the specified string or sequence argument *input*.

By default, leading or trailing white space will be removed from the result. and all sequences of white space will be compressed to a single space character. Additionally, any **$(** and **$)** character sequences will be stripped from the returned string, The optional *raw* argument may be set to **1** if you want to preserve white space and **$(-$)** sequences. The *raw* argument may be set to **2** if you want to strip all characters between any **$(** and **$)** pairs (as is done for signature calculation).

If the input is a sequence (list or tuple), the individual elements of the sequence will be expanded, and the results will be returned as a list.

The optional *target* and *source* keyword arguments must be set to lists of target and source nodes, respectively, if you want the **$TARGET**, **$TARGETS**, **$SOURCE** and **$SOURCES** to be available for expansion. This is usually necessary if you are calling **env.subst**() from within a Python function used as an SCons action.

Returned string values or sequence elements are converted to their string representation by default. The optional *conv* argument may specify a conversion function that will be used in place of the default. For example, if you want Python objects (including SCons Nodes) to be returned as Python objects, you can use the Python **lambda** idiom to pass in an unnamed function that simply returns its unconverted argument.

Example:

```
print env.subst("The C compiler is: $CC")

def compile(target, source, env):
    sourceDir = env.subst("${SOURCE.srcdir}",
                          target=target,
                          source=source)

source_nodes = env.subst('$EXPAND_TO_NODELIST',
                         conv=lambda x: x)
```

SourceSignatures(*type*)

env.SourceSignatures(*type*)

Note: Although it is not yet officially deprecated, use of this function is discouraged. See the **Decider**() function for a more flexible and straightforward way to configure SCons' decision-making.

The **SourceSignatures**() function tells **scons** how to decide if a source file (a file that is not built from any other files) has changed since the last time it was used to build a particular target file. Legal values are **MD5** or **timestamp**.

If the environment method is used, the specified type of source signature is only used when deciding whether targets built with that environment are up-to-date or must be rebuilt. If the global function is used, the specified type of source signature becomes the default used for all decisions about whether targets are up-to-date.

MD5 means **scons** decides that a source file has changed if the MD5 checksum of its contents has changed since the last time it was used to rebuild a particular target file.

timestamp means **scons** decides that a source file has changed if its timestamp (modification time) has changed since the last time it was used to rebuild a particular target file. (Note that although this is similar to the behavior of Make, by default it will also rebuild if the dependency is *older* than the last time it was used to rebuild the target file.)

There is no different between the two behaviors for Python **Value**() node objects.

MD5 signatures take longer to compute, but are more accurate than **timestamp** signatures. The default value is **MD5**.

Note that the default **TargetSignatures**() setting (see below) is to use this **SourceSignatures**() setting for any target files that are used to build other target files. Consequently, changing the value of **SourceSignatures**() will, by default, affect the up-to-date decision for all files in the build (or all files built with a specific construction environment when **env.SourceSignatures**() is used).

Split(*arg*)

env.Split(*arg*)

Returns a list of file names or other objects. If arg is a string, it will be split on strings of white-space characters within the string, making it easier to write long lists of file names. If arg is already a list, the list will be returned untouched. If arg is any other type of object, it will be returned as a list containing just the object.

Example:

```
files = Split("f1.c f2.c f3.c")
files = env.Split("f4.c f5.c f6.c")
files = Split("""
        f7.c
        f8.c
        f9.c
""")
```

Tag(*node*, *tags*)

Annotates file or directory Nodes with information about how the **Package**() Builder should package those files or directories. All tags are optional.

Examples:

```
# makes sure the built library will be installed with 0644 file
# access mode
Tag( Library( 'lib.c' ), UNIX_ATTR="0644" )

# marks file2.txt to be a documentation file
Tag( 'file2.txt', DOC )
```

</summary> </builder>

TargetSignatures(*type*)

env.TargetSignatures(*type*)

Note: Although it is not yet officially deprecated, use of this function is discouraged. See the **Decider**() function for a more flexible and straightforward way to configure SCons' decision-making.

The **TargetSignatures**() function tells **scons** how to decide if a target file (a file that *is* built from any other files) has changed since the last time it was used to build some other target file. Legal values are **build**; **content** (or its synonym **MD5**); **timestamp**; or **source**.

If the environment method is used, the specified type of target signature is only used for targets built with that environment. If the global function is used, the specified type of signature becomes the default used for all target files that don't have an explicit target signature type specified for their environments.

content (or its synonym **MD5**) means **scons** decides that a target file has changed if the MD5 checksum of its contents has changed since the last time it was used to rebuild some other target file. This means **scons** will open up MD5 sum the contents of target files after they're built, and may decide that it does not need to rebuild "downstream" target files if a file was rebuilt with exactly the same contents as the last time.

timestamp means **scons** decides that a target file has changed if its timestamp (modification time) has changed since the last time it was used to rebuild some other target file. (Note that although this is similar to the behavior of Make, by default it will also rebuild if the dependency is *older* than the last time it was used to rebuild the target file.)

source means **scons** decides that a target file has changed as specified by the corresponding **SourceSignatures**() setting (**MD5** or **timestamp**). This means that **scons** will treat all input files to a target the same way, regardless of whether they are source files or have been built from other files.

build means **scons** decides that a target file has changed if it has been rebuilt in this invocation or if its content or timestamp have changed as specified by the corresponding **SourceSignatures**() setting. This "propagates" the status of a rebuilt file so that other "downstream" target files will always be rebuilt, even if the contents or the timestamp have not changed.

build signatures are fastest because **content** (or **MD5**) signatures take longer to compute, but are more accurate than **timestamp** signatures, and can prevent unnecessary "downstream" rebuilds when a target file is rebuilt to the exact same contents as the previous build. The **source** setting provides the most consistent behavior when other target files may be rebuilt from both source and target input files. The default value is **source**.

Because the default setting is **source**, using **SourceSignatures**() is generally preferable to **TargetSignatures**(), so that the up-to-date decision will be consistent for all files (or all files built with a specific construction environment). Use of **TargetSignatures**() provides specific control for how built target files affect their "downstream" dependencies.

Tool(*string*[,*toolpath*, ***kw*])

Returns a callable object that can be used to initialize a construction environment using the tools keyword of the Environment() method. The object may be called with a construction environment as an argument, in which case the object will add the necessary variables to the construction environment and the name of the tool will be added to the **$TOOLS** construction variable.

Additional keyword arguments are passed to the tool's **generate()** method.

Examples:

```
env = Environment(tools = [ Tool('msvc') ])

env = Environment()
t = Tool('msvc')
t(env)  # adds 'msvc' to the TOOLS variable
u = Tool('opengl', toolpath = ['tools'])
u(env)  # adds 'opengl' to the TOOLS variable
```

env.Tool(*string*[,*toolpath*, ***kw*])

Applies the callable object for the specified tool *string* to the environment through which the method was called.

Additional keyword arguments are passed to the tool's **generate()** method.

```
env.Tool('gcc')
env.Tool('opengl', toolpath = ['build/tools'])
```

Value(*value*, [*built_value*])

env.Value(*value*, [*built_value*])

Returns a Node object representing the specified Python value. Value Nodes can be used as dependencies of targets. If the result of calling **str(**value**)** changes between SCons runs, any targets depending on **Value(**value**)** will be rebuilt. (This is true even when using timestamps to decide if files are up-to-date.) When using timestamp source signatures, Value Nodes' timestamps are equal to the system time when the Node is created.

The returned Value Node object has a **write**() method that can be used to "build" a Value Node by setting a new value. The optional *built_value* argument can be specified when the Value Node is created to indicate the Node should already be considered "built." There is a corresponding **read**() method that will return the built value of the Node.

Examples:

```
env = Environment()

def create(target, source, env):
    # A function that will write a 'prefix=$SOURCE'
    # string into the file name specified as the
    # $TARGET.
    f = open(str(target[0]), 'wb')
    f.write('prefix=' + source[0].get_contents())

# Fetch the prefix= argument, if any, from the command
# line, and use /usr/local as the default.
prefix = ARGUMENTS.get('prefix', '/usr/local')

# Attach a .Config() builder for the above function action
# to the construction environment.
env['BUILDERS']['Config'] = Builder(action = create)
env.Config(target = 'package-config', source = Value(prefix))

def build_value(target, source, env):
    # A function that "builds" a Python Value by updating
    # the the Python value with the contents of the file
    # specified as the source of the Builder call ($SOURCE).
    target[0].write(source[0].get_contents())

output = env.Value('before')
input = env.Value('after')

# Attach a .UpdateValue() builder for the above function
# action to the construction environment.
env['BUILDERS']['UpdateValue'] = Builder(action = build_value)
env.UpdateValue(target = Value(output), source = Value(input))
```

VariantDir(*variant_dir*, *src_dir*, [*duplicate*])

env.VariantDir(*variant_dir*, *src_dir*, [*duplicate*])

In effect, the *src_dir* directory tree is copied to *variant_dir* so a build can be performed there. **VariantDir**() can be called multiple times with the same *src_dir* to set up multiple builds with different options (*variants*). *src_dir* must be in or underneath the SConstruct file's directory, and *variant_dir* may not be underneath the *src_dir* .

The default behavior is for **scons** to duplicate the source files in the variant tree and then build the derived files within the variant tree. This guarantees correct builds regardless of whether intermediate source files are generated during the build, whether preprocessors or other scanners search for included files relative to the source file, or whether individual compilers or other invoked tools are hard-coded to put derived files in the same directory as source files.

If possible on the platform, the duplication is performed by linking rather than copying; see also the *--duplicate* command-line option. Moreover, only the files needed for the build are duplicated; files and directories that are not used are not present in *variant_dir*.

Duplicating the source tree may be disabled by setting *duplicate* to 0. This will cause **scons** to invoke Builders using the path names of source files in *src_dir* and the path names of derived files within *variant_dir*. This is always more efficient than *duplicate*=1, and is usually safe for most builds (but see above for cases that may cause problems).

Note that **VariantDir**() works most naturally with a subsidiary SConscript file. However, you would then call the subsidiary SConscript file not in the source directory, but in the *variant_dir* , regardless of the value of *duplicate*. This is how you tell **scons** which variant of a source tree to build. For example:

```
VariantDir('build-variant1', 'src')
SConscript('build-variant1/SConscript')
VariantDir('build-variant2', 'src')
SConscript('build-variant2/SConscript')
```

See also the **SConscript**() function, described above, for another way to specify a variant directory in conjunction with calling a subsidiary SConscript file.

WhereIs(*program*, [*path*, *pathext*, *reject*])

env.WhereIs(*program*, [*path*, *pathext*, *reject*])

Searches for the specified executable *program,* returning the full path name to the program if it is found, and returning None if not. Searches the specified *path,* the value of the calling environment's PATH (env['ENV']['PATH']), or the user's current external PATH (os.environ['PATH']) by default. On Windows systems, searches for executable programs with any of the file extensions listed in the specified *pathext,* the calling environment's PATHEXT (env['ENV']['PATHEXT']) or the user's current PATHEXT (os.environ['PATHEXT']) by default. Will not select any path name or names in the specified *reject* list, if any.

SConscript Variables

In addition to the global functions and methods, **scons** supports a number of Python variables that can be used in SConscript files to affect how you want the build to be performed. These variables may be accessed from custom Python modules that you import into an SConscript file by adding the following to the Python module:

```
from SCons.Script import *
```

ARGLIST

A list *keyword*=*value* arguments specified on the command line. Each element in the list is a tuple containing the (*keyword*,*value*) of the argument. The separate *keyword* and *value* elements of the tuple can be accessed by subscripting for element **[0]** and **[1]** of the tuple, respectively.

Example:

```
print "first keyword, value =", ARGLIST[0][0], ARGLIST[0][1]
print "second keyword, value =", ARGLIST[1][0], ARGLIST[1][1]
third_tuple = ARGLIST[2]
print "third keyword, value =", third_tuple[0], third_tuple[1]
for key, value in ARGLIST:
    # process key and value
```

ARGUMENTS

A dictionary of all the *keyword*=*value* arguments specified on the command line. The dictionary is not in order, and if a given keyword has more than one value assigned to it on the command line, the last (right-most) value is the one in the **ARGUMENTS** dictionary.

Example:

```
if ARGUMENTS.get('debug', 0):
    env = Environment(CCFLAGS = '-g')
else:
    env = Environment()
```

BUILD_TARGETS

A list of the targets which **scons** will actually try to build, regardless of whether they were specified on the command line or via the **Default**() function or method. The elements of this list may be strings *or* nodes, so you should run the list through the Python **str** function to make sure any Node path names are converted to strings.

Because this list may be taken from the list of targets specified using the **Default**() function or method, the contents of the list may change on each successive call to **Default**(). See the **DEFAULT_TARGETS** list, below, for additional information.

Example:

```
if 'foo' in BUILD_TARGETS:
    print "Don't forget to test the `foo' program!"
if 'special/program' in BUILD_TARGETS:
    SConscript('special')
```

Note that the **BUILD_TARGETS** list only contains targets expected listed on the command line or via calls to the **Default**() function or method. It does *not* contain all dependent targets that will be built as a result of making the sure the explicitly-specified targets are up to date.

COMMAND_LINE_TARGETS

A list of the targets explicitly specified on the command line. If there are no targets specified on the command line, the list is empty. This can be used, for example, to take specific actions only when a certain target or targets is explicitly being built.

Example:

```
if 'foo' in COMMAND_LINE_TARGETS:
    print "Don't forget to test the `foo' program!"
if 'special/program' in COMMAND_LINE_TARGETS:
    SConscript('special')
```

DEFAULT_TARGETS

A list of the target *nodes* that have been specified using the **Default**() function or method. The elements of the list are nodes, so you need to run them through the Python **str** function to get at the path name for each Node.

Example:

```
print str(DEFAULT_TARGETS[0])
if 'foo' in map(str, DEFAULT_TARGETS):
```

```
print "Don't forget to test the `foo' program!"
```

The contents of the **DEFAULT_TARGETS** list change on on each successive call to the **Default**() function:

```
print map(str, DEFAULT_TARGETS)   # originally []
Default('foo')
print map(str, DEFAULT_TARGETS)   # now a node ['foo']
Default('bar')
print map(str, DEFAULT_TARGETS)   # now a node ['foo', 'bar']
Default(None)
print map(str, DEFAULT_TARGETS)   # back to []
```

Consequently, be sure to use **DEFAULT_TARGETS** only after you've made all of your **Default**() calls, or else simply be careful of the order of these statements in your SConscript files so that you don't look for a specific default target before it's actually been added to the list.

Construction Variables

A construction environment has an associated dictionary of *construction variables* that are used by built-in or user-supplied build rules. Construction variables must follow the same rules for Python identifiers: the initial character must be an underscore or letter, followed by any number of underscores, letters, or digits.

A number of useful construction variables are automatically defined by scons for each supported platform, and additional construction variables can be defined by the user. The following is a list of the automatically defined construction variables:

AR
: The static library archiver.

ARCHITECTURE
: Specifies the system architecture for which the package is being built. The default is the system architecture of the machine on which SCons is running. This is used to fill in the **Architecture:** field in an Ipkg **control** file, and as part of the name of a generated RPM file.

ARCOM
: The command line used to generate a static library from object files.

ARCOMSTR
: The string displayed when an object file is generated from an assembly-language source file. If this is not set, then $ARCOM (the command line) is displayed.

```
env = Environment(ARCOMSTR = "Archiving $TARGET")
```

ARFLAGS
: General options passed to the static library archiver.

AS
: The assembler.

ASCOM
: The command line used to generate an object file from an assembly-language source file.

ASCOMSTR
The string displayed when an object file is generated from an assembly-language source file. If this is not set, then $ASCOM (the command line) is displayed.

```
env = Environment(ASCOMSTR = "Assembling $TARGET")
```

ASFLAGS
General options passed to the assembler.

ASPPCOM
The command line used to assemble an assembly-language source file into an object file after first running the file through the C preprocessor. Any options specified in the $ASFLAGS and $CPPFLAGS construction variables are included on this command line.

ASPPCOMSTR
The string displayed when an object file is generated from an assembly-language source file after first running the file through the C preprocessor. If this is not set, then $ASPPCOM (the command line) is displayed.

```
env = Environment(ASPPCOMSTR = "Assembling $TARGET")
```

ASPPFLAGS
General options when an assembling an assembly-language source file into an object file after first running the file through the C preprocessor. The default is to use the value of $ASFLAGS.

BIBTEX
The bibliography generator for the TeX formatter and typesetter and the LaTeX structured formatter and typesetter.

BIBTEXCOM
The command line used to call the bibliography generator for the TeX formatter and typesetter and the LaTeX structured formatter and typesetter.

BIBTEXCOMSTR
The string displayed when generating a bibliography for TeX or LaTeX. If this is not set, then $BIBTEXCOM (the command line) is displayed.

```
env = Environment(BIBTEXCOMSTR = "Generating bibliography $TARGET")
```

BIBTEXFLAGS
General options passed to the bibliography generator for the TeX formatter and typesetter and the LaTeX structured formatter and typesetter.

BITKEEPER
The BitKeeper executable.

BITKEEPERCOM
The command line for fetching source files using BitKeeper.

BITKEEPERCOMSTR

The string displayed when fetching a source file using BitKeeper. If this is not set, then $BITKEEPERCOM (the command line) is displayed.

BITKEEPERGET

The command ($BITKEEPER) and subcommand for fetching source files using BitKeeper.

BITKEEPERGETFLAGS

Options that are passed to the BitKeeper **get** subcommand.

BUILDERS

A dictionary mapping the names of the builders available through this environment to underlying Builder objects. Builders named Alias, CFile, CXXFile, DVI, Library, Object, PDF, PostScript, and Program are available by default. If you initialize this variable when an Environment is created:

```
env = Environment(BUILDERS = {'NewBuilder' : foo})
```

the default Builders will no longer be available. To use a new Builder object in addition to the default Builders, add your new Builder object like this:

```
env = Environment()
env.Append(BUILDERS = {'NewBuilder' : foo})
```

or this:

```
env = Environment()
env['BUILDERS]['NewBuilder'] = foo
```

CC The C compiler.

CCCOM

The command line used to compile a C source file to a (static) object file. Any options specified in the $CFLAGS, $CCFLAGS and $CPPFLAGS construction variables are included on this command line.

CCCOMSTR

The string displayed when a C source file is compiled to a (static) object file. If this is not set, then $CCCOM (the command line) is displayed.

```
env = Environment(CCCOMSTR = "Compiling static object $TARGET")
```

CCFLAGS

General options that are passed to the C and C++ compilers.

CCPCHFLAGS

Options added to the compiler command line to support building with precompiled headers. The default value expands expands to the appropriate Microsoft Visual C++ command-line options when the $PCH construction variable is set.

CCPDBFLAGS

Options added to the compiler command line to support storing debugging information in a Microsoft Visual C++ PDB file. The default value expands expands to appropriate Microsoft Visual C++ command-line options when the $PDB construction variable is set.

The Visual C++ compiler option that SCons uses by default to generate PDB information is **/Z7**. This works correctly with parallel (**−j**) builds because it embeds the debug information in the intermediate object files, as opposed to sharing a single PDB file between multiple object files. This is also the only way to get debug information embedded into a static library. Using the **/Zi** instead may yield improved link-time performance, although parallel builds will no longer work.

You can generate PDB files with the **/Zi** switch by overriding the default $CCPDBFLAGS variable as follows:

```
import SCons.Util
env['CCPDBFLAGS'] = SCons.Util.CLVar(['${(PDB and "/Zi /Fd%s" % File(PDB)) or ""}'])
```

An alternative would be to use the **/Zi** to put the debugging information in a separate **.pdb** file for each object file by overriding the $CCPDBFLAGS variable as follows:

```
env['CCPDBFLAGS'] = '/Zi /Fd${TARGET}.pdb'
```

CCVERSION

The version number of the C compiler. This may or may not be set, depending on the specific C compiler being used.

CFILESUFFIX

The suffix for C source files. This is used by the internal CFile builder when generating C files from Lex (.l) or YACC (.y) input files. The default suffix, of course, is **.c** (lower case). On case-insensitive systems (like Windows), SCons also treats **.C** (upper case) files as C files.

CFLAGS

General options that are passed to the C compiler (C only; not C++).

CHANGE_SPECFILE

A hook for modifying the file that controls the packaging build (the **.spec** for RPM, the **control** for Ipkg, the **.wxs** for MSI). If set, the function will be called after the SCons template for the file has been written. XXX

CHANGELOG

The name of a file containing the change log text to be included in the package. This is included as the **%changelog** section of the RPM **.spec** file.

_concat

A function used to produce variables like $_CPPINCFLAGS. It takes four or five arguments: a prefix to concatenate onto each element, a list of elements, a suffix to concatenate onto each element, an environment for variable interpolation, and an optional function that will be called to transform the list before concatenation.

```
env['_CPPINCFLAGS'] = '$( ${_concat(INCPREFIX, CPPPATH, INCSUFFIX, __env__, RDirs)} $)',
```

CONFIGUREDIR
The name of the directory in which Configure context test files are written. The default is **.sconf_temp** in the top-level directory containing the **SConstruct** file.

CONFIGURELOG
The name of the Configure context log file. The default is **config.log** in the top-level directory containing the **SConstruct** file.

