SCons User Guide 2.3.0

Steven Knight

SCons User Guide 2.3.0
by Steven Knight

Revision (2013/03/03 09:48:35) Edition
Published 2004, 2005, 2006, 2007, 2008, 2009, 2010
Copyright © 2004, 2005, 2006, 2007, 2008, 2009, 2010 Steven Knight

SCons User's Guide Copyright (c) 2004, 2005, 2006, 2007 Steven Knight

Table of Contents

Preface ..... **ix**
1. SCons Principles ix
2. A Caveat About This Guide's Completeness ix
3. Acknowledgements x
4. Contact xi

1. Building and Installing SCons **1**
1.1. Installing Python 1
1.2. Installing SCons From Pre-Built Packages 1
1.2.1. Installing SCons on Red Hat (and Other RPM-based) Linux Systems 2
1.2.2. Installing SCons on Debian Linux Systems 2
1.2.3. Installing SCons on Windows Systems 2
1.3. Building and Installing SCons on Any System 3
1.3.1. Building and Installing Multiple Versions of SCons Side-by-Side 3
1.3.2. Installing SCons in Other Locations 4
1.3.3. Building and Installing SCons Without Administrative Privileges 4

2. Simple Builds **5**
2.1. Building Simple C / C++ Programs 5
2.2. Building Object Files 6
2.3. Simple Java Builds 7
2.4. Cleaning Up After a Build 7
2.5. The `SConstruct` File 8
2.5.1. `SConstruct` Files Are Python Scripts 8
2.5.2. SCons Functions Are Order-Independent 8
2.6. Making the SCons Output Less Verbose 10

3. Less Simple Things to Do With Builds **11**
3.1. Specifying the Name of the Target (Output) File 11
3.2. Compiling Multiple Source Files 11
3.3. Making a list of files with `Glob` 12
3.4. Specifying Single Files Vs. Lists of Files 13
3.5. Making Lists of Files Easier to Read 14
3.6. Keyword Arguments 15
3.7. Compiling Multiple Programs 15
3.8. Sharing Source Files Between Multiple Programs 16

4. Building and Linking with Libraries **18**
4.1. Building Libraries 18
4.1.1. Building Libraries From Source Code or Object Files 18
4.1.2. Building Static Libraries Explicitly: the `StaticLibrary` Builder 19
4.1.3. Building Shared (DLL) Libraries: the `SharedLibrary` Builder 19
4.2. Linking with Libraries 20
4.3. Finding Libraries: the $LIBPATH Construction Variable 21

5. Node Objects 23
5.1. Builder Methods Return Lists of Target Nodes 23
5.2. Explicitly Creating File and Directory Nodes 24
5.3. Printing `Node` File Names 24
5.4. Using a `Node`'s File Name as a String 25
5.5. `GetBuildPath`: Getting the Path From a `Node` or String 26
6. Dependencies 27
6.1. Deciding When an Input File Has Changed: the `Decider` Function 27
6.1.1. Using MD5 Signatures to Decide if a File Has Changed 27
6.1.2. Using Time Stamps to Decide If a File Has Changed 29
6.1.3. Deciding If a File Has Changed Using Both MD Signatures and Time Stamps 30
6.1.4. Writing Your Own Custom `Decider` Function 31
6.1.5. Mixing Different Ways of Deciding If a File Has Changed 33
6.2. Older Functions for Deciding When an Input File Has Changed 33
6.2.1. The `SourceSignatures` Function 34
6.2.2. The `TargetSignatures` Function 34
6.3. Implicit Dependencies: The $CPPPATH Construction Variable 35
6.4. Caching Implicit Dependencies 37
6.4.1. The `--implicit-deps-changed` Option 38
6.4.2. The `--implicit-deps-unchanged` Option 38
6.5. Explicit Dependencies: the `Depends` Function 39
6.6. Dependencies From External Files: the `ParseDepends` Function 39
6.7. Ignoring Dependencies: the `Ignore` Function 41
6.8. Order-Only Dependencies: the `Requires` Function 42
6.9. The `AlwaysBuild` Function 44
7. Environments 46
7.1. Using Values From the External Environment 46
7.2. Construction Environments 47
7.2.1. Creating a `Construction Environment`: the `Environment` Function 47
7.2.2. Fetching Values From a `Construction Environment` 48
7.2.3. Expanding Values From a `Construction Environment`: the `subst` Method 49
7.2.4. Handling Problems With Value Expansion 50
7.2.5. Controlling the Default `Construction Environment`: the `DefaultEnvironment` Function 51
7.2.6. Multiple `Construction Environments` 52
7.2.7. Making Copies of `Construction Environments`: the `Clone` Method 53
7.2.8. Replacing Values: the `Replace` Method 54
7.2.9. Setting Values Only If They're Not Already Defined: the `SetDefault` Method 56
7.2.10. Appending to the End of Values: the `Append` Method 56
7.2.11. Appending Unique Values: the `AppendUnique` Method 57
7.2.12. Appending to the Beginning of Values: the `Prepend` Method 57
7.2.13. Prepending Unique Values: the `PrependUnique` Method 58
7.3. Controlling the Execution Environment for Issued Commands 58
7.3.1. Propagating `PATH` From the External Environment 59
7.3.2. Adding to `PATH` Values in the Execution Environment 59

8. Automatically Putting Command-line Options into their Construction Variables 61
8.1. Merging Options into the Environment: the `MergeFlags` Function 61
8.2. Separating Compile Arguments into their Variables: the `ParseFlags` Function 62
8.3. Finding Installed Library Information: the `ParseConfig` Function 64
9. Controlling Build Output 66
9.1. Providing Build Help: the `Help` Function 66
9.2. Controlling How SCons Prints Build Commands: the $*COMSTR Variables 67
9.3. Providing Build Progress Output: the `Progress` Function 69
9.4. Printing Detailed Build Status: the `GetBuildFailures` Function 71
10. Controlling a Build From the Command Line 74
10.1. Command-Line Options 74
10.1.1. Not Having to Specify Command-Line Options Each Time: the `SCONSFLAGS` Environment Variable 74
10.1.2. Getting Values Set by Command-Line Options: the `GetOption` Function 75
10.1.3. Setting Values of Command-Line Options: the `SetOption` Function 75
10.1.4. Strings for Getting or Setting Values of SCons Command-Line Options 77
10.1.5. Adding Custom Command-Line Options: the `AddOption` Function 78
10.2. Command-Line `variable=value` Build Variables 79
10.2.1. Controlling Command-Line Build Variables 80
10.2.2. Providing Help for Command-Line Build Variables 81
10.2.3. Reading Build Variables From a File 82
10.2.4. Pre-Defined Build Variable Functions 83
10.2.5. Adding Multiple Command-Line Build Variables at Once 90
10.2.6. Handling Unknown Command-Line Build Variables: the `UnknownVariables` Function 91
10.3. Command-Line Targets 92
10.3.1. Fetching Command-Line Targets: the `COMMAND_LINE_TARGETS` Variable 92
10.3.2. Controlling the Default Targets: the `Default` Function 92
10.3.3. Fetching the List of Build Targets, Regardless of Origin: the `BUILD_TARGETS` Variable 96
11. Installing Files in Other Directories: the `Install` Builder 98
11.1. Installing Multiple Files in a Directory 98
11.2. Installing a File Under a Different Name 99
11.3. Installing Multiple Files Under Different Names 100
12. Platform-Independent File System Manipulation 101
12.1. Copying Files or Directories: The `Copy` Factory 101
12.2. Deleting Files or Directories: The `Delete` Factory 102
12.3. Moving (Renaming) Files or Directories: The `Move` Factory 103
12.4. Updating the Modification Time of a File: The `Touch` Factory 103
12.5. Creating a Directory: The `Mkdir` Factory 104
12.6. Changing File or Directory Permissions: The `Chmod` Factory 104
12.7. Executing an action immediately: the `Execute` Function 105
13. Controlling Removal of Targets 107
13.1. Preventing target removal during build: the `Precious` Function 107
13.2. Preventing target removal during clean: the `NoClean` Function 107
13.3. Removing additional files during clean: the `Clean` Function 108

14. Hierarchical Builds 110
14.1. SConscript Files 110
14.2. Path Names Are Relative to the SConscript Directory 110
14.3. Top-Level Path Names in Subsidiary SConscript Files 111
14.4. Absolute Path Names 112
14.5. Sharing Environments (and Other Variables) Between SConscript Files 113
14.5.1. Exporting Variables 113
14.5.2. Importing Variables 114
14.5.3. Returning Values From an SConscript File 115
15. Separating Source and Build Directories 116
15.1. Specifying a Variant Directory Tree as Part of an SConscript Call 116
15.2. Why SCons Duplicates Source Files in a Variant Directory Tree 116
15.3. Telling SCons to Not Duplicate Source Files in the Variant Directory Tree 117
15.4. The VariantDir Function 118
15.5. Using VariantDir With an SConscript File 119
15.6. Using Glob with VariantDir 119
16. Variant Builds 121
17. Internationalization and localization with gettext 123
17.1. Prerequisites 123
17.2. Simple project 123
18. Writing Your Own Builders 129
18.1. Writing Builders That Execute External Commands 129
18.2. Attaching a Builder to a Construction Environment 129
18.3. Letting SCons Handle The File Suffixes 130
18.4. Builders That Execute Python Functions 131
18.5. Builders That Create Actions Using a Generator 132
18.6. Builders That Modify the Target or Source Lists Using an Emitter 133
18.7. Where To Put Your Custom Builders and Tools 135
19. Not Writing a Builder: the Command Builder 138
20. Pseudo-Builders: the AddMethod function 140
21. Writing Scanners 142
21.1. A Simple Scanner Example 142
21.2. Adding a search path to a scanner: FindPathDirs 143
22. Building From Code Repositories 145
22.1. The Repository Method 145
22.2. Finding source files in repositories 145
22.3. Finding #include files in repositories 146
22.3.1. Limitations on #include files in repositories 147
22.4. Finding the SConstruct file in repositories 148
22.5. Finding derived files in repositories 148
22.6. Guaranteeing local copies of files 149

23. Multi-Platform Configuration (Autoconf Functionality) 151
23.1. `Configure Contexts` 151
23.2. Checking for the Existence of Header Files 151
23.3. Checking for the Availability of a Function 152
23.4. Checking for the Availability of a Library 152
23.5. Checking for the Availability of a `typedef` 153
23.6. Adding Your Own Custom Checks 153
23.7. Not Configuring When Cleaning Targets 155
24. Caching Built Files 157
24.1. Specifying the Shared Cache Directory 157
24.2. Keeping Build Output Consistent 157
24.3. Not Using the Shared Cache for Specific Files 158
24.4. Disabling the Shared Cache 159
24.5. Populating a Shared Cache With Already-Built Files 159
24.6. Minimizing Cache Contention: the `--random` Option 160
25. Alias Targets 162
26. Java Builds 164
26.1. Building Java Class Files: the `Java` Builder 164
26.2. How SCons Handles Java Dependencies 164
26.3. Building Java Archive (`.jar`) Files: the `Jar` Builder 165
26.4. Building C Header and Stub Files: the `JavaH` Builder 166
26.5. Building RMI Stub and Skeleton Class Files: the `RMIC` Builder 167
27. Miscellaneous Functionality 169
27.1. Verifying the Python Version: the `EnsurePythonVersion` Function 169
27.2. Verifying the SCons Version: the `EnsureSConsVersion` Function 169
27.3. Explicitly Terminating SCons While Reading `SConscript` Files: the `Exit` Function 170
27.4. Searching for Files: the `FindFile` Function 170
27.5. Handling Nested Lists: the `Flatten` Function 172
27.6. Finding the Invocation Directory: the `GetLaunchDir` Function 173
28. Troubleshooting 175
28.1. Why is That Target Being Rebuilt? the `--debug=explain` Option 175
28.2. What's in That Construction Environment? the `Dump` Method 177
28.3. What Dependencies Does SCons Know About? the `--tree` Option 181
28.4. How is SCons Constructing the Command Lines It Executes? the `--debug=presub` Option 187
28.5. Where is SCons Searching for Libraries? the `--debug=findlibs` Option 188
28.6. Where is SCons Blowing Up? the `--debug=stacktrace` Option 188
28.7. How is SCons Making Its Decisions? the `--taskmastertrace` Option 189
28.8. Watch SCons prepare targets for building: the `--debug=prepare` Option 191
28.9. Why is a file disappearing? the --debug=duplicate Option 191
A. Construction Variables 193
B. Builders 251
C. Tools 275
D. Functions and Environment Methods 293
E. Handling Common Tasks 335

List of Examples

E-1. Wildcard globbing to create a list of filenames....335
E-2. Filename extension substitution....335
E-3. Appending a path prefix to a list of filenames....335
E-4. Substituting a path prefix with another one....335
E-5. Filtering a filename list to exclude/retain only a specific set of extensions....335
E-6. The "backtick function": run a shell command and capture the output....335
E-7. Generating source code: how code can be generated and used by SCons....335

Preface

Thank you for taking the time to read about SCons. SCons is a next-generation software construction tool, or make tool--that is, a software utility for building software (or other files) and keeping built software up-to-date whenever the underlying input files change.

The most distinctive thing about SCons is that its configuration files are actually *scripts*, written in the Python programming language. This is in contrast to most alternative build tools, which typically invent a new language to configure the build. SCons still has a learning curve, of course, because you have to know what functions to call to set up your build properly, but the underlying syntax used should be familiar to anyone who has ever looked at a Python script.

Paradoxically, using Python as the configuration file format makes SCons *easier* for non-programmers to learn than the cryptic languages of other build tools, which are usually invented by programmers for other programmers. This is in no small part due to the consistency and readability that are hallmarks of Python. It just so happens that making a real, live scripting language the basis for the configuration files makes it a snap for more accomplished programmers to do more complicated things with builds, as necessary.

1. SCons Principles

There are a few overriding principles we try to live up to in designing and implementing SCons:

Correctness

: First and foremost, by default, SCons guarantees a correct build even if it means sacrificing performance a little. We strive to guarantee the build is correct regardless of how the software being built is structured, how it may have been written, or how unusual the tools are that build it.

Performance

: Given that the build is correct, we try to make SCons build software as quickly as possible. In particular, wherever we may have needed to slow down the default SCons behavior to guarantee a correct build, we also try to make it easy to speed up SCons through optimization options that let you trade off guaranteed correctness in all end cases for a speedier build in the usual cases.

Convenience

: SCons tries to do as much for you out of the box as reasonable, including detecting the right tools on your system and using them correctly to build the software.

In a nutshell, we try hard to make SCons just "do the right thing" and build software correctly, with a minimum of hassles.

2. A Caveat About This Guide's Completeness

One word of warning as you read through this Guide: Like too much Open Source software out there, the SCons documentation isn't always kept up-to-date with the available features. In other words, there's a lot that SCons can do that isn't yet covered in this User's Guide. (Come to think of it, that also describes a lot of proprietary software, doesn't it?)

Although this User's Guide isn't as complete as we'd like it to be, our development process does emphasize making sure that the SCons man page is kept up-to-date with new features. So if you're trying to figure out how to do something that SCons supports but can't find enough (or any) information here, it would be worth your while to look at the man page to see if the information is covered there. And if you do, maybe you'd even consider contributing a section to the User's Guide so the next person looking for that information won't have to go through the same thing...?

3. Acknowledgements

SCons would not exist without a lot of help from a lot of people, many of whom may not even be aware that they helped or served as inspiration. So in no particular order, and at the risk of leaving out someone:

First and foremost, SCons owes a tremendous debt to Bob Sidebotham, the original author of the classic Perl-based Cons tool which Bob first released to the world back around 1996. Bob's work on Cons classic provided the underlying architecture and model of specifying a build configuration using a real scripting language. My real-world experience working on Cons informed many of the design decisions in SCons, including the improved parallel build support, making Builder objects easily definable by users, and separating the build engine from the wrapping interface.

Greg Wilson was instrumental in getting SCons started as a real project when he initiated the Software Carpentry design competition in February 2000. Without that nudge, marrying the advantages of the Cons classic architecture with the readability of Python might have just stayed no more than a nice idea.

The entire SCons team have been absolutely wonderful to work with, and SCons would be nowhere near as useful a tool without the energy, enthusiasm and time people have contributed over the past few years. The "core team" of Chad Austin, Anthony Roach, Bill Deegan, Charles Crain, Steve Leblanc, Greg Noel, Gary Oberbrunner, Greg Spencer and Christoph Wiedemann have been great about reviewing my (and other) changes and catching problems before they get in the code base. Of particular technical note: Anthony's outstanding and innovative work on the tasking engine has given SCons a vastly superior parallel build model; Charles has been the master of the crucial Node infrastructure; Christoph's work on the Configure infrastructure has added crucial Autoconf-like functionality; and Greg has provided excellent support for Microsoft Visual Studio.

Special thanks to David Snopek for contributing his underlying "Autoscons" code that formed the basis of Christoph's work with the Configure functionality. David was extremely generous in making this code

available to SCons, given that he initially released it under the GPL and SCons is released under a less-restrictive MIT-style license.

Thanks to Peter Miller for his splendid change management system, Aegis, which has provided the SCons project with a robust development methodology from day one, and which showed me how you could integrate incremental regression tests into a practical development cycle (years before eXtreme Programming arrived on the scene).

And last, thanks to Guido van Rossum for his elegant scripting language, which is the basis not only for the SCons implementation, but for the interface itself.

4. Contact

The best way to contact people involved with SCons, including the author, is through the SCons mailing lists.

If you want to ask general questions about how to use SCons send email to `users@scons.tigris.org`.

If you want to contact the SCons development community directly, send email to `dev@scons.tigris.org`.

If you want to receive announcements about SCons, join the low-volume `announce@scons.tigris.org` mailing list.

Chapter 1. Building and Installing SCons

This chapter will take you through the basic steps of installing SCons on your system, and building SCons if you don't have a pre-built package available (or simply prefer the flexibility of building it yourself). Before that, however, this chapter will also describe the basic steps involved in installing Python on your system, in case that is necessary. Fortunately, both SCons and Python are very easy to install on almost any system, and Python already comes installed on many systems.

1.1. Installing Python

Because SCons is written in Python, you must obviously have Python installed on your system to use SCons. Before you try to install Python, you should check to see if Python is already available on your system by typing **`python -V`** (capital 'V') or **`python --version`** at your system's command-line prompt.

```
$ python -V
Python 2.5.1
```

And on a Windows system with Python installed:

```
C:\>python -V
Python 2.5.1
```

If Python is not installed on your system, you will see an error message stating something like "command not found" (on UNIX or Linux) or "'python' is not recognized as an internal or external command, operable progam or batch file" (on Windows). In that case, you need to install Python before you can install SCons.

The standard location for information about downloading and installing Python is http://www.python.org/download/. See that page for information about how to download and install Python on your system.

SCons will work with any 2.x version of Python from 2.4 on; 3.0 and later are not yet supported. If you need to install Python and have a choice, we recommend using the most recent 2.x Python version available. Newer Pythons have significant improvements that help speed up the performance of SCons.

1.2. Installing SCons From Pre-Built Packages

SCons comes pre-packaged for installation on a number of systems, including Linux and Windows

systems. You do not need to read this entire section, you should need to read only the section appropriate to the type of system you're running on.

1.2.1. Installing SCons on Red Hat (and Other RPM-based) Linux Systems

SCons comes in RPM (Red Hat Package Manager) format, pre-built and ready to install on Red Hat Linux, Fedora, or any other Linux distribution that uses RPM. Your distribution may already have an SCons RPM built specifically for it; many do, including SUSE, Mandrake and Fedora. You can check for the availability of an SCons RPM on your distribution's download servers, or by consulting an RPM search site like http://www.rpmfind.net/ or http://rpm.pbone.net/.

If your distribution supports installation via yum, you should be able to install SCons by running:

```
# yum install scons
```

If your Linux distribution does not already have a specific SCons RPM file, you can download and install from the generic RPM provided by the SCons project. This will install the SCons script(s) in `/usr/bin`, and the SCons library modules in `/usr/lib/scons`.

To install from the command line, simply download the appropriate `.rpm` file, and then run:

```
# rpm -Uvh scons-2.3.0-1.noarch.rpm
```

Or, you can use a graphical RPM package manager. See your package manager application's documention for specific instructions about how to use it to install a downloaded RPM.

1.2.2. Installing SCons on Debian Linux Systems

Debian Linux systems use a different package management format that also makes it very easy to install SCons.

If your system is connected to the Internet, you can install the latest official Debian package by running:

```
# apt-get install scons
```

1.2.3. Installing SCons on Windows Systems

SCons provides a Windows installer that makes installation extremely easy. Download the `scons-2.3.0.win32.exe` file from the SCons download page at http://www.scons.org/download.php. Then all you need to do is execute the file (usually by clicking on its icon in Windows Explorer). These will take you through a small sequence of windows that will install SCons on your system.

1.3. Building and Installing SCons on Any System

If a pre-built SCons package is not available for your system, then you can still easily build and install SCons using the native Python `distutils` package.

The first step is to download either the `scons-2.3.0.tar.gz` or `scons-2.3.0.zip`, which are available from the SCons download page at http://www.scons.org/download.html.

Unpack the archive you downloaded, using a utility like tar on Linux or UNIX, or WinZip on Windows. This will create a directory called `scons-2.3.0`, usually in your local directory. Then change your working directory to that directory and install SCons by executing the following commands:

```
# cd scons-2.3.0
# python setup.py install
```

This will build SCons, install the `scons` script in the python which is used to run the setup.py's scripts directory (`/usr/local/bin` or `C:\Python25\Scripts`), and will install the SCons build engine in the corresponding library directory for the python used (`/usr/local/lib/scons` or `C:\Python25\scons`). Because these are system directories, you may need root (on Linux or UNIX) or Administrator (on Windows) privileges to install SCons like this.

1.3.1. Building and Installing Multiple Versions of SCons Side-by-Side

The SCons `setup.py` script has some extensions that support easy installation of multiple versions of SCons in side-by-side locations. This makes it easier to download and experiment with different versions of SCons before moving your official build process to a new version, for example.

To install SCons in a version-specific location, add the `--version-lib` option when you call `setup.py`:

```
# python setup.py install --version-lib
```

This will install the SCons build engine in the `/usr/lib/scons-2.3.0` or `C:\Python25\scons-2.3.0` directory, for example.

If you use the `--version-lib` option the first time you install SCons, you do not need to specify it each time you install a new version. The SCons `setup.py` script will detect the version-specific directory name(s) and assume you want to install all versions in version-specific directories. You can override that assumption in the future by explicitly specifying the `--standalone-lib` option.

1.3.2. Installing SCons in Other Locations

You can install SCons in locations other than the default by specifying the `--prefix=` option:

```
# python setup.py install --prefix=/opt/scons
```

This would install the scons script in `/opt/scons/bin` and the build engine in `/opt/scons/lib/scons`,

Note that you can specify both the `--prefix=` and the `--version-lib` options at the same type, in which case `setup.py` will install the build engine in a version-specific directory relative to the specified prefix. Adding `--version-lib` to the above example would install the build engine in `/opt/scons/lib/scons-2.3.0`.

1.3.3. Building and Installing SCons Without Administrative Privileges

If you don't have the right privileges to install SCons in a system location, simply use the `--prefix=` option to install it in a location of your choosing. For example, to install SCons in appropriate locations relative to the user's `$HOME` directory, the `scons` script in `$HOME/bin` and the build engine in `$HOME/lib/scons`, simply type:

```
$ python setup.py install --prefix=$HOME
```

You may, of course, specify any other location you prefer, and may use the `--version-lib` option if you would like to install version-specific directories relative to the specified prefix.

This can also be used to experiment with a newer version of SCons than the one installed in your system locations. Of course, the location in which you install the newer version of the `scons` script (`$HOME/bin` in the above example) must be configured in your `PATH` variable before the directory containing the system-installed version of the `scons` script.

Chapter 2. Simple Builds

In this chapter, you will see several examples of very simple build configurations using SCons, which will demonstrate how easy it is to use SCons to build programs from several different programming languages on different types of systems.

2.1. Building Simple C / C++ Programs

Here's the famous "Hello, World!" program in C:

```
int
main()
{
    printf("Hello, world!\n");
}
```

And here's how to build it using SCons. Enter the following into a file named `SConstruct`:

```
Program('hello.c')
```

This minimal configuration file gives SCons two pieces of information: what you want to build (an executable program), and the input file from which you want it built (the `hello.c` file). `Program` is a *builder_method*, a Python call that tells SCons that you want to build an executable program.

That's it. Now run the `scons` command to build the program. On a POSIX-compliant system like Linux or UNIX, you'll see something like:

```
% scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
cc -o hello.o -c hello.c
cc -o hello hello.o
scons: done building targets.
```

On a Windows system with the Microsoft Visual C++ compiler, you'll see something like:

```
C:\>scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
cl /Fohello.obj /c hello.c /nologo
```

```
link /nologo /OUT:hello.exe hello.obj
embedManifestExeCheck(target, source, env)
scons: done building targets.
```

First, notice that you only need to specify the name of the source file, and that SCons correctly deduces the names of the object and executable files to be built from the base of the source file name.

Second, notice that the same input `SConstruct` file, without any changes, generates the correct output file names on both systems: `hello.o` and `hello` on POSIX systems, `hello.obj` and `hello.exe` on Windows systems. This is a simple example of how SCons makes it extremely easy to write portable software builds.

(Note that we won't provide duplicate side-by-side POSIX and Windows output for all of the examples in this guide; just keep in mind that, unless otherwise specified, any of the examples should work equally well on both types of systems.)

2.2. Building Object Files

The `Program` builder method is only one of many builder methods that SCons provides to build different types of files. Another is the `Object` builder method, which tells SCons to build an object file from the specified source file:

```
Object('hello.c')
```

Now when you run the `scons` command to build the program, it will build just the `hello.o` object file on a POSIX system:

```
% scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
cc -o hello.o -c hello.c
scons: done building targets.
```

And just the `hello.obj` object file on a Windows system (with the Microsoft Visual C++ compiler):

```
C:\>scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
cl /Fohello.obj /c hello.c /nologo
scons: done building targets.
```

2.3. Simple Java Builds

SCons also makes building with Java extremely easy. Unlike the `Program` and `Object` builder methods, however, the `Java` builder method requires that you specify the name of a destination directory in which you want the class files placed, followed by the source directory in which the `.java` files live:

```
Java('classes', 'src')
```

If the `src` directory contains a single `hello.java` file, then the output from running the `scons` command would look something like this (on a POSIX system):

```
% scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
javac -d classes -sourcepath src src/hello.java
scons: done building targets.
```

We'll cover Java builds in more detail, including building Java archive (`.jar`) and other types of file, in Chapter 26.

2.4. Cleaning Up After a Build

When using SCons, it is unnecessary to add special commands or target names to clean up after a build. Instead, you simply use the `-c` or `--clean` option when you invoke SCons, and SCons removes the appropriate built files. So if we build our example above and then invoke `scons -c` afterwards, the output on POSIX looks like:

```
% scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
cc -o hello.o -c hello.c
cc -o hello hello.o
scons: done building targets.
% scons -c
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Cleaning targets ...
Removed hello.o
Removed hello
scons: done cleaning targets.
```

And the output on Windows looks like:

```
C:\>scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
cl /Fohello.obj /c hello.c /nologo
link /nologo /OUT:hello.exe hello.obj
embedManifestExeCheck(target, source, env)
scons: done building targets.
C:\>scons -c
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Cleaning targets ...
Removed hello.obj
Removed hello.exe
scons: done cleaning targets.
```

Notice that SCons changes its output to tell you that it is `Cleaning targets ...` and `done cleaning targets.`

2.5. The `SConstruct` File

If you're used to build systems like Make you've already figured out that the `SConstruct` file is the SCons equivalent of a `Makefile`. That is, the `SConstruct` file is the input file that SCons reads to control the build.

2.5.1. `SConstruct` Files Are Python Scripts

There is, however, an important difference between an `SConstruct` file and a `Makefile`: the `SConstruct` file is actually a Python script. If you're not already familiar with Python, don't worry. This User's Guide will introduce you step-by-step to the relatively small amount of Python you'll need to know to be able to use SCons effectively. And Python is very easy to learn.

One aspect of using Python as the scripting language is that you can put comments in your `SConstruct` file using Python's commenting convention; that is, everything between a '#' and the end of the line will be ignored:

```
# Arrange to build the "hello" program.
Program('hello.c')    # "hello.c" is the source file.
```

You'll see throughout the remainder of this Guide that being able to use the power of a real scripting language can greatly simplify the solutions to complex requirements of real-world builds.

2.5.2. SCons Functions Are Order-Independent

One important way in which the `SConstruct` file is not exactly like a normal Python script, and is more like a `Makefile`, is that the order in which the SCons functions are called in the `SConstruct` file does *not* affect the order in which SCons actually builds the programs and object files you want it to build.[1] In other words, when you call the `Program` builder (or any other builder method), you're not telling SCons to build the program at the instant the builder method is called. Instead, you're telling SCons to build the program that you want, for example, a program built from a file named `hello.c`, and it's up to SCons to build that program (and any other files) whenever it's necessary. (We'll learn more about how SCons decides when building or rebuilding a file is necessary in Chapter 6, below.)

SCons reflects this distinction between *calling a builder method like* `Program` and *actually building the program* by printing the status messages that indicate when it's "just reading" the `SConstruct` file, and when it's actually building the target files. This is to make it clear when SCons is executing the Python statements that make up the `SConstruct` file, and when SCons is actually executing the commands or other actions to build the necessary files.

Let's clarify this with an example. Python has a `print` statement that prints a string of characters to the screen. If we put `print` statements around our calls to the `Program` builder method:

```
print "Calling Program('hello.c')"
Program('hello.c')
print "Calling Program('goodbye.c')"
Program('goodbye.c')
print "Finished calling Program()"
```

Then when we execute SCons, we see the output from the `print` statements in between the messages about reading the `SConscript` files, indicating that that is when the Python statements are being executed:

```
% scons
scons: Reading SConscript files ...
Calling Program('hello.c')
Calling Program('goodbye.c')
Finished calling Program()
scons: done reading SConscript files.
scons: Building targets ...
cc -o goodbye.o -c goodbye.c
cc -o goodbye goodbye.o
cc -o hello.o -c hello.c
cc -o hello hello.o
scons: done building targets.
```

Notice also that SCons built the `goodbye` program first, even though the "reading `SConscript`" output shows that we called `Program('hello.c')` first in the `SConstruct` file.

2.6. Making the SCons Output Less Verbose

You've already seen how SCons prints some messages about what it's doing, surrounding the actual commands used to build the software:

```
C:\>scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
cl /Fohello.obj /c hello.c /nologo
link /nologo /OUT:hello.exe hello.obj
embedManifestExeCheck(target, source, env)
scons: done building targets.
```

These messages emphasize the order in which SCons does its work: all of the configuration files (generically referred to as `SConscript` files) are read and executed first, and only then are the target files built. Among other benefits, these messages help to distinguish between errors that occur while the configuration files are read, and errors that occur while targets are being built.

One drawback, of course, is that these messages clutter the output. Fortunately, they're easily disabled by using the `-Q` option when invoking SCons:

```
C:\>scons -Q
cl /Fohello.obj /c hello.c /nologo
link /nologo /OUT:hello.exe hello.obj
embedManifestExeCheck(target, source, env)
```

Because we want this User's Guide to focus on what SCons is actually doing, we're going to use the `-Q` option to remove these messages from the output of all the remaining examples in this Guide.

Notes

1. In programming parlance, the `SConstruct` file is *declarative*, meaning you tell SCons what you want done and let it figure out the order in which to do it, rather than strictly *imperative*, where you specify explicitly the order in which to do things.

Chapter 3. Less Simple Things to Do With Builds

In this chapter, you will see several examples of very simple build configurations using SCons, which will demonstrate how easy it is to use SCons to build programs from several different programming languages on different types of systems.

3.1. Specifying the Name of the Target (Output) File

You've seen that when you call the `Program` builder method, it builds the resulting program with the same base name as the source file. That is, the following call to build an executable program from the `hello.c` source file will build an executable program named `hello` on POSIX systems, and an executable program named `hello.exe` on Windows systems:

```
Program('hello.c')
```

If you want to build a program with a different name than the base of the source file name, you simply put the target file name to the left of the source file name:

```
Program('new_hello', 'hello.c')
```

(SCons requires the target file name first, followed by the source file name, so that the order mimics that of an assignment statement in most programming languages, including Python: `"program = source files"`.)

Now SCons will build an executable program named `new_hello` when run on a POSIX system:

```
% scons -Q
cc -o hello.o -c hello.c
cc -o new_hello hello.o
```

And SCons will build an executable program named `new_hello.exe` when run on a Windows system:

```
C:\>scons -Q
cl /Fohello.obj /c hello.c /nologo
link /nologo /OUT:new_hello.exe hello.obj
embedManifestExeCheck(target, source, env)
```

3.2. Compiling Multiple Source Files

You've just seen how to configure SCons to compile a program from a single source file. It's more common, of course, that you'll need to build a program from many input source files, not just one. To do this, you need to put the source files in a Python list (enclosed in square brackets), like so:

```
Program(['prog.c', 'file1.c', 'file2.c'])
```

A build of the above example would look like:

```
% scons -Q
cc -o file1.o -c file1.c
cc -o file2.o -c file2.c
cc -o prog.o -c prog.c
cc -o prog prog.o file1.o file2.o
```

Notice that SCons deduces the output program name from the first source file specified in the list--that is, because the first source file was `prog.c`, SCons will name the resulting program `prog` (or `prog.exe` on a Windows system). If you want to specify a different program name, then (as we've seen in the previous section) you slide the list of source files over to the right to make room for the output program file name. (SCons puts the output file name to the left of the source file names so that the order mimics that of an assignment statement: "program = source files".) This makes our example:

```
Program('program', ['prog.c', 'file1.c', 'file2.c'])
```

On Linux, a build of this example would look like:

```
% scons -Q
cc -o file1.o -c file1.c
cc -o file2.o -c file2.c
cc -o prog.o -c prog.c
cc -o program prog.o file1.o file2.o
```

Or on Windows:

```
C:\>scons -Q
cl /Fofile1.obj /c file1.c /nologo
cl /Fofile2.obj /c file2.c /nologo
cl /Foprog.obj /c prog.c /nologo
link /nologo /OUT:program.exe prog.obj file1.obj file2.obj
embedManifestExeCheck(target, source, env)
```

3.3. Making a list of files with `Glob`

You can also use the `Glob` function to find all files matching a certain template, using the standard shell pattern matching characters `*`, `?` and `[abc]` to match any of `a`, `b` or `c`. `[!abc]` is also supported, to match any character *except* `a`, `b` or `c`. This makes many multi-source-file builds quite easy:

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

The SCons man page has more details on using `Glob` with variant directories (see Chapter 16, below) and repositories (see Chapter 22, below), and returning strings rather than Nodes.

3.4. Specifying Single Files Vs. Lists of Files

We've now shown you two ways to specify the source for a program, one with a list of files:

```
Program('hello', ['file1.c', 'file2.c'])
```

And one with a single file:

```
Program('hello', 'hello.c')
```

You could actually put a single file name in a list, too, which you might prefer just for the sake of consistency:

```
Program('hello', ['hello.c'])
```

SCons functions will accept a single file name in either form. In fact, internally, SCons treats all input as lists of files, but allows you to omit the square brackets to cut down a little on the typing when there's only a single file name.

Important: Although SCons functions are forgiving about whether or not you use a string vs. a list for a single file name, Python itself is more strict about treating lists and strings differently. So where SCons allows either a string or list:

```
# The following two calls both work correctly:
Program('program1', 'program1.c')
Program('program2', ['program2.c'])
```

Trying to do "Python things" that mix strings and lists will cause errors or lead to incorrect results:

```
common_sources = ['file1.c', 'file2.c']
```

```
# THE FOLLOWING IS INCORRECT AND GENERATES A PYTHON ERROR
# BECAUSE IT TRIES TO ADD A STRING TO A LIST:
Program('program1', common_sources + 'program1.c')

# The following works correctly, because it's adding two
# lists together to make another list.
Program('program2', common_sources + ['program2.c'])
```

3.5. Making Lists of Files Easier to Read

One drawback to the use of a Python list for source files is that each file name must be enclosed in quotes (either single quotes or double quotes). This can get cumbersome and difficult to read when the list of file names is long. Fortunately, SCons and Python provide a number of ways to make sure that the `SConstruct` file stays easy to read.

To make long lists of file names easier to deal with, SCons provides a `Split` function that takes a quoted list of file names, with the names separated by spaces or other white-space characters, and turns it into a list of separate file names. Using the `Split` function turns the previous example into:

```
Program('program', Split('main.c file1.c file2.c'))
```

(If you're already familiar with Python, you'll have realized that this is similar to the `split()` method in the Python standard `string` module. Unlike the `split()` member function of strings, however, the `Split` function does not require a string as input and will wrap up a single non-string object in a list, or return its argument untouched if it's already a list. This comes in handy as a way to make sure arbitrary values can be passed to SCons functions without having to check the type of the variable by hand.)

Putting the call to the `Split` function inside the `Program` call can also be a little unwieldy. A more readable alternative is to assign the output from the `Split` call to a variable name, and then use the variable when calling the `Program` function:

```
src_files = Split('main.c file1.c file2.c')
Program('program', src_files)
```

Lastly, the `Split` function doesn't care how much white space separates the file names in the quoted string. This allows you to create lists of file names that span multiple lines, which often makes for easier editing:

```
src_files = Split("""main.c
                     file1.c
                     file2.c""")
```

```
Program('program', src_files)
```

(Note in this example that we used the Python "triple-quote" syntax, which allows a string to contain multiple lines. The three quotes can be either single or double quotes.)