_CPPDEFFLAGS
An automatically-generated construction variable containing the C preprocessor command-line options to define values. The value of $_CPPDEFFLAGS is created by appending $CPPDEFPREFIX and $CPPDEFSUFFIX to the beginning and end of each directory in $CPPDEFINES.

CPPDEFINES
A platform independent specification of C preprocessor definitions. The definitions will be added to command lines through the automatically-generated $_CPPDEFFLAGS construction variable (see above), which is constructed according to the type of value of $CPPDEFINES:

If $CPPDEFINES is a string, the values of the $CPPDEFPREFIX and $CPPDEFSUFFIX construction variables will be added to the beginning and end.

```
# Will add -Dxyz to POSIX compiler command lines,
# and /Dxyz to Microsoft Visual C++ command lines.
env = Environment(CPPDEFINES='xyz')
```

If $CPPDEFINES is a list, the values of the $CPPDEFPREFIX and $CPPDEFSUFFIX construction variables will be appended to the beginning and end of each element in the list. If any element is a list or tuple, then the first item is the name being defined and the second item is its value:

```
# Will add -DB=2 -DA to POSIX compiler command lines,
# and /DB=2 /DA to Microsoft Visual C++ command lines.
env = Environment(CPPDEFINES=[('B', 2), 'A'])
```

If $CPPDEFINES is a dictionary, the values of the $CPPDEFPREFIX and $CPPDEFSUFFIX construction variables will be appended to the beginning and end of each item from the dictionary. The key of each dictionary item is a name being defined to the dictionary item's corresponding value; if the value is **None**, then the name is defined without an explicit value. Note that the resulting flags are sorted by keyword to ensure that the order of the options on the command line is consistent each time **scons** is run.

```
# Will add -DA -DB=2 to POSIX compiler command lines,
# and /DA /DB=2 to Microsoft Visual C++ command lines.
env = Environment(CPPDEFINES={'B':2, 'A':None})
```

CPPDEFPREFIX
The prefix used to specify preprocessor definitions on the C compiler command line. This will be appended to the beginning of each definition in the $CPPDEFINES construction variable when the $_CPPDEFFLAGS variable is automatically generated.

CPPDEFSUFFIX
The suffix used to specify preprocessor definitions on the C compiler command line. This will be appended to the end of each definition in the $CPPDEFINES construction variable when the $_CPPDEFFLAGS variable is automatically generated.

CPPFLAGS

User-specified C preprocessor options. These will be included in any command that uses the C preprocessor, including not just compilation of C and C++ source files via the $CCCOM, $SHCCCOM, $CXXCOM and $SHCXXCOM command lines, but also the $FORTRANPPCOM, $SHFORTRANPPCOM, $F77PPCOM and $SHF77PPCOM command lines used to compile a Fortran source file, and the $ASPPCOM command line used to assemble an assembly language source file, after first running each file through the C preprocessor. Note that this variable does *not* contain **–I** (or similar) include search path options that scons generates automatically from $CPPPATH. See $_CPPINCFLAGS, below, for the variable that expands to those options.

_CPPINCFLAGS

An automatically-generated construction variable containing the C preprocessor command-line options for specifying directories to be searched for include files. The value of $_CPPINCFLAGS is created by appending $INCPREFIX and $INCSUFFIX to the beginning and end of each directory in $CPPPATH.

CPPPATH

The list of directories that the C preprocessor will search for include directories. The C/C++ implicit dependency scanner will search these directories for include files. Don't explicitly put include directory arguments in CCFLAGS or CXXFLAGS because the result will be non-portable and the directories will not be searched by the dependency scanner. Note: directory names in CPPPATH will be looked-up relative to the SConscript directory when they are used in a command. To force **scons** to look-up a directory relative to the root of the source tree use #:

```
env = Environment(CPPPATH='#/include')
```

The directory look-up can also be forced using the **Dir**() function:

```
include = Dir('include')
env = Environment(CPPPATH=include)
```

The directory list will be added to command lines through the automatically-generated $_CPPINCFLAGS construction variable, which is constructed by appending the values of the $INCPREFIX and $INCSUFFIX construction variables to the beginning and end of each directory in $CPPPATH. Any command lines you define that need the CPPPATH directory list should include $_CPPINCFLAGS:

```
env = Environment(CCCOM="my_compiler $_CPPINCFLAGS -c -o $TARGET $SOURCE")
```

CPPSUFFIXES

The list of suffixes of files that will be scanned for C preprocessor implicit dependencies (#include lines). The default list is:

```
[".c", ".C", ".cxx", ".cpp", ".c++", ".cc",
 ".h", ".H", ".hxx", ".hpp", ".hh",
 ".F", ".fpp", ".FPP",
 ".m", ".mm",
 ".S", ".spp", ".SPP"]
```

CVS

The CVS executable.

CVSCOFLAGS

Options that are passed to the CVS checkout subcommand.

CVSCOM
The command line used to fetch source files from a CVS repository.

CVSCOMSTR
The string displayed when fetching a source file from a CVS repository. If this is not set, then $CVSCOM (the command line) is displayed.

CVSFLAGS
General options that are passed to CVS. By default, this is set to **–d $CVSREPOSITORY** to specify from where the files must be fetched.

CVSREPOSITORY
The path to the CVS repository. This is referenced in the default $CVSFLAGS value.

CXX
The C++ compiler.

CXXCOM
The command line used to compile a C++ source file to an object file. Any options specified in the $CXXFLAGS and $CPPFLAGS construction variables are included on this command line.

CXXCOMSTR
The string displayed when a C++ source file is compiled to a (static) object file. If this is not set, then $CXXCOM (the command line) is displayed.

```
env = Environment(CXXCOMSTR = "Compiling static object $TARGET")
```

CXXFILESUFFIX
The suffix for C++ source files. This is used by the internal CXXFile builder when generating C++ files from Lex (.ll) or YACC (.yy) input files. The default suffix is **.cc**. SCons also treats files with the suffixes **.cpp**, **.cxx**, **.c++**, and **.C++** as C++ files, and files with **.mm** suffixes as Objective C++ files. On case-sensitive systems (Linux, UNIX, and other POSIX-alikes), SCons also treats **.C** (upper case) files as C++ files.

CXXFLAGS
General options that are passed to the C++ compiler. By default, this includes the value of $CCFLAGS, so that setting $CCFLAGS affects both C and C++ compilation. If you want to add C++-specific flags, you must set or override the value of $CXXFLAGS.

CXXVERSION
The version number of the C++ compiler. This may or may not be set, depending on the specific C++ compiler being used.

DESCRIPTION
A long description of the project being packaged. This is included in the relevant section of the file that controls the packaging build.

DESCRIPTION_lang
A language-specific long description for the specified *lang*. This is used to populate a **%description -l** section of an RPM **.spec** file.

Dir A function that converts a string into a Dir instance relative to the target being built.

Dirs A function that converts a list of strings into a list of Dir instances relative to the target being built.

DSUFFIXES
The list of suffixes of files that will be scanned for imported D package files. The default list is:

```
['.d']
```

DVIPDF
The TeX DVI file to PDF file converter.

DVIPDFCOM
The command line used to convert TeX DVI files into a PDF file.

DVIPDFCOMSTR
The string displayed when a TeX DVI file is converted into a PDF file. If this is not set, then $DVIPDFCOM (the command line) is displayed.

DVIPDFFLAGS
General options passed to the TeX DVI file to PDF file converter.

DVIPS The TeX DVI file to PostScript converter.

DVIPSFLAGS
General options passed to the TeX DVI file to PostScript converter.

ENV A dictionary of environment variables to use when invoking commands. When $ENV is used in a command all list values will be joined using the path separator and any other non-string values will simply be coerced to a string. Note that, by default, **scons** does *not* propagate the environment in force when you execute **scons** to the commands used to build target files. This is so that builds will be guaranteed repeatable regardless of the environment variables set at the time **scons** is invoked.

If you want to propagate your environment variables to the commands executed to build target files, you must do so explicitly:

```
import os
env = Environment(ENV = os.environ)
```

Note that you can choose only to propagate certain environment variables. A common example is the system **PATH** environment variable, so that **scons** uses the same utilities as the invoking shell (or other process):

```
import os
env = Environment(ENV = {'PATH' : os.environ['PATH']})
```

ESCAPE
A function that will be called to escape shell special characters in command lines. The function should take one argument: the command line string to escape; and should return the escaped command line.

F77
The Fortran 77 compiler. You should normally set the $FORTRAN variable, which specifies the default Fortran compiler for all Fortran versions. You only need to set $F77 if you need to use a specific compiler or compiler version for Fortran 77 files.

F77COM
The command line used to compile a Fortran 77 source file to an object file. You only need to set $F77COM if you need to use a specific command line for Fortran 77 files. You should normally set the $FORTRANCOM variable, which specifies the default command line for all Fortran versions.

F77COMSTR
The string displayed when a Fortran 77 source file is compiled to an object file. If this is not set, then $F77COM or $FORTRANCOM (the command line) is displayed.

F77FILESUFFIXES
The list of file extensions for which the F77 dialect will be used. By default, this is ['.f77']

F77FLAGS
General user-specified options that are passed to the Fortran 77 compiler. Note that this variable does *not* contain **−I** (or similar) include search path options that scons generates automatically from $F77PATH. See $_F77INCFLAGS below, for the variable that expands to those options. You only need to set $F77FLAGS if you need to define specific user options for Fortran 77 files. You should normally set the $FORTRANFLAGS variable, which specifies the user-specified options passed to the default Fortran compiler for all Fortran versions.

_F77INCFLAGS
An automatically-generated construction variable containing the Fortran 77 compiler command-line options for specifying directories to be searched for include files. The value of $_F77INCFLAGS is created by appending $INCPREFIX and $INCSUFFIX to the beginning and end of each directory in $F77PATH.

F77PATH
The list of directories that the Fortran 77 compiler will search for include directories. The implicit dependency scanner will search these directories for include files. Don't explicitly put include directory arguments in $F77FLAGS because the result will be non-portable and the directories will not be searched by the dependency scanner. Note: directory names in $F77PATH will be looked-up relative to the SConscript directory when they are used in a command. To force **scons** to look-up a directory relative to the root of the source tree use #: You only need to set $F77PATH if you need to define a specific include path for Fortran 77 files. You should normally set the $FORTRANPATH variable, which specifies the include path for the default Fortran compiler for all Fortran versions.

```
env = Environment(F77PATH='#/include')
```

The directory look-up can also be forced using the **Dir**() function:

```
include = Dir('include')
env = Environment(F77PATH=include)
```

The directory list will be added to command lines through the automatically-generated $_F77INCFLAGS construction variable, which is constructed by appending the values of the $INCPREFIX and $INCSUFFIX construction variables to the beginning and end of each directory in $F77PATH. Any command lines you define that need the F77PATH directory list should include $_F77INCFLAGS:

```
env = Environment(F77COM="my_compiler $_F77INCFLAGS -c -o $TARGET $SOURCE")
```

F77PPCOM
The command line used to compile a Fortran 77 source file to an object file after first running the file through the C preprocessor. Any options specified in the $F77FLAGS and $CPPFLAGS construction variables are included on this command line. You only need to set $F77PPCOM if you need to use a specific C-preprocessor command line for Fortran 77 files. You should normally set the $FORTRANPPCOM variable, which specifies the default C-preprocessor command line for all Fortran versions.

F77PPCOMSTR
The string displayed when a Fortran 77 source file is compiled to an object file after first running the file through the C preprocessor. If this is not set, then $F77PPCOM or $FORTRANPPCOM (the command line) is displayed.

F77PPFILESUFFIXES
The list of file extensions for which the compilation + preprocessor pass for F77 dialect will be used. By default, this is empty

F90
The Fortran 90 compiler. You should normally set the $FORTRAN variable, which specifies the default Fortran compiler for all Fortran versions. You only need to set $F90 if you need to use a specific compiler or compiler version for Fortran 90 files.

F90COM
The command line used to compile a Fortran 90 source file to an object file. You only need to set $F90COM if you need to use a specific command line for Fortran 90 files. You should normally set the $FORTRANCOM variable, which specifies the default command line for all Fortran versions.

F90COMSTR
The string displayed when a Fortran 90 source file is compiled to an object file. If this is not set, then $F90COM or $FORTRANCOM (the command line) is displayed.

F90FILESUFFIXES
The list of file extensions for which the F90 dialect will be used. By default, this is ['.f90']

F90FLAGS
General user-specified options that are passed to the Fortran 90 compiler. Note that this variable does *not* contain **−I** (or similar) include search path options that scons generates automatically from $F90PATH. See $_F90INCFLAGS below, for the variable that expands to those options. You only need to set $F90FLAGS if you need to define specific user options for Fortran 90 files. You should normally set the $FORTRANFLAGS variable, which specifies the user-specified options passed to the default Fortran compiler for all Fortran versions.

_F90INCFLAGS
An automatically-generated construction variable containing the Fortran 90 compiler command-line options for specifying directories to be searched for include files. The value of $_F90INCFLAGS is created by appending $INCPREFIX and $INCSUFFIX to the beginning and end of each directory in $F90PATH.

F90PATH

The list of directories that the Fortran 90 compiler will search for include directories. The implicit dependency scanner will search these directories for include files. Don't explicitly put include directory arguments in $F90FLAGS because the result will be non-portable and the directories will not be searched by the dependency scanner. Note: directory names in $F90PATH will be looked-up relative to the SConscript directory when they are used in a command. To force **scons** to look-up a directory relative to the root of the source tree use #: You only need to set $F90PATH if you need to define a specific include path for Fortran 90 files. You should normally set the $FORTRANPATH variable, which specifies the include path for the default Fortran compiler for all Fortran versions.

```
env = Environment(F90PATH='#/include')
```

The directory look-up can also be forced using the **Dir**() function:

```
include = Dir('include')
env = Environment(F90PATH=include)
```

The directory list will be added to command lines through the automatically-generated $_F90INCFLAGS construction variable, which is constructed by appending the values of the $INCPREFIX and $INCSUFFIX construction variables to the beginning and end of each directory in $F90PATH. Any command lines you define that need the F90PATH directory list should include $_F90INCFLAGS:

```
env = Environment(F90COM="my_compiler $_F90INCFLAGS -c -o $TARGET $SOURCE")
```

F90PPCOM

The command line used to compile a Fortran 90 source file to an object file after first running the file through the C preprocessor. Any options specified in the $F90FLAGS and $CPPFLAGS construction variables are included on this command line. You only need to set $F90PPCOM if you need to use a specific C-preprocessor command line for Fortran 90 files. You should normally set the $FORTRANPPCOM variable, which specifies the default C-preprocessor command line for all Fortran versions.

F90PPCOMSTR

The string displayed when a Fortran 90 source file is compiled after first running the file through the C preprocessor. If this is not set, then $F90PPCOM or $FORTRANPPCOM (the command line) is displayed.

F90PPFILESUFFIXES

The list of file extensions for which the compilation + preprocessor pass for F90 dialect will be used. By default, this is empty

F95

The Fortran 95 compiler. You should normally set the $FORTRAN variable, which specifies the default Fortran compiler for all Fortran versions. You only need to set $F95 if you need to use a specific compiler or compiler version for Fortran 95 files.

F95COM

The command line used to compile a Fortran 95 source file to an object file. You only need to set $F95COM if you need to use a specific command line for Fortran 95 files. You should normally set the $FORTRANCOM variable, which specifies the default command line for all Fortran versions.

F95COMSTR

The string displayed when a Fortran 95 source file is compiled to an object file. If this is not set, then $F95COM or $FORTRANCOM (the command line) is displayed.

F95FILESUFFIXES

The list of file extensions for which the F95 dialect will be used. By default, this is ['.f95']

F95FLAGS

General user-specified options that are passed to the Fortran 95 compiler. Note that this variable does *not* contain **–I** (or similar) include search path options that scons generates automatically from $F95PATH. See $_F95INCFLAGS below, for the variable that expands to those options. You only need to set $F95FLAGS if you need to define specific user options for Fortran 95 files. You should normally set the $FORTRANFLAGS variable, which specifies the user-specified options passed to the default Fortran compiler for all Fortran versions.

_F95INCFLAGS

An automatically-generated construction variable containing the Fortran 95 compiler command-line options for specifying directories to be searched for include files. The value of $_F95INCFLAGS is created by appending $INCPREFIX and $INCSUFFIX to the beginning and end of each directory in $F95PATH.

F95PATH

The list of directories that the Fortran 95 compiler will search for include directories. The implicit dependency scanner will search these directories for include files. Don't explicitly put include directory arguments in $F95FLAGS because the result will be non-portable and the directories will not be searched by the dependency scanner. Note: directory names in $F95PATH will be looked-up relative to the SConscript directory when they are used in a command. To force **scons** to look-up a directory relative to the root of the source tree use #: You only need to set $F95PATH if you need to define a specific include path for Fortran 95 files. You should normally set the $FORTRANPATH variable, which specifies the include path for the default Fortran compiler for all Fortran versions.

```
env = Environment(F95PATH='#/include')
```

The directory look-up can also be forced using the **Dir**() function:

```
include = Dir('include')
env = Environment(F95PATH=include)
```

The directory list will be added to command lines through the automatically-generated $_F95INCFLAGS construction variable, which is constructed by appending the values of the $INCPREFIX and $INCSUFFIX construction variables to the beginning and end of each directory in $F95PATH. Any command lines you define that need the F95PATH directory list should include $_F95INCFLAGS:

```
env = Environment(F95COM="my_compiler $_F95INCFLAGS -c -o $TARGET $SOURCE")
```

F95PPCOM

The command line used to compile a Fortran 95 source file to an object file after first running the file through the C preprocessor. Any options specified in the $F95FLAGS and $CPPFLAGS construction variables are included on this command line. You only need to set $F95PPCOM if you need to use a specific C-preprocessor command line for Fortran 95 files. You should normally set the $FORTRANPPCOM variable, which specifies the default C-preprocessor command line for all Fortran versions.

F95PPCOMSTR
The string displayed when a Fortran 95 source file is compiled to an object file after first running the file through the C preprocessor. If this is not set, then $F95PPCOM or $FORTRANPPCOM (the command line) is displayed.

F95PPFILESUFFIXES
The list of file extensions for which the compilation + preprocessor pass for F95 dialect will be used. By default, this is empty

File
A function that converts a string into a File instance relative to the target being built.

FORTRAN
The default Fortran compiler for all versions of Fortran.

FORTRANCOM
The command line used to compile a Fortran source file to an object file. By default, any options specified in the $FORTRANFLAGS, $CPPFLAGS, $_CPPDEFFLAGS, $_FORTRANMODFLAG, and $_FORTRANINCFLAGS construction variables are included on this command line.

FORTRANCOMSTR
The string displayed when a Fortran source file is compiled to an object file. If this is not set, then $FORTRANCOM (the command line) is displayed.

FORTRANFILESUFFIXES
The list of file extensions for which the FORTRAN dialect will be used. By default, this is ['.f', '.for', '.ftn']

FORTRANFLAGS
General user-specified options that are passed to the Fortran compiler. Note that this variable does *not* contain **-I** (or similar) include or module search path options that scons generates automatically from $FORTRANPATH. See $_FORTRANINCFLAGS and $_FORTRANMODFLAG, below, for the variables that expand those options.

_FORTRANINCFLAGS
An automatically-generated construction variable containing the Fortran compiler command-line options for specifying directories to be searched for include files and module files. The value of $_FORTRANINCFLAGS is created by prepending/appending $INCPREFIX and $INCSUFFIX to the beginning and end of each directory in $FORTRANPATH.

FORTRANMODDIR
Directory location where the Fortran compiler should place any module files it generates. This variable is empty, by default. Some Fortran compilers will internally append this directory in the search path for module files, as well.

FORTRANMODDIRPREFIX
The prefix used to specify a module directory on the Fortran compiler command line. This will be appended to the beginning of the directory in the $FORTRANMODDIR construction variables when the $_FORTRANMODFLAG variables is automatically generated.

FORTRANMODDIRSUFFIX

The suffix used to specify a module directory on the Fortran compiler command line. This will be appended to the beginning of the directory in the $FORTRANMODDIR construction variables when the $_FORTRANMODFLAG variables is automatically generated.

_FORTRANMODFLAG

An automatically-generated construction variable containing the Fortran compiler command-line option for specifying the directory location where the Fortran compiler should place any module files that happen to get generated during compilation. The value of $_FORTRANMODFLAG is created by prepending/appending $FORTRANMODDIRPREFIX and $FORTRANMODDIRSUFFIX to the beginning and end of the directory in $FORTRANMODDIR.

FORTRANMODPREFIX

The module file prefix used by the Fortran compiler. SCons assumes that the Fortran compiler follows the quasi-standard naming convention for module files of **module_name.mod**. As a result, this variable is left empty, by default. For situations in which the compiler does not necessarily follow the normal convention, the user may use this variable. Its value will be appended to every module file name as scons attempts to resolve dependencies.

FORTRANMODSUFFIX

The module file suffix used by the Fortran compiler. SCons assumes that the Fortran compiler follows the quasi-standard naming convention for module files of **module_name.mod**. As a result, this variable is set to ".mod", by default. For situations in which the compiler does not necessarily follow the normal convention, the user may use this variable. Its value will be appended to every module file name as scons attempts to resolve dependencies.

FORTRANPATH

The list of directories that the Fortran compiler will search for include files and (for some compilers) module files. The Fortran implicit dependency scanner will search these directories for include files (but not module files since they are autogenerated and, as such, may not actually exist at the time the scan takes place). Don't explicitly put include directory arguments in FORTRANFLAGS because the result will be non-portable and the directories will not be searched by the dependency scanner. Note: directory names in FORTRANPATH will be looked-up relative to the SConscript directory when they are used in a command. To force **scons** to look-up a directory relative to the root of the source tree use #:

```
env = Environment(FORTRANPATH='#/include')
```

The directory look-up can also be forced using the **Dir**() function:

```
include = Dir('include')
env = Environment(FORTRANPATH=include)
```

The directory list will be added to command lines through the automatically-generated $_FORTRANINCFLAGS construction variable, which is constructed by appending the values of the $INCPREFIX and $INCSUFFIX construction variables to the beginning and end of each directory in $FORTRANPATH. Any command lines you define that need the FORTRANPATH directory list should include $_FORTRANINCFLAGS:

```
env = Environment(FORTRANCOM="my_compiler $_FORTRANINCFLAGS -c -o $TARGET $SOURCE")
```

FORTRANPPCOM

The command line used to compile a Fortran source file to an object file after first running the file through the C preprocessor. By default, any options specified in the $FORTRANFLAGS,

$CPPFLAGS, $_CPPDEFFLAGS, $_FORTRANMODFLAG, and $_FORTRANINCFLAGS construction variables are included on this command line.

FORTRANPPCOMSTR
The string displayed when a Fortran source file is compiled to an object file after first running the file throught the C preprocessor. If this is not set, then $FORTRANPPCOM (the command line) is displayed.

FORTRANPPFILESUFFIXES
The list of file extensions for which the compilation + preprocessor pass for FORTRAN dialect will be used. By default, this is ['.fpp', '.FPP']

FORTRANSUFFIXES
The list of suffixes of files that will be scanned for Fortran implicit dependencies (INCLUDE lines and USE statements). The default list is:

```
[".f", ".F", ".for", ".FOR", ".ftn", ".FTN", ".fpp", ".FPP",
".f77", ".F77", ".f90", ".F90", ".f95", ".F95"]
```

FRAMEWORKPATH
On Mac OS X with gcc, a list containing the paths to search for frameworks. Used by the compiler to find framework-style includes like #include <Fmwk/Header.h>. Used by the linker to find user-specified frameworks when linking (see $FRAMEWORKS). For example:

```
env.AppendUnique(FRAMEWORKPATH='#myframeworkdir')
```

will add

```
... -Fmyframeworkdir
```

to the compiler and linker command lines.

_FRAMEWORKPATH
On Mac OS X with gcc, an automatically-generated construction variable containing the linker command-line options corresponding to $FRAMEWORKPATH.

FRAMEWORKPATHPREFIX
On Mac OS X with gcc, the prefix to be used for the FRAMEWORKPATH entries. (see $FRAMEWORKPATH). The default value is **–F**.

FRAMEWORKPREFIX
On Mac OS X with gcc, the prefix to be used for linking in frameworks (see $FRAMEWORKS). The default value is **–framework**.

_FRAMEWORKS
On Mac OS X with gcc, an automatically-generated construction variable containing the linker command-line options for linking with FRAMEWORKS.

FRAMEWORKS
On Mac OS X with gcc, a list of the framework names to be linked into a program or shared library or bundle. The default value is the empty list. For example:

env.AppendUnique(FRAMEWORKS=Split('System Cocoa SystemConfiguration'))

FRAMEWORKSFLAGS
On Mac OS X with gcc, general user-supplied frameworks options to be added at the end of a command line building a loadable module. (This has been largely superceded by the $FRAMEWORKPATH, $FRAMEWORKPATHPREFIX, $FRAMEWORKPREFIX and $FRAMEWORKS variables described above.)

GS The Ghostscript program used to convert PostScript to PDF files.

GSCOM
The Ghostscript command line used to convert PostScript to PDF files.

GSCOMSTR
The string displayed when Ghostscript is used to convert a PostScript file to a PDF file. If this is not set, then $GSCOM (the command line) is displayed.

GSFLAGS
General options passed to the Ghostscript program when converting PostScript to PDF files.

IDLSUFFIXES
The list of suffixes of files that will be scanned for IDL implicit dependencies (#include or import lines). The default list is:

[".idl", ".IDL"]

IMPLICIT_COMMAND_DEPENDENCIES
Controls whether or not SCons will add implicit dependencies for the commands executed to build targets.

By default, SCons will add to each target an implicit dependency on the command represented by the first argument on any command line it executes. The specific file for the dependency is found by searching the *PATH* variable in the *ENV* environment used to execute the command.

If the construction variable $IMPLICIT_COMMAND_DEPENDENCIES is set to a false value (**None**, **False**, **0**, etc.), then the implicit dependency will not be added to the targets built with that construction environment.

env = Environment(IMPLICIT_COMMAND_DEPENDENCIES = 0)

INCPREFIX
The prefix used to specify an include directory on the C compiler command line. This will be appended to the beginning of each directory in the $CPPPATH and $FORTRANPATH construction variables when the $_CPPINCFLAGS and $_FORTRANINCFLAGS variables are automatically generated.

INCSUFFIX
The suffix used to specify an include directory on the C compiler command line. This will be appended to the end of each directory in the $CPPPATH and $FORTRANPATH construction variables when the $_CPPINCFLAGS and $_FORTRANINCFLAGS variables are automatically

generated.

INSTALL

A function to be called to install a file into a destination file name. The default function copies the file into the destination (and sets the destination file's mode and permission bits to match the source file's). The function takes the following arguments:

```
def install(dest, source, env):
```

dest is the path name of the destination file. *source* is the path name of the source file. *env* is the construction environment (a dictionary of construction values) in force for this file installation.

INSTALLSTR

The string displayed when a file is installed into a destination file name. The default is:
Install file: "$SOURCE" as "$TARGET"

INTEL_C_COMPILER_VERSION

Set by the "intelc" Tool to the major version number of the Intel C compiler selected for use.

JAR The Java archive tool.

JARCHDIR

The directory to which the Java archive tool should change (using the **−C** option).

JARCOM

The command line used to call the Java archive tool.

JARCOMSTR

The string displayed when the Java archive tool is called If this is not set, then $JARCOM (the command line) is displayed.

```
env = Environment(JARCOMSTR = "JARchiving $SOURCES into $TARGET")
```

JARFLAGS

General options passed to the Java archive tool. By default this is set to **cf** to create the necessary **jar** file.

JARSUFFIX

The suffix for Java archives: **.jar** by default.

JAVABOOTCLASSPATH

Specifies the list of directories that will be added to the &javac; command line via the **−bootclasspath** option. The individual directory names will be separated by the operating system's path separate character (**:** on UNIX/Linux/POSIX, **;** on Windows).

JAVAC The Java compiler.

JAVACCOM

The command line used to compile a directory tree containing Java source files to corresponding Java class files. Any options specified in the $JAVACFLAGS construction variable are included on this command line.

JAVACCOMSTR
The string displayed when compiling a directory tree of Java source files to corresponding Java class files. If this is not set, then $JAVACCOM (the command line) is displayed.

env = Environment(JAVACCOMSTR = "Compiling class files $TARGETS from $SOURCES")

JAVACFLAGS
General options that are passed to the Java compiler.

JAVACLASSDIR
The directory in which Java class files may be found. This is stripped from the beginning of any Java .class file names supplied to the **JavaH** builder.

JAVACLASSPATH
Specifies the list of directories that will be searched for Java **.class** file. The directories in this list will be added to the &javac; and &javah; command lines via the **−classpath** option. The individual directory names will be separated by the operating system's path separate character (**:** on UNIX/Linux/POSIX, **;** on Windows).

Note that this currently just adds the specified directory via the **−classpath** option. &SCons; does not currently search the $JAVACLASSPATH directories for dependency **.class** files.

JAVACLASSSUFFIX
The suffix for Java class files; **.class** by default.

JAVAH The Java generator for C header and stub files.

JAVAHCOM
The command line used to generate C header and stub files from Java classes. Any options specified in the $JAVAHFLAGS construction variable are included on this command line.

JAVAHCOMSTR
The string displayed when C header and stub files are generated from Java classes. If this is not set, then $JAVAHCOM (the command line) is displayed.

env = Environment(JAVAHCOMSTR = "Generating header/stub file(s) $TARGETS from $SOURCES")

JAVAHFLAGS
General options passed to the C header and stub file generator for Java classes.

JAVASOURCEPATH
Specifies the list of directories that will be searched for input **.java** file. The directories in this list will be added to the &javac; command line via the **−sourcepath** option. The individual directory names will be separated by the operating system's path separate character (**:** on UNIX/Linux/POSIX, **;** on Windows).

Note that this currently just adds the specified directory via the **−sourcepath** option. &SCons; does not currently search the $JAVASOURCEPATH directories for dependency **.java** files.

JAVASUFFIX

The suffix for Java files; **.java** by default.

JAVAVERSION

Specifies the Java version being used by the **Java**() builder. This is *not* currently used to select one version of the Java compiler vs. another. Instead, you should set this to specify the version of Java supported by your &javac; compiler. The default is **1.4**.

This is sometimes necessary because Java 1.5 changed the file names that are created for nested anonymous inner classes, which can cause a mismatch with the files that &SCons; expects will be generated by the &javac; compiler. Setting $JAVAVERSION to **1.5** (or **1.6**, as appropriate) can make &SCons; realize that a Java 1.5 or 1.6 build is actually up to date.

LATEX

The LaTeX structured formatter and typesetter.

LATEXCOM

The command line used to call the LaTeX structured formatter and typesetter.

LATEXCOMSTR

The string displayed when calling the LaTeX structured formatter and typesetter. If this is not set, then $LATEXCOM (the command line) is displayed.

```
env = Environment(LATEXCOMSTR = "Building $TARGET from LaTeX input $SOURCES")
```

LATEXFLAGS

General options passed to the LaTeX structured formatter and typesetter.

LATEXRETRIES

The maximum number of times that LaTeX will be re-run if the **.log** generated by the $LATEXCOM command indicates that there are undefined references. The default is to try to resolve undefined references by re-running LaTeX up to three times.

LATEXSUFFIXES

The list of suffixes of files that will be scanned for LaTeX implicit dependencies (**\include** or **\import** files). The default list is:

```
[".tex", ".ltx", ".latex"]
```

LDMODULE

The linker for building loadable modules. By default, this is the same as $SHLINK.

LDMODULECOM

The command line for building loadable modules. On Mac OS X, this uses the $LDMODULE, $LDMODULEFLAGS and $FRAMEWORKSFLAGS variables. On other systems, this is the same as $SHLINK.

LDMODULECOMSTR

The string displayed when building loadable modules. If this is not set, then $LDMODULECOM (the command line) is displayed.

LDMODULEFLAGS
General user options passed to the linker for building loadable modules.

LDMODULEPREFIX
The prefix used for loadable module file names. On Mac OS X, this is null; on other systems, this is the same as $SHLIBPREFIX.

LDMODULESUFFIX
The suffix used for loadable module file names. On Mac OS X, this is null; on other systems, this is the same as $SHLIBSUFFIX.

LEX The lexical analyzer generator.

LEXCOM
The command line used to call the lexical analyzer generator to generate a source file.

LEXCOMSTR
The string displayed when generating a source file using the lexical analyzer generator. If this is not set, then $LEXCOM (the command line) is displayed.

```
env = Environment(LEXCOMSTR = "Lex'ing $TARGET from $SOURCES")
```

LEXFLAGS
General options passed to the lexical analyzer generator.

_LIBDIRFLAGS
An automatically-generated construction variable containing the linker command-line options for specifying directories to be searched for library. The value of $_LIBDIRFLAGS is created by appending $LIBDIRPREFIX and $LIBDIRSUFFIX to the beginning and end of each directory in $LIBPATH.

LIBDIRPREFIX
The prefix used to specify a library directory on the linker command line. This will be appended to the beginning of each directory in the $LIBPATH construction variable when the $_LIBDIRFLAGS variable is automatically generated.

LIBDIRSUFFIX
The suffix used to specify a library directory on the linker command line. This will be appended to the end of each directory in the $LIBPATH construction variable when the $_LIBDIRFLAGS variable is automatically generated.

_LIBFLAGS
An automatically-generated construction variable containing the linker command-line options for specifying libraries to be linked with the resulting target. The value of $_LIBFLAGS is created by appending $LIBLINKPREFIX and $LIBLINKSUFFIX to the beginning and end of each filename in $LIBS.

LIBLINKPREFIX
The prefix used to specify a library to link on the linker command line. This will be appended to the beginning of each library in the $LIBS construction variable when the $_LIBFLAGS variable

is automatically generated.

LIBLINKSUFFIX
: The suffix used to specify a library to link on the linker command line. This will be appended to the end of each library in the $LIBS construction variable when the $_LIBFLAGS variable is automatically generated.

LIBPATH
: The list of directories that will be searched for libraries. The implicit dependency scanner will search these directories for include files. Don't explicitly put include directory arguments in $LINKFLAGS or $SHLINKFLAGS because the result will be non-portable and the directories will not be searched by the dependency scanner. Note: directory names in LIBPATH will be looked-up relative to the SConscript directory when they are used in a command. To force **scons** to look-up a directory relative to the root of the source tree use #:

```
env = Environment(LIBPATH='#/libs')
```

The directory look-up can also be forced using the **Dir**() function:

```
libs = Dir('libs')
env = Environment(LIBPATH=libs)
```

The directory list will be added to command lines through the automatically-generated $_LIBDIRFLAGS construction variable, which is constructed by appending the values of the $LIBDIRPREFIX and $LIBDIRSUFFIX construction variables to the beginning and end of each directory in $LIBPATH. Any command lines you define that need the LIBPATH directory list should include $_LIBDIRFLAGS:

```
env = Environment(LINKCOM="my_linker $_LIBDIRFLAGS $_LIBFLAGS -o $TARGET $SOURCE")
```

LIBPREFIX
: The prefix used for (static) library file names. A default value is set for each platform (posix, win32, os2, etc.), but the value is overridden by individual tools (ar, mslib, sgiar, sunar, tlib, etc.) to reflect the names of the libraries they create.

LIBPREFIXES
: A list of all legal prefixes for library file names. When searching for library dependencies, SCons will look for files with these prefixes, the base library name, and suffixes in the $LIBSUFFIXES list.

LIBS
: A list of one or more libraries that will be linked with any executable programs created by this environment.

The library list will be added to command lines through the automatically-generated $_LIBFLAGS construction variable, which is constructed by appending the values of the $LIBLINKPREFIX and $LIBLINKSUFFIX construction variables to the beginning and end of each filename in $LIBS. Any command lines you define that need the LIBS library list should include $_LIBFLAGS:

```
env = Environment(LINKCOM="my_linker $_LIBDIRFLAGS $_LIBFLAGS -o $TARGET $SOURCE")
```

If you add a File object to the $LIBS list, the name of that file will be added to $_LIBFLAGS, and thus the link line, as is, without $LIBLINKPREFIX or $LIBLINKSUFFIX. For example:

env.Append(LIBS=File('/tmp/mylib.so'))

In all cases, scons will add dependencies from the executable program to all the libraries in this list.

LIBSUFFIX
The suffix used for (static) library file names. A default value is set for each platform (posix, win32, os2, etc.), but the value is overridden by individual tools (ar, mslib, sgiar, sunar, tlib, etc.) to reflect the names of the libraries they create.

LIBSUFFIXES
A list of all legal suffixes for library file names. When searching for library dependencies, SCons will look for files with prefixes, in the $LIBPREFIXES list, the base library name, and these suffixes.

LICENSE
The abbreviated name of the license under which this project is released (gpl, lpgl, bsd etc.). See http://www.opensource.org/licenses/alphabetical for a list of license names.

LINK The linker.

LINKCOM
The command line used to link object files into an executable.

LINKCOMSTR
The string displayed when object files are linked into an executable. If this is not set, then $LINKCOM (the command line) is displayed.

env = Environment(LINKCOMSTR = "Linking $TARGET")

LINKFLAGS
General user options passed to the linker. Note that this variable should *not* contain **–l** (or similar) options for linking with the libraries listed in $LIBS, nor **–L** (or similar) library search path options that scons generates automatically from $LIBPATH. See $_LIBFLAGS above, for the variable that expands to library-link options, and $_LIBDIRFLAGS above, for the variable that expands to library search path options.

M4 The M4 macro preprocessor.

M4COM
The command line used to pass files through the M4 macro preprocessor.

M4COMSTR
The string displayed when a file is passed through the M4 macro preprocessor. If this is not set, then $M4COM (the command line) is displayed.

M4FLAGS
General options passed to the M4 macro preprocessor.

MAKEINDEX
The makeindex generator for the TeX formatter and typesetter and the LaTeX structured formatter and typesetter.

MAKEINDEXCOM
The command line used to call the makeindex generator for the TeX formatter and typesetter and the LaTeX structured formatter and typesetter.

MAKEINDEXCOMSTR
The string displayed when calling the makeindex generator for the TeX formatter and typesetter and the LaTeX structured formatter and typesetter. If this is not set, then $MAKEINDEXCOM (the command line) is displayed.

MAKEINDEXFLAGS
General options passed to the makeindex generator for the TeX formatter and typesetter and the LaTeX structured formatter and typesetter.

MAXLINELENGTH
The maximum number of characters allowed on an external command line. On Win32 systems, link lines longer than this many characters are linked via a temporary file name.

MIDL The Microsoft IDL compiler.

MIDLCOM
The command line used to pass files to the Microsoft IDL compiler.

MIDLCOMSTR
The string displayed when the Microsoft IDL copmiler is called. If this is not set, then $MIDL-COM (the command line) is displayed.

MIDLFLAGS
General options passed to the Microsoft IDL compiler.

MSVS When the Microsoft Visual Studio tools are initialized, they set up this dictionary with the following keys:

VERSION: the version of MSVS being used (can be set via MSVS_VERSION)

VERSIONS: the available versions of MSVS installed

VCINSTALLDIR: installed directory of Visual C++

VSINSTALLDIR: installed directory of Visual Studio

FRAMEWORKDIR: installed directory of the .NET framework

FRAMEWORKVERSIONS: list of installed versions of the .NET framework, sorted latest to oldest.

FRAMEWORKVERSION: latest installed version of the .NET framework

FRAMEWORKSDKDIR: installed location of the .NET SDK.

PLATFORMSDKDIR: installed location of the Platform SDK.

PLATFORMSDK_MODULES: dictionary of installed Platform SDK modules, where the dictionary keys are keywords for the various modules, and the values are 2-tuples where the first is the release date, and the second is the version number.

If a value isn't set, it wasn't available in the registry.

MSVS_IGNORE_IDE_PATHS
Tells the MS Visual Studio tools to use minimal INCLUDE, LIB, and PATH settings, instead of the settings from the IDE.

For Visual Studio, SCons will (by default) automatically determine where MSVS is installed, and use the LIB, INCLUDE, and PATH variables set by the IDE. You can override this behavior by setting these variables after Environment initialization, or by setting **MSVS_IGNORE_IDE_PATHS = 1** in the Environment initialization. Specifying this will not leave these unset, but will set them to a minimal set of paths needed to run the tools successfully.

For VS6, the mininimal set is:
INCLUDE:'<VSDir>\VC98\ATL\include;<VSDir>\VC98\MFC\include;<VSDir>\VC98\include'
LIB:'<VSDir>\VC98\MFC\lib;<VSDir>\VC98\lib'
PATH:'<VSDir>\Common\MSDev98\bin;<VSDir>\VC98\bin'
For VS7, it is:
INCLUDE:'<VSDir>\Vc7\atlmfc\include;<VSDir>\Vc7\include'
LIB:'<VSDir>\Vc7\atlmfc\lib;<VSDir>\Vc7\lib'
PATH:'<VSDir>\Common7\Tools\bin;<VSDir>\Common7\Tools;<VSDir>\Vc7\bin'

Where '<VSDir>' is the installed location of Visual Studio.

MSVS_PROJECT_BASE_PATH
The string placed in a generated Microsoft Visual Studio solution file as the value of the **SccProjectFilePathRelativizedFromConnection0** and **SccProjectFilePathRelativizedFromConnection1** attributes of the **GlobalSection(SourceCodeControl)** section. There is no default value.

MSVS_PROJECT_GUID
The string placed in a generated Microsoft Visual Studio project file as the value of the **ProjectGUID** attribute. The string is also placed in the **SolutionUniqueID** attribute of the **GlobalSection(SourceCodeControl)** section of the Microsoft Visual Studio solution file. There is no default value.

MSVS_SCC_AUX_PATH
The path name placed in a generated Microsoft Visual Studio project file as the value of the **SccAuxPath** attribute if the **MSVS_SCC_PROVIDER** construction variable is also set. There is no default value.

MSVS_SCC_LOCAL_PATH
The path name placed in a generated Microsoft Visual Studio project file as the value of the **SccLocalPath** attribute if the **MSVS_SCC_PROVIDER** construction variable is also set. The path name is also placed in the **SccLocalPath0** and **SccLocalPath1** attributes of the **GlobalSection(SourceCodeControl)** section of the Microsoft Visual Studio solution file. There is no default value.

MSVS_SCC_PROJECT_NAME
The project name placed in a generated Microsoft Visual Studio project file as the value of the **SccProjectName** attribute. There is no default value.