3.6. Keyword Arguments

SCons also allows you to identify the output file and input source files using Python keyword arguments. The output file is known as the *target*, and the source file(s) are known (logically enough) as the *source*. The Python syntax for this is:

```
src_files = Split('main.c file1.c file2.c')
Program(target = 'program', source = src_files)
```

Because the keywords explicitly identify what each argument is, you can actually reverse the order if you prefer:

```
src_files = Split('main.c file1.c file2.c')
Program(source = src_files, target = 'program')
```

Whether or not you choose to use keyword arguments to identify the target and source files, and the order in which you specify them when using keywords, are purely personal choices; SCons functions the same regardless.

3.7. Compiling Multiple Programs

In order to compile multiple programs within the same `SConstruct` file, simply call the `Program` method multiple times, once for each program you need to build:

```
Program('foo.c')
Program('bar', ['bar1.c', 'bar2.c'])
```

SCons would then build the programs as follows:

```
% scons -Q
cc -o bar1.o -c bar1.c
cc -o bar2.o -c bar2.c
cc -o bar bar1.o bar2.o
cc -o foo.o -c foo.c
cc -o foo foo.o
```

Notice that SCons does not necessarily build the programs in the same order in which you specify them in the `SConstruct` file. SCons does, however, recognize that the individual object files must be built before the resulting program can be built. We'll discuss this in greater detail in the "Dependencies" section, below.

3.8. Sharing Source Files Between Multiple Programs

It's common to re-use code by sharing source files between multiple programs. One way to do this is to create a library from the common source files, which can then be linked into resulting programs. (Creating libraries is discussed in Chapter 4, below.)

A more straightforward, but perhaps less convenient, way to share source files between multiple programs is simply to include the common files in the lists of source files for each program:

```
Program(Split('foo.c common1.c common2.c'))
Program('bar', Split('bar1.c bar2.c common1.c common2.c'))
```

SCons recognizes that the object files for the `common1.c` and `common2.c` source files each need to be built only once, even though the resulting object files are each linked in to both of the resulting executable programs:

```
% scons -Q
cc -o bar1.o -c bar1.c
cc -o bar2.o -c bar2.c
cc -o common1.o -c common1.c
cc -o common2.o -c common2.c
cc -o bar bar1.o bar2.o common1.o common2.o
cc -o foo.o -c foo.c
cc -o foo foo.o common1.o common2.o
```

If two or more programs share a lot of common source files, repeating the common files in the list for each program can be a maintenance problem when you need to change the list of common files. You can simplify this by creating a separate Python list to hold the common file names, and concatenating it with other lists using the Python + operator:

```
common = ['common1.c', 'common2.c']
foo_files = ['foo.c'] + common
bar_files = ['bar1.c', 'bar2.c'] + common
Program('foo', foo_files)
Program('bar', bar_files)
```

This is functionally equivalent to the previous example.

Chapter 4. Building and Linking with Libraries

It's often useful to organize large software projects by collecting parts of the software into one or more libraries. SCons makes it easy to create libraries and to use them in the programs.

4.1. Building Libraries

You build your own libraries by specifying `Library` instead of `Program`:

```
Library('foo', ['f1.c', 'f2.c', 'f3.c'])
```

SCons uses the appropriate library prefix and suffix for your system. So on POSIX or Linux systems, the above example would build as follows (although ranlib may not be called on all systems):

```
% scons -Q
cc -o f1.o -c f1.c
cc -o f2.o -c f2.c
cc -o f3.o -c f3.c
ar rc libfoo.a f1.o f2.o f3.o
ranlib libfoo.a
```

On a Windows system, a build of the above example would look like:

```
C:\>scons -Q
cl /Fof1.obj /c f1.c /nologo
cl /Fof2.obj /c f2.c /nologo
cl /Fof3.obj /c f3.c /nologo
lib /nologo /OUT:foo.lib f1.obj f2.obj f3.obj
```

The rules for the target name of the library are similar to those for programs: if you don't explicitly specify a target library name, SCons will deduce one from the name of the first source file specified, and SCons will add an appropriate file prefix and suffix if you leave them off.

4.1.1. Building Libraries From Source Code or Object Files

The previous example shows building a library from a list of source files. You can, however, also give the `Library` call object files, and it will correctly realize In fact, you can arbitrarily mix source code files and object files in the source list:

```
Library('foo', ['f1.c', 'f2.o', 'f3.c', 'f4.o'])
```

And SCons realizes that only the source code files must be compiled into object files before creating the final library:

```
% scons -Q
cc -o f1.o -c f1.c
cc -o f3.o -c f3.c
ar rc libfoo.a f1.o f2.o f3.o f4.o
ranlib libfoo.a
```

Of course, in this example, the object files must already exist for the build to succeed. See Chapter 5, below, for information about how you can build object files explicitly and include the built files in a library.

4.1.2. Building Static Libraries Explicitly: the `StaticLibrary` Builder

The `Library` function builds a traditional static library. If you want to be explicit about the type of library being built, you can use the synonym `StaticLibrary` function instead of `Library`:

```
StaticLibrary('foo', ['f1.c', 'f2.c', 'f3.c'])
```

There is no functional difference between the `StaticLibrary` and `Library` functions.

4.1.3. Building Shared (DLL) Libraries: the `SharedLibrary` Builder

If you want to build a shared library (on POSIX systems) or a DLL file (on Windows systems), you use the `SharedLibrary` function:

```
SharedLibrary('foo', ['f1.c', 'f2.c', 'f3.c'])
```

The output on POSIX:

```
% scons -Q
cc -o f1.os -c f1.c
cc -o f2.os -c f2.c
cc -o f3.os -c f3.c
cc -o libfoo.so -shared f1.os f2.os f3.os
```

And the output on Windows:

```
C:\>scons -Q
cl /Fof1.obj /c f1.c /nologo
cl /Fof2.obj /c f2.c /nologo
cl /Fof3.obj /c f3.c /nologo
link /nologo /dll /out:foo.dll /implib:foo.lib f1.obj f2.obj f3.obj
RegServerFunc(target, source, env)
embedManifestDllCheck(target, source, env)
```

Notice again that SCons takes care of building the output file correctly, adding the `-shared` option for a POSIX compilation, and the `/dll` option on Windows.

4.2. Linking with Libraries

Usually, you build a library because you want to link it with one or more programs. You link libraries with a program by specifying the libraries in the $LIBS construction variable, and by specifying the directory in which the library will be found in the $LIBPATH construction variable:

```
Library('foo', ['f1.c', 'f2.c', 'f3.c'])
Program('prog.c', LIBS=['foo', 'bar'], LIBPATH='.')
```

Notice, of course, that you don't need to specify a library prefix (like `lib`) or suffix (like `.a` or `.lib`). SCons uses the correct prefix or suffix for the current system.

On a POSIX or Linux system, a build of the above example would look like:

```
% scons -Q
cc -o f1.o -c f1.c
cc -o f2.o -c f2.c
cc -o f3.o -c f3.c
ar rc libfoo.a f1.o f2.o f3.o
ranlib libfoo.a
cc -o prog.o -c prog.c
cc -o prog prog.o -L. -lfoo -lbar
```

On a Windows system, a build of the above example would look like:

```
C:\>scons -Q
cl /Fof1.obj /c f1.c /nologo
cl /Fof2.obj /c f2.c /nologo
cl /Fof3.obj /c f3.c /nologo
lib /nologo /OUT:foo.lib f1.obj f2.obj f3.obj
cl /Foprog.obj /c prog.c /nologo
link /nologo /OUT:prog.exe /LIBPATH:. foo.lib bar.lib prog.obj
```

```
embedManifestExeCheck(target, source, env)
```

As usual, notice that SCons has taken care of constructing the correct command lines to link with the specified library on each system.

Note also that, if you only have a single library to link with, you can specify the library name in single string, instead of a Python list, so that:

```
Program('prog.c', LIBS='foo', LIBPATH='.')
```

is equivalent to:

```
Program('prog.c', LIBS=['foo'], LIBPATH='.')
```

This is similar to the way that SCons handles either a string or a list to specify a single source file.

4.3. Finding Libraries: the $LIBPATH Construction Variable

By default, the linker will only look in certain system-defined directories for libraries. SCons knows how to look for libraries in directories that you specify with the $LIBPATH construction variable. $LIBPATH consists of a list of directory names, like so:

```
Program('prog.c', LIBS = 'm',
                  LIBPATH = ['/usr/lib', '/usr/local/lib'])
```

Using a Python list is preferred because it's portable across systems. Alternatively, you could put all of the directory names in a single string, separated by the system-specific path separator character: a colon on POSIX systems:

```
LIBPATH = '/usr/lib:/usr/local/lib'
```

or a semi-colon on Windows systems:

```
LIBPATH = 'C:\\lib;D:\\lib'
```

(Note that Python requires that the backslash separators in a Windows path name be escaped within strings.)

When the linker is executed, SCons will create appropriate flags so that the linker will look for libraries in the same directories as SCons. So on a POSIX or Linux system, a build of the above example would look like:

```
% scons -Q
cc -o prog.o -c prog.c
cc -o prog prog.o -L/usr/lib -L/usr/local/lib -lm
```

On a Windows system, a build of the above example would look like:

```
C:\>scons -Q
cl /Foprog.obj /c prog.c /nologo
link /nologo /OUT:prog.exe /LIBPATH:\usr\lib /LIBPATH:\usr\local\lib m.lib prog.obj
embedManifestExeCheck(target, source, env)
```

Note again that SCons has taken care of the system-specific details of creating the right command-line options.

Chapter 5. Node Objects

Internally, SCons represents all of the files and directories it knows about as `Nodes`. These internal objects (not object *files*) can be used in a variety of ways to make your `SConscript` files portable and easy to read.

5.1. Builder Methods Return Lists of Target Nodes

All builder methods return a list of `Node` objects that identify the target file or files that will be built. These returned `Nodes` can be passed as arguments to other builder methods.

For example, suppose that we want to build the two object files that make up a program with different options. This would mean calling the `Object` builder once for each object file, specifying the desired options:

```
Object('hello.c', CCFLAGS='-DHELLO')
Object('goodbye.c', CCFLAGS='-DGOODBYE')
```

One way to combine these object files into the resulting program would be to call the `Program` builder with the names of the object files listed as sources:

```
Object('hello.c', CCFLAGS='-DHELLO')
Object('goodbye.c', CCFLAGS='-DGOODBYE')
Program(['hello.o', 'goodbye.o'])
```

The problem with specifying the names as strings is that our `SConstruct` file is no longer portable across operating systems. It won't, for example, work on Windows because the object files there would be named `hello.obj` and `goodbye.obj`, not `hello.o` and `goodbye.o`.

A better solution is to assign the lists of targets returned by the calls to the `Object` builder to variables, which we can then concatenate in our call to the `Program` builder:

```
hello_list = Object('hello.c', CCFLAGS='-DHELLO')
goodbye_list = Object('goodbye.c', CCFLAGS='-DGOODBYE')
Program(hello_list + goodbye_list)
```

This makes our `SConstruct` file portable again, the build output on Linux looking like:

```
% scons -Q
cc -o goodbye.o -c -DGOODBYE goodbye.c
cc -o hello.o -c -DHELLO hello.c
```

```
cc -o hello hello.o goodbye.o
```

And on Windows:

```
C:\>scons -Q
cl /Fogoodbye.obj /c goodbye.c -DGOODBYE
cl /Fohello.obj /c hello.c -DHELLO
link /nologo /OUT:hello.exe hello.obj goodbye.obj
embedManifestExeCheck(target, source, env)
```

We'll see examples of using the list of nodes returned by builder methods throughout the rest of this guide.

5.2. Explicitly Creating File and Directory Nodes

It's worth mentioning here that SCons maintains a clear distinction between Nodes that represent files and Nodes that represent directories. SCons supports `File` and `Dir` functions that, respectively, return a file or directory Node:

```
hello_c = File('hello.c')
Program(hello_c)

classes = Dir('classes')
Java(classes, 'src')
```

Normally, you don't need to call `File` or `Dir` directly, because calling a builder method automatically treats strings as the names of files or directories, and translates them into the Node objects for you. The `File` and `Dir` functions can come in handy in situations where you need to explicitly instruct SCons about the type of Node being passed to a builder or other function, or unambiguously refer to a specific file in a directory tree.

There are also times when you may need to refer to an entry in a file system without knowing in advance whether it's a file or a directory. For those situations, SCons also supports an `Entry` function, which returns a Node that can represent either a file or a directory.

```
xyzzy = Entry('xyzzy')
```

The returned `xyzzy` Node will be turned into a file or directory Node the first time it is used by a builder method or other function that requires one vs. the other.

5.3. Printing Node File Names

One of the most common things you can do with a Node is use it to print the file name that the node represents. Keep in mind, though, that because the object returned by a builder call is a *list* of Nodes, you must use Python subscripts to fetch individual Nodes from the list. For example, the following SConstruct file:

```
hello_c = File('hello.c')
Program(hello_c)

classes = Dir('classes')
Java(classes, 'src')

object_list = Object('hello.c')
program_list = Program(object_list)
print "The object file is:", object_list[0]
print "The program file is:", program_list[0]
```

Would print the following file names on a POSIX system:

```
% scons -Q
The object file is: hello.o
The program file is: hello
cc -o hello.o -c hello.c
cc -o hello hello.o
```

And the following file names on a Windows system:

```
C:\>scons -Q
The object file is: hello.obj
The program file is: hello.exe
cl /Fohello.obj /c hello.c /nologo
link /nologo /OUT:hello.exe hello.obj
embedManifestExeCheck(target, source, env)
```

Note that in the above example, the object_list[0] extracts an actual Node *object* from the list, and the Python print statement converts the object to a string for printing.

5.4. Using a Node's File Name as a String

Printing a Node's name as described in the previous section works because the string representation of a Node object is the name of the file. If you want to do something other than print the name of the file, you can fetch it by using the builtin Python str function. For example, if you want to use the Python

`os.path.exists` to figure out whether a file exists while the `SConstruct` file is being read and executed, you can fetch the string as follows:

```
import os.path
program_list = Program('hello.c')
program_name = str(program_list[0])
if not os.path.exists(program_name):
    print program_name, "does not exist!"
```

Which executes as follows on a POSIX system:

```
% scons -Q
hello does not exist!
cc -o hello.o -c hello.c
cc -o hello hello.o
```

5.5. `GetBuildPath`: Getting the Path From a `Node` or String

`env.GetBuildPath(file_or_list)` returns the path of a `Node` or a string representing a path. It can also take a list of `Node`s and/or strings, and returns the list of paths. If passed a single `Node`, the result is the same as calling `str(node)` (see above). The string(s) can have embedded construction variables, which are expanded as usual, using the calling environment's set of variables. The paths can be files or directories, and do not have to exist.

```
env=Environment(VAR="value")
n=File("foo.c")
print env.GetBuildPath([n, "sub/dir/$VAR"])
```

Would print the following file names:

```
% scons -Q
['foo.c', 'sub/dir/value']
scons: `.' is up to date.
```

There is also a function version of `GetBuildPath` which can be called without an `Environment`; that uses the default SCons `Environment` to do substitution on any string arguments.

Chapter 6. Dependencies

So far we've seen how SCons handles one-time builds. But one of the main functions of a build tool like SCons is to rebuild only what is necessary when source files change--or, put another way, SCons should *not* waste time rebuilding things that don't need to be rebuilt. You can see this at work simply by re-invoking SCons after building our simple `hello` example:

```
% scons -Q
cc -o hello.o -c hello.c
cc -o hello hello.o
% scons -Q
scons: `.' is up to date.
```

The second time it is executed, SCons realizes that the `hello` program is up-to-date with respect to the current `hello.c` source file, and avoids rebuilding it. You can see this more clearly by naming the `hello` program explicitly on the command line:

```
% scons -Q hello
cc -o hello.o -c hello.c
cc -o hello hello.o
% scons -Q hello
scons: `hello' is up to date.
```

Note that SCons reports `"...is up to date"` only for target files named explicitly on the command line, to avoid cluttering the output.

6.1. Deciding When an Input File Has Changed: the `Decider` Function

Another aspect of avoiding unnecessary rebuilds is the fundamental build tool behavior of *rebuilding* things when an input file changes, so that the built software is up to date. By default, SCons keeps track of this through an MD5 `signature`, or checksum, of the contents of each file, although you can easily configure SCons to use the modification times (or time stamps) instead. You can even specify your own Python function for deciding if an input file has changed.

6.1.1. Using MD5 Signatures to Decide if a File Has Changed

By default, SCons keeps track of whether a file has changed based on an MD5 checksum of the file's contents, not the file's modification time. This means that you may be surprised by the default SCons behavior if you are used to the Make convention of forcing a rebuild by updating the file's modification time (using the touch command, for example):

```
% scons -Q hello
cc -o hello.o -c hello.c
cc -o hello hello.o
% touch hello.c
% scons -Q hello
scons: 'hello' is up to date.
```

Even though the file's modification time has changed, SCons realizes that the contents of the `hello.c` file have *not* changed, and therefore that the `hello` program need not be rebuilt. This avoids unnecessary rebuilds when, for example, someone rewrites the contents of a file without making a change. But if the contents of the file really do change, then SCons detects the change and rebuilds the program as required:

```
% scons -Q hello
cc -o hello.o -c hello.c
cc -o hello hello.o
% edit hello.c
    [CHANGE THE CONTENTS OF hello.c]
% scons -Q hello
cc -o hello.o -c hello.c
cc -o hello hello.o
```

Note that you can, if you wish, specify this default behavior (MD5 signatures) explicitly using the `Decider` function as follows:

```
Program('hello.c')
Decider('MD5')
```

You can also use the string `'content'` as a synonym for `'MD5'` when calling the `Decider` function.

6.1.1.1. Ramifications of Using MD5 Signatures

Using MD5 signatures to decide if an input file has changed has one surprising benefit: if a source file has been changed in such a way that the contents of the rebuilt target file(s) will be exactly the same as the last time the file was built, then any "downstream" target files that depend on the rebuilt-but-not-changed target file actually need not be rebuilt.

So if, for example, a user were to only change a comment in a `hello.c` file, then the rebuilt `hello.o` file would be exactly the same as the one previously built (assuming the compiler doesn't put any build-specific information in the object file). SCons would then realize that it would not need to rebuild the `hello` program as follows:

```
% scons -Q hello
cc -o hello.o -c hello.c
cc -o hello hello.o
```

```
% edit hello.c
  [CHANGE A COMMENT IN hello.c]
% scons -Q hello
cc -o hello.o -c hello.c
scons: 'hello' is up to date.
```

In essence, SCons "short-circuits" any dependent builds when it realizes that a target file has been rebuilt to exactly the same file as the last build. This does take some extra processing time to read the contents of the target (`hello.o`) file, but often saves time when the rebuild that was avoided would have been time-consuming and expensive.

6.1.2. Using Time Stamps to Decide If a File Has Changed

If you prefer, you can configure SCons to use the modification time of a file, not the file contents, when deciding if a target needs to be rebuilt. SCons gives you two ways to use time stamps to decide if an input file has changed since the last time a target has been built.

The most familiar way to use time stamps is the way Make does: that is, have SCons decide that a target must be rebuilt if a source file's modification time is *newer* than the target file. To do this, call the `Decider` function as follows:

```
Object('hello.c')
Decider('timestamp-newer')
```

This makes SCons act like Make when a file's modification time is updated (using the touch command, for example):

```
% scons -Q hello.o
cc -o hello.o -c hello.c
% touch hello.c
% scons -Q hello.o
cc -o hello.o -c hello.c
```

And, in fact, because this behavior is the same as the behavior of Make, you can also use the string `'make'` as a synonym for `'timestamp-newer'` when calling the `Decider` function:

```
Object('hello.c')
Decider('make')
```

One drawback to using times stamps exactly like Make is that if an input file's modification time suddenly becomes *older* than a target file, the target file will not be rebuilt. This can happen if an old

copy of a source file is restored from a backup archive, for example. The contents of the restored file will likely be different than they were the last time a dependent target was built, but the target won't be rebuilt because the modification time of the source file is not newer than the target.

Because SCons actually stores information about the source files' time stamps whenever a target is built, it can handle this situation by checking for an exact match of the source file time stamp, instead of just whether or not the source file is newer than the target file. To do this, specify the argument `'timestamp-match'` when calling the `Decider` function:

```
Object('hello.c')
Decider('timestamp-match')
```

When configured this way, SCons will rebuild a target whenever a source file's modification time has changed. So if we use the `touch -t` option to change the modification time of `hello.c` to an old date (January 1, 1989), SCons will still rebuild the target file:

```
% scons -Q hello.o
cc -o hello.o -c hello.c
% touch -t 198901010000 hello.c
% scons -Q hello.o
cc -o hello.o -c hello.c
```

In general, the only reason to prefer `timestamp-newer` instead of `timestamp-match`, would be if you have some specific reason to require this Make-like behavior of not rebuilding a target when an otherwise-modified source file is older.

6.1.3. Deciding If a File Has Changed Using Both MD Signatures and Time Stamps

As a performance enhancement, SCons provides a way to use MD5 checksums of file contents but to read those contents only when the file's timestamp has changed. To do this, call the `Decider` function with `'MD5-timestamp'` argument as follows:

```
Program('hello.c')
Decider('MD5-timestamp')
```

So configured, SCons will still behave like it does when using `Decider('MD5')`:

```
% scons -Q hello
cc -o hello.o -c hello.c
cc -o hello hello.o
% touch hello.c
% scons -Q hello
```

```
scons: 'hello' is up to date.
% edit hello.c
    [CHANGE THE CONTENTS OF hello.c]
% scons -Q hello
cc -o hello.o -c hello.c
cc -o hello hello.o
```

However, the second call to SCons in the above output, when the build is up-to-date, will have been performed by simply looking at the modification time of the `hello.c` file, not by opening it and performing an MD5 checksum calcuation on its contents. This can significantly speed up many up-to-date builds.

The only drawback to using `Decider('MD5-timestamp')` is that SCons will *not* rebuild a target file if a source file was modified within one second of the last time SCons built the file. While most developers are programming, this isn't a problem in practice, since it's unlikely that someone will have built and then thought quickly enough to make a substantive change to a source file within one second. Certain build scripts or continuous integration tools may, however, rely on the ability to apply changes to files automatically and then rebuild as quickly as possible, in which case use of `Decider('MD5-timestamp')` may not be appropriate.

6.1.4. Writing Your Own Custom `Decider` Function

The different string values that we've passed to the `Decider` function are essentially used by SCons to pick one of several specific internal functions that implement various ways of deciding if a dependency (usually a source file) has changed since a target file has been built. As it turns out, you can also supply your own function to decide if a dependency has changed.

For example, suppose we have an input file that contains a lot of data, in some specific regular format, that is used to rebuild a lot of different target files, but each target file really only depends on one particular section of the input file. We'd like to have each target file depend on only its section of the input file. However, since the input file may contain a lot of data, we want to open the input file only if its timestamp has changed. This could be done with a custom `Decider` function that might look something like this:

```
Program('hello.c')
def decide_if_changed(dependency, target, prev_ni):
    if self.get_timestamp() != prev_ni.timestamp:
        dep = str(dependency)
        tgt = str(target)
        if specific_part_of_file_has_changed(dep, tgt):
            return True
    return False
Decider(decide_if_changed)
```

Note that in the function definition, the `dependency` (input file) is the first argument, and then the `target`. Both of these are passed to the functions as SCons `Node` objects, which we convert to strings using the Python `str()`.

The third argument, `prev_ni`, is an object that holds the signature or timestamp information that was recorded about the dependency the last time the target was built. A `prev_ni` object can hold different information, depending on the type of thing that the `dependency` argument represents. For normal files, the `prev_ni` object has the following attributes:

.csig

: The *content signature*, or MD5 checksum, of the contents of the `dependency` file the list time the `target` was built.

.size

: The size in bytes of the `dependency` file the list time the target was built.

.timestamp

: The modification time of the `dependency` file the list time the `target` was built.

Note that ignoring some of the arguments in your custom `Decider` function is a perfectly normal thing to do, if they don't impact the way you want to decide if the dependency file has changed.

Another thing to look out for is the fact that the three attributes above may not be present at the time of the first run. Without any prior build, no targets have been created and no `.sconsign` DB file exists yet. So, you should always check whether the `prev_ni` attribute in question is available.

We finally present a small example for a `csig`-based decider function. Note how the signature information for the `dependency` file has to get initialized via `get_csig` during each function call (this is mandatory!).

```
env = Environment()

def config_file_decider(dependency, target, prev_ni):
    import os.path

    # We always have to init the .csig value...
    dep_csig = dependency.get_csig()
    # .csig may not exist, because no target was built yet...
    if 'csig' not in dir(prev_ni):
        return True
    # Target file may not exist yet
    if not os.path.exists(str(target.abspath)):
        return True
    if dep_csig != prev_ni.csig:
        # Some change on source file => update installed one
        return True
    return False
```

```
def update_file():
    f = open("test.txt","a")
    f.write("some line\n")
    f.close()

update_file()

# Activate our own decider function
env.Decider(config_file_decider)

env.Install("install","test.txt")
```

6.1.5. Mixing Different Ways of Deciding If a File Has Changed

The previous examples have all demonstrated calling the global `Decider` function to configure all dependency decisions that SCons makes. Sometimes, however, you want to be able to configure different decision-making for different targets. When that's necessary, you can use the `env.Decider` method to affect only the configuration decisions for targets built with a specific construction environment.

For example, if we arbitrarily want to build one program using MD5 checkums and another using file modification times from the same source we might configure it this way:

```
env1 = Environment(CPPPATH = ['.'])
env2 = env1.Clone()
env2.Decider('timestamp-match')
env1.Program('prog-MD5', 'program1.c')
env2.Program('prog-timestamp', 'program2.c')
```

If both of the programs include the same `inc.h` file, then updating the modification time of `inc.h` (using the touch command) will cause only `prog-timestamp` to be rebuilt:

```
% scons -Q
cc -o program1.o -c -I. program1.c
cc -o prog-MD5 program1.o
cc -o program2.o -c -I. program2.c
cc -o prog-timestamp program2.o
% touch inc.h
% scons -Q
cc -o program2.o -c -I. program2.c
cc -o prog-timestamp program2.o
```

6.2. Older Functions for Deciding When an Input File Has Changed

SCons still supports two functions that used to be the primary methods for configuring the decision about whether or not an input file has changed. These functions have been officially deprecated as SCons version 2.0, and their use is discouraged, mainly because they rely on a somewhat confusing distinction between how source files and target files are handled. These functions are documented here mainly in case you encounter them in older `SConscript` files.

6.2.1. The `SourceSignatures` Function

The `SourceSignatures` function is fairly straightforward, and supports two different argument values to configure whether source file changes should be decided using MD5 signatures:

```
Program('hello.c')
SourceSignatures('MD5')
```

Or using time stamps:

```
Program('hello.c')
SourceSignatures('timestamp')
```

These are roughly equivalent to specifying `Decider('MD5')` or `Decider('timestamp-match')`, respectively, although it only affects how SCons makes decisions about dependencies on *source* files--that is, files that are not built from any other files.

6.2.2. The `TargetSignatures` Function

The `TargetSignatures` function specifies how SCons decides when a target file has changed *when it is used as a dependency of (input to) another target*--that is, the `TargetSignatures` function configures how the signatures of "intermediate" target files are used when deciding if a "downstream" target file must be rebuilt. [1]

The `TargetSignatures` function supports the same `'MD5'` and `'timestamp'` argument values that are supported by the `SourceSignatures`, with the same meanings, but applied to target files. That is, in the example:

```
Program('hello.c')
TargetSignatures('MD5')
```

The MD5 checksum of the `hello.o` target file will be used to decide if it has changed since the last time the "downstream" `hello` target file was built. And in the example:

```
Program('hello.c')
TargetSignatures('timestamp')
```

The modification time of the `hello.o` target file will be used to decide if it has changed since the last time the "downstream" `hello` target file was built.

The `TargetSignatures` function supports two additional argument values: `'source'` and `'build'`. The `'source'` argument specifies that decisions involving whether target files have changed since a previous build should use the same behavior for the decisions configured for source files (using the `SourceSignatures` function). So in the example:

```
Program('hello.c')
TargetSignatures('source')
SourceSignatures('timestamp')
```

All files, both targets and sources, will use modification times when deciding if an input file has changed since the last time a target was built.

Lastly, the `'build'` argument specifies that SCons should examine the build status of a target file and always rebuild a "downstream" target if the target file was itself rebuilt, without re-examining the contents or timestamp of the newly-built target file. If the target file was not rebuilt during this `scons` invocation, then the target file will be examined the same way as configured by the `SourceSignature` call to decide if it has changed.

This mimics the behavior of `build signatures` in earlier versions of SCons. A `build signature` re-combined signatures of all the input files that went into making the target file, so that the target file itself did not need to have its contents read to compute an MD5 signature. This can improve performance for some configurations, but is generally not as effective as using `Decider('MD5-timestamp')`.

6.3. Implicit Dependencies: The $CPPPATH Construction Variable

Now suppose that our "Hello, World!" program actually has an `#include` line to include the `hello.h` file in the compilation:

```
#include <hello.h>
int
main()
```

```
{
    printf("Hello, %s!\n", string);
}
```

And, for completeness, the `hello.h` file looks like this:

```
#define string    "world"
```

In this case, we want SCons to recognize that, if the contents of the `hello.h` file change, the `hello` program must be recompiled. To do this, we need to modify the `SConstruct` file like so:

```
Program('hello.c', CPPPATH = '.')
```

The $CPPPATH value tells SCons to look in the current directory (`'.'`) for any files included by C source files (`.c` or `.h` files). With this assignment in the `SConstruct` file:

```
% scons -Q hello
cc -o hello.o -c -I. hello.c
cc -o hello hello.o
% scons -Q hello
scons: `hello' is up to date.
% edit hello.h
    [CHANGE THE CONTENTS OF hello.h]
% scons -Q hello
cc -o hello.o -c -I. hello.c
cc -o hello hello.o
```

First, notice that SCons added the `-I.` argument from the $CPPPATH variable so that the compilation would find the `hello.h` file in the local directory.

Second, realize that SCons knows that the `hello` program must be rebuilt because it scans the contents of the `hello.c` file for the `#include` lines that indicate another file is being included in the compilation. SCons records these as *implicit dependencies* of the target file, Consequently, when the `hello.h` file changes, SCons realizes that the `hello.c` file includes it, and rebuilds the resulting `hello` program that depends on both the `hello.c` and `hello.h` files.

Like the $LIBPATH variable, the $CPPPATH variable may be a list of directories, or a string separated by the system-specific path separation character (':' on POSIX/Linux, ';' on Windows). Either way, SCons creates the right command-line options so that the following example:

```
Program('hello.c', CPPPATH = ['include', '/home/project/inc'])
```

Will look like this on POSIX or Linux:

```
% scons -Q hello
cc -o hello.o -c -Iinclude -I/home/project/inc hello.c
cc -o hello hello.o
```

And like this on Windows:

```
C:\>scons -Q hello.exe
cl /Fohello.obj /c hello.c /nologo /Iinclude /I\home\project\inc
link /nologo /OUT:hello.exe hello.obj
embedManifestExeCheck(target, source, env)
```

6.4. Caching Implicit Dependencies

Scanning each file for `#include` lines does take some extra processing time. When you're doing a full build of a large system, the scanning time is usually a very small percentage of the overall time spent on the build. You're most likely to notice the scanning time, however, when you *rebuild* all or part of a large system: SCons will likely take some extra time to "think about" what must be built before it issues the first build command (or decides that everything is up to date and nothing must be rebuilt).

In practice, having SCons scan files saves time relative to the amount of potential time lost to tracking down subtle problems introduced by incorrect dependencies. Nevertheless, the "waiting time" while SCons scans files can annoy individual developers waiting for their builds to finish. Consequently, SCons lets you cache the implicit dependencies that its scanners find, for use by later builds. You can do this by specifying the `--implicit-cache` option on the command line:

```
% scons -Q --implicit-cache hello
cc -o hello.o -c hello.c
cc -o hello hello.o
% scons -Q hello
scons: 'hello' is up to date.
```

If you don't want to specify `--implicit-cache` on the command line each time, you can make it the default behavior for your build by setting the `implicit_cache` option in an `SConscript` file:

```
SetOption('implicit_cache', 1)
```

SCons does not cache implicit dependencies like this by default because the `--implicit-cache` causes SCons to simply use the implicit dependencies stored during the last run, without any checking for

whether or not those dependencies are still correct. Specifically, this means `--implicit-cache` instructs SCons to *not* rebuild "correctly" in the following cases:

- When `--implicit-cache` is used, SCons will ignore any changes that may have been made to search paths (like $CPPPATH or $LIBPATH,). This can lead to SCons not rebuilding a file if a change to $CPPPATH would normally cause a different, same-named file from a different directory to be used.
- When `--implicit-cache` is used, SCons will not detect if a same-named file has been added to a directory that is earlier in the search path than the directory in which the file was found last time.

6.4.1. The `--implicit-deps-changed` Option

When using cached implicit dependencies, sometimes you want to "start fresh" and have SCons re-scan the files for which it previously cached the dependencies. For example, if you have recently installed a new version of external code that you use for compilation, the external header files will have changed and the previously-cached implicit dependencies will be out of date. You can update them by running SCons with the `--implicit-deps-changed` option:

```
% scons -Q --implicit-deps-changed hello
cc -o hello.o -c hello.c
cc -o hello hello.o
% scons -Q hello
scons: 'hello' is up to date.
```

In this case, SCons will re-scan all of the implicit dependencies and cache updated copies of the information.

6.4.2. The `--implicit-deps-unchanged` Option

By default when caching dependencies, SCons notices when a file has been modified and re-scans the file for any updated implicit dependency information. Sometimes, however, you may want to force SCons to use the cached implicit dependencies, even if the source files changed. This can speed up a build for example, when you have changed your source files but know that you haven't changed any `#include` lines. In this case, you can use the `--implicit-deps-unchanged` option:

```
% scons -Q --implicit-deps-unchanged hello
cc -o hello.o -c hello.c
cc -o hello hello.o
% scons -Q hello
scons: 'hello' is up to date.
```

In this case, SCons will assume that the cached implicit dependencies are correct and will not bother to re-scan changed files. For typical builds after small, incremental changes to source files, the savings may not be very big, but sometimes every bit of improved performance counts.

6.5. Explicit Dependencies: the `Depends` Function

Sometimes a file depends on another file that is not detected by an SCons scanner. For this situation, SCons allows you to specific explicitly that one file depends on another file, and must be rebuilt whenever that file changes. This is specified using the `Depends` method:

```
hello = Program('hello.c')
Depends(hello, 'other_file')
```

```
% scons -Q hello
cc -c hello.c -o hello.o
cc -o hello hello.o
% scons -Q hello
scons: 'hello' is up to date.
% edit other_file
    [CHANGE THE CONTENTS OF other_file]
% scons -Q hello
cc -c hello.c -o hello.o
cc -o hello hello.o
```

Note that the dependency (the second argument to `Depends`) may also be a list of Node objects (for example, as returned by a call to a Builder):

```
hello = Program('hello.c')
goodbye = Program('goodbye.c')
Depends(hello, goodbye)
```

in which case the dependency or dependencies will be built before the target(s):

```
% scons -Q hello
cc -c goodbye.c -o goodbye.o
cc -o goodbye goodbye.o
cc -c hello.c -o hello.o
cc -o hello hello.o
```

6.6. Dependencies From External Files: the `ParseDepends` Function

SCons has built-in scanners for a number of languages. Sometimes these scanners fail to extract certain implicit dependencies due to limitations of the scanner implementation.

The following example illustrates a case where the built-in C scanner is unable to extract the implicit dependency on a header file.

```
#define FOO_HEADER <foo.h>
#include FOO_HEADER

int main() {
    return FOO;
}
```

```
% scons -Q
cc -o hello.o -c -I. hello.c
cc -o hello hello.o
% edit foo.h
   [CHANGE CONTENTS OF foo.h]
% scons -Q
scons: `.' is up to date.
```

Apparently, the scanner does not know about the header dependency. Being not a full-fledged C preprocessor, the scanner does not expand the macro.

In these cases, you may also use the compiler to extract the implicit dependencies. `ParseDepends` can parse the contents of the compiler output in the style of Make, and explicitly establish all of the listed dependencies.

The following example uses `ParseDepends` to process a compiler generated dependency file which is generated as a side effect during compilation of the object file:

```
obj = Object('hello.c', CCFLAGS='-MD -MF hello.d', CPPPATH='.')
SideEffect('hello.d', obj)
ParseDepends('hello.d')
Program('hello', obj)
```

```
% scons -Q
cc -o hello.o -c -MD -MF hello.d -I. hello.c
cc -o hello hello.o
% edit foo.h
   [CHANGE CONTENTS OF foo.h]
% scons -Q
cc -o hello.o -c -MD -MF hello.d -I. hello.c
```

Parsing dependencies from a compiler-generated `.d` file has a chicken-and-egg problem, that causes unnecessary rebuilds:

```
% scons -Q
cc -o hello.o -c -MD -MF hello.d -I. hello.c
```

```
cc -o hello hello.o
% scons -Q --debug=explain
scons: rebuilding `hello.o' because `foo.h' is a new dependency
cc -o hello.o -c -MD -MF hello.d -I. hello.c
% scons -Q
scons: `.' is up to date.
```

In the first pass, the dependency file is generated while the object file is compiled. At that time, SCons does not know about the dependency on `foo.h`. In the second pass, the object file is regenerated because `foo.h` is detected as a new dependency.