MSVS_SCC_PROVIDER
The string placed in a generated Microsoft Visual Studio project file as the value of the **SccProvider** attribute. The string is also placed in the **SccProvider1** attribute of the **GlobalSection(SourceCodeControl)** section of the Microsoft Visual Studio solution file. There is no default value.

MSVS_USE_MFC_DIRS
Tells the MS Visual Studio tool(s) to use the MFC directories in its default paths for compiling and linking. The $MSVS_USE_MFC_DIRS variable has no effect if the **INCLUDE** or **LIB** environment variables are set explictly.

Under Visual Studio version 6, setting $MSVS_USE_MFC_DIRS to a non-zero value adds the **ATL\ATL** and **MFC\MFC** directories to the default **INCLUDE** external environment variable, and adds the **MFC\MFC** directory to the default **LIB** external environment variable.

Under Visual Studio version 7, setting $MSVS_USE_MFC_DIRS to a non-zero value adds the **atlmfc\atlmfc** directory to the default **INCLUDE** external environment variable, and adds the **atlmfc\atlmfc** directory to the default **LIB** external environment variable.

Under Visual Studio version 8, setting $MSVS_USE_MFC_DIRS to a non-zero value will, by default, add the **atlmfc\atlmfc** directory to the default **INCLUDE** external environment variable, and the **atlmfc\atlmfc** directory to the default **LIB** external environment variable. If, however, the **['MSVS']['PLATFORMSDKDIR']** variable is set, then the **mfc** and the **atl** subdirectories of the **PLATFORMSDKDIR** are added to the default value of the **INCLUDE** external environment variable, and the default value of the **LIB** external environment variable is left untouched.

MSVS_VERSION
Sets the preferred version of MSVS to use.

SCons will (by default) select the latest version of MSVS installed on your machine. So, if you have version 6 and version 7 (MSVS .NET) installed, it will prefer version 7. You can override this by specifying the **MSVS_VERSION** variable in the Environment initialization, setting it to the appropriate version ('6.0' or '7.0', for example). If the given version isn't installed, tool initialization will fail.

MSVSBUILDCOM
The build command line placed in a generated Microsoft Visual Studio project file. The default is to have Visual Studio invoke SCons with any specified build targets.

MSVSCLEANCOM
The clean command line placed in a generated Microsoft Visual Studio project file. The default is to have Visual Studio invoke SCons with the -c option to remove any specified targets.

MSVSENCODING
The encoding string placed in a generated Microsoft Visual Studio project file. The default is encoding **Windows-1252**.

MSVSPROJECTCOM
The action used to generate Microsoft Visual Studio project files.

MSVSPROJECTSUFFIX
The suffix used for Microsoft Visual Studio project (DSP) files. The default value is **.vcproj** when using Visual Studio version 7.x (.NET) or later version, and **.dsp** when using earlier versions of Visual Studio.

MSVSREBUILDCOM
The rebuild command line placed in a generated Microsoft Visual Studio project file. The default is to have Visual Studio invoke SCons with any specified rebuild targets.

MSVSSCONS
The SCons used in generated Microsoft Visual Studio project files. The default is the version of SCons being used to generate the project file.

MSVSSCONSCOM
The default SCons command used in generated Microsoft Visual Studio project files.

MSVSSCONSCRIPT
The sconscript file (that is, **SConstruct** or **SConscript** file) that will be invoked by Visual Studio project files (through the $MSVSSCONSCOM variable). The default is the same sconscript file that contains the call to **MSVSProject**() to build the project file.

MSVSSCONSFLAGS
The SCons flags used in generated Microsoft Visual Studio project files.

MSVSSOLUTIONCOM
The action used to generate Microsoft Visual Studio solution files.

MSVSSOLUTIONSUFFIX
The suffix used for Microsoft Visual Studio solution (DSW) files. The default value is **.sln** when using Visual Studio version 7.x (.NET), and **.dsw** when using earlier versions of Visual Studio.

MWCW_VERSION
The version number of the MetroWerks CodeWarrior C compiler to be used.

MWCW_VERSIONS
A list of installed versions of the MetroWerks CodeWarrior C compiler on this system.

NAME Specfies the name of the project to package.

no_import_lib
When set to non-zero, suppresses creation of a corresponding Windows static import lib by the **SharedLibrary** builder when used with MinGW, Microsoft Visual Studio or Metrowerks. This also suppresses creation of an export (.exp) file when using Microsoft Visual Studio.

OBJPREFIX
The prefix used for (static) object file names.

OBJSUFFIX
The suffix used for (static) object file names.

P4 The Perforce executable.

P4COM
The command line used to fetch source files from Perforce.

P4COMSTR
The string displayed when fetching a source file from Perforce. If this is not set, then $P4COM (the command line) is displayed.

P4FLAGS
General options that are passed to Perforce.

PACKAGEROOT
Specifies the directory where all files in resulting archive will be placed if applicable. The default value is "$NAME-$VERSION".

PACKAGETYPE
Selects the package type to build. Currently these are available:

* msi - Microsoft Installer
* rpm - Redhat Package Manger
* ipkg - Itsy Package Management System
* tarbz2 - compressed tar
* targz - compressed tar
* zip - zip file
* src_tarbz2 - compressed tar source
* src_targz - compressed tar source
* src_zip - zip file source

This may be overridden with the "package_type" command line option.

PACKAGEVERSION
The version of the package (not the underlying project). This is currently only used by the rpm packager and should reflect changes in the packaging, not the underlying project code itself.

PCH The Microsoft Visual C++ precompiled header that will be used when compiling object files. This variable is ignored by tools other than Microsoft Visual C++. When this variable is defined SCons will add options to the compiler command line to cause it to use the precompiled header, and will also set up the dependencies for the PCH file. Example:

```
env['PCH'] = 'StdAfx.pch'
```

PCHCOM
The command line used by the **PCH**() builder to generated a precompiled header.

PCHCOMSTR
The string displayed when generating a precompiled header. If this is not set, then $PCHCOM (the command line) is displayed.

PCHPDBFLAGS
A construction variable that, when expanded, adds the **/yD** flag to the command line only if the $PDB construction variable is set.

PCHSTOP
This variable specifies how much of a source file is precompiled. This variable is ignored by tools other than Microsoft Visual C++, or when the PCH variable is not being used. When this variable is define it must be a string that is the name of the header that is included at the end of the precompiled portion of the source files, or the empty string if the "#pragma hrdstop" construct is being used:

```
env['PCHSTOP'] = 'StdAfx.h'
```

PDB
The Microsoft Visual C++ PDB file that will store debugging information for object files, shared libraries, and programs. This variable is ignored by tools other than Microsoft Visual C++. When this variable is defined SCons will add options to the compiler and linker command line to cause them to generate external debugging information, and will also set up the dependencies for the PDB file. Example:

```
env['PDB'] = 'hello.pdb'
```

The Visual C++ compiler switch that SCons uses by default to generate PDB information is **/Z7**. This works correctly with parallel (**–j**) builds because it embeds the debug information in the intermediate object files, as opposed to sharing a single PDB file between multiple object files. This is also the only way to get debug information embedded into a static library. Using the **/Zi** instead may yield improved link-time performance, although parallel builds will no longer work. You can generate PDB files with the **/Zi** switch by overriding the default $CCPDBFLAGS variable; see the entry for that variable for specific examples.

PDFCOM
A deprecated synonym for $DVIPDFCOM.

PDFLATEX
The &pdflatex; utility.

PDFLATEXCOM
The command line used to call the &pdflatex; utility.

PDFLATEXCOMSTR
The string displayed when calling the &pdflatex; utility. If this is not set, then $PDFLATEXCOM (the command line) is displayed.

```
env = Environment(PDFLATEX;COMSTR = "Building $TARGET from LaTeX input $SOURCES")
```

PDFLATEXFLAGS
General options passed to the &pdflatex; utility.

PDFPREFIX
The prefix used for PDF file names.

PDFSUFFIX
The suffix used for PDF file names.

PDFTEX
The &pdftex; utility.

PDFTEXCOM
The command line used to call the &pdftex; utility.

PDFTEXCOMSTR
The string displayed when calling the &pdftex; utility. If this is not set, then $PDFTEXCOM (the command line) is displayed.

```
env = Environment(PDFTEXCOMSTR = "Building $TARGET from TeX input $SOURCES")
```

PDFTEXFLAGS
General options passed to the &pdftex; utility.

PKGCHK
On Solaris systems, the package-checking program that will be used (along with $PKGINFO) to look for installed versions of the Sun PRO C++ compiler. The default is **/usr/sbin/pgkchk**.

PKGINFO
On Solaris systems, the package information program that will be used (along with $PKGCHK) to look for installed versions of the Sun PRO C++ compiler. The default is **pkginfo**.

PLATFORM
The name of the platform used to create the Environment. If no platform is specified when the Environment is created, **scons** autodetects the platform.

```
env = Environment(tools = [])
if env['PLATFORM'] == 'cygwin':
   Tool('mingw')(env)
else:
   Tool('msvc')(env)
```

PRINT_CMD_LINE_FUNC
A Python function used to print the command lines as they are executed (assuming command printing is not disabled by the **−q** or **−s** options or their equivalents). The function should take four arguments: *s*, the command being executed (a string), *target*, the target being built (file node, list, or string name(s)), *source*, the source(s) used (file node, list, or string name(s)), and *env*, the environment being used.

The function must do the printing itself. The default implementation, used if this variable is not set or is None, is:

```
def print_cmd_line(s, target, source, env):
  sys.stdout.write(s + "\n")
```

Here's an example of a more interesting function:

```
def print_cmd_line(s, target, source, env):
   sys.stdout.write("Building %s -> %s...\n" %
    (' and '.join([str(x) for x in source]),
     ' and '.join([str(x) for x in target])))
env=Environment(PRINT_CMD_LINE_FUNC=print_cmd_line)
env.Program('foo', 'foo.c')
```

This just prints "Building *targetname* from *sourcename*..." instead of the actual commands. Such a function could also log the actual commands to a log file, for example.

PROGPREFIX
: The prefix used for executable file names.

PROGSUFFIX
: The suffix used for executable file names.

PSCOM
: The command line used to convert TeX DVI files into a PostScript file.

PSCOMSTR
: The string displayed when a TeX DVI file is converted into a PostScript file. If this is not set, then $PSCOM (the command line) is displayed.

PSPREFIX
: The prefix used for PostScript file names.

PSSUFFIX
: The prefix used for PostScript file names.

QT_AUTOSCAN
: Turn off scanning for mocable files. Use the Moc Builder to explicitely specify files to run moc on.

QT_BINPATH
: The path where the qt binaries are installed. The default value is '$QTDIR/bin'.

QT_CPPPATH
: The path where the qt header files are installed. The default value is '$QTDIR/include'. Note: If you set this variable to None, the tool won't change the $CPPPATH construction variable.

QT_DEBUG
: Prints lots of debugging information while scanning for moc files.

QT_LIB
: Default value is 'qt'. You may want to set this to 'qt-mt'. Note: If you set this variable to None, the tool won't change the $LIBS variable.

QT_LIBPATH
The path where the qt libraries are installed. The default value is '$QTDIR/lib'. Note: If you set this variable to None, the tool won't change the $LIBPATH construction variable.

QT_MOC
Default value is '$QT_BINPATH/moc'.

QT_MOCCXXPREFIX
Default value is ''. Prefix for moc output files, when source is a cxx file.

QT_MOCCXXSUFFIX
Default value is '.moc'. Suffix for moc output files, when source is a cxx file.

QT_MOCFROMCXXCOM
Command to generate a moc file from a cpp file.

QT_MOCFROMCXXCOMSTR
The string displayed when generating a moc file from a cpp file. If this is not set, then $QT_MOCFROMCXXCOM (the command line) is displayed.

QT_MOCFROMCXXFLAGS
Default value is '-i'. These flags are passed to moc, when moccing a C++ file.

QT_MOCFROMHCOM
Command to generate a moc file from a header.

QT_MOCFROMHCOMSTR
The string displayed when generating a moc file from a cpp file. If this is not set, then $QT_MOCFROMHCOM (the command line) is displayed.

QT_MOCFROMHFLAGS
Default value is ''. These flags are passed to moc, when moccing a header file.

QT_MOCHPREFIX
Default value is 'moc_'. Prefix for moc output files, when source is a header.

QT_MOCHSUFFIX
Default value is '$CXXFILESUFFIX'. Suffix for moc output files, when source is a header.

QT_UIC
Default value is '$QT_BINPATH/uic'.

QT_UICCOM
Command to generate header files from .ui files.

QT_UICCOMSTR
The string displayed when generating header files from .ui files. If this is not set, then $QT_UICCOM (the command line) is displayed.

QT_UICDECLFLAGS
Default value is ''. These flags are passed to uic, when creating a a h file from a .ui file.

QT_UICDECLPREFIX
Default value is ''. Prefix for uic generated header files.

QT_UICDECLSUFFIX
Default value is '.h'. Suffix for uic generated header files.

QT_UICIMPLFLAGS
Default value is ''. These flags are passed to uic, when creating a cxx file from a .ui file.

QT_UICIMPLPREFIX
Default value is 'uic_'. Prefix for uic generated implementation files.

QT_UICIMPLSUFFIX
Default value is '$CXXFILESUFFIX'. Suffix for uic generated implementation files.

QT_UISUFFIX
Default value is '.ui'. Suffix of designer input files.

QTDIR The qt tool tries to take this from os.environ. It also initializes all QT_* construction variables listed below. (Note that all paths are constructed with python's os.path.join() method, but are listed here with the '/' separator for easier reading.) In addition, the construction environment variables $CPPPATH, $LIBPATH and $LIBS may be modified and the variables PROGEMITTER, SHLIBEMITTER and LIBEMITTER are modified. Because the build-performance is affected when using this tool, you have to explicitly specify it at Environment creation:

Environment(tools=['default','qt'])

The qt tool supports the following operations:

Automatic moc file generation from header files. You do not have to specify moc files explicitly, the tool does it for you. However, there are a few preconditions to do so: Your header file must have the same filebase as your implementation file and must stay in the same directory. It must have one of the suffixes .h, .hpp, .H, .hxx, .hh. You can turn off automatic moc file generation by setting QT_AUTOSCAN to 0. See also the corresponding builder method **Moc()**

Automatic moc file generation from cxx files. As stated in the qt documentation, include the moc file at the end of the cxx file. Note that you have to include the file, which is generated by the transformation ${QT_MOCCXXPREFIX}<basename>${QT_MOCCXXSUFFIX}, by default <basename>.moc. A warning is generated after building the moc file, if you do not include the correct file. If you are using VariantDir, you may need to specify duplicate=1. You can turn off automatic moc file generation by setting QT_AUTOSCAN to 0. See also the corresponding **Moc**() builder method.

Automatic handling of .ui files. The implementation files generated from .ui files are handled much the same as yacc or lex files. Each .ui file given as a source of Program, Library or SharedLibrary will generate three files, the declaration file, the implementation file and a moc file. Because there are also generated headers, you may need to specify duplicate=1 in calls to VariantDir. See also the corresponding **Uic**() builder method.

RANLIB
The archive indexer.

RANLIBCOM
The command line used to index a static library archive.

RANLIBCOMSTR
The string displayed when a static library archive is indexed. If this is not set, then $RANLIBCOM (the command line) is displayed.

env = Environment(RANLIBCOMSTR = "Indexing $TARGET")

RANLIBFLAGS
General options passed to the archive indexer.

RC
The resource compiler used to build a Microsoft Visual C++ resource file.

RCCOM
The command line used to build a Microsoft Visual C++ resource file.

RCCOMSTR
The string displayed when invoking the resource compiler to build a Microsoft Visual C++ resource file. If this is not set, then $RCCOM (the command line) is displayed.

RCFLAGS
The flags passed to the resource compiler by the RES builder.

RCINCFLAGS
An automatically-generated construction variable containing the command-line options for specifying directories to be searched by the resource compiler. The value of $RCINCFLAGS is created by appending $RCINCPREFIX and $RCINCSUFFIX to the beginning and end of each directory in $CPPPATH.

RCINCPREFIX
The prefix (flag) used to specify an include directory on the resource compiler command line. This will be appended to the beginning of each directory in the $CPPPATH construction variable when the $RCINCFLAGS variable is expanded.

RCINCSUFFIX
The suffix used to specify an include directory on the resource compiler command line. This will be appended to the end of each directory in the $CPPPATH construction variable when the $RCINCFLAGS variable is expanded.

RCS
The RCS executable. Note that this variable is not actually used for the command to fetch source files from RCS; see the $RCS_CO construction variable, below.

RCS_CO
The RCS "checkout" executable, used to fetch source files from RCS.

RCS_COCOM
The command line used to fetch (checkout) source files from RCS.

RCS_COCOMSTR
The string displayed when fetching a source file from RCS. If this is not set, then $RCS_COCOM (the command line) is displayed.

RCS_COFLAGS
Options that are passed to the $RCS_CO command.

RDirs A function that converts a string into a list of Dir instances by searching the repositories.

REGSVR
The program used on Windows systems to register a newly-built DLL library whenever the **SharedLibrary**() builder is passed a keyword argument of **register=1**.

REGSVRCOM
The command line used on Windows systems to register a newly-built DLL library whenever the **SharedLibrary**() builder is passed a keyword argument of **register=1**.

REGSVRCOMSTR
The string displayed when registering a newly-built DLL file. If this is not set, then $REGSVRCOM (the command line) is displayed.

REGSVRFLAGS
Flags passed to the DLL registration program on Windows systems when a newly-built DLL library is registered. By default, this includes the **/s** that prevents dialog boxes from popping up and requiring user attention.

RMIC The Java RMI stub compiler.

RMICCOM
The command line used to compile stub and skeleton class files from Java classes that contain RMI implementations. Any options specified in the $RMICFLAGS construction variable are included on this command line.

RMICCOMSTR
The string displayed when compiling stub and skeleton class files from Java classes that contain RMI implementations. If this is not set, then $RMICCOM (the command line) is displayed.

```
env = Environment(RMICCOMSTR = "Generating stub/skeleton class files $TARGETS from $SOURCES")
```

RMICFLAGS
General options passed to the Java RMI stub compiler.

_RPATH
An automatically-generated construction variable containing the rpath flags to be used when linking a program with shared libraries. The value of $_RPATH is created by appending $RPATHPREFIX and $RPATHSUFFIX to the beginning and end of each directory in $RPATH.

RPATH A list of paths to search for shared libraries when running programs. Currently only used in the GNU (gnulink), IRIX (sgilink) and Sun (sunlink) linkers. Ignored on platforms and toolchains that don't support it. Note that the paths added to RPATH are not transformed by **scons** in any way: if you want an absolute path, you must make it absolute yourself.

RPATHPREFIX
The prefix used to specify a directory to be searched for shared libraries when running programs. This will be appended to the beginning of each directory in the $RPATH construction variable when the $_RPATH variable is automatically generated.

RPATHSUFFIX
The suffix used to specify a directory to be searched for shared libraries when running programs. This will be appended to the end of each directory in the $RPATH construction variable when the $_RPATH variable is automatically generated.

RPCGEN
The RPC protocol compiler.

RPCGENCLIENTFLAGS
Options passed to the RPC protocol compiler when generating client side stubs. These are in addition to any flags specified in the $RPCGENFLAGS construction variable.

RPCGENFLAGS
General options passed to the RPC protocol compiler.

RPCGENHEADERFLAGS
Options passed to the RPC protocol compiler when generating a header file. These are in addition to any flags specified in the $RPCGENFLAGS construction variable.

RPCGENSERVICEFLAGS
Options passed to the RPC protocol compiler when generating server side stubs. These are in addition to any flags specified in the $RPCGENFLAGS construction variable.

RPCGENXDRFLAGS
Options passed to the RPC protocol compiler when generating XDR routines. These are in addition to any flags specified in the $RPCGENFLAGS construction variable.

SCANNERS
A list of the available implicit dependency scanners. New file scanners may be added by appending to this list, although the more flexible approach is to associate scanners with a specific Builder. See the sections "Builder Objects" and "Scanner Objects," below, for more information.

SCCS The SCCS executable.

SCCSCOM
The command line used to fetch source files from SCCS.

SCCSCOMSTR
The string displayed when fetching a source file from a CVS repository. If this is not set, then $SCCSCOM (the command line) is displayed.

SCCSFLAGS
General options that are passed to SCCS.

SCCSGETFLAGS
Options that are passed specifically to the SCCS "get" subcommand. This can be set, for example, to **−e** to check out editable files from SCCS.

SCONS_HOME
The (optional) path to the SCons library directory, initialized from the external environment. If set, this is used to construct a shorter and more efficient search path in the $MSVSSCONS command line executed from Microsoft Visual Studio project files.

SHCC The C compiler used for generating shared-library objects.

SHCCCOM
The command line used to compile a C source file to a shared-library object file. Any options specified in the $SHCFLAGS, $SHCCFLAGS and $CPPFLAGS construction variables are included on this command line.

SHCCCOMSTR
The string displayed when a C source file is compiled to a shared object file. If this is not set, then $SHCCCOM (the command line) is displayed.

```
env = Environment(SHCCCOMSTR = "Compiling shared object $TARGET")
```

SHCCFLAGS
Options that are passed to the C and C++ compilers to generate shared-library objects.

SHCFLAGS
Options that are passed to the C compiler (only; not C++) to generate shared-library objects.

SHCXX
The C++ compiler used for generating shared-library objects.

SHCXXCOM
The command line used to compile a C++ source file to a shared-library object file. Any options specified in the $SHCXXFLAGS and $CPPFLAGS construction variables are included on this command line.

SHCXXCOMSTR
The string displayed when a C++ source file is compiled to a shared object file. If this is not set, then $SHCXXCOM (the command line) is displayed.

```
env = Environment(SHCXXCOMSTR = "Compiling shared object $TARGET")
```

SHCXXFLAGS
Options that are passed to the C++ compiler to generate shared-library objects.

SHELL
A string naming the shell program that will be passed to the $SPAWN function. See the $SPAWN construction variable for more information.

SHF77
The Fortran 77 compiler used for generating shared-library objects. You should normally set the $SHFORTRAN variable, which specifies the default Fortran compiler for all Fortran versions. You only need to set $SHF77 if you need to use a specific compiler or compiler version for Fortran 77 files.

SHF77COM
The command line used to compile a Fortran 77 source file to a shared-library object file. You only need to set $SHF77COM if you need to use a specific command line for Fortran 77 files. You should normally set the $SHFORTRANCOM variable, which specifies the default command line for all Fortran versions.

SHF77COMSTR
The string displayed when a Fortran 77 source file is compiled to a shared-library object file. If this is not set, then $SHF77COM or $SHFORTRANCOM (the command line) is displayed.

SHF77FLAGS
Options that are passed to the Fortran 77 compiler to generated shared-library objects. You only need to set $SHF77FLAGS if you need to define specific user options for Fortran 77 files. You should normally set the $SHFORTRANFLAGS variable, which specifies the user-specified options passed to the default Fortran compiler for all Fortran versions.

SHF77PPCOM
The command line used to compile a Fortran 77 source file to a shared-library object file after first running the file through the C preprocessor. Any options specified in the $SHF77FLAGS and $CPPFLAGS construction variables are included on this command line. You only need to set $SHF77PPCOM if you need to use a specific C-preprocessor command line for Fortran 77 files. You should normally set the $SHFORTRANPPCOM variable, which specifies the default C-preprocessor command line for all Fortran versions.

SHF77PPCOMSTR
The string displayed when a Fortran 77 source file is compiled to a shared-library object file after first running the file through the C preprocessor. If this is not set, then $SHF77PPCOM or $SHFORTRANPPCOM (the command line) is displayed.

SHF90
The Fortran 90 compiler used for generating shared-library objects. You should normally set the $SHFORTRAN variable, which specifies the default Fortran compiler for all Fortran versions. You only need to set $SHF90 if you need to use a specific compiler or compiler version for Fortran 90 files.

SHF90COM
The command line used to compile a Fortran 90 source file to a shared-library object file. You only need to set $SHF90COM if you need to use a specific command line for Fortran 90 files. You should normally set the $SHFORTRANCOM variable, which specifies the default command line for all Fortran versions.

SHF90COMSTR
The string displayed when a Fortran 90 source file is compiled to a shared-library object file. If this is not set, then $SHF90COM or $SHFORTRANCOM (the command line) is displayed.

SHF90FLAGS
Options that are passed to the Fortran 90 compiler to generated shared-library objects. You only need to set $SHF90FLAGS if you need to define specific user options for Fortran 90 files. You should normally set the $SHFORTRANFLAGS variable, which specifies the user-specified options passed to the default Fortran compiler for all Fortran versions.

SHF90PPCOM
The command line used to compile a Fortran 90 source file to a shared-library object file after first running the file through the C preprocessor. Any options specified in the $SHF90FLAGS and $CPPFLAGS construction variables are included on this command line. You only need to set $SHF90PPCOM if you need to use a specific C-preprocessor command line for Fortran 90 files. You should normally set the $SHFORTRANPPCOM variable, which specifies the default C-preprocessor command line for all Fortran versions.

SHF90PPCOMSTR
The string displayed when a Fortran 90 source file is compiled to a shared-library object file after first running the file through the C preprocessor. If this is not set, then $SHF90PPCOM or $SHFORTRANPPCOM (the command line) is displayed.

SHF95
The Fortran 95 compiler used for generating shared-library objects. You should normally set the $SHFORTRAN variable, which specifies the default Fortran compiler for all Fortran versions. You only need to set $SHF95 if you need to use a specific compiler or compiler version for Fortran 95 files.

SHF95COM
The command line used to compile a Fortran 95 source file to a shared-library object file. You only need to set $SHF95COM if you need to use a specific command line for Fortran 95 files. You should normally set the $SHFORTRANCOM variable, which specifies the default command line for all Fortran versions.

SHF95COMSTR
The string displayed when a Fortran 95 source file is compiled to a shared-library object file. If this is not set, then $SHF95COM or $SHFORTRANCOM (the command line) is displayed.

SHF95FLAGS
Options that are passed to the Fortran 95 compiler to generated shared-library objects. You only need to set $SHF95FLAGS if you need to define specific user options for Fortran 95 files. You should normally set the $SHFORTRANFLAGS variable, which specifies the user-specified options passed to the default Fortran compiler for all Fortran versions.

SHF95PPCOM
The command line used to compile a Fortran 95 source file to a shared-library object file after first running the file through the C preprocessor. Any options specified in the $SHF95FLAGS and $CPPFLAGS construction variables are included on this command line. You only need to set $SHF95PPCOM if you need to use a specific C-preprocessor command line for Fortran 95 files. You should normally set the $SHFORTRANPPCOM variable, which specifies the default C-preprocessor command line for all Fortran versions.

SHF95PPCOMSTR

The string displayed when a Fortran 95 source file is compiled to a shared-library object file after first running the file through the C preprocessor. If this is not set, then $SHF95PPCOM or $SHFORTRANPPCOM (the command line) is displayed.

SHFORTRAN

The default Fortran compiler used for generating shared-library objects.

SHFORTRANCOM

The command line used to compile a Fortran source file to a shared-library object file.

SHFORTRANCOMSTR

The string displayed when a Fortran source file is compiled to a shared-library object file. If this is not set, then $SHFORTRANCOM (the command line) is displayed.

SHFORTRANFLAGS

Options that are passed to the Fortran compiler to generate shared-library objects.

SHFORTRANPPCOM

The command line used to compile a Fortran source file to a shared-library object file after first running the file through the C preprocessor. Any options specified in the $SHFORTRANFLAGS and $CPPFLAGS construction variables are included on this command line.

SHFORTRANPPCOMSTR

The string displayed when a Fortran source file is compiled to a shared-library object file after first running the file throught the C preprocessor. If this is not set, then $SHFORTRANPPCOM (the command line) is displayed.

SHLIBPREFIX

The prefix used for shared library file names.

SHLIBSUFFIX

The suffix used for shared library file names.

SHLINK

The linker for programs that use shared libraries.

SHLINKCOM

The command line used to link programs using shared libaries.

SHLINKCOMSTR

The string displayed when programs using shared libraries are linked. If this is not set, then $SHLINKCOM (the command line) is displayed.

```
env = Environment(SHLINKCOMSTR = "Linking shared $TARGET")
```

SHLINKFLAGS

General user options passed to the linker for programs using shared libraries. Note that this variable should *not* contain **–l** (or similar) options for linking with the libraries listed in $LIBS, nor **–L** (or similar) include search path options that scons generates automatically from $LIBPATH.

See $_LIBFLAGS above, for the variable that expands to library-link options, and $_LIBDIRFLAGS above, for the variable that expands to library search path options.

SHOBJPREFIX
The prefix used for shared object file names.

SHOBJSUFFIX
The suffix used for shared object file names.

SOURCE
A reserved variable name that may not be set or used in a construction environment. (See "Variable Substitution," below.)

SOURCE_URL
The URL (web address) of the location from which the project was retrieved. This is used to fill in the **Source:** field in the controlling information for Ipkg and RPM packages.

SOURCES
A reserved variable name that may not be set or used in a construction environment. (See "Variable Substitution," below.)

SPAWN
A command interpreter function that will be called to execute command line strings. The function must expect the following arguments:

```
def spawn(shell, escape, cmd, args, env):
```

sh is a string naming the shell program to use. *escape* is a function that can be called to escape shell special characters in the command line. *cmd* is the path to the command to be executed. *args* is the arguments to the command. *env* is a dictionary of the environment variables in which the command should be executed.

SUMMARY
A short summary of what the project is about. This is used to fill in the **Summary:** field in the controlling information for Ipkg and RPM packages, and as the **Description:** field in MSI packages.

SWIG The scripting language wrapper and interface generator.

SWIGCFILESUFFIX
The suffix that will be used for intermediate C source files generated by the scripting language wrapper and interface generator. The default value is **_wrap**$CFILESUFFIX. By default, this value is used whenever the **−c++** option is *not* specified as part of the $SWIGFLAGS construction variable.

SWIGCOM
The command line used to call the scripting language wrapper and interface generator.

SWIGCOMSTR
The string displayed when calling the scripting language wrapper and interface generator. If this is not set, then $SWIGCOM (the command line) is displayed.

SWIGCXXFILESUFFIX
The suffix that will be used for intermediate C++ source files generated by the scripting language wrapper and interface generator. The default value is **_wrap**$CFILESUFFIX. By default, this value is used whenever the **−c++** option is specified as part of the $SWIGFLAGS construction variable.

SWIGFLAGS
General options passed to the scripting language wrapper and interface generator. This is where you should set **−python**, **−perl5**, **−tcl**, or whatever other options you want to specify to SWIG. If you set the **−c++** option in this variable, **scons** will, by default, generate a C++ intermediate source file with the extension that is specified as the $CXXFILESUFFIX variable.

_SWIGINCFLAGS
An automatically-generated construction variable containing the SWIG command-line options for specifying directories to be searched for included files. The value of $_SWIGINCFLAGS is created by appending $SWIGINCPREFIX and $SWIGINCSUFFIX to the beginning and end of each directory in $SWIGPATH.

SWIGINCPREFIX
The prefix used to specify an include directory on the SWIG command line. This will be appended to the beginning of each directory in the $SWIGPATH construction variable when the $_SWIGINCFLAGS variable is automatically generated.

SWIGINCSUFFIX
The suffix used to specify an include directory on the SWIG command line. This will be appended to the end of each directory in the $SWIGPATH construction variable when the $_SWIGINCFLAGS variable is automatically generated.

SWIGOUTDIR
Specifies the output directory in which the scripting language wrapper and interface generator should place generated language-specific files. This will be used by SCons to identify the files that will be generated by the &swig; call, and translated into the **swig -outdir** option on the command line.

SWIGPATH
The list of directories that the scripting language wrapper and interface generate will search for included files. The SWIG implicit dependency scanner will search these directories for include files. The default is to use the same path specified as $CPPPATH.

Don't explicitly put include directory arguments in SWIGFLAGS; the result will be non-portable and the directories will not be searched by the dependency scanner. Note: directory names in SWIGPATH will be looked-up relative to the SConscript directory when they are used in a command. To force **scons** to look-up a directory relative to the root of the source tree use #:

```
env = Environment(SWIGPATH='#/include')
```

The directory look-up can also be forced using the **Dir**() function:

```
include = Dir('include')
env = Environment(SWIGPATH=include)
```

The directory list will be added to command lines through the automatically-generated $_SWIGINCFLAGS construction variable, which is constructed by appending the values of the

$SWIGINCPREFIX and $SWIGINCSUFFIX construction variables to the beginning and end of each directory in $SWIGPATH. Any command lines you define that need the SWIGPATH directory list should include $_SWIGINCFLAGS:

```
env = Environment(SWIGCOM="my_swig -o $TARGET $_SWIGINCFLAGS $SORUCES")
```

TAR
The tar archiver.

TARCOM
The command line used to call the tar archiver.

TARCOMSTR
The string displayed when archiving files using the tar archiver. If this is not set, then $TARCOM (the command line) is displayed.

```
env = Environment(TARCOMSTR = "Archiving $TARGET")
```

TARFLAGS
General options passed to the tar archiver.

TARGET
A reserved variable name that may not be set or used in a construction environment. (See "Variable Substitution," below.)

TARGETS
A reserved variable name that may not be set or used in a construction environment. (See "Variable Substitution," below.)

TARSUFFIX
The suffix used for tar file names.

TEMPFILEPREFIX
The prefix for a temporary file used to execute lines longer than $MAXLINELENGTH. The default is '@'. This may be set for toolchains that use other values, such as '-@' for the diab compiler or '-via' for ARM toolchain.

TEX
The TeX formatter and typesetter.

TEXCOM
The command line used to call the TeX formatter and typesetter.

TEXCOMSTR
The string displayed when calling the TeX formatter and typesetter. If this is not set, then $TEXCOM (the command line) is displayed.

```
env = Environment(TEXCOMSTR = "Building $TARGET from TeX input $SOURCES")
```

TEXFLAGS
General options passed to the TeX formatter and typesetter.

TEXINPUTS
List of directories that the LaTeX programm will search for include directories. The LaTeX implicit dependency scanner will search these directories for \include and \import files.

TOOLS
A list of the names of the Tool specifications that are part of this construction environment.

VENDOR
The person or organization who supply the packaged software. This is used to fill in the **Vendor:** field in the controlling information for RPM packages, and the **Manufacturer:** field in the controlling information for MSI packages.

VERSION
The version of the project, specified as a string.

WIN32_INSERT_DEF
A deprecated synonym for $WINDOWS_INSERT_DEF.

WIN32DEFPREFIX
A deprecated synonym for $WINDOWSDEFPREFIX.

WIN32DEFSUFFIX
A deprecated synonym for $WINDOWSDEFSUFFIX.

WIN32EXPPREFIX
A deprecated synonym for $WINDOWSEXPSUFFIX.

WIN32EXPSUFFIX
A deprecated synonym for $WINDOWSEXPSUFFIX.

WINDOWS_INSERT_DEF
When this is set to true, a library build of a Windows shared library (**.dll**file) will also build a corresponding **.def** file at the same time, if a **.def** file is not already listed as a build target. The default is 0 (do not build a **.def** file).

WINDOWS_INSERT_MANIFEST
When this is set to true, **scons** will be aware of the **.manifest** files generated by Microsoft Visua C/C++ 8.

WINDOWSDEFPREFIX
The prefix used for Windows **.def**file names.

WINDOWSDEFSUFFIX
The suffix used for Windows **.def** file names.

WINDOWSEXPPREFIX
The prefix used for Windows **.exp** file names.

WINDOWSEXPSUFFIX
The suffix used for Windows **.exp** file names.

WINDOWSPROGMANIFESTPREFIX
The prefix used for executable program **.manifest** files generated by Microsoft Visual C/C++.

WINDOWSPROGMANIFESTSUFFIX
The suffix used for executable program **.manifest** files generated by Microsoft Visual C/C++.

WINDOWSSHLIBMANIFESTPREFIX
The prefix used for shared library **.manifest** files generated by Microsoft Visual C/C++.

WINDOWSSHLIBMANIFESTSUFFIX
The suffix used for shared library **.manifest** files generated by Microsoft Visual C/C++.

X_IPK_DEPENDS
This is used to fill in the **Depends:** field in the controlling information for Ipkg packages.

X_IPK_DESCRIPTION
This is used to fill in the **Description:** field in the controlling information for Ipkg packages. The default value is **$SUMMARY\$SUMMARY**

X_IPK_MAINTAINER
This is used to fill in the **Maintainer:** field in the controlling information for Ipkg packages.

X_IPK_PRIORITY
This is used to fill in the **Priority:** field in the controlling information for Ipkg packages.

X_IPK_SECTION
This is used to fill in the **Section:** field in the controlling information for Ipkg packages.

X_MSI_LANGUAGE
This is used to fill in the **Language:** attribute in the controlling information for MSI packages.

X_MSI_LICENSE_TEXT
The text of the software license in RTF format. Carriage return characters will be replaced with the RTF equivalent \ar.

X_MSI_UPGRADE_CODE
TODO

X_RPM_AUTOREQPROV
This is used to fill in the **AutoReqProv:** field in the RPM **.spec** file.

X_RPM_BUILD
internal, but overridable

X_RPM_BUILDREQUIRES
This is used to fill in the **BuildRequires:** field in the RPM **.spec** file.

X_RPM_BUILDROOT
internal, but overridable

X_RPM_CLEAN
internal, but overridable

X_RPM_CONFLICTS
This is used to fill in the **Conflicts:** field in the RPM **.spec** file.

X_RPM_DEFATTR
This value is used as the default attributes for the files in the RPM package. The default value is **(-,root,root)**.

X_RPM_DISTRIBUTION
This is used to fill in the **Distribution:** field in the RPM **.spec** file.

X_RPM_EPOCH
This is used to fill in the **Epoch:** field in the controlling information for RPM packages.

X_RPM_EXCLUDEARCH
This is used to fill in the **ExcludeArch:** field in the RPM **.spec** file.

X_RPM_EXLUSIVEARCH
This is used to fill in the **ExclusiveArch:** field in the RPM **.spec** file.

X_RPM_GROUP
This is used to fill in the **Group:** field in the RPM **.spec** file.

X_RPM_GROUP_lang
This is used to fill in the **Group(lang):** field in the RPM **.spec** file. Note that *lang* is not literal and should be replaced by the appropriate language code.

X_RPM_ICON
This is used to fill in the **Icon:** field in the RPM **.spec** file.

X_RPM_INSTALL
internal, but overridable

X_RPM_PACKAGER
This is used to fill in the **Packager:** field in the RPM **.spec** file.

X_RPM_POSTINSTALL
This is used to fill in the **%post:** section in the RPM **.spec** file.

X_RPM_POSTUNINSTALL
This is used to fill in the **%postun:** section in the RPM **.spec** file.

X_RPM_PREFIX
This is used to fill in the **Prefix:** field in the RPM **.spec** file.

X_RPM_PREINSTALL
This is used to fill in the **%pre:** section in the RPM **.spec** file.

X_RPM_PREP
internal, but overridable

X_RPM_PREUNINSTALL
This is used to fill in the **%preun:** section in the RPM **.spec** file.

X_RPM_PROVIDES
This is used to fill in the **Provides:** field in the RPM **.spec** file.

X_RPM_REQUIRES
This is used to fill in the **Requires:** field in the RPM **.spec** file.

X_RPM_SERIAL
This is used to fill in the **Serial:** field in the RPM **.spec** file.

X_RPM_URL
This is used to fill in the **Url:** field in the RPM **.spec** file.

YACC The parser generator.

YACCCOM
The command line used to call the parser generator to generate a source file.

YACCCOMSTR
The string displayed when generating a source file using the parser generator. If this is not set, then $YACCCOM (the command line) is displayed.

```
env = Environment(YACCCOMSTR = "Yacc'ing $TARGET from $SOURCES")
```

YACCFLAGS
General options passed to the parser generator. If $YACCFLAGS contains a **-d** option, SCons assumes that the call will also create a .h file (if the yacc source file ends in a .y suffix) or a .hpp file (if the yacc source file ends in a .yy suffix)

YACCHFILESUFFIX
The suffix of the C header file generated by the parser generator when the **-d** option is used. Note that setting this variable does not cause the parser generator to generate a header file with the specified suffix, it exists to allow you to specify what suffix the parser generator will use of its own accord. The default value is **.h**.

YACCHXXFILESUFFIX
The suffix of the C++ header file generated by the parser generator when the **−d** option is used. Note that setting this variable does not cause the parser generator to generate a header file with the specified suffix, it exists to allow you to specify what suffix the parser generator will use of its own accord. The default value is **.hpp**, except on Mac OS X, where the default is **${TARGET.suffix}.h**. because the default &bison; parser generator just appends **.h** to the name of the generated C++ file.

YACCVCGFILESUFFIX
The suffix of the file containing the VCG grammar automaton definition when the **−-graph=** option is used. Note that setting this variable does not cause the parser generator to generate a VCG file with the specified suffix, it exists to allow you to specify what suffix the parser generator will use of its own accord. The default value is **.vcg**.

ZIP The zip compression and file packaging utility.

ZIPCOM
The command line used to call the zip utility, or the internal Python function used to create a zip archive.

ZIPCOMPRESSION
The *compression* flag from the Python **zipfile** module used by the internal Python function to control whether the zip archive is compressed or not. The default value is **zipfile.ZIP_DEFLATED**, which creates a compressed zip archive. This value has no effect when using Python 1.5.2 or if the **zipfile** module is otherwise unavailable.

ZIPCOMSTR
The string displayed when archiving files using the zip utility. If this is not set, then $ZIPCOM (the command line or internal Python function) is displayed.

```
env = Environment(ZIPCOMSTR = "Zipping $TARGET")
```

ZIPFLAGS
General options passed to the zip utility.

ZIPSUFFIX
The suffix used for zip file names.

Construction variables can be retrieved and set using the **Dictionary** method of the construction environment:

```
dict = env.Dictionary()
dict["CC"] = "cc"
```

or using the [] operator:

```
env["CC"] = "cc"
```

Construction variables can also be passed to the construction environment constructor:

```
env = Environment(CC="cc")
```

or when copying a construction environment using the **Clone** method:

```
env2 = env.Clone(CC="cl.exe")
```

Configure Contexts

scons supports *configure contexts,* an integrated mechanism similar to the various AC_CHECK macros in GNU autoconf for testing for the existence of C header files, libraries, etc. In contrast to autoconf, **scons** does not maintain an explicit cache of the tested values, but uses its normal dependency tracking to keep the checked values up to date. However, users may override this behaviour with the **--config** command line option.