`ParseDepends` immediately reads the specified file at invocation time and just returns if the file does not exist. A dependency file generated during the build process is not automatically parsed again. Hence, the compiler-extracted dependencies are not stored in the signature database during the same build pass. This limitation of `ParseDepends` leads to unnecessary recompilations. Therefore, `ParseDepends` should only be used if scanners are not available for the employed language or not powerful enough for the specific task.

6.7. Ignoring Dependencies: the `Ignore` Function

Sometimes it makes sense to not rebuild a program, even if a dependency file changes. In this case, you would tell SCons specifically to ignore a dependency as follows:

```
hello_obj=Object('hello.c')
hello = Program(hello_obj)
Ignore(hello_obj, 'hello.h')
```

```
% scons -Q hello
cc -c -o hello.o hello.c
cc -o hello hello.o
% scons -Q hello
scons: `hello' is up to date.
% edit hello.h
  [CHANGE THE CONTENTS OF hello.h]
% scons -Q hello
scons: `hello' is up to date.
```

Now, the above example is a little contrived, because it's hard to imagine a real-world situation where you wouldn't want to rebuild `hello` if the `hello.h` file changed. A more realistic example might be if the `hello` program is being built in a directory that is shared between multiple systems that have different copies of the `stdio.h` include file. In that case, SCons would notice the differences between the different systems' copies of `stdio.h` and would rebuild `hello` each time you change systems. You could avoid these rebuilds as follows:

```
hello = Program('hello.c', CPPPATH=['/usr/include'])
Ignore(hello, '/usr/include/stdio.h')
```

`Ignore` can also be used to prevent a generated file from being built by default. This is due to the fact that directories depend on their contents. So to ignore a generated file from the default build, you specify that the directory should ignore the generated file. Note that the file will still be built if the user specifically requests the target on scons command line, or if the file is a dependency of another file which is requested and/or is built by default.

```
hello_obj=Object('hello.c')
hello = Program(hello_obj)
Ignore('.',[hello,hello_obj])
```

```
% scons -Q
scons: `.' is up to date.
% scons -Q hello
cc -o hello.o -c hello.c
cc -o hello hello.o
% scons -Q hello
scons: `hello' is up to date.
```

6.8. Order-Only Dependencies: the `Requires` Function

Occasionally, it may be useful to specify that a certain file or directory must, if necessary, be built or created before some other target is built, but that changes to that file or directory do *not* require that the target itself be rebuilt. Such a relationship is called an *order-only dependency* because it only affects the order in which things must be built--the dependency before the target--but it is not a strict dependency relationship because the target should not change in response to changes in the dependent file.

For example, suppose that you want to create a file every time you run a build that identifies the time the build was performed, the version number, etc., and which is included in every program that you build. The version file's contents will change every build. If you specify a normal dependency relationship, then every program that depends on that file would be rebuilt every time you ran SCons. For example, we could use some Python code in a `SConstruct` file to create a new `version.c` file with a string containing the current date every time we run SCons, and then link a program with the resulting object file by listing `version.c` in the sources:

```
import time

version_c_text = """
char *date = "%s";
""" % time.ctime(time.time())
open('version.c', 'w').write(version_c_text)
```

```
hello = Program(['hello.c', 'version.c'])
```

If we list `version.c` as an actual source file, though, then the `version.o` file will get rebuilt every time we run SCons (because the `SConstruct` file itself changes the contents of `version.c`) and the `hello` executable will get re-linked every time (because the `version.o` file changes):

```
% scons -Q hello
cc -o hello.o -c hello.c
cc -o version.o -c version.c
cc -o hello hello.o version.o
% sleep 1
% scons -Q hello
cc -o version.o -c version.c
cc -o hello hello.o version.o
% sleep 1
% scons -Q hello
cc -o version.o -c version.c
cc -o hello hello.o version.o
```

(Note that for the above example to work, we sleep for one second in between each run, so that the `SConstruct` file will create a `version.c` file with a time string that's one second later than the previous run.)

One solution is to use the `Requires` function to specify that the `version.o` must be rebuilt before it is used by the link step, but that changes to `version.o` should not actually cause the `hello` executable to be re-linked:

```
import time

version_c_text = """
char *date = "%s";
""" % time.ctime(time.time())
open('version.c', 'w').write(version_c_text)

version_obj = Object('version.c')

hello = Program('hello.c',
                LINKFLAGS = str(version_obj[0]))

Requires(hello, version_obj)
```

Notice that because we can no longer list `version.c` as one of the sources for the `hello` program, we have to find some other way to get it into the link command line. For this example, we're cheating a bit and stuffing the object file name (extracted from `version_obj` list returned by the `Object` call) into the $LINKFLAGS variable, because $LINKFLAGS is already included in the $LINKCOM command line.

With these changes, we get the desired behavior of only re-linking the `hello` executable when the `hello.c` has changed, even though the `version.o` is rebuilt (because the `SConstruct` file still changes the `version.c` contents directly each run):

```
% scons -Q hello
cc -o version.o -c version.c
cc -o hello.o -c hello.c
cc -o hello version.o hello.o
% sleep 1
% scons -Q hello
cc -o version.o -c version.c
scons: 'hello' is up to date.
% sleep 1
% edit hello.c
    [CHANGE THE CONTENTS OF hello.c]
% scons -Q hello
cc -o version.o -c version.c
cc -o hello.o -c hello.c
cc -o hello version.o hello.o
% sleep 1
% scons -Q hello
cc -o version.o -c version.c
scons: 'hello' is up to date.
```

6.9. The `AlwaysBuild` Function

How SCons handles dependencies can also be affected by the `AlwaysBuild` method. When a file is passed to the `AlwaysBuild` method, like so:

```
hello = Program('hello.c')
AlwaysBuild(hello)
```

Then the specified target file (`hello` in our example) will always be considered out-of-date and rebuilt whenever that target file is evaluated while walking the dependency graph:

```
% scons -Q
cc -o hello.o -c hello.c
cc -o hello hello.o
% scons -Q
cc -o hello hello.o
```

The `AlwaysBuild` function has a somewhat misleading name, because it does not actually mean the target file will be rebuilt every single time SCons is invoked. Instead, it means that the target will, in fact, be rebuilt whenever the target file is encountered while evaluating the targets specified on the command

line (and their dependencies). So specifying some other target on the command line, a target that does *not* itself depend on the `AlwaysBuild` target, will still be rebuilt only if it's out-of-date with respect to its dependencies:

```
% scons -Q
cc -o hello.o -c hello.c
cc -o hello hello.o
% scons -Q hello.o
scons: 'hello.o' is up to date.
```

Notes

1. This easily-overlooked distinction between how SCons decides if the target itself must be rebuilt and how the target is then used to decide if a different target must be rebuilt is one of the confusing things that has led to the `TargetSignatures` and `SourceSignatures` functions being replaced by the simpler `Decider` function.

Chapter 7. Environments

An `environment` is a collection of values that can affect how a program executes. SCons distinguishes between three different types of environments that can affect the behavior of SCons itself (subject to the configuration in the `SConscript` files), as well as the compilers and other tools it executes:

External Environment

The `external environment` is the set of variables in the user's environment at the time the user runs SCons. These variables are available within the `SConscript` files through the Python `os.environ` dictionary. See Section 7.1, below.

`Construction Environment`

A `construction environment` is a distinct object creating within a `SConscript` file and and which contains values that affect how SCons decides what action to use to build a target, and even to define which targets should be built from which sources. One of the most powerful features of SCons is the ability to create multiple `construction environments`, including the ability to clone a new, customized `construction environment` from an existing `construction environment`. See Section 7.2, below.

Execution Environment

An `execution environment` is the values that SCons sets when executing an external command (such as a compiler or linker) to build one or more targets. Note that this is not the same as the `external environment` (see above). See Section 7.3, below.

Unlike Make, SCons does not automatically copy or import values between different environments (with the exception of explicit clones of `construction environments`, which inherit values from their parent). This is a deliberate design choice to make sure that builds are, by default, repeatable regardless of the values in the user's external environment. This avoids a whole class of problems with builds where a developer's local build works because a custom variable setting causes a different compiler or build option to be used, but the checked-in change breaks the official build because it uses different environment variable settings.

Note that the `SConscript` writer can easily arrange for variables to be copied or imported between environments, and this is often very useful (or even downright necessary) to make it easy for developers to customize the build in appropriate ways. The point is *not* that copying variables between different environments is evil and must always be avoided. Instead, it should be up to the implementer of the build system to make conscious choices about how and when to import a variable from one environment to another, making informed decisions about striking the right balance between making the build repeatable on the one hand and convenient to use on the other.

7.1. Using Values From the External Environment

The external environment variable settings that the user has in force when executing SCons are available through the normal Python os.environ dictionary. This means that you must add an import os statement to any SConscript file in which you want to use values from the user's external environment.

```
import os
```

More usefully, you can use the os.environ dictionary in your SConscript files to initialize construction environments with values from the user's external environment. See the next section, Section 7.2, for information on how to do this.

7.2. Construction Environments

It is rare that all of the software in a large, complicated system needs to be built the same way. For example, different source files may need different options enabled on the command line, or different executable programs need to be linked with different libraries. SCons accommodates these different build requirements by allowing you to create and configure multiple construction environments that control how the software is built. A construction environment is an object that has a number of associated construction variables, each with a name and a value. (A construction environment also has an attached set of Builder methods, about which we'll learn more later.)

7.2.1. Creating a Construction Environment: the Environment Function

A construction environment is created by the Environment method:

```
env = Environment()
```

By default, SCons initializes every new construction environment with a set of construction variables based on the tools that it finds on your system, plus the default set of builder methods necessary for using those tools. The construction variables are initialized with values describing the C compiler, the Fortran compiler, the linker, etc., as well as the command lines to invoke them.

When you initialize a construction environment you can set the values of the environment's construction variables to control how a program is built. For example:

```
import os

env = Environment(CC = 'gcc',
                  CCFLAGS = '-O2')
```

```
env.Program('foo.c')
```

The construction environment in this example is still initialized with the same default construction variable values, except that the user has explicitly specified use of the GNU C compiler gcc, and further specifies that the `-O2` (optimization level two) flag should be used when compiling the object file. In other words, the explicit initializations of $CC and $CCFLAGS override the default values in the newly-created construction environment. So a run from this example would look like:

```
% scons -Q
gcc -o foo.o -c -O2 foo.c
gcc -o foo foo.o
```

7.2.2. Fetching Values From a `Construction Environment`

You can fetch individual construction variables using the normal syntax for accessing individual named items in a Python dictionary:

```
env = Environment()
print "CC is:", env['CC']
```

This example `SConstruct` file doesn't build anything, but because it's actually a Python script, it will print the value of $CC for us:

```
% scons -Q
CC is: cc
scons: `.' is up to date.
```

A construction environment, however, is actually an object with associated methods, etc. If you want to have direct access to only the dictionary of construction variables, you can fetch this using the `Dictionary` method:

```
env = Environment(FOO = 'foo', BAR = 'bar')
dict = env.Dictionary()
for key in ['OBJSUFFIX', 'LIBSUFFIX', 'PROGSUFFIX']:
    print "key = %s, value = %s" % (key, dict[key])
```

This `SConstruct` file will print the specified dictionary items for us on POSIX systems as follows:

```
% scons -Q
key = OBJSUFFIX, value = .o
```

```
key = LIBSUFFIX, value = .a
key = PROGSUFFIX, value =
scons: `.' is up to date.
```

And on Windows:

```
C:\>scons -Q
key = OBJSUFFIX, value = .obj
key = LIBSUFFIX, value = .lib
key = PROGSUFFIX, value = .exe
scons: `.' is up to date.
```

If you want to loop and print the values of all of the construction variables in a construction environment, the Python code to do that in sorted order might look something like:

```
env = Environment()
for item in sorted(env.Dictionary().items()):
    print "construction variable = '%s', value = '%s'" % item
```

7.2.3. Expanding Values From a `Construction Environment`: the `subst` Method

Another way to get information from a construction environment is to use the `subst` method on a string containing `$` expansions of construction variable names. As a simple example, the example from the previous section that used `env['CC']` to fetch the value of $CC could also be written as:

```
env = Environment()
print "CC is:", env.subst('$CC')
```

One advantage of using `subst` to expand strings is that construction variables in the result get re-expanded until there are no expansions left in the string. So a simple fetch of a value like $CCCOM:

```
env = Environment(CCFLAGS = '-DFOO')
print "CCCOM is:", env['CCCOM']
```

Will print the unexpanded value of $CCCOM, showing us the construction variables that still need to be expanded:

```
% scons -Q
CCCOM is: $CC $CCFLAGS $CPPFLAGS $_CPPDEFFLAGS $_CPPINCFLAGS -c -o $TARGET $SOURC
scons: `.' is up to date.
```

Calling the `subst` method on `$CCOM`, however:

```
env = Environment(CCFLAGS = '-DFOO')
print "CCCOM is:", env.subst('$CCCOM')
```

Will recursively expand all of the construction variables prefixed with `$` (dollar signs), showing us the final output:

```
% scons -Q
CCCOM is: gcc -DFOO -c -o
scons: `.' is up to date.
```

Note that because we're not expanding this in the context of building something there are no target or source files for $TARGET and $SOURCES to expand.

7.2.4. Handling Problems With Value Expansion

If a problem occurs when expanding a construction variable, by default it is expanded to `″` (a null string), and will not cause scons to fail.

```
env = Environment()
print "value is:", env.subst( '->$MISSING<-' )
```

```
% scons -Q
value is: -><-
scons: `.' is up to date.
```

This default behaviour can be changed using the `AllowSubstExceptions` function. When a problem occurs with a variable expansion it generates an exception, and the `AllowSubstExceptions` function controls which of these exceptions are actually fatal and which are allowed to occur safely. By default, `NameError` and `IndexError` are the two exceptions that are allowed to occur: so instead of causing scons to fail, these are caught, the variable expanded to `″` and scons execution continues. To require that all construction variable names exist, and that indexes out of range are not allowed, call `AllowSubstExceptions` with no extra arguments.

```
AllowSubstExceptions()
env = Environment()
print "value is:", env.subst( '->$MISSING<-' )
```

```
% scons -Q
```

```
value is:
scons: *** NameError 'MISSING' trying to evaluate '$MISSING'
File "/home/my/project/SConstruct", line 3, in <module>
```

This can also be used to allow other exceptions that might occur, most usefully with the `${...}` construction variable syntax. For example, this would allow zero-division to occur in a variable expansion in addition to the default exceptions allowed

```
AllowSubstExceptions(IndexError, NameError, ZeroDivisionError)
env = Environment()
print "value is:", env.subst( '->${1 / 0}<-' )
```

```
% scons -Q
value is: -><-
scons: `.' is up to date.
```

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

7.2.5. Controlling the Default `Construction Environment`: the `DefaultEnvironment` Function

All of the `Builder` functions that we've introduced so far, like `Program` and `Library`, actually use a default `construction environment` that contains settings for the various compilers and other tools that SCons configures by default, or otherwise knows about and has discovered on your system. The goal of the default construction environment is to make many configurations to "just work" to build software using readily available tools with a minimum of configuration changes.

You can, however, control the settings in the default construction environment by using the `DefaultEnvironment` function to initialize various settings:

```
DefaultEnvironment(CC = '/usr/local/bin/gcc')
```

When configured as above, all calls to the `Program` or `Object` Builder will build object files with the `/usr/local/bin/gcc` compiler.

Note that the `DefaultEnvironment` function returns the initialized default construction environment object, which can then be manipulated like any other construction environment. So the following would be equivalent to the previous example, setting the $CC variable to `/usr/local/bin/gcc` but as a separate step after the default construction environment has been initialized:

```
env = DefaultEnvironment()
env['CC'] = '/usr/local/bin/gcc'
```

One very common use of the `DefaultEnvironment` function is to speed up SCons initialization. As part of trying to make most default configurations "just work," SCons will actually search the local system for installed compilers and other utilities. This search can take time, especially on systems with slow or networked file systems. If you know which compiler(s) and/or other utilities you want to configure, you can control the search that SCons performs by specifying some specific tool modules with which to initialize the default construction environment:

```
env = DefaultEnvironment(tools = ['gcc', 'gnulink'],
                         CC = '/usr/local/bin/gcc')
```

So the above example would tell SCons to explicitly configure the default environment to use its normal GNU Compiler and GNU Linker settings (without having to search for them, or any other utilities for that matter), and specifically to use the compiler found at `/usr/local/bin/gcc`.

7.2.6. Multiple `Construction Environments`

The real advantage of construction environments is that you can create as many different construction environments as you need, each tailored to a different way to build some piece of software or other file. If, for example, we need to build one program with the `-O2` flag and another with the `-g` (debug) flag, we would do this like so:

```
opt = Environment(CCFLAGS = '-O2')
dbg = Environment(CCFLAGS = '-g')

opt.Program('foo', 'foo.c')

dbg.Program('bar', 'bar.c')
```

```
% scons -Q
cc -o bar.o -c -g bar.c
cc -o bar bar.o
cc -o foo.o -c -O2 foo.c
cc -o foo foo.o
```

We can even use multiple construction environments to build multiple versions of a single program. If you do this by simply trying to use the `Program` builder with both environments, though, like this:

```
opt = Environment(CCFLAGS = '-O2')
dbg = Environment(CCFLAGS = '-g')

opt.Program('foo', 'foo.c')

dbg.Program('foo', 'foo.c')
```

Then SCons generates the following error:

```
% scons -Q

scons: *** Two environments with different actions were specified for the same t
File "/home/my/project/SConstruct", line 6, in <module>
```

This is because the two `Program` calls have each implicitly told SCons to generate an object file named `foo.o`, one with a $CCFLAGS value of `-O2` and one with a $CCFLAGS value of `-g`. SCons can't just decide that one of them should take precedence over the other, so it generates the error. To avoid this problem, we must explicitly specify that each environment compile `foo.c` to a separately-named object file using the `Object` builder, like so:

```
opt = Environment(CCFLAGS = '-O2')
dbg = Environment(CCFLAGS = '-g')

o = opt.Object('foo-opt', 'foo.c')
opt.Program(o)

d = dbg.Object('foo-dbg', 'foo.c')
dbg.Program(d)
```

Notice that each call to the `Object` builder returns a value, an internal SCons object that represents the object file that will be built. We then use that object as input to the `Program` builder. This avoids having to specify explicitly the object file name in multiple places, and makes for a compact, readable `SConstruct` file. Our SCons output then looks like:

```
% scons -Q
cc -o foo-dbg.o -c -g foo.c
cc -o foo-dbg foo-dbg.o
cc -o foo-opt.o -c -O2 foo.c
cc -o foo-opt foo-opt.o
```

7.2.7. Making Copies of `Construction Environments`: the `Clone` Method

Sometimes you want more than one construction environment to share the same values for one or more variables. Rather than always having to repeat all of the common variables when you create each construction environment, you can use the `Clone` method to create a copy of a construction environment.

Like the `Environment` call that creates a construction environment, the `Clone` method takes `construction variable` assignments, which will override the values in the copied construction environment. For example, suppose we want to use gcc to create three versions of a program, one optimized, one debug, and one with neither. We could do this by creating a "base" construction environment that sets $CC to gcc, and then creating two copies, one which sets $CCFLAGS for optimization and the other which sets $CCFLAGS for debugging:

```
env = Environment(CC = 'gcc')
opt = env.Clone(CCFLAGS = '-O2')
dbg = env.Clone(CCFLAGS = '-g')

env.Program('foo', 'foo.c')

o = opt.Object('foo-opt', 'foo.c')
opt.Program(o)

d = dbg.Object('foo-dbg', 'foo.c')
dbg.Program(d)
```

Then our output would look like:

```
% scons -Q
gcc -o foo.o -c foo.c
gcc -o foo foo.o
gcc -o foo-dbg.o -c -g foo.c
gcc -o foo-dbg foo-dbg.o
gcc -o foo-opt.o -c -O2 foo.c
gcc -o foo-opt foo-opt.o
```

7.2.8. Replacing Values: the `Replace` Method

You can replace existing construction variable values using the `Replace` method:

```
env = Environment(CCFLAGS = '-DDEFINE1')
env.Replace(CCFLAGS = '-DDEFINE2')
env.Program('foo.c')
```

The replacing value (`-DDEFINE2` in the above example) completely replaces the value in the construction environment:

```
% scons -Q
cc -o foo.o -c -DDEFINE2 foo.c
cc -o foo foo.o
```

You can safely call `Replace` for construction variables that don't exist in the construction environment:

```
env = Environment()
env.Replace(NEW_VARIABLE = 'xyzzy')
print "NEW_VARIABLE =", env['NEW_VARIABLE']
```

In this case, the construction variable simply gets added to the construction environment:

```
% scons -Q
NEW_VARIABLE = xyzzy
scons: `.' is up to date.
```

Because the variables aren't expanded until the construction environment is actually used to build the targets, and because SCons function and method calls are order-independent, the last replacement "wins" and is used to build all targets, regardless of the order in which the calls to Replace() are interspersed with calls to builder methods:

```
env = Environment(CCFLAGS = '-DDEFINE1')
print "CCFLAGS =", env['CCFLAGS']
env.Program('foo.c')

env.Replace(CCFLAGS = '-DDEFINE2')
print "CCFLAGS =", env['CCFLAGS']
env.Program('bar.c')
```

The timing of when the replacement actually occurs relative to when the targets get built becomes apparent if we run `scons` without the `-Q` option:

```
% scons
scons: Reading SConscript files ...
CCFLAGS = -DDEFINE1
CCFLAGS = -DDEFINE2
scons: done reading SConscript files.
scons: Building targets ...
cc -o bar.o -c -DDEFINE2 bar.c
cc -o bar bar.o
cc -o foo.o -c -DDEFINE2 foo.c
cc -o foo foo.o
scons: done building targets.
```

Because the replacement occurs while the SConscript files are being read, the $CCFLAGS variable has already been set to -DDEFINE2 by the time the foo.o target is built, even though the call to the Replace method does not occur until later in the SConscript file.

7.2.9. Setting Values Only If They're Not Already Defined: the SetDefault Method

Sometimes it's useful to be able to specify that a construction variable should be set to a value only if the construction environment does not already have that variable defined You can do this with the SetDefault method, which behaves similarly to the set_default method of Python dictionary objects:

```
env.SetDefault(SPECIAL_FLAG = '-extra-option')
```

This is especially useful when writing your own Tool modules to apply variables to construction environments.

7.2.10. Appending to the End of Values: the Append Method

You can append a value to an existing construction variable using the Append method:

```
env = Environment(CCFLAGS = ['-DMY_VALUE'])
env.Append(CCFLAGS = ['-DLAST'])
env.Program('foo.c')
```

SCons then supplies both the -DMY_VALUE and -DLAST flags when compiling the object file:

```
% scons -Q
cc -o foo.o -c -DMY_VALUE -DLAST foo.c
cc -o foo foo.o
```

If the construction variable doesn't already exist, the Append method will create it:

```
env = Environment()
env.Append(NEW_VARIABLE = 'added')
print "NEW_VARIABLE =", env['NEW_VARIABLE']
```

Which yields:

```
% scons -Q
NEW_VARIABLE = added
scons: `.' is up to date.
```

Note that the `Append` function tries to be "smart" about how the new value is appended to the old value. If both are strings, the previous and new strings are simply concatenated. Similarly, if both are lists, the lists are concatenated. If, however, one is a string and the other is a list, the string is added as a new element to the list.

7.2.11. Appending Unique Values: the `AppendUnique` Method

Some times it's useful to add a new value only if the existing construction variable doesn't already contain the value. This can be done using the `AppendUnique` method:

```
env.AppendUnique(CCFLAGS=['-g'])
```

In the above example, the `-g` would be added only if the $CCFLAGS variable does not already contain a `-g` value.

7.2.12. Appending to the Beginning of Values: the `Prepend` Method

You can append a value to the beginning of an existing construction variable using the `Prepend` method:

```
env = Environment(CCFLAGS = ['-DMY_VALUE'])
env.Prepend(CCFLAGS = ['-DFIRST'])
env.Program('foo.c')
```

SCons then supplies both the `-DFIRST` and `-DMY_VALUE` flags when compiling the object file:

```
% scons -Q
cc -o foo.o -c -DFIRST -DMY_VALUE foo.c
cc -o foo foo.o
```

If the construction variable doesn't already exist, the `Prepend` method will create it:

```
env = Environment()
env.Prepend(NEW_VARIABLE = 'added')
print "NEW_VARIABLE =", env['NEW_VARIABLE']
```

Which yields:

```
% scons -Q
NEW_VARIABLE = added
scons: `.' is up to date.
```

Like the `Append` function, the `Prepend` function tries to be "smart" about how the new value is appended to the old value. If both are strings, the previous and new strings are simply concatenated. Similarly, if both are lists, the lists are concatenated. If, however, one is a string and the other is a list, the string is added as a new element to the list.

7.2.13. Prepending Unique Values: the `PrependUnique` Method

Some times it's useful to add a new value to the beginning of a construction variable only if the existing value doesn't already contain the to-be-added value. This can be done using the `PrependUnique` method:

```
env.PrependUnique(CCFLAGS=['-g'])
```

In the above example, the `-g` would be added only if the $CCFLAGS variable does not already contain a `-g` value.

7.3. Controlling the Execution Environment for Issued Commands

When SCons builds a target file, it does not execute the commands with the same external environment that you used to execute SCons. Instead, it uses the dictionary stored in the $ENV construction variable as the external environment for executing commands.

The most important ramification of this behavior is that the `PATH` environment variable, which controls where the operating system will look for commands and utilities, is not the same as in the external environment from which you called SCons. This means that SCons will not, by default, necessarily find all of the tools that you can execute from the command line.

The default value of the `PATH` environment variable on a POSIX system is `/usr/local/bin:/bin:/usr/bin`. The default value of the `PATH` environment variable on a Windows system comes from the Windows registry value for the command interpreter. If you want to execute any commands--compilers, linkers, etc.--that are not in these default locations, you need to set the `PATH` value in the $ENV dictionary in your construction environment.

The simplest way to do this is to initialize explicitly the value when you create the construction environment; this is one way to do that:

```
path = ['/usr/local/bin', '/bin', '/usr/bin']
env = Environment(ENV = {'PATH' : path})
```

Assign a dictionary to the $ENV construction variable in this way completely resets the external environment so that the only variable that will be set when external commands are executed will be the PATH value. If you want to use the rest of the values in $ENV and only set the value of PATH, the most straightforward way is probably:

```
env['ENV']['PATH'] = ['/usr/local/bin', '/bin', '/usr/bin']
```

Note that SCons does allow you to define the directories in the PATH in a string, separated by the pathname-separator character for your system (':' on POSIX systems, ';' on Windows):

```
env['ENV']['PATH'] = '/usr/local/bin:/bin:/usr/bin'
```

But doing so makes your SConscript file less portable, (although in this case that may not be a huge concern since the directories you list are likley system-specific, anyway).

7.3.1. Propagating PATH From the External Environment

You may want to propagate the external PATH to the execution environment for commands. You do this by initializing the PATH variable with the PATH value from the os.environ dictionary, which is Python's way of letting you get at the external environment:

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

Alternatively, you may find it easier to just propagate the entire external environment to the execution environment for commands. This is simpler to code than explicity selecting the PATH value:

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

Either of these will guarantee that SCons will be able to execute any command that you can execute from the command line. The drawback is that the build can behave differently if it's run by people with different PATH values in their environment--for example, if both the /bin and /usr/local/bin directories have different cc commands, then which one will be used to compile programs will depend on which directory is listed first in the user's PATH variable.

7.3.2. Adding to PATH Values in the Execution Environment

One of the most common requirements for manipulating a variable in the execution environment is to add one or more custom directories to a search like the $PATH variable on Linux or POSIX systems, or the %PATH% variable on Windows, so that a locally-installed compiler or other utility can be found when SCons tries to execute it to update a target. SCons provides `PrependENVPath` and `AppendENVPath` functions to make adding things to execution variables convenient. You call these functions by specifying the variable to which you want the value added, and then value itself. So to add some `/usr/local` directories to the $PATH and $LIB variables, you might:

```
env = Environment(ENV = os.environ)
env.PrependENVPath('PATH', '/usr/local/bin')
env.AppendENVPath('LIB', '/usr/local/lib')
```

Note that the added values are strings, and if you want to add multiple directories to a variable like $PATH, you must include the path separate character (`:` on Linux or POSIX, `;` on Windows) in the string.

Chapter 8. Automatically Putting Command-line Options into their Construction Variables

This chapter describes the `MergeFlags`, `ParseFlags`, and `ParseConfig` methods of a `construction environment`.

8.1. Merging Options into the Environment: the `MergeFlags` Function

SCons construction environments have a `MergeFlags` method that merges a dictionary of values into the construction environment. `MergeFlags` treats each value in the dictionary as a list of options such as one might pass to a command (such as a compiler or linker). `MergeFlags` will not duplicate an option if it already exists in the construction environment variable.

`MergeFlags` tries to be intelligent about merging options. When merging options to any variable whose name ends in `PATH`, `MergeFlags` keeps the leftmost occurrence of the option, because in typical lists of directory paths, the first occurrence "wins." When merging options to any other variable name, `MergeFlags` keeps the rightmost occurrence of the option, because in a list of typical command-line options, the last occurrence "wins."

```
env = Environment()
env.Append(CCFLAGS = '-option -O3 -O1')
flags = { 'CCFLAGS' : '-whatever -O3' }
env.MergeFlags(flags)
print env['CCFLAGS']
```

```
% scons -Q
['-option', '-O1', '-whatever', '-O3']
scons: `.' is up to date.
```

Note that the default value for $CCFLAGS is an internal SCons object which automatically converts the options we specified as a string into a list.

```
env = Environment()
env.Append(CPPPATH = ['/include', '/usr/local/include', '/usr/include'])
flags = { 'CPPPATH' : ['/usr/opt/include', '/usr/local/include'] }
env.MergeFlags(flags)
print env['CPPPATH']
```

```
% scons -Q
['/include', '/usr/local/include', '/usr/include', '/usr/opt/include']
```

```
scons: `.' is up to date.
```

Note that the default value for $CPPPATH is a normal Python list, so we must specify its values as a list in the dictionary we pass to the `MergeFlags` function.

If `MergeFlags` is passed anything other than a dictionary, it calls the `ParseFlags` method to convert it into a dictionary.

```
env = Environment()
env.Append(CCFLAGS = '-option -O3 -O1')
env.Append(CPPPATH = ['/include', '/usr/local/include', '/usr/include'])
env.MergeFlags('-whatever -I/usr/opt/include -O3 -I/usr/local/include')
print env['CCFLAGS']
print env['CPPPATH']
```

```
% scons -Q
['-option', '-O1', '-whatever', '-O3']
['/include', '/usr/local/include', '/usr/include', '/usr/opt/include']
scons: `.' is up to date.
```

In the combined example above, `ParseFlags` has sorted the options into their corresponding variables and returned a dictionary for `MergeFlags` to apply to the construction variables in the specified construction environment.

8.2. Separating Compile Arguments into their Variables: the `ParseFlags` Function

SCons has a bewildering array of construction variables for different types of options when building programs. Sometimes you may not know exactly which variable should be used for a particular option.

SCons construction environments have a `ParseFlags` method that takes a set of typical command-line options and distrbutes them into the appropriate construction variables. Historically, it was created to support the `ParseConfig` method, so it focuses on options used by the GNU Compiler Collection (GCC) for the C and C++ toolchains.

`ParseFlags` returns a dictionary containing the options distributed into their respective construction variables. Normally, this dictionary would be passed to `MergeFlags` to merge the options into a `construction environment`, but the dictionary can be edited if desired to provide additional functionality. (Note that if the flags are not going to be edited, calling `MergeFlags` with the options directly will avoid an additional step.)

```
env = Environment()
d = env.ParseFlags("-I/opt/include -L/opt/lib -lfoo")
for k,v in sorted(d.items()):
    if v:
        print k, v
env.MergeFlags(d)
env.Program('f1.c')
```

```
% scons -Q
CPPPATH ['/opt/include']
LIBPATH ['/opt/lib']
LIBS ['foo']
cc -o f1.o -c -I/opt/include f1.c
cc -o f1 f1.o -L/opt/lib -lfoo
```

Note that if the options are limited to generic types like those above, they will be correctly translated for other platform types:

```
C:\>scons -Q
CPPPATH ['/opt/include']
LIBPATH ['/opt/lib']
LIBS ['foo']
cl /Fof1.obj /c f1.c /nologo /I\opt\include
link /nologo /OUT:f1.exe /LIBPATH:\opt\lib foo.lib f1.obj
embedManifestExeCheck(target, source, env)
```

Since the assumption is that the flags are used for the GCC toolchain, unrecognized flags are placed in $CCFLAGS so they will be used for both C and C++ compiles:

```
env = Environment()
d = env.ParseFlags("-whatever")
for k,v in sorted(d.items()):
    if v:
        print k, v
env.MergeFlags(d)
env.Program('f1.c')
```

```
% scons -Q
CCFLAGS -whatever
cc -o f1.o -c -whatever f1.c
cc -o f1 f1.o
```

`ParseFlags` will also accept a (recursive) list of strings as input; the list is flattened before the strings are processed:

```
env = Environment()
```

```
d = env.ParseFlags(["-I/opt/include", ["-L/opt/lib", "-lfoo"]])
for k,v in sorted(d.items()):
    if v:
        print k, v
env.MergeFlags(d)
env.Program('f1.c')
```

```
% scons -Q
CPPPATH ['/opt/include']
LIBPATH ['/opt/lib']
LIBS ['foo']
cc -o f1.o -c -I/opt/include f1.c
cc -o f1 f1.o -L/opt/lib -lfoo
```

If a string begins with a "!" (an exclamation mark, often called a bang), the string is passed to the shell for execution. The output of the command is then parsed:

```
env = Environment()
d = env.ParseFlags(["!echo -I/opt/include", "!echo -L/opt/lib", "-lfoo"])
for k,v in sorted(d.items()):
    if v:
        print k, v
env.MergeFlags(d)
env.Program('f1.c')
```

```
% scons -Q
CPPPATH ['/opt/include']
LIBPATH ['/opt/lib']
LIBS ['foo']
cc -o f1.o -c -I/opt/include f1.c
cc -o f1 f1.o -L/opt/lib -lfoo
```

`ParseFlags` is regularly updated for new options; consult the man page for details about those currently recognized.

8.3. Finding Installed Library Information: the `ParseConfig` Function

Configuring the right options to build programs to work with libraries--especially shared libraries--that are available on POSIX systems can be very complicated. To help this situation, various utilies with names that end in `config` return the command-line options for the GNU Compiler Collection (GCC) that are needed to use these libraries; for example, the command-line options to use a library named `lib` would be found by calling a utility named `lib-config`.

A more recent convention is that these options are available from the generic `pkg-config` program, which has common framework, error handling, and the like, so that all the package creator has to do is provide the set of strings for his particular package.

SCons construction environments have a `ParseConfig` method that executes a `*config` utility (either `pkg-config` or a more specific utility) and configures the appropriate construction variables in the environment based on the command-line options returned by the specified command.

```
env = Environment()
env['CPPPATH'] = ['/lib/compat']
env.ParseConfig("pkg-config x11 --cflags --libs")
print env['CPPPATH']
```

SCons will execute the specified command string, parse the resultant flags, and add the flags to the appropriate environment variables.

```
% scons -Q
['/lib/compat', '/usr/X11/include']
scons: `.' is up to date.
```

In the example above, SCons has added the include directory to `CPPPATH`. (Depending upon what other flags are emitted by the `pkg-config` command, other variables may have been extended as well.)

Note that the options are merged with existing options using the `MergeFlags` method, so that each option only occurs once in the construction variable:

```
env = Environment()
env.ParseConfig("pkg-config x11 --cflags --libs")
env.ParseConfig("pkg-config x11 --cflags --libs")
print env['CPPPATH']
```

```
% scons -Q
['/usr/X11/include']
scons: `.' is up to date.
```

Chapter 9. Controlling Build Output

A key aspect of creating a usable build configuration is providing good output from the build so its users can readily understand what the build is doing and get information about how to control the build. SCons provides several ways of controlling output from the build configuration to help make the build more useful and understandable.

9.1. Providing Build Help: the `Help` Function

It's often very useful to be able to give users some help that describes the specific targets, build options, etc., that can be used for your build. SCons provides the `Help` function to allow you to specify this help text:

```
Help("""
Type: 'scons program' to build the production program,
      'scons debug' to build the debug version.
""")
```

(Note the above use of the Python triple-quote syntax, which comes in very handy for specifying multi-line strings like help text.)

When the `SConstruct` or `SConscript` files contain such a call to the `Help` function, the specified help text will be displayed in response to the SCons `-h` option:

```
% scons -h
scons: Reading SConscript files ...
scons: done reading SConscript files.

Type: 'scons program' to build the production program,
      'scons debug' to build the debug version.

Use scons -H for help about command-line options.
```

The `SConscript` files may contain multiple calls to the `Help` function, in which case the specified text(s) will be concatenated when displayed. This allows you to split up the help text across multiple `SConscript` files. In this situation, the order in which the `SConscript` files are called will determine the order in which the `Help` functions are called, which will determine the order in which the various bits of text will get concatenated.