The following methods can be used to perform checks:

Configure(*env*, [*custom_tests*, *conf_dir*, *log_file*, *config_h*, *clean*, *help])*

env.Configure([*custom_tests*, *conf_dir*, *log_file*, *config_h*, *clean*, *help])*

This creates a configure context, which can be used to perform checks. *env* specifies the environment for building the tests. This environment may be modified when performing checks. *custom_tests* is a dictionary containing custom tests. See also the section about custom tests below. By default, no custom tests are added to the configure context. *conf_dir* specifies a directory where the test cases are built. Note that this directory is not used for building normal targets. The default value is the directory #/.sconf_temp. *log_file* specifies a file which collects the output from commands that are executed to check for the existence of header files, libraries, etc. The default is the file #/config.log. If you are using the **VariantDir**() method, you may want to specify a subdirectory under your variant directory. *config_h* specifies a C header file where the results of tests will be written, e.g. #define HAVE_STDIO_H, #define HAVE_LIBM, etc. The default is to not write a **config.h** file. You can specify the same **config.h** file in multiple calls to Configure, in which case **scons** will concatenate all results in the specified file. Note that SCons uses its normal dependency checking to decide if it's necessary to rebuild the specified *config_h* file. This means that the file is not necessarily re-built each time scons is run, but is only rebuilt if its contents will have changed and some target that depends on the *config_h* file is being built.

The optional **clean** and **help** arguments can be used to suppress execution of the configuration tests when the **-c/--clean** or **-H/-h/--help** options are used, respectively. The default behavior is always to execute configure context tests, since the results of the tests may affect the list of targets to be cleaned or the help text. If the configure tests do not affect these, then you may add the **clean=False** or **help=False** arguments (or both) to avoid unnecessary test execution.

A created **Configure** instance has the following associated methods:

Configure.Finish(*self*)

This method should be called after configuration is done. It returns the environment as modified by the configuration checks performed. After this method is called, no further checks can be performed with this configuration context. However, you can create a new Configure context to perform additional checks. Only one context should be active at a time.

The following Checks are predefined. (This list will likely grow larger as time goes by and developers contribute new useful tests.)

Configure.CheckHeader(*self*, *header*, [*include_quotes*, *language*])

Checks if *header* is usable in the specified language. *header* may be a list, in which case the last item in the list is the header file to be checked, and the previous list items are header files whose **#include** lines should precede the header line being checked for. The optional argument *include_quotes* must be a two character string, where the first character denotes the opening quote and the second character denotes the closing quote. By default, both characters are " (double quote). The optional argument *language* should be either **C** or **C++** and selects the compiler to be used for the check. Returns 1 on success and 0 on failure.

Configure.CheckCHeader(*self*, *header*, [*include_quotes*])

This is a wrapper around **Configure.CheckHeader** which checks if *header* is usable in the C language. *header* may be a list, in which case the last item in the list is the header file to be checked, and the previous list items are header files whose **#include** lines should precede the header line being checked for. The optional argument *include_quotes* must be a two character string, where the first character denotes the opening quote and the second character denotes the closing quote (both default to "). Returns 1 on success and 0 on failure.

Configure.CheckCXXHeader(*self*, *header*, [*include_quotes*])

This is a wrapper around **Configure.CheckHeader** which checks if *header* is usable in the C++ language. *header* may be a list, in which case the last item in the list is the header file to be checked, and the previous list items are header files whose **#include** lines should precede the header line being checked for. The optional argument *include_quotes* must be a two character string, where the first character denotes the opening quote and the second character denotes the closing quote (both default to "). Returns 1 on success and 0 on failure.

Configure.CheckFunc(*self*, *function_name*, [*header*, *language*])

Checks if the specified C or C++ function is available. *function_name* is the name of the function to check for. The optional *header* argument is a string that will be placed at the top of the test file that will be compiled to check if the function exists; the default is:

```
#ifdef __cplusplus
extern "C"
#endif
char function_name();
```

The optional *language* argument should be **C** or **C++** and selects the compiler to be used for the check; the default is "C".

Configure.CheckLib(*self*, [*library*, *symbol*, *header*, *language*, *autoadd=1*])

Checks if *library* provides *symbol*. If the value of *autoadd* is 1 and the library provides the specified *symbol*, appends the library to the LIBS construction environment variable. *library* may also be None (the default), in which case *symbol* is checked with the current LIBS variable, or a list of library names, in which case each library in the list will be checked for *symbol*. If *symbol* is not set or is **None**, then **Configure.CheckLib**() just checks if you can link against the specified *library*. The optional *language* argument should be **C** or **C++** and selects the compiler to be used for the check; the default is "C". The default value for *autoadd* is 1. This method returns 1 on success and 0 on error.

Configure.CheckLibWithHeader(*self*, *library*, *header*, *language*, [*call*, *autoadd*])

In contrast to the Configure.CheckLib call, this call provides a more sophisticated way to check against libraries. Again, *library* specifies the library or a list of libraries to check. *header* specifies a header to check for. *header* may be a list, in which case the last item in the list is the header file to be checked, and the previous list items are header files whose **#include** lines should precede the header line being checked for. *language* may be one of 'C','c','CXX','cxx','C++' and 'c++'. *call* can be any valid expression (with a trailing ';'). If *call* is not set, the default simply checks that you can link against the specified *library*. *autoadd* specifies whether to add the library to the environment (only if the check succeeds). This method returns 1 on success and 0 on error.

Configure.CheckType(*self*, *type_name*, [*includes*, *language*])

Checks for the existence of a type defined by **typedef**. *type_name* specifies the typedef name to check for. *includes* is a string containing one or more **#include** lines that will be inserted into the program that will be run to test for the existence of the type. The optional *language* argument should be **C** or **C++** and selects the compiler to be used for the check; the default is "C".

Example of a typical Configure usage:

```
env = Environment()
conf = Configure( env )
if not conf.CheckCHeader( 'math.h' ):
    print 'We really need math.h!'
    Exit(1)
if conf.CheckLibWithHeader( 'qt', 'qapp.h', 'c++', 'QApplication qapp(0,0);' ):
    # do stuff for qt - usage, e.g.
    conf.env.Append( CPPFLAGS = '-DWITH_QT' )
env = conf.Finish()
```

Configure.CheckTypeSize(*self*, *type_name*, [*header*, *language*, *expect*])

Checks for the size of a type defined by **typedef**. *type_name* specifies the typedef name to check for. The optional *header* argument is a string that will be placed at the top of the test file that will be compiled to check if the function exists; the default is empty. The optional *language* argument should be **C** or **C++** and selects the compiler to be used for the check; the default is "C". The optional *expect* argument should be an integer. If this argument is used, the function will only check whether the type given in type_name has the expected size (in bytes). For example, **CheckTypeSize('short', expect = 2)** will return success only if short is two bytes.

Configure.CheckDeclaration(*self*, *symbol*, [*includes*, *language*])

Checks if the specified *symbol* is declared. *includes* is a string containing one or more **#include** lines that will be inserted into the program that will be run to test for the existence of the type. The optional *language* argument should be **C** or **C++** and selects the compiler to be used for the check; the default is "C".

Configure.Define(self, symbol, *[*value, comment*])*

This function does not check for anything, but defines a preprocessor symbol that will be added to the configuration header file. It is the equivalent of AC_DEFINE, and defines the symbol *name* with the optional **value** and the optional comment **comment**.

Examples:

```
env = Environment()
conf = Configure( env )

# Puts the following line in the config header file:
#    #define A_SYMBOL
conf.Define('A_SYMBOL')

# Puts the following line in the config header file:
#    #define A_SYMBOL 1
conf.Define('A_SYMBOL', 1)
```

Be careful about quoting string values, though:

```
env = Environment()
conf = Configure( env )

# Puts the following line in the config header file:
#    #define A_SYMBOL YA
conf.Define('A_SYMBOL', "YA")

# Puts the following line in the config header file:
#    #define A_SYMBOL "YA"
conf.Define('A_SYMBOL', '"YA"')
```

For comment:

```
env = Environment()
conf = Configure( env )

# Puts the following lines in the config header file:
#    /* Set to 1 if you have a symbol */
#    #define A_SYMBOL 1
conf.Define('A_SYMBOL', 1, 'Set to 1 if you have a symbol')
```

You can define your own custom checks. in addition to the predefined checks. These are passed in a dictionary to the Configure function. This dictionary maps the names of the checks to user defined Python callables (either Python functions or class instances implementing the *__call__* method). The first argument of the call is always a *CheckContext* instance followed by the arguments, which must be supplied by the user of the check. These CheckContext instances define the following methods:

CheckContext.Message(*self*, *text*)

Usually called before the check is started. *text* will be displayed to the user, e.g. 'Checking for library X...'

CheckContext.Result(*self,*, *res*)

Usually called after the check is done. *res* can be either an integer or a string. In the former case, 'ok' (res != 0) or 'failed' (res == 0) is displayed to the user, in the latter case the given string is displayed.

CheckContext.TryCompile(*self*, *text*, *extension*)

Checks if a file with the specified *extension* (e.g. '.c') containing *text* can be compiled using the environment's **Object** builder. Returns 1 on success and 0 on failure.

CheckContext.TryLink(*self*, *text*, *extension*)

Checks, if a file with the specified *extension* (e.g. '.c') containing *text* can be compiled using the environment's **Program** builder. Returns 1 on success and 0 on failure.

CheckContext.TryRun(*self*, *text*, *extension*)

Checks, if a file with the specified *extension* (e.g. '.c') containing *text* can be compiled using the environment's **Program** builder. On success, the program is run. If the program executes successfully (that is, its return status is 0), a tuple *(1, outputStr)* is returned, where *outputStr* is the standard output of the program. If the program fails execution (its return status is non-zero), then (0, '') is returned.

CheckContext.TryAction(*self*, *action*, [*text*, *extension*])
: Checks if the specified *action* with an optional source file (contents *text* , extension *extension* = '') can be executed. *action* may be anything which can be converted to a **scons** Action. On success, *(1, outputStr)* is returned, where *outputStr* is the content of the target file. On failure *(0, '')* is returned.

CheckContext.TryBuild(*self*, *builder*, [*text*, *extension*])
: Low level implementation for testing specific builds; the methods above are based on this method. Given the Builder instance *builder* and the optional *text* of a source file with optional *extension*, this method returns 1 on success and 0 on failure. In addition, *self.lastTarget* is set to the build target node, if the build was successful.

Example for implementing and using custom tests:

```
def CheckQt(context, qtdir):
    context.Message( 'Checking for qt ...' )
    lastLIBS = context.env['LIBS']
    lastLIBPATH = context.env['LIBPATH']
    lastCPPPATH= context.env['CPPPATH']
    context.env.Append(LIBS = 'qt', LIBPATH = qtdir + '/lib', CPPPATH = qtdir + '/include' )
    ret = context.TryLink("""
#include <qapp.h>
int main(int argc, char **argv) {
  QApplication qapp(argc, argv);
  return 0;
}
""")
    if not ret:
        context.env.Replace(LIBS = lastLIBS, LIBPATH=lastLIBPATH, CPPPATH=lastCPPPATH)
    context.Result( ret )
    return ret

env = Environment()
conf = Configure( env, custom_tests = { 'CheckQt' : CheckQt } )
if not conf.CheckQt('/usr/lib/qt'):
    print 'We really need qt!'
    Exit(1)
env = conf.Finish()
```

Command-Line Construction Variables

Often when building software, some variables must be specified at build time. For example, libraries needed for the build may be in non-standard locations, or site-specific compiler options may need to be passed to the compiler. **scons** provides a **Variables** object to support overriding construction variables on the command line:

```
$ scons VARIABLE=foo
```

The variable values can also be specified in a text-based SConscript file. To create a Variables object, call the Variables() function:

Variables([*files*], [*args*])

This creates a Variables object that will read construction variables from the file or list of filenames specified in *files*. If no files are specified, or the *files* argument is **None**, then no files will be read. The optional argument *args* is a dictionary of values that will override anything read from the specified files; it is primarily intended to be passed the **ARGUMENTS** dictionary that holds variables specified on the command line. Example:

```
vars = Variables('custom.py')
vars = Variables('overrides.py', ARGUMENTS)
vars = Variables(None, {FOO:'expansion', BAR:7})
```

Variables objects have the following methods:

Add(*key*, [*help*, *default*, *validator*, *converter*])

This adds a customizable construction variable to the Variables object. *key* is the name of the variable. *help* is the help text for the variable. *default* is the default value of the variable; if the default value is **None** and there is no explicit value specified, the construction variable will *not* be added to the construction environment. *validator* is called to validate the value of the variable, and should take three arguments: key, value, and environment. The recommended way to handle an invalid value is to raise an exception (see example below). *converter* is called to convert the value before putting it in the environment, and should take either a value, or the value and environment, as parameters. The *converter* must return a value, which will be converted into a string before being validated by the *validator* (if any) and then added to the environment.

Examples:

```
vars.Add('CC', 'The C compiler')

def validate_color(key, val, env):
    if not val in ['red', 'blue', 'yellow']:
        raise "Invalid color value '%s'" % val
vars.Add('COLOR', validator=valid_color)
```

AddVariables(*list*)

A wrapper script that adds multiple customizable construction variables to a Variables object. *list* is a list of tuple or list objects that contain the arguments for an individual call to the **Add** method.

```
opt.AddVariables(
      ('debug', '', 0),
      ('CC', 'The C compiler'),
      ('VALIDATE', 'An option for testing validation',
       'notset', validator, None),
    )
```

Update(*env*, [*args*])

This updates a construction environment *env* with the customized construction variables. Any specified variables that are *not* configured for the Variables object will be saved and may be retrieved with the **UnknownVariables**() method, below.

Normally this method is not called directly, but is called indirectly by passing the Variables object to the Environment() function:

```
env = Environment(variables=vars)
```

The text file(s) that were specified when the Variables object was created are executed as Python scripts, and the values of (global) Python variables set in the file are added to the construction environment.

Example:

```
CC = 'my_cc'
```

UnknownVariables()

Returns a dictionary containing any variables that were specified either in the files or the dictionary with which the Variables object was initialized, but for which the Variables object was not configured.

```
env = Environment(variables=vars)
for key, value in vars.UnknownVariables():
    print "unknown variable:  %s=%s" % (key, value)
```

Save(*filename*, *env*)

This saves the currently set variables into a script file named *filename* that can be used on the next invocation to automatically load the current settings. This method combined with the Variables method can be used to support caching of variables between runs.

```
env = Environment()
vars = Variables(['variables.cache', 'custom.py'])
vars.Add(...)
vars.Update(env)
vars.Save('variables.cache', env)
```

GenerateHelpText(*env*, [*sort*])

This generates help text documenting the customizable construction variables suitable to passing in to the Help() function. *env* is the construction environment that will be used to get the actual values of customizable variables. Calling with an optional *sort* function will cause the output to be sorted by the specified argument. The specific *sort* function should take two arguments and return -1, 0 or 1 (like the standard Python *cmp* function).

```
Help(vars.GenerateHelpText(env))
Help(vars.GenerateHelpText(env, sort=cmp))
```

FormatVariableHelpText(*env*, *opt*, *help*, *default*, *actual*)

This method returns a formatted string containing the printable help text for one option. It is normally not called directly, but is called by the *GenerateHelpText*() method to create the returned help text. It may be overridden with your own function that takes the arguments specified above and returns a string of help text formatted to your liking. Note that the *GenerateHelpText*() will not put any blank lines or extra characters in between the entries, so you must add those characters to the returned string if you want the entries separated.

```
def my_format(env, opt, help, default, actual):
    fmt = "119s: default=%s actual=%s (%s)0
    return fmt % (opt, default. actual, help)
vars.FormatVariableHelpText = my_format
```

To make it more convenient to work with customizable Variables, **scons** provides a number of functions that make it easy to set up various types of Variables:

BoolVariable(*key*, *help*, *default*)

Return a tuple of arguments to set up a Boolean option. The option will use the specified name *key*, have a default value of *default*, and display the specified *help* text. The option will interpret the values **y**, **yes**, **t**, **true**, **1**, **on** and **all** as true, and the values **n**, **no**, **f**, **false**, **0**, **off** and **none** as false.

EnumVariable(*key*, *help*, *default*, *allowed_values*, [*map*, *ignorecase*])

Return a tuple of arguments to set up an option whose value may be one of a specified list of legal enumerated values. The option will use the specified name *key*, have a default value of *default*, and display the specified *help* text. The option will only support those values in the *allowed_values* list. The optional *map* argument is a dictionary that can be used to convert input values into specific legal values in the *allowed_values* list. If the value of *ignore_case* is **0** (the default), then the values are case-sensitive. If the value of *ignore_case* is **1**, then values will be matched case-insensitive. If the value of *ignore_case* is **1**, then values will be matched case-insensitive, and all input values will be converted to lower case.

ListVariable(*key*, *help*, *default*, *names*, [,map*])*

Return a tuple of arguments to set up an option whose value may be one or more of a specified list of legal enumerated values. The option will use the specified name *key*, have a default value of *default*, and display the specified *help* text. The option will only support the values **all**, **none**, or the values in the *names* list. More than one value may be specified, with all values separated by commas. The default may be a string of comma-separated default values, or a list of the default values. The optional *map* argument is a dictionary that can be used to convert input values into specific legal values in the *names* list.

PackageVariable(*key*, *help*, *default*)

Return a tuple of arguments to set up an option whose value is a path name of a package that may be enabled, disabled or given an explicit path name. The option will use the specified name *key*, have a default value of *default*, and display the specified *help* text. The option will support the values **yes**, **true**, **on**, **enable** or **search**, in which case the specified *default* will be used, or the option may be set to an arbitrary string (typically the path name to a package that is being enabled). The option will also support the values **no**, **false**, **off** or **disable** to disable use of the specified option.

PathVariable(*key*, *help*, *default*, [*validator*])

Return a tuple of arguments to set up an option whose value is expected to be a path name. The option will use the specified name *key*, have a default value of *default*, and display the specified *help* text. An additional *validator* may be specified that will be called to verify that the specified path is acceptable. SCons supplies the following ready-made validators: **PathVariable.PathExists** (the default), which verifies that the specified path exists; **PathVariable.PathIsFile**, which verifies that the specified path is an existing file; **PathVariable.PathIsDir**, which verifies that the specified path is an existing directory; **PathVariable.PathIsDirCreate**, which verifies that the specified path is a directory and will create the specified directory if the path does not exist; and **PathVariable.PathAccept**, which simply accepts the specific path name argument without validation, and which is suitable if you want your users to be able to specify a directory path that will be created as part of the build process, for example. You may supply your own *validator* function, which must take three arguments (*key*, the name of the variable to be set; *val*, the specified value being checked; and *env*, the construction environment) and should raise an exception if the specified value is not acceptable.

These functions make it convenient to create a number of variables with consistent behavior in a single call to the **AddVariables** method:

```
vars.AddVariables(
  BoolVariable('warnings', 'compilation with -Wall and similiar', 1),
  EnumVariable('debug', 'debug output and symbols', 'no'
        allowed_values=('yes', 'no', 'full'),
        map={}, ignorecase=0),  # case sensitive
  ListVariable('shared',
        'libraries to build as shared libraries',
        'all',
        names = list_of_libs),
  PackageVariable('x11',
         'use X11 installed here (yes = search some places)',
         'yes'),
  PathVariable('qtdir', 'where the root of Qt is installed', qtdir),
  PathVariable('foopath', 'where the foo library is installed', foopath,
        PathVariable.PathIsDir),

)
```

File and Directory Nodes

The *File*() and *Dir*() functions return *File* and *Dir* Nodes, respectively. python objects, respectively. Those objects have several user-visible attributes and methods that are often useful:

path
: The build path of the given file or directory. This path is relative to the top-level directory (where the **SConstruct** file is found). The build path is the same as the source path if *variant_dir* is not being used.

abspath
: The absolute build path of the given file or directory.

srcnode()
: The *srcnode*() method returns another *File* or *Dir* object representing the *source* path of the given *File* or *Dir*. The

```
# Get the current build dir's path, relative to top.
Dir('.').path
# Current dir's absolute path
Dir('.').abspath
# Next line is always '.', because it is the top dir's path relative to itself.
Dir('#.').path
File('foo.c').srcnode().path   # source path of the given source file.

# Builders also return File objects:
foo = env.Program('foo.c')
print "foo will be built in %s"%foo.path
```

A *Dir* Node or *File* Node can also be used to create file and subdirectory Nodes relative to the generating Node. A *Dir* Node will place the new Nodes within the directory it represents. A *File* node will place the new Nodes within its parent directory (that is, "beside" the file in question). If *d* is a *Dir* (directory) Node and *f* is a *File* (file) Node, then these methods are available:

d.Dir(*name*)
: Returns a directory Node for a subdirectory of *d* named *name*.

d.File(*name*)
: Returns a file Node for a file within *d* named *name*.

d.Entry(*name*)
: Returns an unresolved Node within *d* named *name*.

f.Dir(*name*)
: Returns a directory named *name* within the parent directory of *f*.

f.File(*name*)
: Returns a file named *name* within the parent directory of *f*.

f.Entry(*name*)
: Returns an unresolved Node named *name* within the parent directory of *f*.

For example:

```
# Get a Node for a file within a directory
incl = Dir('include')
f = incl.File('header.h')

# Get a Node for a subdirectory within a directory
dist = Dir('project-3.2.1)
src = dist.Dir('src')

# Get a Node for a file in the same directory
cfile = File('sample.c')
hfile = cfile.File('sample.h')

# Combined example
docs = Dir('docs')
html = docs.Dir('html')
index = html.File('index.html')
css = index.File('app.css')
```

EXTENDING SCONS

Builder Objects

scons can be extended to build different types of targets by adding new Builder objects to a construction environment. *In general*, you should only need to add a new Builder object when you want to build a new type of file or other external target. If you just want to invoke a different compiler or other tool to build a Program, Object, Library, or any other type of output file for which **scons** already has an existing Builder, it is generally much easier to use those existing Builders in a construction environment that sets the appropriate construction variables (CC, LINK, etc.).

Builder objects are created using the **Builder** function. The **Builder** function accepts the following arguments:

action
: The command line string used to build the target from the source. **action** can also be: a list of strings representing the command to be executed and its arguments (suitable for enclosing white space in an argument), a dictionary mapping source file name suffixes to any combination of command line strings (if the builder should accept multiple source file extensions), a Python function; an Action object (see the next section); or a list of any of the above.

An action function takes three arguments: *source* - a list of source nodes, *target* - a list of target nodes, *env* - the construction environment.

prefix The prefix that will be prepended to the target file name. This may be specified as a:

* *string*,

* *callable object* - a function or other callable that takes two arguments (a construction environment and a list of sources) and returns a prefix,

* *dictionary* - specifies a mapping from a specific source suffix (of the first source specified) to a corresponding target prefix. Both the source suffix and target prefix specifications may use environment variable substitution, and the target prefix (the 'value' entries in the dictionary) may also be a callable object. The default target prefix may be indicated by a dictionary entry with a key value of None.

```
b = Builder("build_it < $SOURCE > $TARGET"
        prefix = "file-")

def gen_prefix(env, sources):
    return "file-" + env['PLATFORM'] + '-'
b = Builder("build_it < $SOURCE > $TARGET",
        prefix = gen_prefix)

b = Builder("build_it < $SOURCE > $TARGET",
        suffix = { None: "file-",
                "$SRC_SFX_A": gen_prefix })
```

suffix The suffix that will be appended to the target file name. This may be specified in the same manner as the prefix above. If the suffix is a string, then **scons** will append a '.' to the beginning of the suffix if it's not already there. The string returned by callable object (or obtained from the dictionary) is untouched and must append its own '.' to the beginning if one is desired.

```
b = Builder("build_it < $SOURCE > $TARGET"
        suffix = "-file")

def gen_suffix(env, sources):
    return "." + env['PLATFORM'] + "-file"
b = Builder("build_it < $SOURCE > $TARGET",
        suffix = gen_suffix)

b = Builder("build_it < $SOURCE > $TARGET",
        suffix = { None: ".sfx1",
                "$SRC_SFX_A": gen_suffix })
```

ensure_suffix
When set to any true value, causes **scons** to add the target suffix specified by the *suffix* keyword to any target strings that have a different suffix. (The default behavior is to leave untouched any target file name that looks like it already has any suffix.)

```
b1 = Builder("build_it < $SOURCE > $TARGET"
         suffix = ".out")
```

```
b2 = Builder("build_it < $SOURCE > $TARGET"
          suffix = ".out",
          ensure_suffix)
env = Environment()
env['BUILDERS']['B1'] = b1
env['BUILDERS']['B2'] = b2

# Builds "foo.txt" because ensure_suffix is not set.
env.B1('foo.txt', 'foo.in')

# Builds "bar.txt.out" because ensure_suffix is set.
env.B2('bar.txt', 'bar.in')
```

src_suffix
The expected source file name suffix. This may be a string or a list of strings.

target_scanner
A Scanner object that will be invoked to find implicit dependencies for this target file. This keyword argument should be used for Scanner objects that find implicit dependencies based only on the target file and the construction environment, *not* for implicit (See the section "Scanner Objects," below, for information about creating Scanner objects.)

source_scanner
A Scanner object that will be invoked to find implicit dependences in any source files used to build this target file. This is where you would specify a scanner to find things like **#include** lines in source files. The pre-built **DirScanner** Scanner object may be used to indicate that this Builder should scan directory trees for on-disk changes to files that **scons** does not know about from other Builder or function calls. (See the section "Scanner Objects," below, for information about creating your own Scanner objects.)

target_factory
A factory function that the Builder will use to turn any targets specified as strings into SCons Nodes. By default, SCons assumes that all targets are files. Other useful target_factory values include **Dir**, for when a Builder creates a directory target, and **Entry**, for when a Builder can create either a file or directory target.

Example:

```
MakeDirectoryBuilder = Builder(action=my_mkdir, target_factory=Dir)
env = Environment()
env.Append(BUILDERS = {'MakeDirectory':MakeDirectoryBuilder})
env.MakeDirectory('new_directory', [])
```

Note that the call to the MakeDirectory Builder needs to specify an empty source list to make the string represent the builder's target; without that, it would assume the argument is the source, and would try to deduce the target name from it, which in the absence of an automatically-added prefix or suffix would lead to a matching target and source name and a circular dependency.

source_factory
A factory function that the Builder will use to turn any sources specified as strings into SCons Nodes. By default, SCons assumes that all source are files. Other useful source_factory values include **Dir**, for when a Builder uses a directory as a source, and **Entry**, for when a Builder can

use files or directories (or both) as sources.

Example:

```
CollectBuilder = Builder(action=my_mkdir, source_factory=Entry)
env = Environment()
env.Append(BUILDERS = {'Collect':CollectBuilder})
env.Collect('archive', ['directory_name', 'file_name'])
```

emitter A function or list of functions to manipulate the target and source lists before dependencies are established and the target(s) are actually built. **emitter** can also be a string containing a construction variable to expand to an emitter function or list of functions, or a dictionary mapping source file suffixes to emitter functions. (Only the suffix of the first source file is used to select the actual emitter function from an emitter dictionary.)

An emitter function takes three arguments: *source* - a list of source nodes, *target* - a list of target nodes, *env* - the construction environment. An emitter must return a tuple containing two lists, the list of targets to be built by this builder, and the list of sources for this builder.

Example:

```
def e(target, source, env):
    return (target + ['foo.foo'], source + ['foo.src'])

# Simple association of an emitter function with a Builder.
b = Builder("my_build < $TARGET > $SOURCE",
            emitter = e)

def e2(target, source, env):
    return (target + ['bar.foo'], source + ['bar.src'])

# Simple association of a list of emitter functions with a Builder.
b = Builder("my_build < $TARGET > $SOURCE",
            emitter = [e, e2])

# Calling an emitter function through a construction variable.
env = Environment(MY_EMITTER = e)
b = Builder("my_build < $TARGET > $SOURCE",
            emitter = '$MY_EMITTER')

# Calling a list of emitter functions through a construction variable.
env = Environment(EMITTER_LIST = [e, e2])
b = Builder("my_build < $TARGET > $SOURCE",
            emitter = '$EMITTER_LIST')

# Associating multiple emitters with different file
# suffixes using a dictionary.
def e_suf1(target, source, env):
    return (target + ['another_target_file'], source)
def e_suf2(target, source, env):
    return (target, source + ['another_source_file'])
b = Builder("my_build < $TARGET > $SOURCE",
            emitter = {'.suf1' : e_suf1,
                       '.suf2' : e_suf2})
```

multi
: Specifies whether this builder is allowed to be called multiple times for the same target file(s). The default is 0, which means the builder can not be called multiple times for the same target file(s). Calling a builder multiple times for the same target simply adds additional source files to the target; it is not allowed to change the environment associated with the target, specify addition environment overrides, or associate a different builder with the target.

env
: A construction environment that can be used to fetch source code using this Builder. (Note that this environment is *not* used for normal builds of normal target files, which use the environment that was used to call the Builder for the target file.)

generator
: A function that returns a list of actions that will be executed to build the target(s) from the source(s). The returned action(s) may be an Action object, or anything that can be converted into an Action object (see the next section).

The generator function takes four arguments: *source* - a list of source nodes, *target* - a list of target nodes, *env* - the construction environment, *for_signature* - a Boolean value that specifies whether the generator is being called for generating a build signature (as opposed to actually executing the command). Example:

```
def g(source, target, env, for_signature):
    return [["gcc", "-c", "-o"] + target + source]

b = Builder(generator=g)
```

The *generator* and *action* arguments must not both be used for the same Builder.

src_builder
: Specifies a builder to use when a source file name suffix does not match any of the suffixes of the builder. Using this argument produces a multi-stage builder.

single_source
: Specifies that this builder expects exactly one source file per call. Giving more than one source files without target files results in implicitely calling the builder multiple times (once for each source given). Giving multiple source files together with target files results in a UserError exception.

The *generator* and *action* arguments must not both be used for the same Builder.

source_ext_match
: When the specified *action* argument is a dictionary, the default behavior when a builder is passed multiple source files is to make sure that the extensions of all the source files match. If it is legal for this builder to be called with a list of source files with different extensions, this check can be suppressed by setting **source_ext_match** to **None** or some other non-true value. When **source_ext_match** is disable, **scons** will use the suffix of the first specified source file to select the appropriate action from the *action* dictionary.

In the following example, the setting of **source_ext_match** prevents **scons** from exiting with an error due to the mismatched suffixes of **foo.in** and **foo.extra**.

```
b = Builder(action={'.in' : 'build $SOURCES > $TARGET'},
        source_ext_match = None)
```

```
env = Environment(BUILDERS = {'MyBuild':b})
env.MyBuild('foo.out', ['foo.in', 'foo.extra'])
```

env A construction environment that can be used to fetch source code using this Builder. (Note that this environment is *not* used for normal builds of normal target files, which use the environment that was used to call the Builder for the target file.)

```
b = Builder(action="build < $SOURCE > $TARGET")
env = Environment(BUILDERS = {'MyBuild' : b})
env.MyBuild('foo.out', 'foo.in', my_arg = 'xyzzy')
```

chdir A directory from which scons will execute the action(s) specified for this Builder. If the **chdir** argument is a string or a directory Node, scons will change to the specified directory. If the **chdir** is not a string or Node and is non-zero, then scons will change to the target file's directory.

Note that scons will *not* automatically modify its expansion of construction variables like **$TARGET** and **$SOURCE** when using the chdir keyword argument--that is, the expanded file names will still be relative to the top-level SConstruct directory, and consequently incorrect relative to the chdir directory. Builders created using chdir keyword argument, will need to use construction variable expansions like **${TARGET.file}** and **${SOURCE.file}** to use just the filename portion of the targets and source.

```
b = Builder(action="build < ${SOURCE.file} > ${TARGET.file}",
        chdir=1)
env = Environment(BUILDERS = {'MyBuild' : b})
env.MyBuild('sub/dir/foo.out', 'sub/dir/foo.in')
```

WARNING: Python only keeps one current directory location for all of the threads. This means that use of the **chdir** argument will *not* work with the SCons **-j** option, because individual worker threads spawned by SCons interfere with each other when they start changing directory.

Any additional keyword arguments supplied when a Builder object is created (that is, when the Builder() function is called) will be set in the executing construction environment when the Builder object is called. The canonical example here would be to set a construction variable to the repository of a source code system.

Any additional keyword arguments supplied when a Builder *object* is called will only be associated with the target created by that particular Builder call (and any other files built as a result of the call).

These extra keyword arguments are passed to the following functions: command generator functions, function Actions, and emitter functions.

Action Objects

The **Builder()** function will turn its **action** keyword argument into an appropriate internal Action object. You can also explicity create Action objects using the **Action**() global function, which can then be passed to the **Builder**() function. This can be used to configure an Action object more flexibly, or it may simply be more efficient than letting each separate Builder object create a separate Action when multiple Builder objects need to do the same thing.

The **Action**() global function returns an appropriate object for the action represented by the type of the first argument:

Action If the first argument is already an Action object, the object is simply returned.

String If the first argument is a string, a command-line Action is returned. Note that the command line string may be preceded by an @ (at-sign) to suppress printing of the specified command line, or by a **–** (hyphen) to ignore the exit status from the specified command. Examples:

```
Action('$CC -c -o $TARGET $SOURCES')

# Doesn't print the line being executed.
Action('@build $TARGET $SOURCES')

# Ignores
Action('-build $TARGET $SOURCES')
```

List If the first argument is a list, then a list of Action objects is returned. An Action object is created as necessary for each element in the list. If an element *within* the list is itself a list, the internal list is the command and arguments to be executed via the command line. This allows white space to be enclosed in an argument by defining a command in a list within a list:

```
Action([['cc', '-c', '-DWHITE SPACE', '-o', '$TARGET', '$SOURCES']])
```

Function

If the first argument is a Python function, a function Action is returned. The Python function takes three keyword arguments, **target** (a Node object representing the target file), **source** (a Node object representing the source file) and **env** (the construction environment used for building the target file). The **target** and **source** arguments may be lists of Node objects if there is more than one target file or source file. The actual target and source file name(s) may be retrieved from their Node objects via the built-in Python str() function:

```
target_file_name = str(target)
source_file_names = map(lambda x: str(x), source)
```

The function should return **0** or **None** to indicate a successful build of the target file(s). The function may raise an exception or return a non-zero exit status to indicate an unsuccessful build.

```
def build_it(target = None, source = None, env = None):
    # build the target from the source
    return 0

a = Action(build_it)
```

If the action argument is not one of the above, None is returned.

The second, optional argument is used to define the output which is printed when the Action is actually performed. In the absence of this parameter, or if it's an empty string, a default output depending on the type of the action is used. For example, a command-line action will print the executed command. The argument is either a python function or a string.

In the first case, it's a function that returns a string to be printed to describe the action being executed. Like a function to build a file, this function takes three arguments: **target** (a Node object representing the target file), **source** (a Node object representing the source file) and **env** (a construction environment). The **target** and **source** arguments may be lists of Node objects if there is more than one target file or source file.

In the second case, you provide the string itself. The string typically contains variables, notably $TARGET(S) and $SOURCE(S), or consists of just a single variable, which is optionally defined somewhere else. SCons itself heavily uses the latter variant.

Examples:

```
def build_it(target, source, env):
    # build the target from the source
    return 0

def string_it(target, source, env):
    return "building '%s' from '%s'" % (target[0], source[0])

# Use a positional argument.
f = Action(build_it, string_it)
s = Action(build_it, "building '$TARGET' from '$SOURCE'")

# Alternatively, use a keyword argument.
f = Action(build_it, strfunction=string_it)
s = Action(build_it, cmdstr="building '$TARGET' from '$SOURCE'")

# You can provide a configurable variable.
l = Action(build_it, '$STRINGIT')
```

The third, also optional argument is a list of construction variables whose values will be included in the signature of the Action when deciding whether a target should be rebuilt because the action changed. This is necessary whenever you want a target to be rebuilt when a specific construction variable changes, because the underlying Python code for a function will not change when the value of the construction variable does.

```
def build_it(target, source, env):
    # build the target from the 'XXX' construction variable
    open(target[0], 'w').write(env['XXX'])
    return 0

# Use positional arguments.
a = Action(build_it, '$STRINGIT', ['XXX'])

# Alternatively, use a keyword argument.
a = Action(build_it, varlist=['XXX'])
```

The **Action**() global function also takes a **chdir** keyword argument which specifies that scons will execute the action after changing to the specified directory. If the chdir argument is a string or a directory Node, scons will change to the specified directory. If the chdir argument is not a string or Node and is non-zero, then scons will change to the target file's directory.

Note that scons will *not* automatically modify its expansion of construction variables like **$TARGET** and **$SOURCE** when using the chdir keyword argument--that is, the expanded file names will still be relative to the top-level SConstruct directory, and consequently incorrect relative to the chdir directory. Builders created using chdir keyword argument, will need to use construction variable expansions like **${TARGET.file}** and **${SOURCE.file}** to use just the filename portion of the targets and source.

```
a = Action("build < ${SOURCE.file} > ${TARGET.file}",
```

```
        chdir=1)
```

The **Action**() global function also takes an **exitstatfunc** keyword argument which specifies a function that is passed the exit status (or return value) from the specified action and can return an arbitrary or modified value. This can be used, for example, to specify that an Action object's return value should be ignored and SCons should, therefore, consider that the action always suceeds:

```
def always_succeed(s):
    # Always return 0, which indicates success.
    return 0
a = Action("build < ${SOURCE.file} > ${TARGET.file}",
        exitstatfunc=always_succeed)
```

Miscellaneous Action Functions

scons supplies a number of functions that arrange for various common file and directory manipulations to be performed. These are similar in concept to "tasks" in the Ant build tool, although the implementation is slightly different. These functions do not actually perform the specified action at the time the function is called, but instead return an Action object that can be executed at the appropriate time. (In Object-Oriented terminology, these are actually Action *Factory* functions that return Action objects.)

In practice, there are two natural ways that these Action Functions are intended to be used.

First, if you need to perform the action at the time the SConscript file is being read, you can use the **Execute** global function to do so:

```
Execute(Touch('file'))
```

Second, you can use these functions to supply Actions in a list for use by the **Command** method. This can allow you to perform more complicated sequences of file manipulation without relying on platform-specific external commands: that

```
env = Environment(TMPBUILD = '/tmp/builddir')
env.Command('foo.out', 'foo.in',
            [Mkdir('$TMPBUILD'),
             Copy('$TMPBUILD', '${SOURCE.dir}'),
             "cd $TMPBUILD && make",
             Delete('$TMPBUILD')])
```

Chmod(*dest*, *mode*)

Returns an Action object that changes the permissions on the specified *dest* file or directory to the specified *mode*. Examples:

```
Execute(Chmod('file', 0755))

env.Command('foo.out', 'foo.in',
            [Copy('$TARGET', '$SOURCE'),
             Chmod('$TARGET', 0755)])
```

Copy(*dest*, *src*)

Returns an Action object that will copy the *src* source file or directory to the *dest* destination file or directory. Examples:

```
Execute(Copy('foo.output', 'foo.input'))

env.Command('bar.out', 'bar.in',
```

```
Copy('$TARGET', '$SOURCE'))
```

Delete(*entry*, [*must_exist*])

Returns an Action that deletes the specified *entry*, which may be a file or a directory tree. If a directory is specified, the entire directory tree will be removed. If the *must_exist* flag is set, then a Python error will be thrown if the specified entry does not exist; the default is **must_exist=0**, that is, the Action will silently do nothing if the entry does not exist. Examples:

```
Execute(Delete('/tmp/buildroot'))

env.Command('foo.out', 'foo.in',
        [Delete('${TARGET.dir}'),
         MyBuildAction])

Execute(Delete('file_that_must_exist', must_exist=1))
```

Mkdir(*dir*)

Returns an Action that creates the specified directory *dir* . Examples:

```
Execute(Mkdir('/tmp/outputdir'))

env.Command('foo.out', 'foo.in',
        [Mkdir('/tmp/builddir',
         Copy('$SOURCE', '/tmp/builddir')
         "cd /tmp/builddir && ])
```

Move(*dest*, *src*)

Returns an Action that moves the specified *src* file or directory to the specified *dest* file or directory. Examples:

```
Execute(Move('file.destination', 'file.source'))

env.Command('output_file', 'input_file',
        [MyBuildAction,
         Move('$TARGET', 'file_created_by_MyBuildAction')])
```

Touch(*file*)

Returns an Action that updates the modification time on the specified *file*. Examples:

```
Execute(Touch('file_to_be_touched'))

env.Command('marker', 'input_file',
        [MyBuildAction,
         Touch('$TARGET')])
```

Variable Substitution

Before executing a command, **scons** performs construction variable interpolation on the strings that make up the command line of builders. Variables are introduced by a **$** prefix. Besides construction variables, scons provides the following variables for each command execution:

TARGET
The file name of the target being built, or the file name of the first target if multiple targets are being built.

TARGETS
The file names of all targets being built.

SOURCE
The file name of the source of the build command, or the file name of the first source if multiple sources are being built.

SOURCES
The file names of the sources of the build command.

(Note that the above variables are reserved and may not be set in a construction environment.)

For example, given the construction variable CC='cc', targets=['foo'], and sources=['foo.c', 'bar.c']:

```
action='$CC -c -o $TARGET $SOURCES'
```

would produce the command line:

```
cc -c -o foo foo.c bar.c
```

Variable names may be surrounded by curly braces ({}) to separate the name from the trailing characters. Within the curly braces, a variable name may have a Python slice subscript appended to select one or more items from a list. In the previous example, the string:

```
${SOURCES[1]}
```

would produce:

```
bar.c
```

Additionally, a variable name may have the following special modifiers appended within the enclosing curly braces to modify the interpolated string:

base The base path of the file name, including the directory path but excluding any suffix.

dir The name of the directory in which the file exists.

file The file name, minus any directory portion.

filebase Just the basename of the file, minus any suffix and minus the directory.

suffix Just the file suffix.

abspath The absolute path name of the file.

posix
: The POSIX form of the path, with directories separated by **/** (forward slashes) not backslashes. This is sometimes necessary on Windows systems when a path references a file on other (POSIX) systems.

srcpath
: The directory and file name to the source file linked to this file through **VariantDir**(). If this file isn't linked, it just returns the directory and filename unchanged.

srcdir
: The directory containing the source file linked to this file through **VariantDir**(). If this file isn't linked, it just returns the directory part of the filename.

rsrcpath
: The directory and file name to the source file linked to this file through **VariantDir**(). If the file does not exist locally but exists in a Repository, the path in the Repository is returned. If this file isn't linked, it just returns the directory and filename unchanged.

rsrcdir
: The Repository directory containing the source file linked to this file through **VariantDir**(). If this file isn't linked, it just returns the directory part of the filename.