Another use would be to make the help text conditional on some variable. For example, suppose you only want to display a line about building a Windows-only version of a program when actually run on Windows. The following `SConstruct` file:

```
env = Environment()

Help("\nType: 'scons program' to build the production program.\n")

if env['PLATFORM'] == 'win32':
    Help("\nType: 'scons windebug' to build the Windows debug version.\n")
```

Will display the complete help text on Windows:

```
C:\>scons -h
scons: Reading SConscript files ...
scons: done reading SConscript files.

Type: 'scons program' to build the production program.

Type: 'scons windebug' to build the Windows debug version.

Use scons -H for help about command-line options.
```

But only show the relevant option on a Linux or UNIX system:

```
% scons -h
scons: Reading SConscript files ...
scons: done reading SConscript files.

Type: 'scons program' to build the production program.

Use scons -H for help about command-line options.
```

If there is no `Help` text in the `SConstruct` or `SConscript` files, SCons will revert to displaying its standard list that describes the SCons command-line options. This list is also always displayed whenever the `-H` option is used.

9.2. Controlling How SCons Prints Build Commands: the $*COMSTR Variables

Sometimes the commands executed to compile object files or link programs (or build other targets) can get very long, long enough to make it difficult for users to distinguish error messages or other important build output from the commands themselves. All of the default $*COM variables that specify the command lines used to build various types of target files have a corresponding $*COMSTR variable that can be set to an alternative string that will be displayed when the target is built.

For example, suppose you want to have SCons display a "Compiling" message whenever it's compiling an object file, and a "Linking" when it's linking an executable. You could write a SConstruct file that looks like:

```
env = Environment(CCCOMSTR = "Compiling $TARGET",
                  LINKCOMSTR = "Linking $TARGET")
env.Program('foo.c')
```

Which would then yield the output:

```
% scons -Q
Compiling foo.o
Linking foo
```

SCons performs complete variable substitution on $*COMSTR variables, so they have access to all of the standard variables like $TARGET $SOURCES, etc., as well as any construction variables that happen to be configured in the construction environment used to build a specific target.

Of course, sometimes it's still important to be able to see the exact command that SCons will execute to build a target. For example, you may simply need to verify that SCons is configured to supply the right options to the compiler, or a developer may want to cut-and-paste a compile command to add a few options for a custom test.

One common way to give users control over whether or not SCons should print the actual command line or a short, configured summary is to add support for a VERBOSE command-line variable to your SConstruct file. A simple configuration for this might look like:

```
env = Environment()
if ARGUMENTS.get('VERBOSE') != "1":
    env['CCCOMSTR'] = "Compiling $TARGET"
    env['LINKCOMSTR'] = "Linking $TARGET"
env.Program('foo.c')
```

By only setting the appropriate $*COMSTR variables if the user specifies VERBOSE=1 on the command line, the user has control over how SCons displays these particular command lines:

```
% scons -Q
Compiling foo.o
Linking foo
% scons -Q -c
Removed foo.o
Removed foo
% scons -Q VERBOSE=1
cc -o foo.o -c foo.c
cc -o foo foo.o
```

9.3. Providing Build Progress Output: the `Progress` Function

Another aspect of providing good build output is to give the user feedback about what SCons is doing even when nothing is being built at the moment. This can be especially true for large builds when most of the targets are already up-to-date. Because SCons can take a long time making absolutely sure that every target is, in fact, up-to-date with respect to a lot of dependency files, it can be easy for users to mistakenly conclude that SCons is hung or that there is some other problem with the build.

One way to deal with this perception is to configure SCons to print something to let the user know what it's "thinking about." The `Progress` function allows you to specify a string that will be printed for every file that SCons is "considering" while it is traversing the dependency graph to decide what targets are or are not up-to-date.

```
Progress('Evaluating $TARGET\n')
Program('f1.c')
Program('f2.c')
```

Note that the `Progress` function does not arrange for a newline to be printed automatically at the end of the string (as does the Python `print` statement), and we must specify the `\n` that we want printed at the end of the configured string. This configuration, then, will have SCons print that it is `Evaluating` each file that it encounters in turn as it traverses the dependency graph:

```
% scons -Q
Evaluating SConstruct
Evaluating f1.c
Evaluating f1.o
cc -o f1.o -c f1.c
Evaluating f1
cc -o f1 f1.o
Evaluating f2.c
Evaluating f2.o
cc -o f2.o -c f2.c
Evaluating f2
cc -o f2 f2.o
Evaluating .
```

Of course, normally you don't want to add all of these additional lines to your build output, as that can make it difficult for the user to find errors or other important messages. A more useful way to display this progress might be to have the file names printed directly to the user's screen, not to the same standard

output stream where build output is printed, and to use a carriage return character (`\r`) so that each file name gets re-printed on the same line. Such a configuration would look like:

```
Progress('$TARGET\r',
         file=open('/dev/tty', 'w'),
         overwrite=True)
Program('f1.c')
Program('f2.c')
```

Note that we also specified the `overwrite=True` argument to the `Progress` function, which causes SCons to "wipe out" the previous string with space characters before printing the next `Progress` string. Without the `overwrite=True` argument, a shorter file name would not overwrite all of the charactes in a longer file name that precedes it, making it difficult to tell what the actual file name is on the output. Also note that we opened up the `/dev/tty` file for direct access (on POSIX) to the user's screen. On Windows, the equivalent would be to open the `con:` file name.

Also, it's important to know that although you can use `$TARGET` to substitute the name of the node in the string, the `Progress` function does *not* perform general variable substitution (because there's not necessarily a construction environment involved in evaluating a node like a source file, for example).

You can also specify a list of strings to the `Progress` function, in which case SCons will display each string in turn. This can be used to implement a "spinner" by having SCons cycle through a sequence of strings:

```
Progress(['-\r', '\\\r', '|\r', '/\r'], interval=5)
Program('f1.c')
Program('f2.c')
```

Note that here we have also used the `interval=` keyword argument to have SCons only print a new "spinner" string once every five evaluated nodes. Using an `interval=` count, even with strings that use `$TARGET` like our examples above, can be a good way to lessen the work that SCons expends printing `Progress` strings, while still giving the user feedback that indicates SCons is still working on evaluating the build.

Lastly, you can have direct control over how to print each evaluated node by passing a Python function (or other Python callable) to the `Progress` function. Your function will be called for each evaluated node, allowing you to implement more sophisticated logic like adding a counter:

```
screen = open('/dev/tty', 'w')
count = 0
def progress_function(node)
    count += 1
    screen.write('Node %4d: %s\r' % (count, node))

Progress(progress_function)
```

Of course, if you choose, you could completely ignore the `node` argument to the function, and just print a count, or anything else you wish.

(Note that there's an obvious follow-on question here: how would you find the total number of nodes that *will be* evaluated so you can tell the user how close the build is to finishing? Unfortunately, in the general case, there isn't a good way to do that, short of having SCons evaluate its dependency graph twice, first to count the total and the second time to actually build the targets. This would be necessary because you can't know in advance which target(s) the user actually requested to be built. The entire build may consist of thousands of Nodes, for example, but maybe the user specifically requested that only a single object file be built.)

9.4. Printing Detailed Build Status: the `GetBuildFailures` Function

SCons, like most build tools, returns zero status to the shell on success and nonzero status on failure. Sometimes it's useful to give more information about the build status at the end of the run, for instance to print an informative message, send an email, or page the poor slob who broke the build.

SCons provides a `GetBuildFailures` method that you can use in a python `atexit` function to get a list of objects describing the actions that failed while attempting to build targets. There can be more than one if you're using `-j`. Here's a simple 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)
```

The `atexit.register` call registers `print_build_failures` as an `atexit` callback, to be called before SCons exits. When that function is called, it calls `GetBuildFailures` to fetch the list of failed objects. See the man page for the detailed contents of the returned objects; some of the more useful attributes are `.node`, `.errstr`, `.filename`, and `.command`. The `filename` is not necessarily the same file as the `node`; the `node` is the target that was being built when the error occurred, while the `filename`is the file or dir that actually caused the error. Note: only call `GetBuildFailures` at the end of the build; calling it at any other time is undefined.

Here is a more complete example showing how to turn each element of `GetBuildFailures` into a string:

```
# Make the build fail if we pass fail=1 on the command line
if ARGUMENTS.get('fail', 0):
```

```
    Command('target', 'source', ['/bin/false'])

def bf_to_str(bf):
    """Convert an element of GetBuildFailures() to a string
    in a useful way."""
    import SCons.Errors
    if bf is None: # unknown targets product None in list
        return '(unknown tgt)'
    elif isinstance(bf, SCons.Errors.StopError):
        return str(bf)
    elif bf.node:
        return str(bf.node) + ': ' + bf.errstr
    elif bf.filename:
        return bf.filename + ': ' + bf.errstr
    return 'unknown failure: ' + bf.errstr
import atexit

def build_status():
    """Convert the build status to a 2-tuple, (status, msg)."""
    from SCons.Script import GetBuildFailures
    bf = GetBuildFailures()
    if bf:
        # bf is normally a list of build failures; if an element is None,
        # it's because of a target that scons doesn't know anything about.
        status = 'failed'
        failures_message = "\n".join(["Failed building %s" % bf_to_str(x)
                            for x in bf if x is not None])
    else:
        # if bf is None, the build completed successfully.
        status = 'ok'
        failures_message = ''
    return (status, failures_message)

def display_build_status():
    """Display the build status.  Called by atexit.
    Here you could do all kinds of complicated things."""
    status, failures_message = build_status()
    if status == 'failed':
       print "FAILED!!!!"  # could display alert, ring bell, etc.
    elif status == 'ok':
       print "Build succeeded."
    print failures_message

atexit.register(display_build_status)
```

When this runs, you'll see the appropriate output:

```
% scons -Q
scons: `.' is up to date.
Build succeeded.
% scons -Q fail=1
```

```
scons: *** [target] Source 'source' not found, needed by target 'target'.
FAILED!!!!
Failed building target: Source 'source' not found, needed by target 'target'.
```

Chapter 10. Controlling a Build From the Command Line

SCons provides a number of ways for the writer of the `SConscript` files to give the users who will run SCons a great deal of control over the build execution. The arguments that the user can specify on the command line are broken down into three types:

Options

Command-line options always begin with one or two - (hyphen) characters. SCons provides ways for you to examine and set options values from within your `SConscript` files, as well as the ability to define your own custom options. See Section 10.1, below.

Variables

Any command-line argument containing an = (equal sign) is considered a variable setting with the form `variable=value`. SCons provides direct access to all of the command-line variable settings, the ability to apply command-line variable settings to construction environments, and functions for configuring specific types of variables (Boolean values, path names, etc.) with automatic validation of the user's specified values. See Section 10.2, below.

Targets

Any command-line argument that is not an option or a variable setting (does not begin with a hyphen and does not contain an equal sign) is considered a target that the user (presumably) wants SCons to build. A list of Node objects representing the target or targets to build. SCons provides access to the list of specified targets, as well as ways to set the default list of targets from within the `SConscript` files. See Section 10.3, below.

10.1. Command-Line Options

SCons has many *command-line options* that control its behavior. A SCons *command-line option* always begins with one or two - (hyphen) characters.

10.1.1. Not Having to Specify Command-Line Options Each Time: the SCONSFLAGS Environment Variable

Users may find themselves supplying the same command-line options every time they run SCons. For example, you might find it saves time to specify a value of `-j 2` to have SCons run up to two build commands in parallel. To avoid having to type `-j 2` by hand every time, you can set the external environment variable `SCONSFLAGS` to a string containing command-line options that you want SCons to use.

If, for example, you're using a POSIX shell that's compatible with the Bourne shell, and you always want SCons to use the `-Q` option, you can set the `SCONSFLAGS` environment as follows:

```
% scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
    ... [build output] ...
scons: done building targets.
% export SCONSFLAGS="-Q"
% scons
    ... [build output] ...
```

Users of csh-style shells on POSIX systems can set the `SCONSFLAGS` environment as follows:

```
$ setenv SCONSFLAGS "-Q"
```

Windows users may typically want to set the `SCONSFLAGS` in the appropriate tab of the `System Properties` window.

10.1.2. Getting Values Set by Command-Line Options: the `GetOption` Function

SCons provides the `GetOption` function to get the values set by the various command-line options. One common use of this is to check whether or not the `-h` or `--help` option has been specified. Normally, SCons does not print its help text until after it has read all of the `SConscript` files, because it's possible that help text has been added by some subsidiary `SConscript` file deep in the source tree hierarchy. Of course, reading all of the `SConscript` files takes extra time.

If you know that your configuration does not define any additional help text in subsidiary `SConscript` files, you can speed up the command-line help available to users by using the `GetOption` function to load the subsidiary `SConscript` files only if the the user has *not* specified the `-h` or `--help` option, like so:

```
if not GetOption('help'):
    SConscript('src/SConscript', export='env')
```

In general, the string that you pass to the `GetOption` function to fetch the value of a command-line option setting is the same as the "most common" long option name (beginning with two hyphen characters), although there are some exceptions. The list of SCons command-line options and the `GetOption` strings for fetching them, are available in the Section 10.1.4 section, below.

10.1.3. Setting Values of Command-Line Options: the `SetOption` Function

You can also set the values of SCons command-line options from within the `SConscript` files by using the `SetOption` function. The strings that you use to set the values of SCons command-line options are available in the Section 10.1.4 section, below.

One use of the `SetOption` function is to specify a value for the `-j` or `--jobs` option, so that users get the improved performance of a parallel build without having to specify the option by hand. A complicating factor is that a good value for the `-j` option is somewhat system-dependent. One rough guideline is that the more processors your system has, the higher you want to set the `-j` value, in order to take advantage of the number of CPUs.

For example, suppose the administrators of your development systems have standardized on setting a `NUM_CPU` environment variable to the number of processors on each system. A little bit of Python code to access the environment variable and the `SetOption` function provide the right level of flexibility:

```
import os
num_cpu = int(os.environ.get('NUM_CPU', 2))
SetOption('num_jobs', num_cpu)
print "running with -j", GetOption('num_jobs')
```

The above snippet of code sets the value of the `--jobs` option to the value specified in the `$NUM_CPU` environment variable. (This is one of the exception cases where the string is spelled differently from the from command-line option. The string for fetching or setting the `--jobs` value is `num_jobs` for historical reasons.) The code in this example prints the `num_jobs` value for illustrative purposes. It uses a default value of `2` to provide some minimal parallelism even on single-processor systems:

```
% scons -Q
running with -j 2
scons: `.' is up to date.
```

But if the `$NUM_CPU` environment variable is set, then we use that for the default number of jobs:

```
% export NUM_CPU="4"
% scons -Q
running with -j 4
scons: `.' is up to date.
```

But any explicit `-j` or `--jobs` value the user specifies an the command line is used first, regardless of whether or not the `$NUM_CPU` environment variable is set:

```
% scons -Q -j 7
running with -j 7
```

```
scons: `.' is up to date.
% export NUM_CPU="4"
% scons -Q -j 3
running with -j 3
scons: `.' is up to date.
```

10.1.4. Strings for Getting or Setting Values of SCons Command-Line Options

The strings that you can pass to the `GetOption` and `SetOption` functions usually correspond to the first long-form option name (beginning with two hyphen characters: `--`), after replacing any remaining hyphen characters with underscores.

The full list of strings and the variables they correspond to is as follows:

String for `GetOption` and `SetOption`	Command-Line Option(s)
cache_debug	--cache-debug
cache_disable	--cache-disable
cache_force	--cache-force
cache_show	--cache-show
clean	-c, --clean, --remove
config	--config
directory	-C, --directory
diskcheck	--diskcheck
duplicate	--duplicate
file	-f, --file, --makefile , --sconstruct
help	-h, --help
ignore_errors	--ignore-errors
implicit_cache	--implicit-cache
implicit_deps_changed	--implicit-deps-changed
implicit_deps_unchanged	--implicit-deps-unchanged
interactive	--interact, --interactive
keep_going	-k, --keep-going
max_drift	--max-drift
no_exec	-n, --no-exec, --just-print, --dry-run, --recon
no_site_dir	--no-site-dir
num_jobs	-j, --jobs
profile_file	--profile

String for `GetOption` and `SetOption`	Command-Line Option(s)
question	-q, --question
random	--random
repository	-Y, --repository, --srcdir
silent	-s, --silent, --quiet
site_dir	--site-dir
stack_size	--stack-size
taskmastertrace_file	--taskmastertrace
warn	--warn --warning

10.1.5. Adding Custom Command-Line Options: the `AddOption` Function

SCons also allows you to define your own command-line options with the `AddOption` function. The `AddOption` function takes the same arguments as the `optparse.add_option` function from the standard Python library. [1] Once you have added a custom command-line option with the `AddOption` function, the value of the option (if any) is immediately available using the standard `GetOption` function. (The value can also be set using `SetOption`, although that's not very useful in practice because a default value can be specified in directly in the `AddOption` call.)

One useful example of using this functionality is to provide a `--prefix` for users:

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

env = Environment(PREFIX = GetOption('prefix'))

installed_foo = env.Install('$PREFIX/usr/bin', 'foo.in')
Default(installed_foo)
```

The above code uses the `GetOption` function to set the `$PREFIX` construction variable to any value that the user specifies with a command-line option of `--prefix`. Because `$PREFIX` will expand to a null string if it's not initialized, running SCons without the option of `--prefix` will install the file in the `/usr/bin/` directory:

```
% scons -Q -n
Install file: "foo.in" as "/usr/bin/foo.in"
```

But specifying `--prefix=/tmp/install` on the command line causes the file to be installed in the `/tmp/install/usr/bin/` directory:

```
% scons -Q -n --prefix=/tmp/install
Install file: "foo.in" as "/tmp/install/usr/bin/foo.in"
```

10.2. Command-Line `variable=value` Build Variables

You may want to control various aspects of your build by allowing the user to specify `variable=value` values on the command line. For example, suppose you want users to be able to build a debug version of a program by running SCons as follows:

```
% scons -Q debug=1
```

SCons provides an `ARGUMENTS` dictionary that stores all of the `variable=value` assignments from the command line. This allows you to modify aspects of your build in response to specifications on the command line. (Note that unless you want to require that users *always* specify a variable, you probably want to use the Python `ARGUMENTS.get()` function, which allows you to specify a default value to be used if there is no specification on the command line.)

The following code sets the $CCFLAGS construction variable in response to the `debug` flag being set in the `ARGUMENTS` dictionary:

```
env = Environment()
debug = ARGUMENTS.get('debug', 0)
if int(debug):
    env.Append(CCFLAGS = '-g')
env.Program('prog.c')
```

This results in the `-g` compiler option being used when `debug=1` is used on the command line:

```
% scons -Q debug=0
cc -o prog.o -c prog.c
cc -o prog prog.o
% scons -Q debug=0
scons: `.' is up to date.
% scons -Q debug=1
cc -o prog.o -c -g prog.c
cc -o prog prog.o
% scons -Q debug=1
scons: `.' is up to date.
```

Notice that SCons keeps track of the last values used to build the object files, and as a result correctly rebuilds the object and executable files only when the value of the `debug` argument has changed.

The `ARGUMENTS` dictionary has two minor drawbacks. First, because it is a dictionary, it can only store one value for each specified keyword, and thus only "remembers" the last setting for each keyword on the command line. This makes the `ARGUMENTS` dictionary inappropriate if users should be able to specify multiple values on the command line for a given keyword. Second, it does not preserve the order in which the variable settings were specified, which is a problem if you want the configuration to behave differently in response to the order in which the build variable settings were specified on the command line.

To accomodate these requirements, SCons provides an `ARGLIST` variable that gives you direct access to `variable=value` settings on the command line, in the exact order they were specified, and without removing any duplicate settings. Each element in the `ARGLIST` variable is itself a two-element list containing the keyword and the value of the setting, and you must loop through, or otherwise select from, the elements of `ARGLIST` to process the specific settings you want in whatever way is appropriate for your configuration. For example, the following code to let the user add to the `CPPDEFINES` construction variable by specifying multiple `define=` settings on the command line:

```
cppdefines = []
for key, value in ARGLIST:
    if key == 'define':
        cppdefines.append(value)
env = Environment(CPPDEFINES = cppdefines)
env.Object('prog.c')
```

Yields the following output:

```
% scons -Q define=FOO
cc -o prog.o -c -DFOO prog.c
% scons -Q define=FOO define=BAR
cc -o prog.o -c -DFOO -DBAR prog.c
```

Note that the `ARGLIST` and `ARGUMENTS` variables do not interfere with each other, but merely provide slightly different views into how the user specified `variable=value` settings on the command line. You can use both variables in the same SCons configuration. In general, the `ARGUMENTS` dictionary is more convenient to use, (since you can just fetch variable settings through a dictionary access), and the `ARGLIST` list is more flexible (since you can examine the specific order in which the user's command-line variabe settings).

10.2.1. Controlling Command-Line Build Variables

Being able to use a command-line build variable like `debug=1` is handy, but it can be a chore to write specific Python code to recognize each such variable, check for errors and provide appropriate messages,

and apply the values to a construction variable. To help with this, SCons supports a class to define such build variables easily, and a mechanism to apply the build variables to a construction environment. This allows you to control how the build variables affect construction environments.

For example, suppose that you want users to set a RELEASE construction variable on the command line whenever the time comes to build a program for release, and that the value of this variable should be added to the command line with the appropriate -D option (or other command line option) to pass the value to the C compiler. Here's how you might do that by setting the appropriate value in a dictionary for the $CPPDEFINES construction variable:

```
vars = Variables(None, ARGUMENTS)
vars.Add('RELEASE', 'Set to 1 to build for release', 0)
env = Environment(variables = vars,
                  CPPDEFINES={'RELEASE_BUILD' : '${RELEASE}'})
env.Program(['foo.c', 'bar.c'])
```

This SConstruct file first creates a Variables object which uses the values from the command-line options dictionary ARGUMENTS (the vars = Variables(None, ARGUMENTS) call). It then uses the object's Add method to indicate that the RELEASE variable can be set on the command line, and that its default value will be 0 (the third argument to the Add method). The second argument is a line of help text; we'll learn how to use it in the next section.

We then pass the created Variables object as a variables keyword argument to the Environment call used to create the construction environment. This then allows a user to set the RELEASE build variable on the command line and have the variable show up in the command line used to build each object from a C source file:

```
% scons -Q RELEASE=1
cc -o bar.o -c -DRELEASE_BUILD=1 bar.c
cc -o foo.o -c -DRELEASE_BUILD=1 foo.c
cc -o foo foo.o bar.o
```

NOTE: Before SCons release 0.98.1, these build variables were known as "command-line build options." The class was actually named the Options class, and in the sections below, the various functions were named BoolOption, EnumOption, ListOption, PathOption, PackageOption and AddOptions. These older names still work, and you may encounter them in older SConscript files, but they have been officially deprecated as of SCons version 2.0.

10.2.2. Providing Help for Command-Line Build Variables

To make command-line build variables most useful, you ideally want to provide some help text that will describe the available variables when the user runs scons -h. You could write this text by hand, but SCons provides an easier way. Variables objects support a GenerateHelpText method that will, as

its name suggests, generate text that describes the various variables that have been added to it. You then pass the output from this method to the `Help` function:

```
vars = Variables(None, ARGUMENTS)
vars.Add('RELEASE', 'Set to 1 to build for release', 0)
env = Environment(variables = vars)
Help(vars.GenerateHelpText(env))
```

SCons will now display some useful text when the `-h` option is used:

```
% scons -Q -h

RELEASE: Set to 1 to build for release
    default: 0
    actual: 0

Use scons -H for help about command-line options.
```

Notice that the help output shows the default value, and the current actual value of the build variable.

10.2.3. Reading Build Variables From a File

Giving the user a way to specify the value of a build variable on the command line is useful, but can still be tedious if users must specify the variable every time they run SCons. We can let users provide customized build variable settings in a local file by providing a file name when we create the `Variables` object:

```
vars = Variables('custom.py')
vars.Add('RELEASE', 'Set to 1 to build for release', 0)
env = Environment(variables = vars,
                  CPPDEFINES={'RELEASE_BUILD' : '${RELEASE}'})
env.Program(['foo.c', 'bar.c'])
Help(vars.GenerateHelpText(env))
```

This then allows the user to control the `RELEASE` variable by setting it in the `custom.py` file:

```
RELEASE = 1
```

Note that this file is actually executed like a Python script. Now when we run SCons:

```
% scons -Q
cc -o bar.o -c -DRELEASE_BUILD=1 bar.c
cc -o foo.o -c -DRELEASE_BUILD=1 foo.c
cc -o foo foo.o bar.o
```

And if we change the contents of `custom.py` to:

```
RELEASE = 0
```

The object files are rebuilt appropriately with the new variable:

```
% scons -Q
cc -o bar.o -c -DRELEASE_BUILD=0 bar.c
cc -o foo.o -c -DRELEASE_BUILD=0 foo.c
cc -o foo foo.o bar.o
```

Finally, you can combine both methods with:

```
vars = Variables('custom.py', ARGUMENTS)
```

where values in the option file `custom.py` get overwritten by the ones specified on the command line.

10.2.4. Pre-Defined Build Variable Functions

SCons provides a number of functions that provide ready-made behaviors for various types of command-line build variables.

10.2.4.1. True/False Values: the `BoolVariable` Build Variable Function

It's often handy to be able to specify a variable that controls a simple Boolean variable with a `true` or `false` value. It would be even more handy to accomodate users who have different preferences for how to represent `true` or `false` values. The `BoolVariable` function makes it easy to accomodate these common representations of `true` or `false`.

The `BoolVariable` function takes three arguments: the name of the build variable, the default value of the build variable, and the help string for the variable. It then returns appropriate information for passing to the `Add` method of a `Variables` object, like so:

```
vars = Variables('custom.py')
vars.Add(BoolVariable('RELEASE', 'Set to build for release', 0))
env = Environment(variables = vars,
                  CPPDEFINES={'RELEASE_BUILD' : '${RELEASE}'})
env.Program('foo.c')
```

With this build variable, the RELEASE variable can now be enabled by setting it to the value yes or t:

```
% scons -Q RELEASE=yes foo.o
cc -o foo.o -c -DRELEASE_BUILD=True foo.c


% scons -Q RELEASE=t foo.o
cc -o foo.o -c -DRELEASE_BUILD=True foo.c
```

Other values that equate to true include y, 1, on and all.

Conversely, RELEASE may now be given a false value by setting it to no or f:

```
% scons -Q RELEASE=no foo.o
cc -o foo.o -c -DRELEASE_BUILD=False foo.c


% scons -Q RELEASE=f foo.o
cc -o foo.o -c -DRELEASE_BUILD=False foo.c
```

Other values that equate to false include n, 0, off and none.

Lastly, if a user tries to specify any other value, SCons supplies an appropriate error message:

```
% scons -Q RELEASE=bad_value foo.o

scons: *** Error converting option: RELEASE
Invalid value for boolean option: bad_value
File "/home/my/project/SConstruct", line 4, in <module>
```

10.2.4.2. Single Value From a List: the EnumVariable Build Variable Function

Suppose that we want a user to be able to set a COLOR variable that selects a background color to be displayed by an application, but that we want to restrict the choices to a specific set of allowed colors. This can be set up quite easily using the EnumVariable, which takes a list of allowed_values in addition to the variable name, default value, and help text arguments:

```
vars = Variables('custom.py')
vars.Add(EnumVariable('COLOR', 'Set background color', 'red',
                      allowed_values=('red', 'green', 'blue')))
env = Environment(variables = vars,
                  CPPDEFINES={'COLOR' : '"${COLOR}"'})
env.Program('foo.c')
```

The user can now explicity set the COLOR build variable to any of the specified allowed values:

```
% scons -Q COLOR=red foo.o
cc -o foo.o -c -DCOLOR="red" foo.c
% scons -Q COLOR=blue foo.o
cc -o foo.o -c -DCOLOR="blue" foo.c
% scons -Q COLOR=green foo.o
cc -o foo.o -c -DCOLOR="green" foo.c
```

But, almost more importantly, an attempt to set COLOR to a value that's not in the list generates an error message:

```
% scons -Q COLOR=magenta foo.o

scons: *** Invalid value for option COLOR: magenta.  Valid values are: ('red',
File "/home/my/project/SConstruct", line 5, in <module>
```

The EnumVariable function also supports a way to map alternate names to allowed values. Suppose, for example, that we want to allow the user to use the word navy as a synonym for blue. We do this by adding a map dictionary that will map its key values to the desired legal value:

```
vars = Variables('custom.py')
vars.Add(EnumVariable('COLOR', 'Set background color', 'red',
                      allowed_values=('red', 'green', 'blue'),
                      map={'navy':'blue'}))
env = Environment(variables = vars,
                  CPPDEFINES={'COLOR' : '"${COLOR}"'})
env.Program('foo.c')
```

As desired, the user can then use navy on the command line, and SCons will translate it into blue when it comes time to use the COLOR variable to build a target:

```
% scons -Q COLOR=navy foo.o
cc -o foo.o -c -DCOLOR="blue" foo.c
```

By default, when using the EnumVariable function, arguments that differ from the legal values only in case are treated as illegal values:

```
% scons -Q COLOR=Red foo.o

scons: *** Invalid value for option COLOR: Red.  Valid values are: ('red', 'gre
File "/home/my/project/SConstruct", line 5, in <module>
% scons -Q COLOR=BLUE foo.o
```

```
scons: *** Invalid value for option COLOR: BLUE.  Valid values are: ('red', 'gr
File "/home/my/project/SConstruct", line 5, in <module>
% scons -Q COLOR=nAvY foo.o

scons: *** Invalid value for option COLOR: nAvY.  Valid values are: ('red', 'gr
File "/home/my/project/SConstruct", line 5, in <module>
```

The `EnumVariable` function can take an additional `ignorecase` keyword argument that, when set to `1`, tells SCons to allow case differences when the values are specified:

```
vars = Variables('custom.py')
vars.Add(EnumVariable('COLOR', 'Set background color', 'red',
                      allowed_values=('red', 'green', 'blue'),
                      map={'navy':'blue'},
                      ignorecase=1))
env = Environment(variables = vars,
                  CPPDEFINES={'COLOR' : '"${COLOR}"'})
env.Program('foo.c')
```

Which yields the output:

```
% scons -Q COLOR=Red foo.o
cc -o foo.o -c -DCOLOR="Red" foo.c
% scons -Q COLOR=BLUE foo.o
cc -o foo.o -c -DCOLOR="BLUE" foo.c
% scons -Q COLOR=nAvY foo.o
cc -o foo.o -c -DCOLOR="blue" foo.c
% scons -Q COLOR=green foo.o
cc -o foo.o -c -DCOLOR="green" foo.c
```

Notice that an `ignorecase` value of `1` preserves the case-spelling that the user supplied. If you want SCons to translate the names into lower-case, regardless of the case used by the user, specify an `ignorecase` value of `2`:

```
vars = Variables('custom.py')
vars.Add(EnumVariable('COLOR', 'Set background color', 'red',
                      allowed_values=('red', 'green', 'blue'),
                      map={'navy':'blue'},
                      ignorecase=2))
env = Environment(variables = vars,
                  CPPDEFINES={'COLOR' : '"${COLOR}"'})
env.Program('foo.c')
```

Now SCons will use values of `red`, `green` or `blue` regardless of how the user spells those values on the command line:

```
% scons -Q COLOR=Red foo.o
cc -o foo.o -c -DCOLOR="red" foo.c
% scons -Q COLOR=nAvY foo.o
cc -o foo.o -c -DCOLOR="blue" foo.c
% scons -Q COLOR=GREEN foo.o
cc -o foo.o -c -DCOLOR="green" foo.c
```

10.2.4.3. Multiple Values From a List: the `ListVariable` Build Variable Function

Another way in which you might want to allow users to control a build variable is to specify a list of one or more legal values. SCons supports this through the `ListVariable` function. If, for example, we want a user to be able to set a `COLORS` variable to one or more of the legal list of values:

```
vars = Variables('custom.py')
vars.Add(ListVariable('COLORS', 'List of colors', 0,
                      ['red', 'green', 'blue']))
env = Environment(variables = vars,
                  CPPDEFINES={'COLORS' : '"${COLORS}"'})
env.Program('foo.c')
```

A user can now specify a comma-separated list of legal values, which will get translated into a space-separated list for passing to the any build commands:

```
% scons -Q COLORS=red,blue foo.o
cc -o foo.o -c -DCOLORS="red blue" foo.c
% scons -Q COLORS=blue,green,red foo.o
cc -o foo.o -c -DCOLORS="blue green red" foo.c
```

In addition, the `ListVariable` function allows the user to specify explicit keywords of `all` or `none` to select all of the legal values, or none of them, respectively:

```
% scons -Q COLORS=all foo.o
cc -o foo.o -c -DCOLORS="red green blue" foo.c
% scons -Q COLORS=none foo.o
cc -o foo.o -c -DCOLORS="" foo.c
```

And, of course, an illegal value still generates an error message:

```
% scons -Q COLORS=magenta foo.o

scons: *** Error converting option: COLORS
Invalid value(s) for option: magenta
File "/home/my/project/SConstruct", line 5, in <module>
```

10.2.4.4. Path Names: the `PathVariable` Build Variable Function

SCons supports a `PathVariable` function to make it easy to create a build variable to control an expected path name. If, for example, you need to define a variable in the preprocessor that controls the location of a configuration file:

```
vars = Variables('custom.py')
vars.Add(PathVariable('CONFIG',
                      'Path to configuration file',
                      '/etc/my_config'))
env = Environment(variables = vars,
                  CPPDEFINES={'CONFIG_FILE' : '"$CONFIG"'})
env.Program('foo.c')
```

This then allows the user to override the `CONFIG` build variable on the command line as necessary:

```
% scons -Q foo.o
cc -o foo.o -c -DCONFIG_FILE="/etc/my_config" foo.c
% scons -Q CONFIG=/usr/local/etc/other_config foo.o
scons: `foo.o' is up to date.
```

By default, `PathVariable` checks to make sure that the specified path exists and generates an error if it doesn't:

```
% scons -Q CONFIG=/does/not/exist foo.o

scons: *** Path for option CONFIG does not exist: /does/not/exist
File "/home/my/project/SConstruct", line 6, in <module>
```

`PathVariable` provides a number of methods that you can use to change this behavior. If you want to ensure that any specified paths are, in fact, files and not directories, use the `PathVariable.PathIsFile` method:

```
vars = Variables('custom.py')
vars.Add(PathVariable('CONFIG',
                      'Path to configuration file',
                      '/etc/my_config',
                      PathVariable.PathIsFile))
env = Environment(variables = vars,
                  CPPDEFINES={'CONFIG_FILE' : '"$CONFIG"'})
env.Program('foo.c')
```

Conversely, to ensure that any specified paths are directories and not files, use the `PathVariable.PathIsDir` method:

```
vars = Variables('custom.py')
vars.Add(PathVariable('DBDIR',
                      'Path to database directory',
                      '/var/my_dbdir',
                      PathVariable.PathIsDir))
env = Environment(variables = vars,
                  CPPDEFINES={'DBDIR' : '"$DBDIR"'})
env.Program('foo.c')
```

If you want to make sure that any specified paths are directories, and you would like the directory created if it doesn't already exist, use the `PathVariable.PathIsDirCreate` method:

```
vars = Variables('custom.py')
vars.Add(PathVariable('DBDIR',
                      'Path to database directory',
                      '/var/my_dbdir',
                      PathVariable.PathIsDirCreate))
env = Environment(variables = vars,
                  CPPDEFINES={'DBDIR' : '"$DBDIR"'})
env.Program('foo.c')
```

Lastly, if you don't care whether the path exists, is a file, or a directory, use the `PathVariable.PathAccept` method to accept any path that the user supplies:

```
vars = Variables('custom.py')
vars.Add(PathVariable('OUTPUT',
                      'Path to output file or directory',
                      None,
                      PathVariable.PathAccept))
env = Environment(variables = vars,
                  CPPDEFINES={'OUTPUT' : '"$OUTPUT"'})
env.Program('foo.c')
```

10.2.4.5. Enabled/Disabled Path Names: the `PackageVariable` Build Variable Function

Sometimes you want to give users even more control over a path name variable, allowing them to explicitly enable or disable the path name by using `yes` or `no` keywords, in addition to allow them to supply an explicit path name. SCons supports the `PackageVariable` function to support this:

```
vars = Variables('custom.py')
vars.Add(PackageVariable('PACKAGE',
```

```
                          'Location package',
                          '/opt/location'))
    env = Environment(variables = vars,
                      CPPDEFINES={'PACKAGE' : '"$PACKAGE"'})
    env.Program('foo.c')
```

When the `SConscript` file uses the `PackageVariable` funciton, user can now still use the default or supply an overriding path name, but can now explicitly set the specified variable to a value that indicates the package should be enabled (in which case the default should be used) or disabled:

```
% scons -Q foo.o
cc -o foo.o -c -DPACKAGE="/opt/location" foo.c
% scons -Q PACKAGE=/usr/local/location foo.o
cc -o foo.o -c -DPACKAGE="/usr/local/location" foo.c
% scons -Q PACKAGE=yes foo.o
cc -o foo.o -c -DPACKAGE="True" foo.c
% scons -Q PACKAGE=no foo.o
cc -o foo.o -c -DPACKAGE="False" foo.c
```

10.2.5. Adding Multiple Command-Line Build Variables at Once

Lastly, SCons provides a way to add multiple build variables to a `Variables` object at once. Instead of having to call the `Add` method multiple times, you can call the `AddVariables` method with a list of build variables to be added to the object. Each build variable is specified as either a tuple of arguments, just like you'd pass to the `Add` method itself, or as a call to one of the pre-defined functions for pre-packaged command-line build variables. in any order:

```
vars = Variables()
vars.AddVariables(
    ('RELEASE', 'Set to 1 to build for release', 0),
    ('CONFIG', 'Configuration file', '/etc/my_config'),
    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),
)
```

10.2.6. Handling Unknown Command-Line Build Variables: the `UnknownVariables` Function

Users may, of course, occasionally misspell variable names in their command-line settings. SCons does not generate an error or warning for any unknown variables the users specifies on the command line. (This is in no small part because you may be processing the arguments directly using the `ARGUMENTS` dictionary, and therefore SCons can't know in the general case whether a given "misspelled" variable is really unknown and a potential problem, or something that your `SConscript` file will handle directly with some Python code.)

If, however, you're using a `Variables` object to define a specific set of command-line build variables that you expect users to be able to set, you may want to provide an error message or warning of your own if the user supplies a variable setting that is *not* among the defined list of variable names known to the `Variables` object. You can do this by calling the `UnknownVariables` method of the `Variables` object:

```
vars = Variables(None)
vars.Add('RELEASE', 'Set to 1 to build for release', 0)
env = Environment(variables = vars,
                  CPPDEFINES={'RELEASE_BUILD' : '${RELEASE}'})
unknown = vars.UnknownVariables()
if unknown:
    print "Unknown variables:", unknown.keys()
    Exit(1)
env.Program('foo.c')
```

The `UnknownVariables` method returns a dictionary containing the keywords and values of any variables the user specified on the command line that are *not* among the variables known to the `Variables` object (from having been specified using the `Variables` object's`Add` method). In the examble above, we check for whether the dictionary returned by the `UnknownVariables` is non-empty, and if so print the Python list containing the names of the unknwown variables and then call the `Exit` function to terminate SCons:

```
% scons -Q NOT_KNOWN=foo
Unknown variables: ['NOT_KNOWN']
```

Of course, you can process the items in the dictionary returned by the `UnknownVariables` function in any way appropriate to your build configuration, including just printing a warning message but not exiting, logging an error somewhere, etc.

Note that you must delay the call of UnknownVariables until after you have applied the Variables object to a construction environment with the variables= keyword argument of an Environment call.

10.3. Command-Line Targets

10.3.1. Fetching Command-Line Targets: the COMMAND_LINE_TARGETS Variable

SCons supports a COMMAND_LINE_TARGETS variable that lets you fetch the list of targets that the user specified on the command line. You can use the targets to manipulate the build in any way you wish. As a simple example, suppose that you want to print a reminder to the user whenever a specific program is built. You can do this by checking for the target in the COMMAND_LINE_TARGETS list:

```
if 'bar' in COMMAND_LINE_TARGETS:
    print "Don't forget to copy `bar' to the archive!"
Default(Program('foo.c'))
Program('bar.c')
```

Then, running SCons with the default target works as it always does, but explicity specifying the bar target on the command line generates the warning message:

```
% scons -Q
cc -o foo.o -c foo.c
cc -o foo foo.o
% scons -Q bar
Don't forget to copy `bar' to the archive!
cc -o bar.o -c bar.c
cc -o bar bar.o
```

Another practical use for the COMMAND_LINE_TARGETS variable might be to speed up a build by only reading certain subsidiary SConscript files if a specific target is requested.

10.3.2. Controlling the Default Targets: the Default Function

One of the most basic things you can control is which targets SCons will build by default--that is, when there are no targets specified on the command line. As mentioned previously, SCons will normally build every target in or below the current directory by default--that is, when you don't explicitly specify one or more targets on the command line. Sometimes, however, you may want to specify explicitly that only certain programs, or programs in certain directories, should be built by default. You do this with the Default function:

```
env = Environment()
hello = env.Program('hello.c')
env.Program('goodbye.c')
Default(hello)
```

This `SConstruct` file knows how to build two programs, `hello` and `goodbye`, but only builds the `hello` program by default:

```
% scons -Q
cc -o hello.o -c hello.c
cc -o hello hello.o
% scons -Q
scons: 'hello' is up to date.
% scons -Q goodbye
cc -o goodbye.o -c goodbye.c
cc -o goodbye goodbye.o
```

Note that, even when you use the `Default` function in your `SConstruct` file, you can still explicitly specify the current directory (`.`) on the command line to tell SCons to build everything in (or below) the current directory:

```
% scons -Q .
cc -o goodbye.o -c goodbye.c
cc -o goodbye goodbye.o
cc -o hello.o -c hello.c
cc -o hello hello.o
```

You can also call the `Default` function more than once, in which case each call adds to the list of targets to be built by default:

```
env = Environment()
prog1 = env.Program('prog1.c')
Default(prog1)
prog2 = env.Program('prog2.c')
prog3 = env.Program('prog3.c')
Default(prog3)
```

Or you can specify more than one target in a single call to the `Default` function:

```
env = Environment()
prog1 = env.Program('prog1.c')
prog2 = env.Program('prog2.c')
prog3 = env.Program('prog3.c')
Default(prog1, prog3)
```

Either of these last two examples will build only the prog1 and prog3 programs by default:

```
% scons -Q
cc -o prog1.o -c prog1.c
cc -o prog1 prog1.o
cc -o prog3.o -c prog3.c
cc -o prog3 prog3.o
% scons -Q .
cc -o prog2.o -c prog2.c
cc -o prog2 prog2.o
```

You can list a directory as an argument to `Default`:

```
env = Environment()
env.Program(['prog1/main.c', 'prog1/foo.c'])
env.Program(['prog2/main.c', 'prog2/bar.c'])
Default('prog1')
```

In which case only the target(s) in that directory will be built by default:

```
% scons -Q
cc -o prog1/foo.o -c prog1/foo.c
cc -o prog1/main.o -c prog1/main.c
cc -o prog1/main prog1/main.o prog1/foo.o
% scons -Q
scons: `prog1' is up to date.
% scons -Q .
cc -o prog2/bar.o -c prog2/bar.c
cc -o prog2/main.o -c prog2/main.c
cc -o prog2/main prog2/main.o prog2/bar.o
```

Lastly, if for some reason you don't want any targets built by default, you can use the Python `None` variable:

```
env = Environment()
prog1 = env.Program('prog1.c')
prog2 = env.Program('prog2.c')
Default(None)
```

Which would produce build output like:

```
% scons -Q
scons: *** No targets specified and no Default() targets found.  Stop.
% scons -Q .
cc -o prog1.o -c prog1.c
cc -o prog1 prog1.o
```

```
cc -o prog2.o -c prog2.c
cc -o prog2 prog2.o
```

10.3.2.1. Fetching the List of Default Targets: the DEFAULT_TARGETS Variable

SCons supports a DEFAULT_TARGETS variable that lets you get at the current list of default targets. The DEFAULT_TARGETS variable has two important differences from the COMMAND_LINE_TARGETS variable. First, the DEFAULT_TARGETS variable is a list of internal SCons nodes, so you need to convert the list elements to strings if you want to print them or look for a specific target name. Fortunately, you can do this easily by using the Python map function to run the list through str:

```
prog1 = Program('prog1.c')
Default(prog1)
print "DEFAULT_TARGETS is", map(str, DEFAULT_TARGETS)
```

(Keep in mind that all of the manipulation of the DEFAULT_TARGETS list takes place during the first phase when SCons is reading up the SConscript files, which is obvious if we leave off the -Q flag when we run SCons:)

```
% scons
scons: Reading SConscript files ...
DEFAULT_TARGETS is ['prog1']
scons: done reading SConscript files.
scons: Building targets ...
cc -o prog1.o -c prog1.c
cc -o prog1 prog1.o
scons: done building targets.
```

Second, the contents of the DEFAULT_TARGETS list change in response to calls to the Default function, as you can see from the following SConstruct file:

```
prog1 = Program('prog1.c')
Default(prog1)
print "DEFAULT_TARGETS is now", map(str, DEFAULT_TARGETS)
prog2 = Program('prog2.c')
Default(prog2)
print "DEFAULT_TARGETS is now", map(str, DEFAULT_TARGETS)
```

Which yields the output:

```
% scons
scons: Reading SConscript files ...
DEFAULT_TARGETS is now ['prog1']
```

```
DEFAULT_TARGETS is now ['prog1', 'prog2']
scons: done reading SConscript files.
scons: Building targets ...
cc -o prog1.o -c prog1.c
cc -o prog1 prog1.o
cc -o prog2.o -c prog2.c
cc -o prog2 prog2.o
scons: done building targets.
```

In practice, this simply means that you need to pay attention to the order in which you call the `Default` function and refer to the `DEFAULT_TARGETS` list, to make sure that you don't examine the list before you've added the default targets you expect to find in it.

10.3.3. Fetching the List of Build Targets, Regardless of Origin: the `BUILD_TARGETS` Variable

We've already been introduced to the `COMMAND_LINE_TARGETS` variable, which contains a list of targets specified on the command line, and the `DEFAULT_TARGETS` variable, which contains a list of targets specified via calls to the `Default` method or function. Sometimes, however, you want a list of whatever targets SCons will try to build, regardless of whether the targets came from the command line or a `Default` call. You could code this up by hand, as follows:

```
if COMMAND_LINE_TARGETS:
    targets = COMMAND_LINE_TARGETS
else:
    targets = DEFAULT_TARGETS
```

SCons, however, provides a convenient `BUILD_TARGETS` variable that eliminates the need for this by-hand manipulation. Essentially, the `BUILD_TARGETS` variable contains a list of the command-line targets, if any were specified, and if no command-line targets were specified, it contains a list of the targets specified via the `Default` method or function.

Because `BUILD_TARGETS` may contain a list of SCons nodes, you must convert the list elements to strings if you want to print them or look for a specific target name, just like the `DEFAULT_TARGETS` list:

```
prog1 = Program('prog1.c')
Program('prog2.c')
Default(prog1)
print "BUILD_TARGETS is", map(str, BUILD_TARGETS)
```

Notice how the value of `BUILD_TARGETS` changes depending on whether a target is specified on the command line:

```
% scons -Q
BUILD_TARGETS is ['prog1']
cc -o prog1.o -c prog1.c
cc -o prog1 prog1.o
% scons -Q prog2
BUILD_TARGETS is ['prog2']
cc -o prog2.o -c prog2.c
cc -o prog2 prog2.o
% scons -Q -c .
BUILD_TARGETS is ['.']
Removed prog1.o
Removed prog1
Removed prog2.o
Removed prog2
```

Notes

1. The `AddOption` function is, in fact, implemented using a subclass of the `optparse.OptionParser`.

Chapter 11. Installing Files in Other Directories: the `Install` Builder

Once a program is built, it is often appropriate to install it in another directory for public use. You use the `Install` method to arrange for a program, or any other file, to be copied into a destination directory:

```
env = Environment()
hello = env.Program('hello.c')
env.Install('/usr/bin', hello)
```

Note, however, that installing a file is still considered a type of file "build." This is important when you remember that the default behavior of SCons is to build files in or below the current directory. If, as in the example above, you are installing files in a directory outside of the top-level `SConstruct` file's directory tree, you must specify that directory (or a higher directory, such as `/`) for it to install anything there:

```
% scons -Q
cc -o hello.o -c hello.c
cc -o hello hello.o
% scons -Q /usr/bin
Install file: "hello" as "/usr/bin/hello"
```

It can, however, be cumbersome to remember (and type) the specific destination directory in which the program (or any other file) should be installed. This is an area where the `Alias` function comes in handy, allowing you, for example, to create a pseudo-target named `install` that can expand to the specified destination directory:

```
env = Environment()
hello = env.Program('hello.c')
env.Install('/usr/bin', hello)
env.Alias('install', '/usr/bin')
```

This then yields the more natural ability to install the program in its destination as follows:

```
% scons -Q
cc -o hello.o -c hello.c
cc -o hello hello.o
% scons -Q install
Install file: "hello" as "/usr/bin/hello"
```

11.1. Installing Multiple Files in a Directory

You can install multiple files into a directory simply by calling the `Install` function multiple times:

```
env = Environment()
hello = env.Program('hello.c')
goodbye = env.Program('goodbye.c')
env.Install('/usr/bin', hello)
env.Install('/usr/bin', goodbye)
env.Alias('install', '/usr/bin')
```

Or, more succinctly, listing the multiple input files in a list (just like you can do with any other builder):

```
env = Environment()
hello = env.Program('hello.c')
goodbye = env.Program('goodbye.c')
env.Install('/usr/bin', [hello, goodbye])
env.Alias('install', '/usr/bin')
```

Either of these two examples yields:

```
% scons -Q install
cc -o goodbye.o -c goodbye.c
cc -o goodbye goodbye.o
Install file: "goodbye" as "/usr/bin/goodbye"
cc -o hello.o -c hello.c
cc -o hello hello.o
Install file: "hello" as "/usr/bin/hello"
```

11.2. Installing a File Under a Different Name

The `Install` method preserves the name of the file when it is copied into the destination directory. If you need to change the name of the file when you copy it, use the `InstallAs` function:

```
env = Environment()
hello = env.Program('hello.c')
env.InstallAs('/usr/bin/hello-new', hello)
env.Alias('install', '/usr/bin')
```

This installs the `hello` program with the name `hello-new` as follows:

```
% scons -Q install
cc -o hello.o -c hello.c
```

```
cc -o hello hello.o
Install file: "hello" as "/usr/bin/hello-new"
```

11.3. Installing Multiple Files Under Different Names

Lastly, if you have multiple files that all need to be installed with different file names, you can either call the InstallAs function multiple times, or as a shorthand, you can supply same-length lists for both the target and source arguments:

```
env = Environment()
hello = env.Program('hello.c')
goodbye = env.Program('goodbye.c')
env.InstallAs(['/usr/bin/hello-new',
               '/usr/bin/goodbye-new'],
              [hello, goodbye])
env.Alias('install', '/usr/bin')
```

In this case, the InstallAs function loops through both lists simultaneously, and copies each source file into its corresponding target file name:

```
% scons -Q install
cc -o goodbye.o -c goodbye.c
cc -o goodbye goodbye.o
Install file: "goodbye" as "/usr/bin/goodbye-new"
cc -o hello.o -c hello.c
cc -o hello hello.o
Install file: "hello" as "/usr/bin/hello-new"
```

Chapter 12. Platform-Independent File System Manipulation

SCons provides a number of platform-independent functions, called `factories`, that perform common file system manipulations like copying, moving or deleting files and directories, or making directories. These functions are `factories` because they don't perform the action at the time they're called, they each return an `Action` object that can be executed at the appropriate time.

12.1. Copying Files or Directories: The `Copy` Factory

Suppose you want to arrange to make a copy of a file, and don't have a suitable pre-existing builder. [1] One way would be to use the `Copy` action factory in conjunction with the `Command` builder:

```
Command("file.out", "file.in", Copy("$TARGET", "$SOURCE"))
```

Notice that the action returned by the `Copy` factory will expand the $TARGET and $SOURCE strings at the time `file.out` is built, and that the order of the arguments is the same as that of a builder itself--that is, target first, followed by source:

```
% scons -Q
Copy("file.out", "file.in")
```

You can, of course, name a file explicitly instead of using $TARGET or $SOURCE:

```
Command("file.out", [], Copy("$TARGET", "file.in"))
```

Which executes as:

```
% scons -Q
Copy("file.out", "file.in")
```

The usefulness of the `Copy` factory becomes more apparent when you use it in a list of actions passed to the `Command` builder. For example, suppose you needed to run a file through a utility that only modifies files in-place, and can't "pipe" input to output. One solution is to copy the source file to a temporary file name, run the utility, and then copy the modified temporary file to the target, which the `Copy` factory makes extremely easy:

```
Command("file.out", "file.in",
        [
```

```
          Copy("tempfile", "$SOURCE"),
          "modify tempfile",
          Copy("$TARGET", "tempfile"),
        ])
```

The output then looks like:

```
% scons -Q
Copy("tempfile", "file.in")
modify tempfile
Copy("file.out", "tempfile")
```

12.2. Deleting Files or Directories: The `Delete` Factory

If you need to delete a file, then the `Delete` factory can be used in much the same way as the `Copy` factory. For example, if we want to make sure that the temporary file in our last example doesn't exist before we copy to it, we could add `Delete` to the beginning of the command list:

```
Command("file.out", "file.in",
        [
          Delete("tempfile"),
          Copy("tempfile", "$SOURCE"),
          "modify tempfile",
          Copy("$TARGET", "tempfile"),
        ])
```

Which then executes as follows:

```
% scons -Q
Delete("tempfile")
Copy("tempfile", "file.in")
modify tempfile
Copy("file.out", "tempfile")
```

Of course, like all of these `Action` factories, the `Delete` factory also expands $TARGET and $SOURCE variables appropriately. For example:

```
Command("file.out", "file.in",
        [
          Delete("$TARGET"),
          Copy("$TARGET", "$SOURCE")
        ])
```

Executes as:

```
% scons -Q
Delete("file.out")
Copy("file.out", "file.in")
```

Note, however, that you typically don't need to call the `Delete` factory explicitly in this way; by default, SCons deletes its target(s) for you before executing any action.

One word of caution about using the `Delete` factory: it has the same variable expansions available as any other factory, including the $SOURCE variable. Specifying `Delete("$SOURCE")` is not something you usually want to do!

12.3. Moving (Renaming) Files or Directories: The `Move` Factory

The `Move` factory allows you to rename a file or directory. For example, if we don't want to copy the temporary file, we could use:

```
Command("file.out", "file.in",
        [
          Copy("tempfile", "$SOURCE"),
          "modify tempfile",
          Move("$TARGET", "tempfile"),
        ])
```

Which would execute as:

```
% scons -Q
Copy("tempfile", "file.in")
modify tempfile
Move("file.out", "tempfile")
```

12.4. Updating the Modification Time of a File: The `Touch` Factory

If you just need to update the recorded modification time for a file, use the `Touch` factory:

```
Command("file.out", "file.in",
```

```
        [
          Copy("$TARGET", "$SOURCE"),
          Touch("$TARGET"),
        ])
```

Which executes as:

```
% scons -Q
Copy("file.out", "file.in")
Touch("file.out")
```

12.5. Creating a Directory: The `Mkdir` Factory

If you need to create a directory, use the `Mkdir` factory. For example, if we need to process a file in a temporary directory in which the processing tool will create other files that we don't care about, you could use:

```
Command("file.out", "file.in",
        [
          Delete("tempdir"),
          Mkdir("tempdir"),
          Copy("tempdir/${SOURCE.file}", "$SOURCE"),
          "process tempdir",
          Move("$TARGET", "tempdir/output_file"),
          Delete("tempdir"),
        ])
```

Which executes as:

```
% scons -Q
Delete("tempdir")
Mkdir("tempdir")
Copy("tempdir/file.in", "file.in")
process tempdir
Move("file.out", "tempdir/output_file")
scons: *** [file.out] tempdir/output_file: No such file or directory
```

12.6. Changing File or Directory Permissions: The `Chmod` Factory

To change permissions on a file or directory, use the `Chmod` factory. The permission argument uses POSIX-style permission bits and should typically be expressed as an octal, not decimal, number:

```
Command("file.out", "file.in",
        [
          Copy("$TARGET", "$SOURCE"),
          Chmod("$TARGET", 0755),
        ])
```

Which executes:

```
% scons -Q
Copy("file.out", "file.in")
Chmod("file.out", 0755)
```

12.7. Executing an action immediately: the `Execute` Function

We've been showing you how to use `Action` factories in the `Command` function. You can also execute an `Action` returned by a factory (or actually, any `Action`) at the time the `SConscript` file is read by using the `Execute` function. For example, if we need to make sure that a directory exists before we build any targets,

```
Execute(Mkdir('/tmp/my_temp_directory'))
```

Notice that this will create the directory while the `SConscript` file is being read:

```
% scons
scons: Reading SConscript files ...
Mkdir("/tmp/my_temp_directory")
scons: done reading SConscript files.
scons: Building targets ...
scons: `.' is up to date.
scons: done building targets.
```

If you're familiar with Python, you may wonder why you would want to use this instead of just calling the native Python `os.mkdir()` function. The advantage here is that the `Mkdir` action will behave

appropriately if the user specifies the SCons `-n` or `-q` options--that is, it will print the action but not actually make the directory when `-n` is specified, or make the directory but not print the action when `-q` is specified.

The `Execute` function returns the exit status or return value of the underlying action being executed. It will also print an error message if the action fails and returns a non-zero value. SCons will *not*, however, actually stop the build if the action fails. If you want the build to stop in response to a failure in an action called by `Execute`, you must do so by explicitly checking the return value and calling the `Exit` function (or a Python equivalent):

```
if Execute(Mkdir('/tmp/my_temp_directory')):
    # A problem occurred while making the temp directory.
    Exit(1)
```

Notes

1. Unfortunately, in the early days of SCons design, we used the name `Copy` for the function that returns a copy of the environment, otherwise that would be the logical choice for a Builder that copies a file or directory tree to a target location.

Chapter 13. Controlling Removal of Targets

There are two occasions when SCons will, by default, remove target files. The first is when SCons determines that an target file needs to be rebuilt and removes the existing version of the target before executing The second is when SCons is invoked with the `-c` option to "clean" a tree of its built targets. These behaviours can be suppressed with the `Precious` and `NoClean` functions, respectively.

13.1. Preventing target removal during build: the `Precious` Function

By default, SCons removes targets before building them. Sometimes, however, this is not what you want. For example, you may want to update a library incrementally, not by having it deleted and then rebuilt from all of the constituent object files. In such cases, you can use the `Precious` method to prevent SCons from removing the target before it is built:

```
env = Environment(RANLIBCOM='')
lib = env.Library('foo', ['f1.c', 'f2.c', 'f3.c'])
env.Precious(lib)
```

Although the output doesn't look any different, SCons does not, in fact, delete the target library before rebuilding it:

```
% scons -Q
cc -o f1.o -c f1.c
cc -o f2.o -c f2.c
cc -o f3.o -c f3.c
ar rc libfoo.a f1.o f2.o f3.o
```

SCons will, however, still delete files marked as `Precious` when the `-c` option is used.

13.2. Preventing target removal during clean: the `NoClean` Function

By default, SCons removes all built targets when invoked with the `-c` option to clean a source tree of built targets. Sometimes, however, this is not what you want. For example, you may want to remove only intermediate generated files (such as object files), but leave the final targets (the libraries) untouched. In such cases, you can use the `NoClean` method to prevent SCons from removing a target during a clean:

```
env = Environment(RANLIBCOM='')
```

```
lib = env.Library('foo', ['f1.c', 'f2.c', 'f3.c'])
env.NoClean(lib)
```

Notice that the `libfoo.a` is not listed as a removed file:

```
% scons -Q
cc -o f1.o -c f1.c
cc -o f2.o -c f2.c
cc -o f3.o -c f3.c
ar rc libfoo.a f1.o f2.o f3.o
% scons -c
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Cleaning targets ...
Removed f1.o
Removed f2.o
Removed f3.o
scons: done cleaning targets.
```

13.3. Removing additional files during clean: the `Clean` Function

There may be additional files that you want removed when the `-c` option is used, but which SCons doesn't know about because they're not normal target files. For example, perhaps a command you invoke creates a log file as part of building the target file you want. You would like the log file cleaned, but you don't want to have to teach SCons that the command "builds" two files.

You can use the `Clean` function to arrange for additional files to be removed when the `-c` option is used. Notice, however, that the `Clean` function takes two arguments, and the *second* argument is the name of the additional file you want cleaned (`foo.log` in this example):

```
t = Command('foo.out', 'foo.in', 'build -o $TARGET $SOURCE')
Clean(t, 'foo.log')
```

The first argument is the target with which you want the cleaning of this additional file associated. In the above example, we've used the return value from the `Command` function, which represents the `foo.out` target. Now whenever the `foo.out` target is cleaned by the `-c` option, the `foo.log` file will be removed as well:

```
% scons -Q
build -o foo.out foo.in
% scons -Q -c
Removed foo.out
```

```
Removed foo.log
```

Chapter 14. Hierarchical Builds

The source code for large software projects rarely stays in a single directory, but is nearly always divided into a hierarchy of directories. Organizing a large software build using SCons involves creating a hierarchy of build scripts using the `SConscript` function.

14.1. `SConscript` Files

As we've already seen, the build script at the top of the tree is called `SConstruct`. The top-level `SConstruct` file can use the `SConscript` function to include other subsidiary scripts in the build. These subsidiary scripts can, in turn, use the `SConscript` function to include still other scripts in the build. By convention, these subsidiary scripts are usually named `SConscript`. For example, a top-level `SConstruct` file might arrange for four subsidiary scripts to be included in the build as follows:

```
SConscript(['drivers/display/SConscript',
            'drivers/mouse/SConscript',
            'parser/SConscript',
            'utilities/SConscript'])
```

In this case, the `SConstruct` file lists all of the `SConscript` files in the build explicitly. (Note, however, that not every directory in the tree necessarily has an `SConscript` file.) Alternatively, the `drivers` subdirectory might contain an intermediate `SConscript` file, in which case the `SConscript` call in the top-level `SConstruct` file would look like:

```
SConscript(['drivers/SConscript',
            'parser/SConscript',
            'utilities/SConscript'])
```

And the subsidiary `SConscript` file in the `drivers` subdirectory would look like:

```
SConscript(['display/SConscript',
            'mouse/SConscript'])
```

Whether you list all of the `SConscript` files in the top-level `SConstruct` file, or place a subsidiary `SConscript` file in intervening directories, or use some mix of the two schemes, is up to you and the needs of your software.

14.2. Path Names Are Relative to the SConscript Directory

Subsidiary SConscript files make it easy to create a build hierarchy because all of the file and directory names in a subsidiary SConscript files are interpreted relative to the directory in which the SConscript file lives. Typically, this allows the SConscript file containing the instructions to build a target file to live in the same directory as the source files from which the target will be built, making it easy to update how the software is built whenever files are added or deleted (or other changes are made).

For example, suppose we want to build two programs prog1 and prog2 in two separate directories with the same names as the programs. One typical way to do this would be with a top-level SConstruct file like this:

```
SConscript(['prog1/SConscript',
            'prog2/SConscript'])
```

And subsidiary SConscript files that look like this:

```
env = Environment()
env.Program('prog1', ['main.c', 'foo1.c', 'foo2.c'])
```

And this:

```
env = Environment()
env.Program('prog2', ['main.c', 'bar1.c', 'bar2.c'])
```

Then, when we run SCons in the top-level directory, our build looks like:

```
% scons -Q
cc -o prog1/foo1.o -c prog1/foo1.c
cc -o prog1/foo2.o -c prog1/foo2.c
cc -o prog1/main.o -c prog1/main.c
cc -o prog1/prog1 prog1/main.o prog1/foo1.o prog1/foo2.o
cc -o prog2/bar1.o -c prog2/bar1.c
cc -o prog2/bar2.o -c prog2/bar2.c
cc -o prog2/main.o -c prog2/main.c
cc -o prog2/prog2 prog2/main.o prog2/bar1.o prog2/bar2.o
```

Notice the following: First, you can have files with the same names in multiple directories, like main.c in the above example. Second, unlike standard recursive use of Make, SCons stays in the top-level directory (where the SConstruct file lives) and issues commands that use the path names from the top-level directory to the target and source files within the hierarchy.

14.3. Top-Level Path Names in Subsidiary SConscript Files

If you need to use a file from another directory, it's sometimes more convenient to specify the path to a file in another directory from the top-level SConstruct directory, even when you're using that file in a subsidiary SConscript file in a subdirectory. You can tell SCons to interpret a path name as relative to the top-level SConstruct directory, not the local directory of the SConscript file, by appending a # (hash mark) to the beginning of the path name:

```
env = Environment()
env.Program('prog', ['main.c', '#lib/foo1.c', 'foo2.c'])
```

In this example, the lib directory is directly underneath the top-level SConstruct directory. If the above SConscript file is in a subdirectory named src/prog, the output would look like:

```
% scons -Q
cc -o lib/foo1.o -c lib/foo1.c
cc -o src/prog/foo2.o -c src/prog/foo2.c
cc -o src/prog/main.o -c src/prog/main.c
cc -o src/prog/prog src/prog/main.o lib/foo1.o src/prog/foo2.o
```

(Notice that the lib/foo1.o object file is built in the same directory as its source file. See Chapter 15, below, for information about how to build the object file in a different subdirectory.)

14.4. Absolute Path Names

Of course, you can always specify an absolute path name for a file--for example:

```
env = Environment()
env.Program('prog', ['main.c', '/usr/joe/lib/foo1.c', 'foo2.c'])
```

Which, when executed, would yield:

```
% scons -Q
cc -o src/prog/foo2.o -c src/prog/foo2.c
cc -o src/prog/main.o -c src/prog/main.c
cc -o /usr/joe/lib/foo1.o -c /usr/joe/lib/foo1.c
cc -o src/prog/prog src/prog/main.o /usr/joe/lib/foo1.o src/prog/foo2.o
```

(As was the case with top-relative path names, notice that the `/usr/joe/lib/foo1.o` object file is built in the same directory as its source file. See Chapter 15, below, for information about how to build the object file in a different subdirectory.)

14.5. Sharing Environments (and Other Variables) Between `SConscript` Files

In the previous example, each of the subsidiary `SConscript` files created its own construction environment by calling `Environment` separately. This obviously works fine, but if each program must be built with the same construction variables, it's cumbersome and error-prone to initialize separate construction environments in the same way over and over in each subsidiary `SConscript` file.

SCons supports the ability to *export* variables from a parent `SConscript` file to its subsidiary `SConscript` files, which allows you to share common initialized values throughout your build hierarchy.

14.5.1. Exporting Variables

There are two ways to export a variable, such as a construction environment, from an `SConscript` file, so that it may be used by other `SConscript` files. First, you can call the `Export` function with a list of variables, or a string of white-space separated variable names. Each call to `Export` adds one or more variables to a global list of variables that are available for import by other `SConscript` files.

```
env = Environment()
Export('env')
```

You may export more than one variable name at a time:

```
env = Environment()
debug = ARGUMENTS['debug']
Export('env', 'debug')
```

Because white space is not legal in Python variable names, the `Export` function will even automatically split a string into separate names for you:

```
Export('env debug')
```

Second, you can specify a list of variables to export as a second argument to the `SConscript` function call:

```
SConscript('src/SConscript', 'env')
```

Or as the `exports` keyword argument:

```
SConscript('src/SConscript', exports='env')
```

These calls export the specified variables to only the listed `SConscript` files. You may, however, specify more than one `SConscript` file in a list:

```
SConscript(['src1/SConscript',
            'src2/SConscript'], exports='env')
```

This is functionally equivalent to calling the `SConscript` function multiple times with the same `exports` argument, one per `SConscript` file.

14.5.2. Importing Variables

Once a variable has been exported from a calling `SConscript` file, it may be used in other `SConscript` files by calling the `Import` function:

```
Import('env')
env.Program('prog', ['prog.c'])
```

The `Import` call makes the `env` construction environment available to the `SConscript` file, after which the variable can be used to build programs, libraries, etc.

Like the `Export` function, the `Import` function can be used with multiple variable names:

```
Import('env', 'debug')
env = env.Clone(DEBUG = debug)
env.Program('prog', ['prog.c'])
```

And the `Import` function will similarly split a string along white-space into separate variable names:

```
Import('env debug')
env = env.Clone(DEBUG = debug)
env.Program('prog', ['prog.c'])
```

Lastly, as a special case, you may import all of the variables that have been exported by supplying an asterisk to the `Import` function:

```
Import('*')
env = env.Clone(DEBUG = debug)
env.Program('prog', ['prog.c'])
```

If you're dealing with a lot of `SConscript` files, this can be a lot simpler than keeping arbitrary lists of imported variables in each file.

14.5.3. Returning Values From an `SConscript` File

Sometimes, you would like to be able to use information from a subsidiary `SConscript` file in some way. For example, suppose that you want to create one library from source files scattered throughout a number of subsidiary `SConscript` files. You can do this by using the `Return` function to return values from the subsidiary `SConscript` files to the calling file.

If, for example, we have two subdirectories `foo` and `bar` that should each contribute a source file to a Library, what we'd like to be able to do is collect the object files from the subsidiary `SConscript` calls like this:

```
env = Environment()
Export('env')
objs = []
for subdir in ['foo', 'bar']:
    o = SConscript('%s/SConscript' % subdir)
    objs.append(o)
env.Library('prog', objs)
```

We can do this by using the `Return` function in the `foo/SConscript` file like this:

```
Import('env')
obj = env.Object('foo.c')
Return('obj')
```

(The corresponding `bar/SConscript` file should be pretty obvious.) Then when we run SCons, the object files from the subsidiary subdirectories are all correctly archived in the desired library:

```
% scons -Q
cc -o bar/bar.o -c bar/bar.c
cc -o foo/foo.o -c foo/foo.c
ar rc libprog.a foo/foo.o bar/bar.o
ranlib libprog.a
```

Chapter 15. Separating Source and Build Directories

It's often useful to keep any built files completely separate from the source files. In SCons, this is usually done by creating one or more separate *variant directory trees* that are used to hold the built objects files, libraries, and executable programs, etc. for a specific flavor, or variant, of build. SCons provides two ways to do this, one through the `SConscript` function that we've already seen, and the second through a more flexible `VariantDir` function.

One historical note: the `VariantDir` function used to be called `BuildDir`. That name is still supported but has been deprecated because the SCons functionality differs from the model of a "build directory" implemented by other build systems like the GNU Autotools.

15.1. Specifying a Variant Directory Tree as Part of an `SConscript` Call

The most straightforward way to establish a variant directory tree uses the fact that the usual way to set up a build hierarchy is to have an `SConscript` file in the source subdirectory. If you then pass a `variant_dir` argument to the `SConscript` function call:

```
SConscript('src/SConscript', variant_dir='build')
```

SCons will then build all of the files in the `build` subdirectory:

```
% ls src
SConscript  hello.c
% scons -Q
cc -o build/hello.o -c build/hello.c
cc -o build/hello build/hello.o
% ls build
SConscript  hello  hello.c  hello.o
```

But wait a minute--what's going on here? SCons created the object file `build/hello.o` in the `build` subdirectory, as expected. But even though our `hello.c` file lives in the `src` subdirectory, SCons has actually compiled a `build/hello.c` file to create the object file.

What's happened is that SCons has *duplicated* the `hello.c` file from the `src` subdirectory to the `build` subdirectory, and built the program from there. The next section explains why SCons does this.

15.2. Why SCons Duplicates Source Files in a Variant Directory Tree

SCons duplicates source files in variant directory trees because it's the most straightforward way to guarantee a correct build *regardless of include-file directory paths, relative references between files, or tool support for putting files in different locations*, and the SCons philosophy is to, by default, guarantee a correct build in all cases.

The most direct reason to duplicate source files in variant directories is simply that some tools (mostly older versions) are written to only build their output files in the same directory as the source files. In this case, the choices are either to build the output file in the source directory and move it to the variant directory, or to duplicate the source files in the variant directory.

Additionally, relative references between files can cause problems if we don't just duplicate the hierarchy of source files in the variant directory. You can see this at work in use of the C preprocessor `#include` mechanism with double quotes, not angle brackets:

```
#include "file.h"
```

The *de facto* standard behavior for most C compilers in this case is to first look in the same directory as the source file that contains the `#include` line, then to look in the directories in the preprocessor search path. Add to this that the SCons implementation of support for code repositories (described below) means not all of the files will be found in the same directory hierarchy, and the simplest way to make sure that the right include file is found is to duplicate the source files into the variant directory, which provides a correct build regardless of the original location(s) of the source files.

Although source-file duplication guarantees a correct build even in these end-cases, it *can* usually be safely disabled. The next section describes how you can disable the duplication of source files in the variant directory.

15.3. Telling SCons to Not Duplicate Source Files in the Variant Directory Tree

In most cases and with most tool sets, SCons can place its target files in a build subdirectory *without* duplicating the source files and everything will work just fine. You can disable the default SCons behavior by specifying `duplicate=0` when you call the `SConscript` function:

```
SConscript('src/SConscript', variant_dir='build', duplicate=0)
```

When this flag is specified, SCons uses the variant directory like most people expect--that is, the output files are placed in the variant directory while the source files stay in the source directory:

```
% ls src
SConscript
hello.c
% scons -Q
cc -c src/hello.c -o build/hello.o
cc -o build/hello build/hello.o
% ls build
hello
hello.o
```

15.4. The `VariantDir` Function

Use the `VariantDir` function to establish that target files should be built in a separate directory from the source files:

```
VariantDir('build', 'src')
env = Environment()
env.Program('build/hello.c')
```

Note that when you're not using an `SConscript` file in the `src` subdirectory, you must actually specify that the program must be built from the `build/hello.c` file that SCons will duplicate in the `build` subdirectory.

When using the `VariantDir` function directly, SCons still duplicates the source files in the variant directory by default:

```
% ls src
hello.c
% scons -Q
cc -o build/hello.o -c build/hello.c
cc -o build/hello build/hello.o
% ls build
hello  hello.c  hello.o
```

You can specify the same `duplicate=0` argument that you can specify for an `SConscript` call:

```
VariantDir('build', 'src', duplicate=0)
env = Environment()
env.Program('build/hello.c')
```

In which case SCons will disable duplication of the source files:

```
% ls src
hello.c
% scons -Q
cc -o build/hello.o -c src/hello.c
cc -o build/hello build/hello.o
% ls build
hello  hello.o
```

15.5. Using `VariantDir` With an `SConscript` File

Even when using the `VariantDir` function, it's much more natural to use it with a subsidiary `SConscript` file. For example, if the `src/SConscript` looks like this:

```
env = Environment()
env.Program('hello.c')
```

Then our `SConstruct` file could look like:

```
VariantDir('build', 'src')
SConscript('build/SConscript')
```

Yielding the following output:

```
% ls src
SConscript  hello.c
% scons -Q
cc -o build/hello.o -c build/hello.c
cc -o build/hello build/hello.o
% ls build
SConscript  hello  hello.c  hello.o
```

Notice that this is completely equivalent to the use of `SConscript` that we learned about in the previous section.