For example, the specified target will expand as follows for the corresponding modifiers:

```
$TARGET              => sub/dir/file.x
${TARGET.base}       => sub/dir/file
${TARGET.dir}        => sub/dir
${TARGET.file}       => file.x
${TARGET.filebase}   => file
${TARGET.suffix}     => .x
${TARGET.abspath}    => /top/dir/sub/dir/file.x

SConscript('src/SConscript', variant_dir='sub/dir')
$SOURCE              => sub/dir/file.x
${SOURCE.srcpath}    => src/file.x
${SOURCE.srcdir}     => src

Repository('/usr/repository')
$SOURCE              => sub/dir/file.x
${SOURCE.rsrcpath}   => /usr/repository/src/file.x
${SOURCE.rsrcdir}    => /usr/repository/src
```

Note that curly braces braces may also be used to enclose arbitrary Python code to be evaluated. (In fact, this is how the above modifiers are substituted, they are simply attributes of the Python objects that represent TARGET, SOURCES, etc.) See the section "Python Code Substitution," below, for more thorough examples of how this can be used.

Lastly, a variable name may be a callable Python function associated with a construction variable in the environment. The function should take four arguments: *target* - a list of target nodes, *source* - a list of source nodes, *env* - the construction environment, *for_signature* - a Boolean value that specifies whether the function is being called for generating a build signature. SCons will insert whatever the called function returns into the expanded string:

```
def foo(target, source, env, for_signature):
    return "bar"
```

```
# Will expand $BAR to "bar baz"
env=Environment(FOO=foo, BAR="$FOO baz")
```

You can use this feature to pass arguments to a Python function by creating a callable class that stores one or more arguments in an object, and then uses them when the **__call__()** method is called. Note that in this case, the entire variable expansion must be enclosed by curly braces so that the arguments will be associated with the instantiation of the class:

```
class foo:
   def __init__(self, arg):
      self.arg = arg

   def __call__(self, target, source, env, for_signature):
      return self.arg + " bar"

# Will expand $BAR to "my argument bar baz"
env=Environment(FOO=foo, BAR="${FOO('my argument')} baz")
```

The special pseudo-variables **$(** and **$)** may be used to surround parts of a command line that may change *without* causing a rebuild--that is, which are not included in the signature of target files built with this command. All text between **$(** and **$)** will be removed from the command line before it is added to file signatures, and the **$(** and **$)** will be removed before the command is executed. For example, the command line:

```
echo Last build occurred $( $TODAY $). > $TARGET
```

would execute the command:

```
echo Last build occurred $TODAY. > $TARGET
```

but the command signature added to any target files would be:

```
echo Last build occurred  . > $TARGET
```

Python Code Substitution

Any python code within **${-}** pairs gets evaluated by python 'eval', with the python globals set to the current environment's set of construction variables. So in the following case:

```
env['COND'] = 0
env.Command('foo.out', 'foo.in',
  '''echo ${COND==1 and 'FOO' or 'BAR'} > $TARGET''')
```

the command executed will be either

```
echo FOO > foo.out
```

or

```
echo BAR > foo.out
```

according to the current value of env['COND'] when the command is executed. The evaluation occurs when the target is being built, not when the SConscript is being read. So if env['COND'] is changed later in the SConscript, the final value will be used.

Here's a more interesting example. Note that all of COND, FOO, and BAR are environment variables, and their values are substituted into the final command. FOO is a list, so its elements are interpolated separated by spaces.

```
env=Environment()
env['COND'] = 0
```

```
env['FOO'] = ['foo1', 'foo2']
env['BAR'] = 'barbar'
env.Command('foo.out', 'foo.in',
    'echo ${COND==1 and FOO or BAR} > $TARGET')

# Will execute this:
#  echo foo1 foo2 > foo.out
```

SCons uses the following rules when converting construction variables into command lines:

String
When the value is a string it is interpreted as a space delimited list of command line arguments.

List
When the value is a list it is interpreted as a list of command line arguments. Each element of the list is converted to a string.

Other
Anything that is not a list or string is converted to a string and interpreted as a single command line argument.

Newline
Newline characters (\n) delimit lines. The newline parsing is done after all other parsing, so it is not possible for arguments (e.g. file names) to contain embedded newline characters. This limitation will likely go away in a future version of SCons.

Scanner Objects

You can use the **Scanner** function to define objects to scan new file types for implicit dependencies. Scanner accepts the following arguments:

function
This can be either: 1) a Python function that will process the Node (file) and return a list of strings (file names) representing the implicit dependencies found in the contents; or: 2) a dictionary that maps keys (typically the file suffix, but see below for more discussion) to other Scanners that should be called.

If the argument is actually a Python function, the function must take three or four arguments:

```
def scanner_function(node, env, path):

def scanner_function(node, env, path, arg=None):
```

The **node** argument is the internal SCons node representing the file. Use **str(node)** to fetch the name of the file, and **node.get_contents()** to fetch contents of the file. Note that the file is *not* guaranteed to exist before the scanner is called, so the scanner function should check that if there's any chance that the scanned file might not exist (for example, if it's built from other files).

The **env** argument is the construction environment for the scan. Fetch values from it using the **env.Dictionary()** method.

The **path** argument is a tuple (or list) of directories that can be searched for files. This will usually be the tuple returned by the **path_function** argument (see below).

The **arg** argument is the argument supplied when the scanner was created, if any.

name The name of the Scanner. This is mainly used to identify the Scanner internally.

argument
An optional argument that, if specified, will be passed to the scanner function (described above) and the path function (specified below).

skeys An optional list that can be used to determine which scanner should be used for a given Node. In the usual case of scanning for file names, this argument will be a list of suffixes for the different file types that this Scanner knows how to scan. If the argument is a string, then it will be expanded into a list by the current environment.

path_function
A Python function that takes four or five arguments: a construction environment, a Node for the directory containing the SConscript file in which the first target was defined, a list of target nodes, a list of source nodes, and an optional argument supplied when the scanner was created. The **path_function** returns a tuple of directories that can be searched for files to be returned by this Scanner object. (Note that the **FindPathDirs**() function can be used to return a ready-made **path_function** for a given construction variable name, instead of having to write your own function from scratch.)

node_class
The class of Node that should be returned by this Scanner object. Any strings or other objects returned by the scanner function that are not of this class will be run through the **node_factory** function.

node_factory
A Python function that will take a string or other object and turn it into the appropriate class of Node to be returned by this Scanner object.

scan_check
An optional Python function that takes two arguments, a Node (file) and a construction environment, and returns whether the Node should, in fact, be scanned for dependencies. This check can be used to eliminate unnecessary calls to the scanner function when, for example, the underlying file represented by a Node does not yet exist.

recursive
An optional flag that specifies whether this scanner should be re-invoked on the dependency files returned by the scanner. When this flag is not set, the Node subsystem will only invoke the scanner on the file being scanned, and not (for example) also on the files specified by the #include lines in the file being scanned. *recursive* may be a callable function, in which case it will be called with a list of Nodes found and should return a list of Nodes that should be scanned recursively; this can be used to select a specific subset of Nodes for additional scanning.

Note that **scons** has a global **SourceFileScanner** object that is used by the **Object**(), **SharedObject**(), and **StaticObject**() builders to decide which scanner should be used for different file extensions. You can using the **SourceFileScanner.add_scanner**() method to add your own Scanner object to the **scons** infrastructure that builds target programs or libraries from a list of source files of different types:

```
def xyz_scan(node, env, path):
    contents = node.get_contents()
    # Scan the contents and return the included files.
```

```
XYZScanner = Scanner(xyz_scan)

SourceFileScanner.add_scanner('.xyx', XYZScanner)

env.Program('my_prog', ['file1.c', 'file2.f', 'file3.xyz'])
```

SYSTEM-SPECIFIC BEHAVIOR

SCons and its configuration files are very portable, due largely to its implementation in Python. There are, however, a few portability issues waiting to trap the unwary.

.C file suffix

SCons handles the upper-case **.C** file suffix differently, depending on the capabilities of the underlying system. On a case-sensitive system such as Linux or UNIX, SCons treats a file with a **.C** suffix as a C++ source file. On a case-insensitive system such as Windows, SCons treats a file with a **.C** suffix as a C source file.

.F file suffix

SCons handles the upper-case **.F** file suffix differently, depending on the capabilities of the underlying system. On a case-sensitive system such as Linux or UNIX, SCons treats a file with a **.F** suffix as a Fortran source file that is to be first run through the standard C preprocessor. On a case-insensitive system such as Windows, SCons treats a file with a **.F** suffix as a Fortran source file that should *not* be run through the C preprocessor.

Windows: Cygwin Tools and Cygwin Python vs. Windows Pythons

Cygwin supplies a set of tools and utilities that let users work on a Windows system using a more POSIX-like environment. The Cygwin tools, including Cygwin Python, do this, in part, by sharing an ability to interpret UNIX-like path names. For example, the Cygwin tools will internally translate a Cygwin path name like /cygdrive/c/mydir to an equivalent Windows pathname of C:/mydir (equivalent to C:\mydir).

Versions of Python that are built for native Windows execution, such as the python.org and ActiveState versions, do not have the Cygwin path name semantics. This means that using a native Windows version of Python to build compiled programs using Cygwin tools (such as gcc, bison, and flex) may yield unpredictable results. "Mixing and matching" in this way can be made to work, but it requires careful attention to the use of path names in your SConscript files.

In practice, users can sidestep the issue by adopting the following rules: When using gcc, use the Cygwin-supplied Python interpreter to run SCons; when using Microsoft Visual C/C++ (or some other Windows compiler) use the python.org or ActiveState version of Python to run SCons.

Windows: scons.bat file

On Windows systems, SCons is executed via a wrapper **scons.bat** file. This has (at least) two ramifications:

First, Windows command-line users that want to use variable assignment on the command line may have to put double quotes around the assignments:

```
scons "FOO=BAR" "BAZ=BLEH"
```

Second, the Cygwin shell does not recognize this file as being the same as an **scons** command issued at the command-line prompt. You can work around this either by executing **scons.bat** from the Cygwin command line, or by creating a wrapper shell script named **scons .**

MinGW

The MinGW bin directory must be in your PATH environment variable or the PATH variable under the ENV construction variable for SCons to detect and use the MinGW tools. When running under the native Windows Python interpreter, SCons will prefer the MinGW tools over the Cygwin tools, if they are both installed, regardless of the order of the bin directories in the PATH variable. If you have both MSVC and

MinGW installed and you want to use MinGW instead of MSVC, then you must explictly tell SCons to use MinGW by passing

```
tools=['mingw']
```

to the Environment() function, because SCons will prefer the MSVC tools over the MinGW tools.

EXAMPLES

To help you get started using SCons, this section contains a brief overview of some common tasks.

Basic Compilation From a Single Source File

```
env = Environment()
env.Program(target = 'foo', source = 'foo.c')
```

Note: Build the file by specifying the target as an argument ("scons foo" or "scons foo.exe"). or by specifying a dot ("scons .").

Basic Compilation From Multiple Source Files

```
env = Environment()
env.Program(target = 'foo', source = Split('f1.c f2.c f3.c'))
```

Setting a Compilation Flag

```
env = Environment(CCFLAGS = '-g')
env.Program(target = 'foo', source = 'foo.c')
```

Search The Local Directory For .h Files

Note: You do *not* need to set CCFLAGS to specify -I options by hand. SCons will construct the right -I options from CPPPATH.

```
env = Environment(CPPPATH = ['.'])
env.Program(target = 'foo', source = 'foo.c')
```

Search Multiple Directories For .h Files

```
env = Environment(CPPPATH = ['include1', 'include2'])
env.Program(target = 'foo', source = 'foo.c')
```

Building a Static Library

```
env = Environment()
env.StaticLibrary(target = 'foo', source = Split('l1.c l2.c'))
env.StaticLibrary(target = 'bar', source = ['l3.c', 'l4.c'])
```

Building a Shared Library

```
env = Environment()
env.SharedLibrary(target = 'foo', source = ['l5.c', 'l6.c'])
env.SharedLibrary(target = 'bar', source = Split('l7.c l8.c'))
```

Linking a Local Library Into a Program

```
env = Environment(LIBS = 'mylib', LIBPATH = ['.'])
env.Library(target = 'mylib', source = Split('l1.c l2.c'))
env.Program(target = 'prog', source = ['p1.c', 'p2.c'])
```

Defining Your Own Builder Object

Notice that when you invoke the Builder, you can leave off the target file suffix, and SCons will add it automatically.

```
bld = Builder(action = 'pdftex < $SOURCES > $TARGET'
          suffix = '.pdf',
          src_suffix = '.tex')
env = Environment(BUILDERS = {'PDFBuilder' : bld})
env.PDFBuilder(target = 'foo.pdf', source = 'foo.tex')

# The following creates "bar.pdf" from "bar.tex"
env.PDFBuilder(target = 'bar', source = 'bar')
```

Note also that the above initialization overwrites the default Builder objects, so the Environment created above can not be used call Builders like env.Program(), env.Object(), env.StaticLibrary(), etc.

Adding Your Own Builder Object to an Environment

```
bld = Builder(action = 'pdftex < $SOURCES > $TARGET'
          suffix = '.pdf',
          src_suffix = '.tex')
env = Environment()
env.Append(BUILDERS = {'PDFBuilder' : bld})
env.PDFBuilder(target = 'foo.pdf', source = 'foo.tex')
env.Program(target = 'bar', source = 'bar.c')
```

You also can use other Pythonic techniques to add to the BUILDERS construction variable, such as:

```
env = Environment()
env['BUILDERS]['PDFBuilder'] = bld
```

Defining Your Own Scanner Object

The following example shows an extremely simple scanner (the **kfile_scan**() function) that doesn't use a search path at all and simply returns the file names present on any **include** lines in the scanned file. This would implicitly assume that all included files live in the top-level directory:

```
import re

include_re = re.compile(r'^include\s+(\S+)$', re.M)

def kfile_scan(node, env, path, arg):
   contents = node.get_contents()
   includes = include_re.findall(contents)
   return includes

kscan = Scanner(name = 'kfile',
            function = kfile_scan,
            argument = None,
            skeys = ['.k'])
scanners = Environment().Dictionary('SCANNERS')
env = Environment(SCANNERS = scanners + [kscan])

env.Command('foo', 'foo.k', 'kprocess < $SOURCES > $TARGET')

bar_in = File('bar.in')
```

```
env.Command('bar', bar_in, 'kprocess $SOURCES > $TARGET')
bar_in.target_scanner = kscan
```

Here is a similar but more complete example that searches a path of directories (specified as the **MYPATH** construction variable) for files that actually exist:

```
include_re = re.compile(r'^include\s+(\S+)$', re.M)

def my_scan(node, env, path, arg):
   contents = node.get_contents()
   includes = include_re.findall(contents)
   if includes == []:
       return []
    results = []
    for inc in includes:
        for dir in path:
            file = dir + os.sep + inc
            if os.path.exists(file):
                results.append(file)
                break
    return results

scanner = Scanner(name = 'myscanner',
                 function = my_scan,
                 argument = None,
                 skeys = ['.x'],
                 path_function = FindPathDirs('MYPATH'),
                 )
scanners = Environment().Dictionary('SCANNERS')
env = Environment(SCANNERS = scanners + [scanner])
```

The **FindPathDirs**() function used in the previous example returns a function (actually a callable Python object) that will return a list of directories specified in the **$MYPATH** construction variable. If you need to customize how the search path is derived, you would provide your own **path_function** argument when creating the Scanner object, as follows:

```
# MYPATH is a list of directories to search for files in
def pf(env, dir, target, source, arg):
    top_dir = Dir('#').abspath
    results = []
    if env.has_key('MYPATH'):
        for p in env['MYPATH']:
            results.append(top_dir + os.sep + p)
    return results

scanner = Scanner(name = 'myscanner',
                 function = my_scan,
                 argument = None,
                 skeys = ['.x'],
                 path_function = pf,
                 )
```

Creating a Hierarchical Build

Notice that the file names specified in a subdirectory's SConscript file are relative to that subdirectory.

```
SConstruct:

    env = Environment()
    env.Program(target = 'foo', source = 'foo.c')

    SConscript('sub/SConscript')

sub/SConscript:

    env = Environment()
    # Builds sub/foo from sub/foo.c
    env.Program(target = 'foo', source = 'foo.c')

    SConscript('dir/SConscript')

sub/dir/SConscript:

    env = Environment()
    # Builds sub/dir/foo from sub/dir/foo.c
    env.Program(target = 'foo', source = 'foo.c')
```

Sharing Variables Between SConscript Files

You must explicitly Export() and Import() variables that you want to share between SConscript files.

```
SConstruct:

    env = Environment()
    env.Program(target = 'foo', source = 'foo.c')

    Export("env")
    SConscript('subdirectory/SConscript')

subdirectory/SConscript:

    Import("env")
    env.Program(target = 'foo', source = 'foo.c')
```

Building Multiple Variants From the Same Source

Use the variant_dir keyword argument to the SConscript function to establish one or more separate variant build directory trees for a given source directory:

```
SConstruct:

    cppdefines = ['FOO']
    Export("cppdefines")
    SConscript('src/SConscript', variant_dir='foo')

    cppdefines = ['BAR']
    Export("cppdefines")
    SConscript('src/SConscript', variant_dir='bar')
```

src/SConscript:

```
Import("cppdefines")
env = Environment(CPPDEFINES = cppdefines)
env.Program(target = 'src', source = 'src.c')
```

Note the use of the Export() method to set the "cppdefines" variable to a different value each time we call the SConscript function.

Hierarchical Build of Two Libraries Linked With a Program

SConstruct:

```
env = Environment(LIBPATH = ['#libA', '#libB'])
Export('env')
SConscript('libA/SConscript')
SConscript('libB/SConscript')
SConscript('Main/SConscript')
```

libA/SConscript:

```
Import('env')
env.Library('a', Split('a1.c a2.c a3.c'))
```

libB/SConscript:

```
Import('env')
env.Library('b', Split('b1.c b2.c b3.c'))
```

Main/SConscript:

```
Import('env')
e = env.Copy(LIBS = ['a', 'b'])
e.Program('foo', Split('m1.c m2.c m3.c'))
```

The '#' in the LIBPATH directories specify that they're relative to the top-level directory, so they don't turn into "Main/libA" when they're used in Main/SConscript.

Specifying only 'a' and 'b' for the library names allows SCons to append the appropriate library prefix and suffix for the current platform (for example, 'liba.a' on POSIX systems,

Customizing construction variables from the command line.

The following would allow the C compiler to be specified on the command line or in the file custom.py.

```
vars = Variables('custom.py')
vars.Add('CC', 'The C compiler.')
env = Environment(variables=vars)
Help(vars.GenerateHelpText(env))
```

The user could specify the C compiler on the command line:

```
scons "CC=my_cc"
```

or in the custom.py file:

```
CC = 'my_cc'
```

or get documentation on the options:

```
$ scons -h

CC: The C compiler.
    default: None
    actual: cc
```

Using Microsoft Visual C++ precompiled headers

Since windows.h includes everything and the kitchen sink, it can take quite some time to compile it over and over again for a bunch of object files, so Microsoft provides a mechanism to compile a set of headers once and then include the previously compiled headers in any object file. This technology is called precompiled headers. The general recipe is to create a file named "StdAfx.cpp" that includes a single header named "StdAfx.h", and then include every header you want to precompile in "StdAfx.h", and finally include "StdAfx.h" as the first header in all the source files you are compiling to object files. For example:

```
StdAfx.h:
       #include <windows.h>
       #include <my_big_header.h>

StdAfx.cpp:
       #include <StdAfx.h>

Foo.cpp:
       #include <StdAfx.h>

       /* do some stuff */

Bar.cpp:
       #include <StdAfx.h>

       /* do some other stuff */

SConstruct:
       env=Environment()
       env['PCHSTOP'] = 'StdAfx.h'
       env['PCH'] = env.PCH('StdAfx.cpp')[0]
       env.Program('MyApp', ['Foo.cpp', 'Bar.cpp'])
```

For more information see the document for the PCH builder, and the PCH and PCHSTOP construction variables. To learn about the details of precompiled headers consult the MSDN documention for /Yc, /Yu, and /Yp.

Using Microsoft Visual C++ external debugging information

Since including debugging information in programs and shared libraries can cause their size to increase significantly, Microsoft provides a mechanism for including the debugging information in an external file called a PDB file. SCons supports PDB files through the PDB construction variable.

```
SConstruct:
       env=Environment()
       env['PDB'] = 'MyApp.pdb'
```

```
env.Program('MyApp', ['Foo.cpp', 'Bar.cpp'])
```

For more information see the document for the PDB construction variable.

ENVIRONMENT

SCONS_LIB_DIR
: Specifies the directory that contains the SCons Python module directory (e.g. /home/aroach/scons-src-0.01/src/engine).

SCONSFLAGS
: A string of options that will be used by scons in addition to those passed on the command line.

SEE ALSO

scons User Manual, **scons** Design Document, **scons** source code.

AUTHORS

Steven Knight <knight@baldmt.com>
Anthony Roach <aroach@electriceyeball.com>