15.6. Using `Glob` with `VariantDir`

The `Glob` file name pattern matching function works just as usual when using `VariantDir`. For example, if the `src/SConscript` looks like this:

```
env = Environment()
env.Program('hello', Glob('*.c'))
```

Then with the same `SConstruct` file as in the previous section, and source files `f1.c` and `f2.c` in src, we would see the following output:

```
% ls src
SConscript  f1.c  f2.c  f2.h
% scons -Q
cc -o build/f1.o -c build/f1.c
cc -o build/f2.o -c build/f2.c
cc -o build/hello build/f1.o build/f2.o
% ls build
SConscript  f1.c  f1.o  f2.c  f2.h  f2.o  hello
```

The `Glob` function returns Nodes in the `build/` tree, as you'd expect.

Chapter 16. Variant Builds

The `variant_dir` keyword argument of the `SConscript` function provides everything we need to show how easy it is to create variant builds using SCons. Suppose, for example, that we want to build a program for both Windows and Linux platforms, but that we want to build it in a shared directory with separate side-by-side build directories for the Windows and Linux versions of the program.

```
platform = ARGUMENTS.get('OS', Platform())

include = "#export/$PLATFORM/include"
lib = "#export/$PLATFORM/lib"
bin = "#export/$PLATFORM/bin"

env = Environment(PLATFORM = platform,
                  BINDIR = bin,
                  INCDIR = include,
                  LIBDIR = lib,
                  CPPPATH = [include],
                  LIBPATH = [lib],
                  LIBS = 'world')

Export('env')

env.SConscript('src/SConscript', variant_dir='build/$PLATFORM')
```

This SConstruct file, when run on a Linux system, yields:

```
% scons -Q OS=linux
Install file: "build/linux/world/world.h" as "export/linux/include/world.h"
cc -o build/linux/hello/hello.o -c -Iexport/linux/include build/linux/hello/hello.c
cc -o build/linux/world/world.o -c -Iexport/linux/include build/linux/world/world.c
ar rc build/linux/world/libworld.a build/linux/world/world.o
ranlib build/linux/world/libworld.a
Install file: "build/linux/world/libworld.a" as "export/linux/lib/libworld.a"
cc -o build/linux/hello/hello build/linux/hello/hello.o -Lexport/linux/lib -lworld
Install file: "build/linux/hello/hello" as "export/linux/bin/hello"
```

The same SConstruct file on Windows would build:

```
C:\>scons -Q OS=windows
Install file: "build/windows/world/world.h" as "export/windows/include/world.h"
cl /Fobuild\windows\hello\hello.obj /c build\windows\hello\hello.c /nologo /Iexport\w
cl /Fobuild\windows\world\world.obj /c build\windows\world\world.c /nologo /Iexport\w
lib /nologo /OUT:build\windows\world\world.lib build\windows\world\world.obj
Install file: "build/windows/world/world.lib" as "export/windows/lib/world.lib"
link /nologo /OUT:build\windows\hello\hello.exe /LIBPATH:export\windows\lib world.lib
embedManifestExeCheck(target, source, env)
Install file: "build/windows/hello/hello.exe" as "export/windows/bin/hello.exe"
```

Chapter 17. Internationalization and localization with gettext

The `gettext` toolset supports internationalization and localization of SCons-based projects. Builders provided by `gettext` automatize generation and updates of translation files. You can manage translations and translation templates similarly to how it's done with autotools.

17.1. Prerequisites

To follow examples provided in this chapter set up your operating system to support two or more languages. In following examples we use locales `en_US`, `de_DE`, and `pl_PL`.

Ensure, that you have GNU gettext utilities (http://www.gnu.org/software/gettext/manual/gettext.html) installed on your system.

To edit translation files you may wish to install poedit (http://www.poedit.net/) editor.

17.2. Simple project

Let's start with a very simple project, the "Hello world" program for example

```
/* hello.c */
#include <stdio.h>
int main(int argc, char* argv[])
{
  printf("Hello world\n");
  return 0;
}
```

Prepare a `SConstruct` to compile the program as usual.

```
# SConstruct
env = Environment()
hello = Program(["hello.c"])
```

Now we'll convert the project to a multi-lingual one. If you don't already have GNU gettext utilities (http://www.gnu.org/software/gettext/manual/gettext.html) installed, install them from your preffered package repository, or download from http://ftp.gnu.org/gnu/gettext/ (http://ftp.gnu.org/gnu/gettext/).

For the purpose of this example, you should have following three locales installed on your system: `en_US`, `de_DE` and `pl_PL`. On debian, for example, you may enable certain locales through **dpkg-reconfigure locales**.

First prepare the `hello.c` program for internationalization. Change the previous code so it reads as follows:

```
/* hello.c */
#include <stdio.h>
#include <libintl.h>
#include <locale.h>
int main(int argc, char* argv[])
{
  bindtextdomain("hello", "locale");
  setlocale(LC_ALL, "");
  textdomain("hello");
  printf(gettext("Hello world\n"));
  return 0;
}
```

Detailed recipes for such conversion can be found at
http://www.gnu.org/software/gettext/manual/gettext.html#Sources
(http://www.gnu.org/software/gettext/manual/gettext.html#Sources). The `gettext("...")` has two purposes. First, it marks messages for the **xgettext(1)** program, which we will use to extract from the sources the messages for localization. Second, it calls the `gettext` library internals to translate the message at runtime.

Now we shall instruct SCons how to generate and maintain translation files. For that, use the `Translate` builder and `MOFiles` builder. The first one takes source files, extracts internationalized messages from them, creates so-called `POT` file (translation template), and then creates `PO` translation files, one for each requested language. Later, during the development lifecycle, the builder keeps all these files up-to date. The `MOFiles` builder compiles the `PO` files to binary form. Then install the `MO` files under directory called `locale`.

The completed `SConstruct` is as follows:

```
# SConstruct
env = Environment( tools = ['default', 'gettext'] )
hello = env.Program(["hello.c"])
env['XGETTEXTFLAGS'] = [
  '--package-name=%s' % 'hello',
  '--package-version=%s' % '1.0',
]
po = env.Translate(["pl","en", "de"], ["hello.c"], POAUTOINIT = 1)
mo = env.MOFiles(po)
InstallAs(["locale/en/LC_MESSAGES/hello.mo"], ["en.mo"])
InstallAs(["locale/pl/LC_MESSAGES/hello.mo"], ["pl.mo"])
InstallAs(["locale/de/LC_MESSAGES/hello.mo"], ["de.mo"])
```

Generate the translation files with **scons po-update**. You should see the output from SCons simillar to this:

```
user@host:$ scons po-update
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
Entering '/home/ptomulik/projects/tmp'
xgettext --package-name=hello --package-version=1.0 -o - hello.c
Leaving '/home/ptomulik/projects/tmp'
Writting 'messages.pot' (new file)
msginit --no-translator -l pl -i messages.pot -o pl.po
Created pl.po.
msginit --no-translator -l en -i messages.pot -o en.po
Created en.po.
msginit --no-translator -l de -i messages.pot -o de.po
Created de.po.
scons: done building targets.
```

If everything is right, you should see following new files.

```
user@host:$ ls *.po*
de.po  en.po  messages.pot  pl.po
```

Open `en.po` in **poedit** and provide the English translation to message `"Hello world\n"`. Do the same for `de.po` (deutsch) and `pl.po` (polish). Let the translations be, for example:

- `en: "Welcome to beautiful world!\n"`
- `de: "Hallo Welt!\n"`
- `pl: "Witaj swiecie!\n"`

Now compile the project by executing **scons**. The output should be similar to this:

```
user@host:$ scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
msgfmt -c -o de.mo de.po
```

```
msgfmt -c -o en.mo en.po
gcc -o hello.o -c hello.c
gcc -o hello hello.o
Install file: "de.mo" as "locale/de/LC_MESSAGES/hello.mo"
Install file: "en.mo" as "locale/en/LC_MESSAGES/hello.mo"
msgfmt -c -o pl.mo pl.po
Install file: "pl.mo" as "locale/pl/LC_MESSAGES/hello.mo"
scons: done building targets.
```

SCons automatically compiled the `PO` files to binary format `MO`, and the `InstallAs` lines installed these files under `locale` folder.

Your program should be now ready. You may try it as follows (linux):

```
user@host:$ LANG=en_US.UTF-8 ./hello
Welcome to beautiful world


user@host:$ LANG=de_DE.UTF-8 ./hello
Hallo Welt


user@host:$ LANG=pl_PL.UTF-8 ./hello
Witaj swiecie
```

To demonstrate the further life of translation files, let's change Polish translation (**poedit pl.po**) to `"Witaj drogi swiecie\n"`. Run **scons** to see how scons reacts to this

```
user@host:$scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
msgfmt -c -o pl.mo pl.po
Install file: "pl.mo" as "locale/pl/LC_MESSAGES/hello.mo"
scons: done building targets.
```

Now, open `hello.c` and add another one `printf` line with new message.

```
/* hello.c */
#include <stdio.h>
#include <libintl.h>
#include <locale.h>
int main(int argc, char* argv[])
{
```

```
  bindtextdomain("hello", "locale");
  setlocale(LC_ALL, "");
  textdomain("hello");
  printf(gettext("Hello world\n"));
  printf(gettext("and good bye\n"));
  return 0;
}
```

Compile project with **scons**. This time, the **msgmerge(1)** program is used by SCons to update `PO` file. The output from compilation is like:

```
user@host:$scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
Entering '/home/ptomulik/projects/tmp'
xgettext --package-name=hello --package-version=1.0 -o - hello.c
Leaving '/home/ptomulik/projects/tmp'
Writting 'messages.pot' (messages in file were outdated)
msgmerge --update de.po messages.pot
... done.
msgfmt -c -o de.mo de.po
msgmerge --update en.po messages.pot
... done.
msgfmt -c -o en.mo en.po
gcc -o hello.o -c hello.c
gcc -o hello hello.o
Install file: "de.mo" as "locale/de/LC_MESSAGES/hello.mo"
Install file: "en.mo" as "locale/en/LC_MESSAGES/hello.mo"
msgmerge --update pl.po messages.pot
... done.
msgfmt -c -o pl.mo pl.po
Install file: "pl.mo" as "locale/pl/LC_MESSAGES/hello.mo"
scons: done building targets.
```

The next example demonstrates what happens if we change the source code in such way that the internationalized messages do not change. The answer is that none of translation files (`POT`, `PO`) are touched (i.e. no content changes, no creation/modification time changed and so on). Let's append another line to the program (after the last printf), so its code becomes:

```
/* hello.c */
#include <stdio.h>
#include <libintl.h>
#include <locale.h>
int main(int argc, char* argv[])
```

```
{
  bindtextdomain("hello", "locale");
  setlocale(LC_ALL, "");
  textdomain("hello");
  printf(gettext("Hello world\n"));
  printf(gettext("and good bye\n"));
  printf("----------------\n");
  return a;
}
```

Compile the project. You'll see on your screen

```
user@host:$scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
Entering '/home/ptomulik/projects/tmp'
xgettext --package-name=hello --package-version=1.0 -o - hello.c
Leaving '/home/ptomulik/projects/tmp'
Not writting 'messages.pot' (messages in file found to be up-to-date)
gcc -o hello.o -c hello.c
gcc -o hello hello.o
scons: done building targets.
```

As you see, the internationalized messages ditn't change, so the `POT` and the rest of translation files have not even been touched.

Chapter 18. Writing Your Own Builders

Although SCons provides many useful methods for building common software products (programs, libraries, documents, etc.), you frequently want to be able to build some other type of file not supported directly by SCons. Fortunately, SCons makes it very easy to define your own `Builder` objects for any custom file types you want to build. (In fact, the SCons interfaces for creating `Builder` objects are flexible enough and easy enough to use that all of the the SCons built-in `Builder` objects are created using the mechanisms described in this section.)

18.1. Writing Builders That Execute External Commands

The simplest `Builder` to create is one that executes an external command. For example, if we want to build an output file by running the contents of the input file through a command named `foobuild`, creating that `Builder` might look like:

```
bld = Builder(action = 'foobuild < $SOURCE > $TARGET')
```

All the above line does is create a free-standing `Builder` object. The next section will show us how to actually use it.

18.2. Attaching a Builder to a `Construction Environment`

A `Builder` object isn't useful until it's attached to a `construction environment` so that we can call it to arrange for files to be built. This is done through the $BUILDERS `construction variable` in an environment. The $BUILDERS variable is a Python dictionary that maps the names by which you want to call various `Builder` objects to the objects themselves. For example, if we want to call the `Builder` we just defined by the name `Foo`, our `SConstruct` file might look like:

```
bld = Builder(action = 'foobuild < $SOURCE > $TARGET')
env = Environment(BUILDERS = {'Foo' : bld})
```

With the `Builder` attached to our `construction environment` with the name `Foo`, we can now actually call it like so:

```
env.Foo('file.foo', 'file.input')
```

Then when we run SCons it looks like:

```
% scons -Q
```

```
foobuild < file.input > file.foo
```

Note, however, that the default $BUILDERS variable in a construction environment comes with a default set of Builder objects already defined: Program, Library, etc. And when we explicitly set the $BUILDERS variable when we create the construction environment, the default Builders are no longer part of the environment:

```
bld = Builder(action = 'foobuild < $SOURCE > $TARGET')
env = Environment(BUILDERS = {'Foo' : bld})
env.Foo('file.foo', 'file.input')
env.Program('hello.c')
```

```
% scons -Q
AttributeError: 'SConsEnvironment' object has no attribute 'Program':
  File "/home/my/project/SConstruct", line 4:
    env.Program('hello.c')
```

To be able to use both our own defined Builder objects and the default Builder objects in the same construction environment, you can either add to the $BUILDERS variable using the Append function:

```
env = Environment()
bld = Builder(action = 'foobuild < $SOURCE > $TARGET')
env.Append(BUILDERS = {'Foo' : bld})
env.Foo('file.foo', 'file.input')
env.Program('hello.c')
```

Or you can explicitly set the appropriately-named key in the $BUILDERS dictionary:

```
env = Environment()
bld = Builder(action = 'foobuild < $SOURCE > $TARGET')
env['BUILDERS']['Foo'] = bld
env.Foo('file.foo', 'file.input')
env.Program('hello.c')
```

Either way, the same construction environment can then use both the newly-defined Foo Builder and the default Program Builder:

```
% scons -Q
foobuild < file.input > file.foo
cc -o hello.o -c hello.c
cc -o hello hello.o
```

18.3. Letting SCons Handle The File Suffixes

By supplying additional information when you create a `Builder`, you can let SCons add appropriate file suffixes to the target and/or the source file. For example, rather than having to specify explicitly that you want the `Foo Builder` to build the `file.foo` target file from the `file.input` source file, you can give the `.foo` and `.input` suffixes to the `Builder`, making for more compact and readable calls to the `Foo Builder`:

```
bld = Builder(action = 'foobuild < $SOURCE > $TARGET',
              suffix = '.foo',
              src_suffix = '.input')
env = Environment(BUILDERS = {'Foo' : bld})
env.Foo('file1')
env.Foo('file2')
```

```
% scons -Q
foobuild < file1.input > file1.foo
foobuild < file2.input > file2.foo
```

You can also supply a `prefix` keyword argument if it's appropriate to have SCons append a prefix to the beginning of target file names.

18.4. Builders That Execute Python Functions

In SCons, you don't have to call an external command to build a file. You can, instead, define a Python function that a `Builder` object can invoke to build your target file (or files). Such a `builder function` definition looks like:

```
def build_function(target, source, env):
    # Code to build "target" from "source"
    return None
```

The arguments of a `builder function` are:

target

A list of Node objects representing the target or targets to be built by this builder function. The file names of these target(s) may be extracted using the Python `str` function.

source

A list of Node objects representing the sources to be used by this builder function to build the targets. The file names of these source(s) may be extracted using the Python `str` function.

env

The construction environment used for building the target(s). The builder function may use any of the environment's construction variables in any way to affect how it builds the targets.

The builder function must return a 0 or None value if the target(s) are built successfully. The builder function may raise an exception or return any non-zero value to indicate that the build is unsuccessful,

Once you've defined the Python function that will build your target file, defining a Builder object for it is as simple as specifying the name of the function, instead of an external command, as the Builder's action argument:

```
def build_function(target, source, env):
    # Code to build "target" from "source"
    return None
bld = Builder(action = build_function,
              suffix = '.foo',
              src_suffix = '.input')
env = Environment(BUILDERS = {'Foo' : bld})
env.Foo('file')
```

And notice that the output changes slightly, reflecting the fact that a Python function, not an external command, is now called to build the target file:

```
% scons -Q
build_function(["file.foo"], ["file.input"])
```

18.5. Builders That Create Actions Using a Generator

SCons Builder objects can create an action "on the fly" by using a function called a generator. This provides a great deal of flexibility to construct just the right list of commands to build your target. A generator looks like:

```
def generate_actions(source, target, env, for_signature):
    return 'foobuild < %s > %s' % (target[0], source[0])
```

The arguments of a generator are:

source

A list of Node objects representing the sources to be built by the command or other action generated by this function. The file names of these source(s) may be extracted using the Python str function.

target

A list of Node objects representing the target or targets to be built by the command or other action generated by this function. The file names of these target(s) may be extracted using the Python `str` function.

env

The `construction environment` used for building the target(s). The generator may use any of the environment's construction variables in any way to determine what command or other action to return.

for_signature

A flag that specifies whether the generator is being called to contribute to a build signature, as opposed to actually executing the command.

The `generator` must return a command string or other action that will be used to build the specified target(s) from the specified source(s).

Once you've defined a `generator`, you create a `Builder` to use it by specifying the generator keyword argument instead of `action`.

```
def generate_actions(source, target, env, for_signature):
    return 'foobuild < %s > %s' % (source[0], target[0])
bld = Builder(generator = generate_actions,
              suffix = '.foo',
              src_suffix = '.input')
env = Environment(BUILDERS = {'Foo' : bld})
env.Foo('file')
```

```
% scons -Q
foobuild < file.input > file.foo
```

Note that it's illegal to specify both an `action` and a `generator` for a `Builder`.

18.6. Builders That Modify the Target or Source Lists Using an `Emitter`

SCons supports the ability for a Builder to modify the lists of target(s) from the specified source(s). You do this by defining an `emitter` function that takes as its arguments the list of the targets passed to the builder, the list of the sources passed to the builder, and the construction environment. The emitter function should return the modified lists of targets that should be built and sources from which the targets will be built.

For example, suppose you want to define a Builder that always calls a `foobuild` program, and you want to automatically add a new target file named `new_target` and a new source file named `new_source` whenever it's called. The `SConstruct` file might look like this:

```
def modify_targets(target, source, env):
    target.append('new_target')
    source.append('new_source')
    return target, source
bld = Builder(action = 'foobuild $TARGETS - $SOURCES',
              suffix = '.foo',
              src_suffix = '.input',
              emitter = modify_targets)
env = Environment(BUILDERS = {'Foo' : bld})
env.Foo('file')
```

And would yield the following output:

```
% scons -Q
foobuild file.foo new_target - file.input new_source
```

One very flexible thing that you can do is use a construction variable to specify different emitter functions for different construction variable. To do this, specify a string containing a construction variable expansion as the emitter when you call the `Builder` function, and set that construction variable to the desired emitter function in different construction environments:

```
bld = Builder(action = 'my_command $SOURCES > $TARGET',
              suffix = '.foo',
              src_suffix = '.input',
              emitter = '$MY_EMITTER')
def modify1(target, source, env):
    return target, source + ['modify1.in']
def modify2(target, source, env):
    return target, source + ['modify2.in']
env1 = Environment(BUILDERS = {'Foo' : bld},
                   MY_EMITTER = modify1)
env2 = Environment(BUILDERS = {'Foo' : bld},
                   MY_EMITTER = modify2)
env1.Foo('file1')
env2.Foo('file2')
import os
env1['ENV']['PATH'] = env2['ENV']['PATH'] + os.pathsep + os.getcwd()
env2['ENV']['PATH'] = env2['ENV']['PATH'] + os.pathsep + os.getcwd()
```

```
bld = Builder(action = 'my_command $SOURCES > $TARGET',
              suffix = '.foo',
              src_suffix = '.input',
              emitter = '$MY_EMITTER')
```

```
def modify1(target, source, env):
    return target, source + ['modify1.in']
def modify2(target, source, env):
    return target, source + ['modify2.in']
env1 = Environment(BUILDERS = {'Foo' : bld},
                   MY_EMITTER = modify1)
env2 = Environment(BUILDERS = {'Foo' : bld},
                   MY_EMITTER = modify2)
env1.Foo('file1')
env2.Foo('file2')
```

In this example, the `modify1.in` and `modify2.in` files get added to the source lists of the different commands:

```
% scons -Q
my_command file1.input modify1.in > file1.foo
my_command file2.input modify2.in > file2.foo
```

18.7. Where To Put Your Custom Builders and Tools

The `site_scons` directories give you a place to put Python modules and packages that you can import into your `SConscript` files (`site_scons`), add-on tools that can integrate into SCons (`site_scons/site_tools`), and a `site_scons/site_init.py` file that gets read before any `SConstruct` or `SConscript` file, allowing you to change SCons's default behavior.

Each system type (Windows, Mac, Linux, etc.) searches a canonical set of directories for site_scons; see the man page for details. The top-level SConstruct's site_scons dir is always searched last, and its dir is placed first in the tool path so it overrides all others.

If you get a tool from somewhere (the SCons wiki or a third party, for instance) and you'd like to use it in your project, a `site_scons` dir is the simplest place to put it. Tools come in two flavors; either a Python function that operates on an `Environment` or a Python module or package containing two functions, `exists()` and `generate()`.

A single-function Tool can just be included in your `site_scons/site_init.py` file where it will be parsed and made available for use. For instance, you could have a `site_scons/site_init.py` file like this:

```
def TOOL_ADD_HEADER(env):
   """A Tool to add a header from $HEADER to the source file"""
   add_header = Builder(action=['echo "$HEADER" > $TARGET',
                                'cat $SOURCE >> $TARGET'])
   env.Append(BUILDERS = {'AddHeader' : add_header})
```

```
env['HEADER'] = '' # set default value
```

and a SConstruct like this:

```
# Use TOOL_ADD_HEADER from site_scons/site_init.py
env=Environment(tools=['default', TOOL_ADD_HEADER], HEADER="=====")
env.AddHeader('tgt', 'src')
```

The TOOL_ADD_HEADER tool method will be called to add the AddHeader tool to the environment.

A more full-fledged tool with exists() and generate() methods can be installed either as a module in the file site_scons/site_tools/toolname.py or as a package in the directory site_scons/site_tools/toolname. In the case of using a package, the exists() and generate() are in the file site_scons/site_tools/toolname/__init__.py. (In all the above case toolname is replaced by the name of the tool.) Since site_scons/site_tools is automatically added to the head of the tool search path, any tool found there will be available to all environments. Furthermore, a tool found there will override a built-in tool of the same name, so if you need to change the behavior of a built-in tool, site_scons gives you the hook you need.

Many people have a library of utility Python functions they'd like to include in SConscripts; just put that module in site_scons/my_utils.py or any valid Python module name of your choice. For instance you can do something like this in site_scons/my_utils.py to add build_id and MakeWorkDir functions:

```
from SCons.Script import *   # for Execute and Mkdir
def build_id():
   """Return a build ID (stub version)"""
   return "100"
def MakeWorkDir(workdir):
   """Create the specified dir immediately"""
   Execute(Mkdir(workdir))
```

And then in your SConscript or any sub-SConscript anywhere in your build, you can import my_utils and use it:

```
import my_utils
print "build_id=" + my_utils.build_id()
my_utils.MakeWorkDir('/tmp/work')
```

Note that although you can put this library in site_scons/site_init.py, it is no better there than site_scons/my_utils.py since you still have to import that module into your SConscript. Also note that in order to refer to objects in the SCons namespace such as Environment or Mkdir or Execute in any file other than a SConstruct or SConscript you always need to do

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

This is true in modules in `site_scons` such as `site_scons/site_init.py` as well.

You can use any of the user- or machine-wide site dirs such as `~/.scons/site_scons` instead of `./site_scons`, or use the `--site-dir` option to point to your own dir. `site_init.py` and `site_tools` will be located under that dir. To avoid using a `site_scons` dir at all, even if it exists, use the `--no-site-dir` option.

Chapter 19. Not Writing a Builder: the `Command` Builder

Creating a `Builder` and attaching it to a `construction environment` allows for a lot of flexibility when you want to re-use actions to build multiple files of the same type. This can, however, be cumbersome if you only need to execute one specific command to build a single file (or group of files). For these situations, SCons supports a `Command Builder` that arranges for a specific action to be executed to build a specific file or files. This looks a lot like the other builders (like `Program`, `Object`, etc.), but takes as an additional argument the command to be executed to build the file:

```
env = Environment()
env.Command('foo.out', 'foo.in', "sed 's/x/y/' < $SOURCE > $TARGET")
```

When executed, SCons runs the specified command, substituting $SOURCE and $TARGET as expected:

```
% scons -Q
sed 's/x/y/' < foo.in > foo.out
```

This is often more convenient than creating a `Builder` object and adding it to the $BUILDERS variable of a `construction environment`

Note that the action you specify to the `Command Builder` can be any legal SCons `Action`, such as a Python function:

```
env = Environment()
def build(target, source, env):
    # Whatever it takes to build
    return None
env.Command('foo.out', 'foo.in', build)
```

Which executes as follows:

```
% scons -Q
build(["foo.out"], ["foo.in"])
```

Note that $SOURCE and $TARGET are expanded in the source and target as well as of SCons 1.1, so you can write:

```
env.Command('${SOURCE.basename}.out', 'foo.in', build)
```

which does the same thing as the previous example, but allows you to avoid repeating yourself.

Chapter 20. Pseudo-Builders: the AddMethod function

The `AddMethod` function is used to add a method to an environment. It's typically used to add a "pseudo-builder," a function that looks like a `Builder` but wraps up calls to multiple other `Builder`s or otherwise processes its arguments before calling one or more `Builder`s. In the following example, we want to install the program into the standard `/usr/bin` directory hierarchy, but also copy it into a local `install/bin` directory from which a package might be built:

```
def install_in_bin_dirs(env, source):
    """Install source in both bin dirs"""
    i1 = env.Install("$BIN", source)
    i2 = env.Install("$LOCALBIN", source)
    return [i1[0], i2[0]] # Return a list, like a normal builder
env = Environment(BIN='/usr/bin', LOCALBIN='#install/bin')
env.AddMethod(install_in_bin_dirs, "InstallInBinDirs")
env.InstallInBinDirs(Program('hello.c')) # installs hello in both bin dirs
```

This produces the following:

```
% scons -Q /
cc -o hello.o -c hello.c
cc -o hello hello.o
Install file: "hello" as "/usr/bin/hello"
Install file: "hello" as "install/bin/hello"
```

As mentioned, a pseudo-builder also provides more flexibility in parsing arguments than you can get with a `Builder`. The next example shows a pseudo-builder with a named argument that modifies the filename, and a separate argument for the resource file (rather than having the builder figure it out by file extension). This example also demonstrates using the global `AddMethod` function to add a method to the global Environment class, so it will be used in all subsequently created environments.

```
def BuildTestProg(env, testfile, resourcefile, testdir="tests"):
    """Build the test program;
    prepends "test_" to src and target,
    and puts target into testdir."""
    srcfile = "test_%s.c" % testfile
    target = "%s/test_%s" % (testdir, testfile)
    if env['PLATFORM'] == 'win32':
        resfile = env.RES(resourcefile)
        p = env.Program(target, [srcfile, resfile])
    else:
        p = env.Program(target, srcfile)
    return p
AddMethod(Environment, BuildTestProg)
```

```
env = Environment()
env.BuildTestProg('stuff', resourcefile='res.rc')
```

This produces the following on Linux:

```
% scons -Q
cc -o test_stuff.o -c test_stuff.c
cc -o tests/test_stuff test_stuff.o
```

And the following on Windows:

```
C:\>scons -Q
rc /fores.res res.rc
cl /Fotest_stuff.obj /c test_stuff.c /nologo
link /nologo /OUT:tests\test_stuff.exe test_stuff.obj res.res
embedManifestExeCheck(target, source, env)
```

Using `AddMethod` is better than just adding an instance method to a `construction environment` because it gets called as a proper method, and because `AddMethod` provides for copying the method to any clones of the `construction environment` instance.

Chapter 21. Writing Scanners

SCons has built-in scanners that know how to look in C, Fortran and IDL source files for information about other files that targets built from those files depend on--for example, in the case of files that use the C preprocessor, the `.h` files that are specified using `#include` lines in the source. You can use the same mechanisms that SCons uses to create its built-in scanners to write scanners of your own for file types that SCons does not know how to scan "out of the box."

21.1. A Simple Scanner Example

Suppose, for example, that we want to create a simple scanner for `.foo` files. A `.foo` file contains some text that will be processed, and can include other files on lines that begin with `include` followed by a file name:

```
include filename.foo
```

Scanning a file will be handled by a Python function that you must supply. Here is a function that will use the Python `re` module to scan for the `include` lines in our example:

```
import re

include_re = re.compile(r'^include\s+(\S+)$', re.M)

def kfile_scan(node, env, path, arg):
    contents = node.get_text_contents()
    return env.File(include_re.findall(contents))
```

It is important to note that you have to return a list of File nodes from the scanner function, simple strings for the file names won't do. As in the examples we are showing here, you can use the `File` function of your current Environment in order to create nodes on the fly from a sequence of file names with relative paths.

The scanner function must accept the four specified arguments and return a list of implicit dependencies. Presumably, these would be dependencies found from examining the contents of the file, although the function can perform any manipulation at all to generate the list of dependencies.

node

An SCons node object representing the file being scanned. The path name to the file can be used by converting the node to a string using the `str()` function, or an internal SCons `get_text_contents()` object method can be used to fetch the contents.

env

The construction environment in effect for this scan. The scanner function may choose to use construction variables from this environment to affect its behavior.

path

A list of directories that form the search path for included files for this scanner. This is how SCons handles the $CPPPATH and $LIBPATH variables.

arg

An optional argument that you can choose to have passed to this scanner function by various scanner instances.

A Scanner object is created using the `Scanner` function, which typically takes an `skeys` argument to associate the type of file suffix with this scanner. The Scanner object must then be associated with the $SCANNERS construction variable of a construction environment, typically by using the `Append` method:

```
kscan = Scanner(function = kfile_scan,
                skeys = ['.k'])
env.Append(SCANNERS = kscan)
```

When we put it all together, it looks like:

```
import re

include_re = re.compile(r'^include\s+(\S+)$', re.M)

def kfile_scan(node, env, path):
    contents = node.get_text_contents()
    includes = include_re.findall(contents)
    return env.File(includes)

kscan = Scanner(function = kfile_scan,
                skeys = ['.k'])

env = Environment(ENV = {'PATH' : '/usr/local/bin'})
env.Append(SCANNERS = kscan)

env.Command('foo', 'foo.k', 'kprocess < $SOURCES > $TARGET')
```

21.2. Adding a search path to a scanner: `FindPathDirs`

Many scanners need to search for included files or dependencies using a path variable; this is how $CPPPATH and $LIBPATH work. The path to search is passed to your scanner as the `path` argument.

Path variables may be lists of nodes, semicolon-separated strings, or even contain SCons variables which need to be expanded. Fortunately, SCons provides the `FindPathDirs` function which itself returns a function to expand a given path (given as a SCons construction variable name) to a list of paths at the time the scanner is called. Deferring evaluation until that point allows, for instance, the path to contain $TARGET references which differ for each file scanned.

Using `FindPathDirs` is quite easy. Continuing the above example, using KPATH as the construction variable with the search path (analogous to $CPPPATH), we just modify the `Scanner` constructor call to include a path keyword arg:

```
kscan = Scanner(function = kfile_scan,
                skeys = ['.k'],
                path=FindPathDirs('KPATH'))
```

FindPathDirs returns a callable object that, when called, will essentially expand the elements in env['KPATH'] and tell the scanner to search in those dirs. It will also properly add related repository and variant dirs to the search list. As a side note, the returned method stores the path in an efficient way so lookups are fast even when variable substitutions may be needed. This is important since many files get scanned in a typical build.

Chapter 22. Building From Code Repositories

Often, a software project will have one or more central repositories, directory trees that contain source code, or derived files, or both. You can eliminate additional unnecessary rebuilds of files by having SCons use files from one or more code repositories to build files in your local build tree.

22.1. The `Repository` Method

It's often useful to allow multiple programmers working on a project to build software from source files and/or derived files that are stored in a centrally-accessible repository, a directory copy of the source code tree. (Note that this is not the sort of repository maintained by a source code management system like BitKeeper, CVS, or Subversion.) You use the `Repository` method to tell SCons to search one or more central code repositories (in order) for any source files and derived files that are not present in the local build tree:

```
env = Environment()
env.Program('hello.c')
Repository('/usr/repository1', '/usr/repository2')
```

Multiple calls to the `Repository` method will simply add repositories to the global list that SCons maintains, with the exception that SCons will automatically eliminate the current directory and any non-existent directories from the list.

22.2. Finding source files in repositories

The above example specifies that SCons will first search for files under the `/usr/repository1` tree and next under the `/usr/repository2` tree. SCons expects that any files it searches for will be found in the same position relative to the top-level directory. In the above example, if the `hello.c` file is not found in the local build tree, SCons will search first for a `/usr/repository1/hello.c` file and then for a `/usr/repository2/hello.c` file to use in its place.

So given the `SConstruct` file above, if the `hello.c` file exists in the local build directory, SCons will rebuild the `hello` program as normal:

```
% scons -Q
cc -o hello.o -c hello.c
cc -o hello hello.o
```

If, however, there is no local `hello.c` file, but one exists in `/usr/repository1`, SCons will recompile the `hello` program from the source file it finds in the repository:

```
% scons -Q
cc -o hello.o -c /usr/repository1/hello.c
cc -o hello hello.o
```

And similarly, if there is no local `hello.c` file and no `/usr/repository1/hello.c`, but one exists in `/usr/repository2`:

```
% scons -Q
cc -o hello.o -c /usr/repository2/hello.c
cc -o hello hello.o
```

22.3. Finding `#include` files in repositories

We've already seen that SCons will scan the contents of a source file for `#include` file names and realize that targets built from that source file also depend on the `#include` file(s). For each directory in the $CPPPATH list, SCons will actually search the corresponding directories in any repository trees and establish the correct dependencies on any `#include` files that it finds in repository directory.

Unless the C compiler also knows about these directories in the repository trees, though, it will be unable to find the `#include` files. If, for example, the `hello.c` file in our previous example includes the `hello.h` in its current directory, and the `hello.h` only exists in the repository:

```
% scons -Q
cc -o hello.o -c hello.c
hello.c:1: hello.h: No such file or directory
```

In order to inform the C compiler about the repositories, SCons will add appropriate `-I` flags to the compilation commands for each directory in the $CPPPATH list. So if we add the current directory to the construction environment $CPPPATH like so:

```
env = Environment(CPPPATH = ['.'])
env.Program('hello.c')
Repository('/usr/repository1')
```

Then re-executing SCons yields:

```
% scons -Q
cc -o hello.o -c -I. -I/usr/repository1 hello.c
cc -o hello hello.o
```

The order of the -I options replicates, for the C preprocessor, the same repository-directory search path that SCons uses for its own dependency analysis. If there are multiple repositories and multiple $CPPPATH directories, SCons will add the repository directories to the beginning of each $CPPPATH directory, rapidly multiplying the number of -I flags. If, for example, the $CPPPATH contains three directories (and shorter repository path names!):

```
env = Environment(CPPPATH = ['dir1', 'dir2', 'dir3'])
env.Program('hello.c')
Repository('/r1', '/r2')
```

Then we'll end up with nine -I options on the command line, three (for each of the $CPPPATH directories) times three (for the local directory plus the two repositories):

```
% scons -Q
cc -o hello.o -c -Idir1 -I/r1/dir1 -I/r2/dir1 -Idir2 -I/r1/dir2 -I/r2/dir2 -Idir3 -
cc -o hello hello.o
```

22.3.1. Limitations on #include files in repositories

SCons relies on the C compiler's -I options to control the order in which the preprocessor will search the repository directories for #include files. This causes a problem, however, with how the C preprocessor handles #include lines with the file name included in double-quotes.

As we've seen, SCons will compile the hello.c file from the repository if it doesn't exist in the local directory. If, however, the hello.c file in the repository contains a #include line with the file name in double quotes:

```
#include "hello.h"
int
main(int argc, char *argv[])
{
    printf(HELLO_MESSAGE);
    return (0);
}
```

Then the C preprocessor will *always* use a hello.h file from the repository directory first, even if there is a hello.h file in the local directory, despite the fact that the command line specifies -I as the first option:

```
% scons -Q
cc -o hello.o -c -I. -I/usr/repository1 /usr/repository1/hello.c
cc -o hello hello.o
```

This behavior of the C preprocessor--always search for a `#include` file in double-quotes first in the same directory as the source file, and only then search the `-I`--can not, in general, be changed. In other words, it's a limitation that must be lived with if you want to use code repositories in this way. There are three ways you can possibly work around this C preprocessor behavior:

1. Some modern versions of C compilers do have an option to disable or control this behavior. If so, add that option to $CFLAGS (or $CXXFLAGS or both) in your construction environment(s). Make sure the option is used for all construction environments that use C preprocessing!
2. Change all occurrences of `#include "file.h"` to `#include <file.h>`. Use of `#include` with angle brackets does not have the same behavior--the `-I` directories are searched first for `#include` files--which gives SCons direct control over the list of directories the C preprocessor will search.
3. Require that everyone working with compilation from repositories check out and work on entire directories of files, not individual files. (If you use local wrapper scripts around your source code control system's command, you could add logic to enforce this restriction there.

22.4. Finding the `SConstruct` file in repositories

SCons will also search in repositories for the `SConstruct` file and any specified `SConscript` files. This poses a problem, though: how can SCons search a repository tree for an `SConstruct` file if the `SConstruct` file itself contains the information about the pathname of the repository? To solve this problem, SCons allows you to specify repository directories on the command line using the `-Y` option:

```
% scons -Q -Y /usr/repository1 -Y /usr/repository2
```

When looking for source or derived files, SCons will first search the repositories specified on the command line, and then search the repositories specified in the `SConstruct` or `SConscript` files.

22.5. Finding derived files in repositories

If a repository contains not only source files, but also derived files (such as object files, libraries, or executables), SCons will perform its normal MD5 signature calculation to decide if a derived file in a repository is up-to-date, or the derived file must be rebuilt in the local build directory. For the SCons signature calculation to work correctly, a repository tree must contain the `.sconsign` files that SCons uses to keep track of signature information.

Usually, this would be done by a build integrator who would run SCons in the repository to create all of its derived files and `.sconsign` files, or who would run SCons in a separate build directory and copy the resulting tree to the desired repository:

```
% cd /usr/repository1
% scons -Q
```

```
cc -o file1.o -c file1.c
cc -o file2.o -c file2.c
cc -o hello.o -c hello.c
cc -o hello hello.o file1.o file2.o
```

(Note that this is safe even if the `SConstruct` file lists `/usr/repository1` as a repository, because SCons will remove the current build directory from its repository list for that invocation.)

Now, with the repository populated, we only need to create the one local source file we're interested in working with at the moment, and use the `-Y` option to tell SCons to fetch any other files it needs from the repository:

```
% cd $HOME/build
% edit hello.c
% scons -Q -Y /usr/repository1
cc -c -o hello.o hello.c
cc -o hello hello.o /usr/repository1/file1.o /usr/repository1/file2.o
```

Notice that SCons realizes that it does not need to rebuild local copies `file1.o` and `file2.o` files, but instead uses the already-compiled files from the repository.

22.6. Guaranteeing local copies of files

If the repository tree contains the complete results of a build, and we try to build from the repository without any files in our local tree, something moderately surprising happens:

```
% mkdir $HOME/build2
% cd $HOME/build2
% scons -Q -Y /usr/all/repository hello
scons: 'hello' is up-to-date.
```

Why does SCons say that the `hello` program is up-to-date when there is no `hello` program in the local build directory? Because the repository (not the local directory) contains the up-to-date `hello` program, and SCons correctly determines that nothing needs to be done to rebuild that up-to-date copy of the file.

There are, however, many times when you want to ensure that a local copy of a file always exists. A packaging or testing script, for example, may assume that certain generated files exist locally. To tell SCons to make a copy of any up-to-date repository file in the local build directory, use the `Local` function:

```
env = Environment()
hello = env.Program('hello.c')
```

```
Local(hello)
```

If we then run the same command, SCons will make a local copy of the program from the repository copy, and tell you that it is doing so:

```
% scons -Y /usr/all/repository hello
Local copy of hello from /usr/all/repository/hello
scons: 'hello' is up-to-date.
```

(Notice that, because the act of making the local copy is not considered a "build" of the `hello` file, SCons still reports that it is up-to-date.)

Chapter 23. Multi-Platform Configuration (Autoconf Functionality)

SCons has integrated support for multi-platform build configuration similar to that offered by GNU Autoconf, such as figuring out what libraries or header files are available on the local system. This section describes how to use this SCons feature.

> **Note:** This chapter is still under development, so not everything is explained as well as it should be. See the SCons man page for additional information.

23.1. `Configure Contexts`

The basic framework for multi-platform build configuration in SCons is to attach a `configure context` to a construction environment by calling the `Configure` function, perform a number of checks for libraries, functions, header files, etc., and to then call the configure context's `Finish` method to finish off the configuration:

```
env = Environment()
conf = Configure(env)
# Checks for libraries, header files, etc. go here!
env = conf.Finish()
```

SCons provides a number of basic checks, as well as a mechanism for adding your own custom checks.

Note that SCons uses its own dependency mechanism to determine when a check needs to be run--that is, SCons does not run the checks every time it is invoked, but caches the values returned by previous checks and uses the cached values unless something has changed. This saves a tremendous amount of developer time while working on cross-platform build issues.

The next sections describe the basic checks that SCons supports, as well as how to add your own custom checks.

23.2. Checking for the Existence of Header Files

Testing the existence of a header file requires knowing what language the header file is. A configure context has a `CheckCHeader` method that checks for the existence of a C header file:

```
env = Environment()
```

```
conf = Configure(env)
if not conf.CheckCHeader('math.h'):
    print 'Math.h must be installed!'
    Exit(1)
if conf.CheckCHeader('foo.h'):
    conf.env.Append('-DHAS_FOO_H')
env = conf.Finish()
```

Note that you can choose to terminate the build if a given header file doesn't exist, or you can modify the construction environment based on the existence of a header file.

If you need to check for the existence a C++ header file, use the `CheckCXXHeader` method:

```
env = Environment()
conf = Configure(env)
if not conf.CheckCXXHeader('vector.h'):
    print 'vector.h must be installed!'
    Exit(1)
env = conf.Finish()
```

23.3. Checking for the Availability of a Function

Check for the availability of a specific function using the `CheckFunc` method:

```
env = Environment()
conf = Configure(env)
if not conf.CheckFunc('strcpy'):
    print 'Did not find strcpy(), using local version'
    conf.env.Append(CPPDEFINES = '-Dstrcpy=my_local_strcpy')
env = conf.Finish()
```

23.4. Checking for the Availability of a Library

Check for the availability of a library using the `CheckLib` method. You only specify the basename of the library, you don't need to add a `lib` prefix or a `.a` or `.lib` suffix:

```
env = Environment()
conf = Configure(env)
if not conf.CheckLib('m'):
    print 'Did not find libm.a or m.lib, exiting!'
    Exit(1)
```

```
env = conf.Finish()
```

Because the ability to use a library successfully often depends on having access to a header file that describes the library's interface, you can check for a library *and* a header file at the same time by using the `CheckLibWithHeader` method:

```
env = Environment()
conf = Configure(env)
if not conf.CheckLibWithHeader('m', 'math.h', 'c'):
    print 'Did not find libm.a or m.lib, exiting!'
    Exit(1)
env = conf.Finish()
```

This is essentially shorthand for separate calls to the `CheckHeader` and `CheckLib` functions.

23.5. Checking for the Availability of a `typedef`

Check for the availability of a `typedef` by using the `CheckType` method:

```
env = Environment()
conf = Configure(env)
if not conf.CheckType('off_t'):
    print 'Did not find off_t typedef, assuming int'
    conf.env.Append(CCFLAGS = '-Doff_t=int')
env = conf.Finish()
```

You can also add a string that will be placed at the beginning of the test file that will be used to check for the `typedef`. This provide a way to specify files that must be included to find the `typedef`:

```
env = Environment()
conf = Configure(env)
if not conf.CheckType('off_t', '#include <sys/types.h>\n'):
    print 'Did not find off_t typedef, assuming int'
    conf.env.Append(CCFLAGS = '-Doff_t=int')
env = conf.Finish()
```

23.6. Adding Your Own Custom Checks

A custom check is a Python function that checks for a certain condition to exist on the running system, usually using methods that SCons supplies to take care of the details of checking whether a compilation

succeeds, a link succeeds, a program is runnable, etc. A simple custom check for the existence of a specific library might look as follows:

```
mylib_test_source_file = """
#include <mylib.h>
int main(int argc, char **argv)
{
    MyLibrary mylib(argc, argv);
    return 0;
}
"""

def CheckMyLibrary(context):
    context.Message('Checking for MyLibrary...')
    result = context.TryLink(mylib_test_source_file, '.c')
    context.Result(result)
    return result
```

The `Message` and `Result` methods should typically begin and end a custom check to let the user know what's going on: the `Message` call prints the specified message (with no trailing newline) and the `Result` call prints `yes` if the check succeeds and `no` if it doesn't. The `TryLink` method actually tests for whether the specified program text will successfully link.

(Note that a custom check can modify its check based on any arguments you choose to pass it, or by using or modifying the configure context environment in the `context.env` attribute.)

This custom check function is then attached to the `configure context` by passing a dictionary to the `Configure` call that maps a name of the check to the underlying function:

```
env = Environment()
conf = Configure(env, custom_tests = {'CheckMyLibrary' : CheckMyLibrary})
```

You'll typically want to make the check and the function name the same, as we've done here, to avoid potential confusion.

We can then put these pieces together and actually call the `CheckMyLibrary` check as follows:

```
mylib_test_source_file = """
#include <mylib.h>
int main(int argc, char **argv)
{
    MyLibrary mylib(argc, argv);
    return 0;
}
"""

def CheckMyLibrary(context):
```

```
    context.Message('Checking for MyLibrary... ')
    result = context.TryLink(mylib_test_source_file, '.c')
    context.Result(result)
    return result

env = Environment()
conf = Configure(env, custom_tests = {'CheckMyLibrary' : CheckMyLibrary})
if not conf.CheckMyLibrary():
    print 'MyLibrary is not installed!'
    Exit(1)
env = conf.Finish()

# We would then add actual calls like Program() to build
# something using the "env" construction environment.
```

If MyLibrary is not installed on the system, the output will look like:

```
% scons
scons: Reading SConscript file ...
Checking for MyLibrary... failed
MyLibrary is not installed!
```

If MyLibrary is installed, the output will look like:

```
% scons
scons: Reading SConscript file ...
Checking for MyLibrary... failed
scons: done reading SConscript
scons: Building targets ...
    .
    .
    .
```

23.7. Not Configuring When Cleaning Targets

Using multi-platform configuration as described in the previous sections will run the configuration commands even when invoking **scons -c** to clean targets:

```
% scons -Q -c
Checking for MyLibrary... yes
Removed foo.o
Removed foo
```

Although running the platform checks when removing targets doesn't hurt anything, it's usually unnecessary. You can avoid this by using the `GetOption` method to check whether the `-c` (clean) option has been invoked on the command line:

```
env = Environment()
if not env.GetOption('clean'):
    conf = Configure(env, custom_tests = {'CheckMyLibrary' : CheckMyLibrary})
    if not conf.CheckMyLibrary():
        print 'MyLibrary is not installed!'
        Exit(1)
    env = conf.Finish()
```

```
% scons -Q -c
Removed foo.o
Removed foo
```

Chapter 24. Caching Built Files

On multi-developer software projects, you can sometimes speed up every developer's builds a lot by allowing them to share the derived files that they build. SCons makes this easy, as well as reliable.

24.1. Specifying the Shared Cache Directory

To enable sharing of derived files, use the `CacheDir` function in any `SConscript` file:

```
CacheDir('/usr/local/build_cache')
```

Note that the directory you specify must already exist and be readable and writable by all developers who will be sharing derived files. It should also be in some central location that all builds will be able to access. In environments where developers are using separate systems (like individual workstations) for builds, this directory would typically be on a shared or NFS-mounted file system.

Here's what happens: When a build has a `CacheDir` specified, every time a file is built, it is stored in the shared cache directory along with its MD5 build signature. [1] On subsequent builds, before an action is invoked to build a file, SCons will check the shared cache directory to see if a file with the exact same build signature already exists. If so, the derived file will not be built locally, but will be copied into the local build directory from the shared cache directory, like so:

```
% scons -Q
cc -o hello.o -c hello.c
cc -o hello hello.o
% scons -Q -c
Removed hello.o
Removed hello
% scons -Q
Retrieved `hello.o' from cache
Retrieved `hello' from cache
```

Note that the `CacheDir` feature still calculates MD5 build sigantures for the shared cache file names even if you configure SCons to use timestamps to decide if files are up to date. (See the Chapter 6 chapter for information about the `Decider` function.) Consequently, using `CacheDir` may reduce or eliminate any potential performance improvements from using timestamps for up-to-date decisions.

24.2. Keeping Build Output Consistent

One potential drawback to using a shared cache is that the output printed by SCons can be inconsistent from invocation to invocation, because any given file may be rebuilt one time and retrieved from the shared cache the next time. This can make analyzing build output more difficult, especially for automated scripts that expect consistent output each time.

If, however, you use the `--cache-show` option, SCons will print the command line that it *would* have executed to build the file, even when it is retrieving the file from the shared cache. This makes the build output consistent every time the build is run:

```
% scons -Q
cc -o hello.o -c hello.c
cc -o hello hello.o
% scons -Q -c
Removed hello.o
Removed hello
% scons -Q --cache-show
cc -o hello.o -c hello.c
cc -o hello hello.o
```

The trade-off, of course, is that you no longer know whether or not SCons has retrieved a derived file from cache or has rebuilt it locally.

24.3. Not Using the Shared Cache for Specific Files

You may want to disable caching for certain specific files in your configuration. For example, if you only want to put executable files in a central cache, but not the intermediate object files, you can use the `NoCache` function to specify that the object files should not be cached:

```
env = Environment()
obj = env.Object('hello.c')
env.Program('hello.c')
CacheDir('cache')
NoCache('hello.o')
```

Then when you run `scons` after cleaning the built targets, it will recompile the object file locally (since it doesn't exist in the shared cache directory), but still realize that the shared cache directory contains an up-to-date executable program that can be retrieved instead of re-linking:

```
% scons -Q
cc -o hello.o -c hello.c
cc -o hello hello.o
% scons -Q -c
```

```
Removed hello.o
Removed hello
% scons -Q
cc -o hello.o -c hello.c
Retrieved 'hello' from cache
```

24.4. Disabling the Shared Cache

Retrieving an already-built file from the shared cache is usually a significant time-savings over rebuilding the file, but how much of a savings (or even whether it saves time at all) can depend a great deal on your system or network configuration. For example, retrieving cached files from a busy server over a busy network might end up being slower than rebuilding the files locally.

In these cases, you can specify the `--cache-disable` command-line option to tell SCons to not retrieve already-built files from the shared cache directory:

```
% scons -Q
cc -o hello.o -c hello.c
cc -o hello hello.o
% scons -Q -c
Removed hello.o
Removed hello
% scons -Q
Retrieved 'hello.o' from cache
Retrieved 'hello' from cache
% scons -Q -c
Removed hello.o
Removed hello
% scons -Q --cache-disable
cc -o hello.o -c hello.c
cc -o hello hello.o
```

24.5. Populating a Shared Cache With Already-Built Files

Sometimes, you may have one or more derived files already built in your local build tree that you wish to make available to other people doing builds. For example, you may find it more effective to perform integration builds with the cache disabled (per the previous section) and only populate the shared cache directory with the built files after the integration build has completed successfully. This way, the cache will only get filled up with derived files that are part of a complete, successful build not with files that might be later overwritten while you debug integration problems.

In this case, you can use the the `--cache-force` option to tell SCons to put all derived files in the cache, even if the files already exist in your local tree from having been built by a previous invocation:

```
% scons -Q --cache-disable
cc -o hello.o -c hello.c
cc -o hello hello.o
% scons -Q -c
Removed hello.o
Removed hello
% scons -Q --cache-disable
cc -o hello.o -c hello.c
cc -o hello hello.o
% scons -Q --cache-force
scons: `.' is up to date.
% scons -Q
scons: `.' is up to date.
```

Notice how the above sample run demonstrates that the `--cache-disable` option avoids putting the built `hello.o` and `hello` files in the cache, but after using the `--cache-force` option, the files have been put in the cache for the next invocation to retrieve.

24.6. Minimizing Cache Contention: the `--random` Option

If you allow multiple builds to update the shared cache directory simultaneously, two builds that occur at the same time can sometimes start "racing" with one another to build the same files in the same order. If, for example, you are linking multiple files into an executable program:

```
Program('prog',
        ['f1.c', 'f2.c', 'f3.c', 'f4.c', 'f5.c'])
```

SCons will normally build the input object files on which the program depends in their normal, sorted order:

```
% scons -Q
cc -o f1.o -c f1.c
cc -o f2.o -c f2.c
cc -o f3.o -c f3.c
cc -o f4.o -c f4.c
cc -o f5.o -c f5.c
cc -o prog f1.o f2.o f3.o f4.o f5.o
```

But if two such builds take place simultaneously, they may each look in the cache at nearly the same time and both decide that `f1.o` must be rebuilt and pushed into the shared cache directory, then both decide that `f2.o` must be rebuilt (and pushed into the shared cache directory), then both decide that `f3.o` must

be rebuilt... This won't cause any actual build problems--both builds will succeed, generate correct output files, and populate the cache--but it does represent wasted effort.

To alleviate such contention for the cache, you can use the `--random` command-line option to tell SCons to build dependencies in a random order:

```
% scons -Q --random
cc -o f3.o -c f3.c
cc -o f1.o -c f1.c
cc -o f5.o -c f5.c
cc -o f2.o -c f2.c
cc -o f4.o -c f4.c
cc -o prog f1.o f2.o f3.o f4.o f5.o
```

Multiple builds using the `--random` option will usually build their dependencies in different, random orders, which minimizes the chances for a lot of contention for same-named files in the shared cache directory. Multiple simultaneous builds might still race to try to build the same target file on occasion, but long sequences of inefficient contention should be rare.

Note, of course, the `--random` option will cause the output that SCons prints to be inconsistent from invocation to invocation, which may be an issue when trying to compare output from different build runs.

If you want to make sure dependencies will be built in a random order without having to specify the `--random` on very command line, you can use the `SetOption` function to set the `random` option within any `SConscript` file:

```
Program('prog',
        ['f1.c', 'f2.c', 'f3.c', 'f4.c', 'f5.c'])

SetOption('random', 1)
Program('prog',
        ['f1.c', 'f2.c', 'f3.c', 'f4.c', 'f5.c'])
```

Notes

1. Actually, the MD5 signature is used as the name of the file in the shared cache directory in which the contents are stored.

Chapter 25. Alias Targets

We've already seen how you can use the `Alias` function to create a target named `install`:

```
env = Environment()
hello = env.Program('hello.c')
env.Install('/usr/bin', hello)
env.Alias('install', '/usr/bin')
```

You can then use this alias on the command line to tell SCons more naturally that you want to install files:

```
% scons -Q install
cc -o hello.o -c hello.c
cc -o hello hello.o
Install file: "hello" as "/usr/bin/hello"
```

Like other `Builder` methods, though, the `Alias` method returns an object representing the alias being built. You can then use this object as input to anothother `Builder`. This is especially useful if you use such an object as input to another call to the `Alias` `Builder`, allowing you to create a hierarchy of nested aliases:

```
env = Environment()
p = env.Program('foo.c')
l = env.Library('bar.c')
env.Install('/usr/bin', p)
env.Install('/usr/lib', l)
ib = env.Alias('install-bin', '/usr/bin')
il = env.Alias('install-lib', '/usr/lib')
env.Alias('install', [ib, il])
```

This example defines separate `install`, `install-bin`, and `install-lib` aliases, allowing you finer control over what gets installed:

```
% scons -Q install-bin
cc -o foo.o -c foo.c
cc -o foo foo.o
Install file: "foo" as "/usr/bin/foo"
% scons -Q install-lib
cc -o bar.o -c bar.c
ar rc libbar.a bar.o
ranlib libbar.a
Install file: "libbar.a" as "/usr/lib/libbar.a"
% scons -Q -c /
Removed foo.o
Removed foo
Removed /usr/bin/foo
```

```
Removed bar.o
Removed libbar.a
Removed /usr/lib/libbar.a
% scons -Q install
cc -o foo.o -c foo.c
cc -o foo foo.o
Install file: "foo" as "/usr/bin/foo"
cc -o bar.o -c bar.c
ar rc libbar.a bar.o
ranlib libbar.a
Install file: "libbar.a" as "/usr/lib/libbar.a"
```

Chapter 26. Java Builds

So far, we've been using examples of building C and C++ programs to demonstrate the features of SCons. SCons also supports building Java programs, but Java builds are handled slightly differently, which reflects the ways in which the Java compiler and tools build programs differently than other languages' tool chains.

26.1. Building Java Class Files: the `Java` Builder

The basic activity when programming in Java, of course, is to take one or more `.java` files containing Java source code and to call the Java compiler to turn them into one or more `.class` files. In SCons, you do this by giving the `Java` Builder a target directory in which to put the `.class` files, and a source directory that contains the `.java` files:

```
Java('classes', 'src')
```

If the `src` directory contains three `.java` source files, then running SCons might look like this:

```
% scons -Q
javac -d classes -sourcepath src src/Example1.java src/Example2.java src/Example3.
```

SCons will actually search the `src` directory tree for all of the `.java` files. The Java compiler will then create the necessary class files in the `classes` subdirectory, based on the class names found in the `.java` files.

26.2. How SCons Handles Java Dependencies

In addition to searching the source directory for `.java` files, SCons actually runs the `.java` files through a stripped-down Java parser that figures out what classes are defined. In other words, SCons knows, without you having to tell it, what `.class` files will be produced by the javac call. So our one-liner example from the preceding section:

```
Java('classes', 'src')
```

Will not only tell you reliably that the `.class` files in the `classes` subdirectory are up-to-date:

```
% scons -Q
javac -d classes -sourcepath src src/Example1.java src/Example2.java src/Example3.
% scons -Q classes
scons: `classes' is up to date.
```

But it will also remove all of the generated `.class` files, even for inner classes, without you having to specify them manually. For example, if our `Example1.java` and `Example3.java` files both define additional classes, and the class defined in `Example2.java` has an inner class, running **`scons -c`** will clean up all of those `.class` files as well:

```
% scons -Q
javac -d classes -sourcepath src src/Example1.java src/Example2.java src/Example3.j
% scons -Q -c classes
Removed classes/Example1.class
Removed classes/AdditionalClass1.class
Removed classes/Example2$Inner2.class
Removed classes/Example2.class
Removed classes/Example3.class
Removed classes/AdditionalClass3.class
```

To ensure correct handling of `.class` dependencies in all cases, you need to tell SCons which Java version is being used. This is needed because Java 1.5 changed the `.class` file names for nested anonymous inner classes. Use the `JAVAVERSION` construction variable to specify the version in use. With Java 1.6, the one-liner example can then be defined like this:

```
Java('classes', 'src', JAVAVERSION='1.6')
```

See `JAVAVERSION` in the man page for more information.

26.3. Building Java Archive (`.jar`) Files: the `Jar` Builder

After building the class files, it's common to collect them into a Java archive (`.jar`) file, which you do by calling the `Jar` Builder method. If you want to just collect all of the class files within a subdirectory, you can just specify that subdirectory as the `Jar` source:

```
Java(target = 'classes', source = 'src')
Jar(target = 'test.jar', source = 'classes')
```

SCons will then pass that directory to the jar command, which will collect all of the underlying `.class` files:

```
% scons -Q
javac -d classes -sourcepath src src/Example1.java src/Example2.java src/Example3.j
jar cf test.jar classes
```

If you want to keep all of the `.class` files for multiple programs in one location, and only archive some of them in each `.jar` file, you can pass the `Jar` builder a list of files as its source. It's extremely simple to create multiple `.jar` files this way, using the lists of target class files created by calls to the `Java` builder as sources to the various `Jar` calls:

```
prog1_class_files = Java(target = 'classes', source = 'prog1')
prog2_class_files = Java(target = 'classes', source = 'prog2')
Jar(target = 'prog1.jar', source = prog1_class_files)
Jar(target = 'prog2.jar', source = prog2_class_files)
```

This will then create `prog1.jar` and `prog2.jar` next to the subdirectories that contain their `.java` files:

```
% scons -Q
javac -d classes -sourcepath prog1 prog1/Example1.java prog1/Example2.java
javac -d classes -sourcepath prog2 prog2/Example3.java prog2/Example4.java
jar cf prog1.jar -C classes Example1.class -C classes Example2.class
jar cf prog2.jar -C classes Example3.class -C classes Example4.class
```

26.4. Building C Header and Stub Files: the `JavaH` Builder

You can generate C header and source files for implementing native methods, by using the `JavaH` Builder. There are several ways of using the `JavaH` Builder. One typical invocation might look like:

```
classes = Java(target = 'classes', source = 'src/pkg/sub')
JavaH(target = 'native', source = classes)
```

The source is a list of class files generated by the call to the `Java` Builder, and the target is the output directory in which we want the C header files placed. The target gets converted into the `-d` when SCons runs javah:

```
% scons -Q
javac -d classes -sourcepath src/pkg/sub src/pkg/sub/Example1.java src/pkg/sub/Exar
javah -d native -classpath classes pkg.sub.Example1 pkg.sub.Example2 pkg.sub.Exampl
```

In this case, the call to javah will generate the header files `native/pkg_sub_Example1.h`, `native/pkg_sub_Example2.h` and `native/pkg_sub_Example3.h`. Notice that SCons remembered that the class files were generated with a target directory of `classes`, and that it then specified that target directory as the `-classpath` option to the call to javah.

Although it's more convenient to use the list of class files returned by the Java Builder as the source of a call to the JavaH Builder, you *can* specify the list of class files by hand, if you prefer. If you do, you need to set the $JAVACLASSDIR construction variable when calling JavaH:

```
Java(target = 'classes', source = 'src/pkg/sub')
class_file_list = ['classes/pkg/sub/Example1.class',
                   'classes/pkg/sub/Example2.class',
                   'classes/pkg/sub/Example3.class']
JavaH(target = 'native', source = class_file_list, JAVACLASSDIR = 'classes')
```

The $JAVACLASSDIR value then gets converted into the -classpath when SCons runs javah:

```
% scons -Q
javac -d classes -sourcepath src/pkg/sub src/pkg/sub/Example1.java src/pkg/sub/Exar
javah -d native -classpath classes pkg.sub.Example1 pkg.sub.Example2 pkg.sub.Exampl
```

Lastly, if you don't want a separate header file generated for each source file, you can specify an explicit File Node as the target of the JavaH Builder:

```
classes = Java(target = 'classes', source = 'src/pkg/sub')
JavaH(target = File('native.h'), source = classes)
```

Because SCons assumes by default that the target of the JavaH builder is a directory, you need to use the File function to make sure that SCons doesn't create a directory named native.h. When a file is used, though, SCons correctly converts the file name into the javah -o option:

```
% scons -Q
javac -d classes -sourcepath src/pkg/sub src/pkg/sub/Example1.java src/pkg/sub/Exar
javah -o native.h -classpath classes pkg.sub.Example1 pkg.sub.Example2 pkg.sub.Exar
```

26.5. Building RMI Stub and Skeleton Class Files: the RMIC Builder

You can generate Remote Method Invocation stubs by using the RMIC Builder. The source is a list of directories, typically returned by a call to the Java Builder, and the target is an output directory where the _Stub.class and _Skel.class files will be placed:

```
classes = Java(target = 'classes', source = 'src/pkg/sub')
RMIC(target = 'outdir', source = classes)
```

As it did with the `JavaH` Builder, SCons remembers the class directory and passes it as the `-classpath` option to rmic:

```
% scons -Q
javac -d classes -sourcepath src/pkg/sub src/pkg/sub/Example1.java src/pkg/sub/Exa
rmic -d outdir -classpath classes pkg.sub.Example1 pkg.sub.Example2
```

This example would generate the files `outdir/pkg/sub/Example1_Skel.class`, `outdir/pkg/sub/Example1_Stub.class`, `outdir/pkg/sub/Example2_Skel.class` and `outdir/pkg/sub/Example2_Stub.class`.

Chapter 27. Miscellaneous Functionality

SCons supports a lot of additional functionality that doesn't readily fit into the other chapters.

27.1. Verifying the Python Version: the `EnsurePythonVersion` Function

Although the SCons code itself will run on any 2.x Python version 2.4 or later, you are perfectly free to make use of Python syntax and modules from more modern versions (for example, Python 2.5 or 2.6) when writing your `SConscript` files or your own local modules. If you do this, it's usually helpful to configure SCons to exit gracefully with an error message if it's being run with a version of Python that simply won't work with your code. This is especially true if you're going to use SCons to build source code that you plan to distribute publicly, where you can't be sure of the Python version that an anonymous remote user might use to try to build your software.

SCons provides an `EnsurePythonVersion` function for this. You simply pass it the major and minor versions numbers of the version of Python you require:

```
EnsurePythonVersion(2, 5)
```

And then SCons will exit with the following error message when a user runs it with an unsupported earlier version of Python:

```
% scons -Q
Python 2.5 or greater required, but you have Python 2.3.6
```

27.2. Verifying the SCons Version: the `EnsureSConsVersion` Function

You may, of course, write your `SConscript` files to use features that were only added in recent versions of SCons. When you publicly distribute software that is built using SCons, it's helpful to have SCons verify the version being used and exit gracefully with an error message if the user's version of SCons won't work with your `SConscript` files. SCons provides an `EnsureSConsVersion` function that verifies the version of SCons in the same the `EnsurePythonVersion` function verifies the version of Python, by passing in the major and minor versions numbers of the version of SCons you require:

```
EnsureSConsVersion(1, 0)
```

And then SCons will exit with the following error message when a user runs it with an unsupported earlier version of SCons:

```
% scons -Q
SCons 1.0 or greater required, but you have SCons 0.98.5
```

27.3. Explicitly Terminating SCons While Reading `SConscript` Files: the `Exit` Function

SCons supports an `Exit` function which can be used to terminate SCons while reading the `SConscript` files, usually because you've detected a condition under which it doesn't make sense to proceed:

```
if ARGUMENTS.get('FUTURE'):
    print "The FUTURE option is not supported yet!"
    Exit(2)
env = Environment()
env.Program('hello.c')
```

```
% scons -Q FUTURE=1
The FUTURE option is not supported yet!
% scons -Q
cc -o hello.o -c hello.c
cc -o hello hello.o
```

The `Exit` function takes as an argument the (numeric) exit status that you want SCons to exit with. If you don't specify a value, the default is to exit with `0`, which indicates successful execution.

Note that the `Exit` function is equivalent to calling the Python `sys.exit` function (which the it actually calls), but because `Exit` is a SCons function, you don't have to import the Python `sys` module to use it.

27.4. Searching for Files: the `FindFile` Function

The `FindFile` function searches for a file in a list of directories. If there is only one directory, it can be given as a simple string. The function returns a File node if a matching file exists, or None if no file is found. (See the documentation for the `Glob` function for an alternative way of searching for entries in a directory.)

```
# one directory
print FindFile('missing', '.')
t = FindFile('exists', '.')
print t.__class__, t
```

```
% scons -Q
None
<class 'SCons.Node.FS.File'> exists
scons: `.' is up to date.
```

```
# several directories
includes = [ '.', 'include', 'src/include']
headers = [ 'nonesuch.h', 'config.h', 'private.h', 'dist.h']
for hdr in headers:
    print '%-12s' % ('%s:' % hdr), FindFile(hdr, includes)
```

```
% scons -Q
nonesuch.h:  None
config.h:    config.h
private.h:   src/include/private.h
dist.h:      include/dist.h
scons: `.' is up to date.
```

If the file exists in more than one directory, only the first occurrence is returned.

```
 print FindFile('multiple', ['sub1', 'sub2', 'sub3'])
 print FindFile('multiple', ['sub2', 'sub3', 'sub1'])
 print FindFile('multiple', ['sub3', 'sub1', 'sub2'])
```

```
% scons -Q
sub1/multiple
sub2/multiple
sub3/multiple
scons: `.' is up to date.
```

In addition to existing files, `FindFile` will also find derived files (that is, non-leaf files) that haven't been built yet. (Leaf files should already exist, or the build will fail!)

```
# Neither file exists, so build will fail
Command('derived', 'leaf', 'cat >$TARGET $SOURCE')
print FindFile('leaf', '.')
print FindFile('derived', '.')
```

```
% scons -Q
None
derived
scons: *** [derived] Source `leaf' not found, needed by target `derived'.
```

```
# Neither file exists, so build will fail
Command('derived', 'leaf', 'cat >$TARGET $SOURCE')
print FindFile('leaf', '.')
print FindFile('derived', '.')

# Only 'leaf' exists
Command('derived', 'leaf', 'cat >$TARGET $SOURCE')
print FindFile('leaf', '.')
print FindFile('derived', '.')
```

```
% scons -Q
leaf
derived
cat > derived leaf
```

If a source file exists, `FindFile` will correctly return the name in the build directory.

```
# Only 'src/leaf' exists
VariantDir('build', 'src')
print FindFile('leaf', 'build')
```

```
% scons -Q
build/leaf
scons: `.' is up to date.
```

27.5. Handling Nested Lists: the `Flatten` Function

SCons supports a `Flatten` function which takes an input Python sequence (list or tuple) and returns a flattened list containing just the individual elements of the sequence. This can be handy when trying to examine a list composed of the lists returned by calls to various Builders. For example, you might collect object files built in different ways into one call to the `Program` Builder by just enclosing them in a list, as follows:

```
objects = [
    Object('prog1.c'),
    Object('prog2.c', CCFLAGS='-DFOO'),
]
Program(objects)
```

Because the Builder calls in SCons flatten their input lists, this works just fine to build the program:

```
% scons -Q
cc -o prog1.o -c prog1.c
```

```
cc -o prog2.o -c -DFOO prog2.c
cc -o prog1 prog1.o prog2.o
```

But if you were debugging your build and wanted to print the absolute path of each object file in the `objects` list, you might try the following simple approach, trying to print each Node's `abspath` attribute:

```
objects = [
    Object('prog1.c'),
    Object('prog2.c', CCFLAGS='-DFOO'),
]
Program(objects)

for object_file in objects:
    print object_file.abspath
```

This does not work as expected because each call to `str` is operating an embedded list returned by each `Object` call, not on the underlying Nodes within those lists:

```
% scons -Q
AttributeError: 'NodeList' object has no attribute 'abspath':
  File "/home/my/project/SConstruct", line 8:
    print object_file.abspath
```

The solution is to use the `Flatten` function so that you can pass each Node to the `str` separately:

```
objects = [
    Object('prog1.c'),
    Object('prog2.c', CCFLAGS='-DFOO'),
]
Program(objects)

for object_file in Flatten(objects):
    print object_file.abspath
```

```
% scons -Q
/home/me/project/prog1.o
/home/me/project/prog2.o
cc -o prog1.o -c prog1.c
cc -o prog2.o -c -DFOO prog2.c
cc -o prog1 prog1.o prog2.o
```

27.6. Finding the Invocation Directory: the `GetLaunchDir` Function

If you need to find the directory from which the user invoked the `scons` command, you can use the `GetLaunchDir` function:

```
env = Environment(
    LAUNCHDIR = GetLaunchDir(),
)
env.Command('directory_build_info',
            '$LAUNCHDIR/build_info'
            Copy('$TARGET', '$SOURCE'))
```

Because SCons is usually invoked from the top-level directory in which the `SConstruct` file lives, the Python `os.getcwd()` is often equivalent. However, the SCons `-u`, `-U` and `-D` command-line options, when invoked from a subdirectory, will cause SCons to change to the directory in which the `SConstruct` file is found. When those options are used, `GetLaunchDir` will still return the path to the user's invoking subdirectory, allowing the `SConscript` configuration to still get at configuration (or other) files from the originating directory.

Chapter 28. Troubleshooting

The experience of configuring any software build tool to build a large code base usually, at some point, involves trying to figure out why the tool is behaving a certain way, and how to get it to behave the way you want. SCons is no different. This appendix contains a number of different ways in which you can get some additional insight into SCons' behavior.

Note that we're always interested in trying to improve how you can troubleshoot configuration problems. If you run into a problem that has you scratching your head, and which there just doesn't seem to be a good way to debug, odds are pretty good that someone else will run into the same problem, too. If so, please let the SCons development team know (preferably by filing a bug report or feature request at our project pages at tigris.org) so that we can use your feedback to try to come up with a better way to help you, and others, get the necessary insight into SCons behavior to help identify and fix configuration issues.

28.1. Why is That Target Being Rebuilt? the `--debug=explain` Option

Let's look at a simple example of a misconfigured build that causes a target to be rebuilt every time SCons is run:

```
# Intentionally misspell the output file name in the
# command used to create the file:
Command('file.out', 'file.in', 'cp $SOURCE file.oout')
```

(Note to Windows users: The POSIX cp command copies the first file named on the command line to the second file. In our example, it copies the `file.in` file to the `file.out` file.)

Now if we run SCons multiple times on this example, we see that it re-runs the cp command every time:

```
% scons -Q
cp file.in file.oout
% scons -Q
cp file.in file.oout
% scons -Q
cp file.in file.oout
```

In this example, the underlying cause is obvious: we've intentionally misspelled the output file name in the cp command, so the command doesn't actually build the `file.out` file that we've told SCons to expect. But if the problem weren't obvious, it would be helpful to specify the `--debug=explain` option on the command line to have SCons tell us very specifically why it's decided to rebuild the target:

```
% scons -Q --debug=explain
scons: building 'file.out' because it doesn't exist
cp file.in file.oout
```

If this had been a more complicated example involving a lot of build output, having SCons tell us that it's trying to rebuild the target file because it doesn't exist would be an important clue that something was wrong with the command that we invoked to build it.

The `--debug=explain` option also comes in handy to help figure out what input file changed. Given a simple configuration that builds a program from three source files, changing one of the source files and rebuilding with the `--debug=explain` option shows very specifically why SCons rebuilds the files that it does:

```
% scons -Q
cc -o file1.o -c file1.c
cc -o file2.o -c file2.c
cc -o file3.o -c file3.c
cc -o prog file1.o file2.o file3.o
% edit file2.c
    [CHANGE THE CONTENTS OF file2.c]
% scons -Q --debug=explain
scons: rebuilding 'file2.o' because 'file2.c' changed
cc -o file2.o -c file2.c
scons: rebuilding 'prog' because 'file2.o' changed
cc -o prog file1.o file2.o file3.o
```

This becomes even more helpful in identifying when a file is rebuilt due to a change in an implicit dependency, such as an incuded `.h` file. If the `file1.c` and `file3.c` files in our example both included a `hello.h` file, then changing that included file and re-running SCons with the `--debug=explain` option will pinpoint that it's the change to the included file that starts the chain of rebuilds:

```
% scons -Q
cc -o file1.o -c -I. file1.c
cc -o file2.o -c -I. file2.c
cc -o file3.o -c -I. file3.c
cc -o prog file1.o file2.o file3.o
% edit hello.h
    [CHANGE THE CONTENTS OF hello.h]
% scons -Q --debug=explain
scons: rebuilding 'file1.o' because 'hello.h' changed
cc -o file1.o -c -I. file1.c
scons: rebuilding 'file3.o' because 'hello.h' changed
cc -o file3.o -c -I. file3.c
scons: rebuilding 'prog' because:
           'file1.o' changed
           'file3.o' changed
cc -o prog file1.o file2.o file3.o
```

(Note that the `--debug=explain` option will only tell you why SCons decided to rebuild necessary targets. It does not tell you what files it examined when deciding *not* to rebuild a target file, which is often a more valuable question to answer.)

28.2. What's in That Construction Environment? the `Dump` Method

When you create a construction environment, SCons populates it with construction variables that are set up for various compilers, linkers and utilities that it finds on your system. Although this is usually helpful and what you want, it might be frustrating if SCons doesn't set certain variables that you expect to be set. In situations like this, it's sometimes helpful to use the construction environment `Dump` method to print all or some of the construction variables. Note that the `Dump` method *returns* the representation of the variables in the environment for you to print (or otherwise manipulate):

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

On a POSIX system with gcc installed, this might generate:

```
% scons
scons: Reading SConscript files ...
{ 'BUILDERS': {'_InternalInstall': <function InstallBuilderWrapper at 0x700000>, '_
  'CONFIGUREDIR': '#/.sconf_temp',
  'CONFIGURELOG': '#/config.log',
  'CPPSUFFIXES': [ '.c',
                   '.C',
                   '.cxx',
                   '.cpp',
                   '.c++',
                   '.cc',
                   '.h',
                   '.H',
                   '.hxx',
                   '.hpp',
                   '.hh',
                   '.F',
                   '.fpp',
                   '.FPP',
                   '.m',
                   '.mm',
                   '.S',
                   '.spp',
                   '.SPP',
                   '.sx'],
  'DSUFFIXES': ['.d'],
  'Dir': <SCons.Defaults.Variable_Method_Caller object at 0x700000>,
  'Dirs': <SCons.Defaults.Variable_Method_Caller object at 0x700000>,
```

```
  'ENV': { 'PATH': '/usr/local/bin:/opt/bin:/bin:/usr/bin'},
  'ESCAPE': <function escape at 0x700000>,
  'File': <SCons.Defaults.Variable_Method_Caller object at 0x700000>,
  'HOST_ARCH': None,
  'HOST_OS': None,
  'IDLSUFFIXES': ['.idl', '.IDL'],
  'INSTALL': <function copyFunc at 0x700000>,
  'LIBPREFIX': 'lib',
  'LIBPREFIXES': ['$LIBPREFIX'],
  'LIBSUFFIX': '.a',
  'LIBSUFFIXES': ['$LIBSUFFIX', '$SHLIBSUFFIX'],
  'MAXLINELENGTH': 128072,
  'OBJPREFIX': '',
  'OBJSUFFIX': '.o',
  'PLATFORM': 'posix',
  'PROGPREFIX': '',
  'PROGSUFFIX': '',
  'PSPAWN': <function piped_env_spawn at 0x700000>,
  'RDirs': <SCons.Defaults.Variable_Method_Caller object at 0x700000>,
  'SCANNERS': [],
  'SHELL': 'sh',
  'SHLIBPREFIX': '$LIBPREFIX',
  'SHLIBSUFFIX': '.so',
  'SHOBJPREFIX': '$OBJPREFIX',
  'SHOBJSUFFIX': '$OBJSUFFIX',
  'SPAWN': <function spawnvpe_spawn at 0x700000>,
  'TARGET_ARCH': None,
  'TARGET_OS': None,
  'TEMPFILE': <class 'SCons.Platform.TempFileMunge'>,
  'TEMPFILEPREFIX': '@',
  'TOOLS': ['install', 'install'],
  '_CPPDEFFLAGS': '${_defines(CPPDEFPREFIX, CPPDEFINES, CPPDEFSUFFIX, __env__)}',
  '_CPPINCFLAGS': '$( ${_concat(INCPREFIX, CPPPATH, INCSUFFIX, __env__, RDirs, TARG
  '_LIBDIRFLAGS': '$( ${_concat(LIBDIRPREFIX, LIBPATH, LIBDIRSUFFIX, __env__, RDirs
  '_LIBFLAGS': '${_concat(LIBLINKPREFIX, LIBS, LIBLINKSUFFIX, __env__)}',
  '__RPATH': '$_RPATH',
  '_concat': <function _concat at 0x700000>,
  '_defines': <function _defines at 0x700000>,
  '_stripixes': <function _stripixes at 0x700000>}
scons: done reading SConscript files.
scons: Building targets ...
scons: `.' is up to date.
scons: done building targets.
```

On a Windows system with Visual C++ the output might look like:

```
C:\>scons
scons: Reading SConscript files ...
{ 'BUILDERS': {'_InternalInstall': <function InstallBuilderWrapper at 0x700000>, 'C
  'CC': 'cl',
  'CCCOM': <SCons.Action.FunctionAction object at 0x700000>,
```

```
'CCFLAGS': ['/nologo'],
'CCPCHFLAGS': ['${(PCH and "/Yu%s \\"/Fp%s\\""%(PCHSTOP or "",File(PCH))) or ""}'
'CCPDBFLAGS': ['${(PDB and "/Z7") or ""}'],
'CFILESUFFIX': '.c',
'CFLAGS': [],
'CONFIGUREDIR': '#/.sconf_temp',
'CONFIGURELOG': '#/config.log',
'CPPDEFPREFIX': '/D',
'CPPDEFSUFFIX': '',
'CPPSUFFIXES': [ '.c',
                 '.C',
                 '.cxx',
                 '.cpp',
                 '.c++',
                 '.cc',
                 '.h',
                 '.H',
                 '.hxx',
                 '.hpp',
                 '.hh',
                 '.F',
                 '.fpp',
                 '.FPP',
                 '.m',
                 '.mm',
                 '.S',
                 '.spp',
                 '.SPP',
                 '.sx'],
'CXX': '$CC',
'CXXCOM': '${TEMPFILE("$CXX $_MSVC_OUTPUT_FLAG /c $CHANGED_SOURCES $CXXFLAGS $CCF
'CXXFILESUFFIX': '.cc',
'CXXFLAGS': ['$(', '/TP', '$)'],
'DSUFFIXES': ['.d'],
'Dir': <SCons.Defaults.Variable_Method_Caller object at 0x700000>,
'Dirs': <SCons.Defaults.Variable_Method_Caller object at 0x700000>,
'ENV': { 'PATH': 'C:\\WINDOWS\\System32',
         'PATHEXT': '.COM;.EXE;.BAT;.CMD',
         'SystemRoot': 'C:\\WINDOWS'},
'ESCAPE': <function escape at 0x700000>,
'File': <SCons.Defaults.Variable_Method_Caller object at 0x700000>,
'HOST_ARCH': '',
'HOST_OS': 'win32',
'IDLSUFFIXES': ['.idl', '.IDL'],
'INCPREFIX': '/I',
'INCSUFFIX': '',
'INSTALL': <function copyFunc at 0x700000>,
'LIBPREFIX': '',
'LIBPREFIXES': ['$LIBPREFIX'],
'LIBSUFFIX': '.lib',
'LIBSUFFIXES': ['$LIBSUFFIX'],
'MAXLINELENGTH': 2048,
'MSVC_SETUP_RUN': True,
```

```
  'OBJPREFIX': '',
  'OBJSUFFIX': '.obj',
  'PCHCOM': '$CXX /Fo${TARGETS[1]} $CXXFLAGS $CCFLAGS $CPPFLAGS $_CPPDEFFLAGS $_CPI
  'PCHPDBFLAGS': ['${(PDB and "/Yd") or ""}'],
  'PLATFORM': 'win32',
  'PROGPREFIX': '',
  'PROGSUFFIX': '.exe',
  'PSPAWN': <function piped_spawn at 0x700000>,
  'RC': 'rc',
  'RCCOM': <SCons.Action.FunctionAction object at 0x700000>,
  'RCFLAGS': [],
  'RCSUFFIXES': ['.rc', '.rc2'],
  'RDirs': <SCons.Defaults.Variable_Method_Caller object at 0x700000>,
  'SCANNERS': [],
  'SHCC': '$CC',
  'SHCCCOM': <SCons.Action.FunctionAction object at 0x700000>,
  'SHCCFLAGS': ['$CCFLAGS'],
  'SHCFLAGS': ['$CFLAGS'],
  'SHCXX': '$CXX',
  'SHCXXCOM': '${TEMPFILE("$SHCXX $_MSVC_OUTPUT_FLAG /c $CHANGED_SOURCES $SHCXXFLAG
  'SHCXXFLAGS': ['$CXXFLAGS'],
  'SHELL': None,
  'SHLIBPREFIX': '',
  'SHLIBSUFFIX': '.dll',
  'SHOBJPREFIX': '$OBJPREFIX',
  'SHOBJSUFFIX': '$OBJSUFFIX',
  'SPAWN': <function spawn at 0x700000>,
  'STATIC_AND_SHARED_OBJECTS_ARE_THE_SAME': 1,
  'TARGET_ARCH': '',
  'TARGET_OS': 'win32',
  'TEMPFILE': <class 'SCons.Platform.TempFileMunge'>,
  'TEMPFILEPREFIX': '@',
  'TOOLS': ['msvc', 'install', 'install'],
  '_CCCOMCOM': '$CPPFLAGS $_CPPDEFFLAGS $_CPPINCFLAGS $CCPCHFLAGS $CCPDBFLAGS',
  '_CPPDEFFLAGS': '${_defines(CPPDEFPREFIX, CPPDEFINES, CPPDEFSUFFIX, __env__)}',
  '_CPPINCFLAGS': '$( ${_concat(INCPREFIX, CPPPATH, INCSUFFIX, __env__, RDirs, TARG
  '_LIBDIRFLAGS': '$( ${_concat(LIBDIRPREFIX, LIBPATH, LIBDIRSUFFIX, __env__, RDirs
  '_LIBFLAGS': '${_concat(LIBLINKPREFIX, LIBS, LIBLINKSUFFIX, __env__)}',
  '_MSVC_OUTPUT_FLAG': <function msvc_output_flag at 0x700000>,
  '_concat': <function _concat at 0x700000>,
  '_defines': <function _defines at 0x700000>,
  '_stripixes': <function _stripixes at 0x700000>}
scons: done reading SConscript files.
scons: Building targets ...
scons: `.' is up to date.
scons: done building targets.
```

The construction environments in these examples have actually been restricted to just gcc and Visual C++, respectively. In a real-life situation, the construction environments will likely contain a great many more variables. Also note that we've massaged the example output above to make the memory address of

all objects a constant 0x700000. In reality, you would see a different hexadecimal number for each object.

To make it easier to see just what you're interested in, the `Dump` method allows you to specify a specific constrcution variable that you want to disply. For example, it's not unusual to want to verify the external environment used to execute build commands, to make sure that the PATH and other environment variables are set up the way they should be. You can do this as follows:

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

Which might display the following when executed on a POSIX system:

```
% scons
scons: Reading SConscript files ...
{ 'PATH': '/usr/local/bin:/opt/bin:/bin:/usr/bin'}
scons: done reading SConscript files.
scons: Building targets ...
scons: `.' is up to date.
scons: done building targets.
```

And the following when executed on a Windows system:

```
C:\>scons
scons: Reading SConscript files ...
{ 'PATH': 'C:\\WINDOWS\\System32',
  'PATHEXT': '.COM;.EXE;.BAT;.CMD',
  'SystemRoot': 'C:\\WINDOWS'}
scons: done reading SConscript files.
scons: Building targets ...
scons: `.' is up to date.
scons: done building targets.
```

28.3. What Dependencies Does SCons Know About? the `--tree` Option

Sometimes the best way to try to figure out what SCons is doing is simply to take a look at the dependency graph that it constructs based on your `SConscript` files. The `--tree` option will display all or part of the SCons dependency graph in an "ASCII art" graphical format that shows the dependency hierarchy.

For example, given the following input `SConstruct` file:

```
env = Environment(CPPPATH = ['.'])
env.Program('prog', ['f1.c', 'f2.c', 'f3.c'])
```

Running SCons with the `--tree=all` option yields:

```
% scons -Q --tree=all
cc -o f1.o -c -I. f1.c
cc -o f2.o -c -I. f2.c
cc -o f3.o -c -I. f3.c
cc -o prog f1.o f2.o f3.o
+-.
  +-SConstruct
  +-f1.c
  +-f1.o
  | +-f1.c
  | +-inc.h
  +-f2.c
  +-f2.o
  | +-f2.c
  | +-inc.h
  +-f3.c
  +-f3.o
  | +-f3.c
  | +-inc.h
  +-inc.h
  +-prog
    +-f1.o
    | +-f1.c
    | +-inc.h
    +-f2.o
    | +-f2.c
    | +-inc.h
    +-f3.o
      +-f3.c
      +-inc.h
```

The tree will also be printed when the `-n` (no execute) option is used, which allows you to examine the dependency graph for a configuration without actually rebuilding anything in the tree.

The `--tree` option only prints the dependency graph for the specified targets (or the default target(s) if none are specified on the command line). So if you specify a target like `f2.o` on the command line, the `--tree` option will only print the dependency graph for that file:

```
% scons -Q --tree=all f2.o
cc -o f2.o -c -I. f2.c
+-f2.o
  +-f2.c
  +-inc.h
```

This is, of course, useful for restricting the output from a very large build configuration to just a portion in which you're interested. Multiple targets are fine, in which case a tree will be printed for each specified target:

```
% scons -Q --tree=all f1.o f3.o
cc -o f1.o -c -I. f1.c
+-f1.o
  +-f1.c
  +-inc.h
cc -o f3.o -c -I. f3.c
+-f3.o
  +-f3.c
  +-inc.h
```

The `status` argument may be used to tell SCons to print status information about each file in the dependency graph:

```
% scons -Q --tree=status
cc -o f1.o -c -I. f1.c
cc -o f2.o -c -I. f2.c
cc -o f3.o -c -I. f3.c
cc -o prog f1.o f2.o f3.o
 E         = exists
  R        = exists in repository only
   b       = implicit builder
   B       = explicit builder
    S      = side effect
     P     = precious
      A    = always build
       C   = current
        N  = no clean
         H = no cache

[E b      ]+-.
[E     C  ]  +-SConstruct
[E     C  ]  +-f1.c
[E B   C  ]  +-f1.o
[E     C  ]  | +-f1.c
[E     C  ]  | +-inc.h
[E     C  ]  +-f2.c
[E B   C  ]  +-f2.o
[E     C  ]  | +-f2.c
[E     C  ]  | +-inc.h
[E     C  ]  +-f3.c
[E B   C  ]  +-f3.o
[E     C  ]  | +-f3.c
[E     C  ]  | +-inc.h
[E     C  ]  +-inc.h
```

```
[E B   C  ]  +-prog
[E B   C  ]    +-f1.o
[E     C  ]    | +-f1.c
[E     C  ]    | +-inc.h
[E B   C  ]    +-f2.o
[E     C  ]    | +-f2.c
[E     C  ]    | +-inc.h
[E B   C  ]    +-f3.o
[E     C  ]      +-f3.c
[E     C  ]      +-inc.h
```

Note that `--tree=all,status` is equivalent; the `all` is assumed if only `status` is present. As an alternative to `all`, you can specify `--tree=derived` to have SCons only print derived targets in the tree output, skipping source files (like `.c` and `.h` files):

```
% scons -Q --tree=derived
cc -o f1.o -c -I. f1.c
cc -o f2.o -c -I. f2.c
cc -o f3.o -c -I. f3.c
cc -o prog f1.o f2.o f3.o
+-.
  +-f1.o
  +-f2.o
  +-f3.o
  +-prog
    +-f1.o
    +-f2.o
    +-f3.o
```

You can use the `status` modifier with `derived` as well:

```
% scons -Q --tree=derived,status
cc -o f1.o -c -I. f1.c
cc -o f2.o -c -I. f2.c
cc -o f3.o -c -I. f3.c
cc -o prog f1.o f2.o f3.o
 E         = exists
  R        = exists in repository only
   b       = implicit builder
   B       = explicit builder
    S      = side effect
     P     = precious
      A    = always build
       C   = current
        N  = no clean
         H = no cache

[E b      ]+-.
[E B   C  ]  +-f1.o
```

```
[E B   C  ]  +-f2.o
[E B   C  ]  +-f3.o
[E B   C  ]  +-prog
[E B   C  ]    +-f1.o
[E B   C  ]    +-f2.o
[E B   C  ]    +-f3.o
```

Note that the order of the `--tree=` arguments doesn't matter; `--tree=status,derived` is completely equivalent.

The default behavior of the `--tree` option is to repeat all of the dependencies each time the library dependency (or any other dependency file) is encountered in the tree. If certain target files share other target files, such as two programs that use the same library:

```
env = Environment(CPPPATH = ['.'],
                  LIBS = ['foo'],
                  LIBPATH = ['.'])
env.Library('foo', ['f1.c', 'f2.c', 'f3.c'])
env.Program('prog1.c')
env.Program('prog2.c')
```

Then there can be a *lot* of repetition in the `--tree=` output:

```
% scons -Q --tree=all
cc -o f1.o -c -I. f1.c
cc -o f2.o -c -I. f2.c
cc -o f3.o -c -I. f3.c
ar rc libfoo.a f1.o f2.o f3.o
ranlib libfoo.a
cc -o prog1.o -c -I. prog1.c
cc -o prog1 prog1.o -L. -lfoo
cc -o prog2.o -c -I. prog2.c
cc -o prog2 prog2.o -L. -lfoo
+-.
  +-SConstruct
  +-f1.c
  +-f1.o
  | +-f1.c
  | +-inc.h
  +-f2.c
  +-f2.o
  | +-f2.c
  | +-inc.h
  +-f3.c
  +-f3.o
  | +-f3.c
  | +-inc.h
  +-inc.h
```

```
+-libfoo.a
| +-f1.o
| | +-f1.c
| | +-inc.h
| +-f2.o
| | +-f2.c
| | +-inc.h
| +-f3.o
|   +-f3.c
|   +-inc.h
+-prog1
| +-prog1.o
| | +-prog1.c
| | +-inc.h
| +-libfoo.a
|   +-f1.o
|   | +-f1.c
|   | +-inc.h
|   +-f2.o
|   | +-f2.c
|   | +-inc.h
|   +-f3.o
|     +-f3.c
|     +-inc.h
+-prog1.c
+-prog1.o
| +-prog1.c
| +-inc.h
+-prog2
| +-prog2.o
| | +-prog2.c
| | +-inc.h
| +-libfoo.a
|   +-f1.o
|   | +-f1.c
|   | +-inc.h
|   +-f2.o
|   | +-f2.c
|   | +-inc.h
|   +-f3.o
|     +-f3.c
|     +-inc.h
+-prog2.c
+-prog2.o
  +-prog2.c
  +-inc.h
```

In a large configuration with many internal libraries and include files, this can very quickly lead to huge output trees. To help make this more manageable, a `prune` modifier may be added to the option list, in which case SCons will print the name of a target that has already been visited during the tree-printing in

`[square brackets]` as an indication that the dependencies of the target file may be found by looking farther up the tree:

```
% scons -Q --tree=prune
cc -o f1.o -c -I. f1.c
cc -o f2.o -c -I. f2.c
cc -o f3.o -c -I. f3.c
ar rc libfoo.a f1.o f2.o f3.o
ranlib libfoo.a
cc -o prog1.o -c -I. prog1.c
cc -o prog1 prog1.o -L. -lfoo
cc -o prog2.o -c -I. prog2.c
cc -o prog2 prog2.o -L. -lfoo
+-.
  +-SConstruct
  +-f1.c
  +-f1.o
  | +-f1.c
  | +-inc.h
  +-f2.c
  +-f2.o
  | +-f2.c
  | +-inc.h
  +-f3.c
  +-f3.o
  | +-f3.c
  | +-inc.h
  +-inc.h
  +-libfoo.a
  | +-[f1.o]
  | +-[f2.o]
  | +-[f3.o]
  +-prog1
  | +-prog1.o
  | | +-prog1.c
  | | +-inc.h
  | +-[libfoo.a]
  +-prog1.c
  +-[prog1.o]
  +-prog2
  | +-prog2.o
  | | +-prog2.c
  | | +-inc.h
  | +-[libfoo.a]
  +-prog2.c
  +-[prog2.o]
```

Like the `status` keyword, the `prune` argument by itself is equivalent to `--tree=all,prune`.

28.4. How is SCons Constructing the Command Lines It Executes? the `--debug=presub` Option

Sometimes it's useful to look at the pre-substitution string that SCons uses to generate the command lines it executes. This can be done with the `--debug=presub` option:

```
% scons -Q --debug=presub
Building prog.o with action:
  $CC -o $TARGET -c $CFLAGS $CCFLAGS $_CCOMCOM $SOURCES
cc -o prog.o -c -I. prog.c
Building prog with action:
  $SMART_LINKCOM
cc -o prog prog.o
```

28.5. Where is SCons Searching for Libraries? the `--debug=findlibs` Option

To get some insight into what library names SCons is searching for, and in which directories it is searching, Use the `--debug=findlibs` option. Given the following input `SConstruct` file:

```
env = Environment(LIBPATH = ['libs1', 'libs2'])
env.Program('prog.c', LIBS=['foo', 'bar'])
```

And the libraries `libfoo.a` and `libbar.a` in `libs1` and `libs2`, respectively, use of the `--debug=findlibs` option yields:

```
% scons -Q --debug=findlibs
  findlibs: looking for 'libfoo.a' in 'libs1' ...
  findlibs: ... FOUND 'libfoo.a' in 'libs1'
  findlibs: looking for 'libfoo.so' in 'libs1' ...
  findlibs: looking for 'libfoo.so' in 'libs2' ...
  findlibs: looking for 'libbar.a' in 'libs1' ...
  findlibs: looking for 'libbar.a' in 'libs2' ...
  findlibs: ... FOUND 'libbar.a' in 'libs2'
  findlibs: looking for 'libbar.so' in 'libs1' ...
  findlibs: looking for 'libbar.so' in 'libs2' ...
cc -o prog.o -c prog.c
cc -o prog prog.o -Llibs1 -Llibs2 -lfoo -lbar
```

28.6. Where is SCons Blowing Up? the `--debug=stacktrace` Option

In general, SCons tries to keep its error messages short and informative. That means we usually try to avoid showing the stack traces that are familiar to experienced Python programmers, since they usually contain much more information than is useful to most people.

For example, the following `SConstruct` file:

```
Program('prog.c')
```

Generates the following error if the `prog.c` file does not exist:

```
% scons -Q
scons: *** [prog.o] Source 'prog.c' not found, needed by target 'prog.o'.
```

In this case, the error is pretty obvious. But if it weren't, and you wanted to try to get more information about the error, the `--debug=stacktrace` option would show you exactly where in the SCons source code the problem occurs:

```
% scons -Q --debug=stacktrace
scons: *** [prog.o] Source 'prog.c' not found, needed by target 'prog.o'.
scons: internal stack trace:
  File "bootstrap/src/engine/SCons/Job.py", line 199, in start
    task.prepare()
  File "bootstrap/src/engine/SCons/Script/Main.py", line 168, in prepare
    return SCons.Taskmaster.OutOfDateTask.prepare(self)
  File "bootstrap/src/engine/SCons/Taskmaster.py", line 189, in prepare
    executor.prepare()
  File "bootstrap/src/engine/SCons/Executor.py", line 392, in prepare
    raise SCons.Errors.StopError(msg % (s, self.batches[0].targets[0]))
```

Of course, if you do need to dive into the SCons source code, we'd like to know if, or how, the error messages or troubleshooting options could have been improved to avoid that. Not everyone has the necessary time or Python skill to dive into the source code, and we'd like to improve SCons for those people as well...

28.7. How is SCons Making Its Decisions? the `--taskmastertrace` Option

The internal SCons subsystem that handles walking the dependency graph and controls the

decision-making about what to rebuild is the `Taskmaster`. SCons supports a `--taskmastertrace` option that tells the Taskmaster to print information about the children (dependencies) of the various Nodes on its walk down the graph, which specific dependent Nodes are being evaluated, and in what order.

The `--taskmastertrace` option takes as an argument the name of a file in which to put the trace output, with – (a single hyphen) indicating that the trace messages should be printed to the standard output:

```
env = Environment(CPPPATH = ['.'])
env.Program('prog.c')
```

```
% scons -Q --taskmastertrace=- prog

Taskmaster: Looking for a node to evaluate
Taskmaster:     Considering node <no_state   0   'prog'> and its children:
Taskmaster:        <no_state   0   'prog.o'>
Taskmaster:      adjusted ref count: <pending    1   'prog'>, child 'prog.o'
Taskmaster:     Considering node <no_state   0   'prog.o'> and its children:
Taskmaster:        <no_state   0   'prog.c'>
Taskmaster:        <no_state   0   'inc.h'>
Taskmaster:      adjusted ref count: <pending    1   'prog.o'>, child 'prog.c'
Taskmaster:      adjusted ref count: <pending    2   'prog.o'>, child 'inc.h'
Taskmaster:     Considering node <no_state   0   'prog.c'> and its children:
Taskmaster: Evaluating <pending    0   'prog.c'>

Task.make_ready_current(): node <pending    0   'prog.c'>
Task.prepare():      node <up_to_date 0   'prog.c'>
Task.executed_with_callbacks(): node <up_to_date 0   'prog.c'>
Task.postprocess():  node <up_to_date 0   'prog.c'>
Task.postprocess():  removing <up_to_date 0   'prog.c'>
Task.postprocess():  adjusted parent ref count <pending    1   'prog.o'>

Taskmaster: Looking for a node to evaluate
Taskmaster:     Considering node <no_state   0   'inc.h'> and its children:
Taskmaster: Evaluating <pending    0   'inc.h'>

Task.make_ready_current(): node <pending    0   'inc.h'>
Task.prepare():      node <up_to_date 0   'inc.h'>
Task.executed_with_callbacks(): node <up_to_date 0   'inc.h'>
Task.postprocess():  node <up_to_date 0   'inc.h'>
Task.postprocess():  removing <up_to_date 0   'inc.h'>
Task.postprocess():  adjusted parent ref count <pending    0   'prog.o'>

Taskmaster: Looking for a node to evaluate
Taskmaster:     Considering node <pending    0   'prog.o'> and its children:
Taskmaster:        <up_to_date 0   'prog.c'>
Taskmaster:        <up_to_date 0   'inc.h'>
Taskmaster: Evaluating <pending    0   'prog.o'>

Task.make_ready_current(): node <pending    0   'prog.o'>
```

```
Task.prepare():      node <executing  0   'prog.o'>
Task.execute():      node <executing  0   'prog.o'>
cc -o prog.o -c -I. prog.c
Task.executed_with_callbacks(): node <executing  0   'prog.o'>
Task.postprocess():  node <executed   0   'prog.o'>
Task.postprocess():  removing <executed   0   'prog.o'>
Task.postprocess():  adjusted parent ref count <pending    0   'prog'>

Taskmaster: Looking for a node to evaluate
Taskmaster:     Considering node <pending    0   'prog'> and its children:
Taskmaster:        <executed   0   'prog.o'>
Taskmaster: Evaluating <pending    0   'prog'>

Task.make_ready_current(): node <pending    0   'prog'>
Task.prepare():      node <executing  0   'prog'>
Task.execute():      node <executing  0   'prog'>
cc -o prog prog.o
Task.executed_with_callbacks(): node <executing  0   'prog'>
Task.postprocess():  node <executed   0   'prog'>

Taskmaster: Looking for a node to evaluate
Taskmaster: No candidate anymore.
```

The `--taskmastertrace` option doesn't provide information about the actual calculations involved in deciding if a file is up-to-date, but it does show all of the dependencies it knows about for each Node, and the order in which those dependencies are evaluated. This can be useful as an alternate way to determine whether or not your SCons configuration, or the implicit dependency scan, has actually identified all the correct dependencies you want it to.

28.8. Watch SCons prepare targets for building: the `--debug=prepare` Option

Sometimes SCons doesn't build the target you want and it's difficult to figure out why. You can use the `--debug=prepare` option to see all the targets SCons is considering, whether they are already up-to-date or not. The message is printed before SCons decides whether to build the target.

28.9. Why is a file disappearing? the --debug=duplicate Option

When using the `Duplicate` option to create variant dirs, sometimes you may find files not getting copied to where you expect (or not at all), or files mysteriously disappearing. These are usually because of a misconfiguration of some kind in the SConstruct/SConscript, but they can be tricky to debug. The --debug=duplicate option shows each time a variant file is unlinked and relinked from its source (or

copied, depending on settings), and also shows a message for removing "stale" variant-dir files that no longer have a corresponding source file. It also prints a line for each target that's removed just before building, since that can also be mistaken for the same thing.

Appendix A. Construction Variables

This appendix contains descriptions of all of the construction variables that are *potentially* available "out of the box" in this version of SCons. Whether or not setting a construction variable in a construction environment will actually have an effect depends on whether any of the Tools and/or Builders that use the variable have been included in the construction environment.

In this appendix, we have appended the initial $ (dollar sign) to the beginning of each variable name when it appears in the text, but left off the dollar sign in the left-hand column where the name appears for each entry.

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:

```
env['CCPDBFLAGS'] = ['${(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

CHANGED_SOURCES

A reserved variable name that may not be set or used in a construction environment. (See "Variable Substitution," below.)

CHANGED_TARGETS

A reserved variable name that may not be set or used in a construction environment. (See "Variable Substitution," below.)

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 definition 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 through 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 superseded 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.

HOST_ARCH

Sets the host architecture for Visual Studio compiler. If not set, default to the detected host architecture: note that this may depend on the python you are using. This variable must be passed as an argument to the Environment() constructor; setting it later has no effect.

Valid values are the same as for $TARGET_ARCH.

This is currently only used on Windows, but in the future it will be used on other OSes as well.

HOST_OS

The name of the host operating system used to create the Environment. If a platform is specified when creating the Environment, then that Platform's logic will handle setting this value. This value is immutable, and should not be changed by the user after the Environment is initialized. Currently only set for Win32.

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 $SOURCE
```

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.

LIBEMITTER

TODO

_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.

LINESEPARATOR

The separator used by the `Substfile` and `Textfile` builders. This value is used between sources when constructing the target. It defaults to the current system line separator.

LINGUAS_FILE

The $LINGUAS_FILE defines file(s) containing list of additional linguas to be processed by `POInit`, `POUpdate` or `MOFiles` builders. It also affects `Translate` builder. If the variable contains a string, it defines name of the list file. The $LINGUAS_FILE may be a list of file names as well. If $LINGUAS_FILE is set to `True` (or non-zero numeric value), the list will be read from default file named `LINGUAS`.

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 $MIDLCOM (the command line) is displayed.

MIDLFLAGS

General options passed to the Microsoft IDL compiler.

MOSUFFIX

Suffix used for `MO` files (default: `'.mo'`). See `msgfmt` tool and `MOFiles` builder.

MSGFMT

Absolute path to **msgfmt(1)** binary, found by `Detect()`. See `msgfmt` tool and `MOFiles` builder.

MSGFMTCOM

Complete command line to run **msgfmt(1)** program. See `msgfmt` tool and `MOFiles` builder.

MSGFMTCOMSTR

String to display when **msgfmt(1)** is invoked (default: `''`, which means "print $MSGFMTCOM"). See `msgfmt` tool and `MOFiles` builder.

MSGFMTFLAGS

Additional flags to **msgfmt(1)**. See `msgfmt` tool and `MOFiles` builder.

MSGINIT

Path to **msginit(1)** program (found via `Detect()`). See `msginit` tool and `POInit` builder.

MSGINITCOM

Complete command line to run **msginit(1)** program. See `msginit` tool and `POInit` builder.

MSGINITCOMSTR

String to display when **msginit(1)** is invoked (default: `''`, which means "print $MSGINITCOM"). See `msginit` tool and `POInit` builder.

MSGINITFLAGS

List of additional flags to **msginit(1)** (default: `[]`). See `msginit` tool and `POInit` builder.

_MSGINITLOCALE

Internal "macro". Computes locale (language) name based on target filename (default: `'${TARGET.filebase}'`).

MSGMERGE

Absolute path to **msgmerge(1)** binary as found by `Detect()`. See `msgmerge` tool and `POUpdate` builder.

MSGMERGECOM

Complete command line to run **msgmerge(1)** command. See `msgmerge` tool and `POUpdate` builder.

MSGMERGECOMSTR

String to be displayed when **msgmerge(1)** is invoked (default: `''`, which means "print $MSGMERGECOM"). See `msgmerge` tool and `POUpdate` builder.

MSGMERGEFLAGS

Additional flags to **msgmerge(1)** command. See `msgmerge` tool and `POUpdate` builder.

MSSDK_DIR

The directory containing the Microsoft SDK (either Platform SDK or Windows SDK) to be used for compilation.

MSSDK_VERSION

The version string of the Microsoft SDK (either Platform SDK or Windows SDK) to be used for compilation. Supported versions include `6.1`, `6.0A`, `6.0`, `2003R2` and `2003R1`.

MSVC_BATCH

When set to any true value, specifies that SCons should batch compilation of object files when calling the Microsoft Visual C/C++ compiler. All compilations of source files from the same source

directory that generate target files in a same output directory and were configured in SCons using the same construction environment will be built in a single call to the compiler. Only source files that have changed since their object files were built will be passed to each compiler invocation (via the $CHANGED_SOURCES construction variable). Any compilations where the object (target) file base name (minus the `.obj`) does not match the source file base name will be compiled separately.

MSVC_USE_SCRIPT

Use a batch script to set up Microsoft Visual Studio compiler

$MSVC_USE_SCRIPT overrides $MSVC_VERSION and $TARGET_ARCH. If set to the name of a Visual Studio .bat file (e.g. vcvars.bat), SCons will run that bat file and extract the relevant variables from the result (typically %INCLUDE%, %LIB%, and %PATH%). Setting MSVC_USE_SCRIPT to None bypasses the Visual Studio autodetection entirely; use this if you are running SCons in a Visual Studio cmd window and importing the shell's environment variables.

MSVC_VERSION

Sets the preferred version of Microsoft Visual C/C++ to use.

If $MSVC_VERSION is not set, SCons will (by default) select the latest version of Visual C/C++ installed on your system. If the specified version isn't installed, tool initialization will fail. This variable must be passed as an argument to the Environment() constructor; setting it later has no effect. Set it to an unexpected value (e.g. "XXX") to see the valid values on your system.

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_ARCH

Sets the architecture for which the generated project(s) should build.

The default value is `x86`. `amd64` is also supported by SCons for some Visual Studio versions. Trying to set $MSVS_ARCH to an architecture that's not supported for a given Visual Studio version will generate an error.

MSVS_PROJECT_GUID

The string placed in a generated Microsoft Visual Studio project file as the value of the `ProjectGUID` attribute. There is no default value. If not defined, a new GUID is generated.

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_CONNECTION_ROOT

The root path of projects in your SCC workspace, i.e the path under which all project and solution files will be generated. It is used as a reference path from which the relative paths of the generated Microsoft Visual Studio project and solution files are computed. The relative project file path is placed as the value of the `SccLocalPath` attribute of the project file and as the values of the `SccProjectFilePathRelativizedFromConnection[i]` (where [i] ranges from 0 to the number of projects in the solution) attributes of the `GlobalSection(SourceCodeControl)` section of the Microsoft Visual Studio solution file. Similarly the relative solution file path is placed as the values of the `SccLocalPath[i]` (where [i] ranges from 0 to the number of projects in the solution) attributes of the `GlobalSection(SourceCodeControl)` section of the Microsoft Visual Studio solution file. This is used only if the MSVS_SCC_PROVIDER construction variable is also set. The default value is the current working directory.

MSVS_SCC_PROJECT_NAME

The project name placed in a generated Microsoft Visual Studio project file as the value of the `SccProjectName` attribute if the MSVS_SCC_PROVIDER construction variable is also set. In this case the string is also placed in the `SccProjectName0` attribute of the

`GlobalSection(SourceCodeControl)` section of the Microsoft Visual Studio solution file. 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 `SccProvider0` attribute of the `GlobalSection(SourceCodeControl)` section of the Microsoft Visual Studio solution file. There is no default value.

MSVS_VERSION

Sets the preferred version of Microsoft Visual Studio to use.

If $MSVS_VERSION is not set, SCons will (by default) select the latest version of Visual Studio installed on your system. 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 specified version isn't installed, tool initialization will fail.

This is obsolete: use $MSVC_VERSION instead. If $MSVS_VERSION is set and $MSVC_VERSION is not, $MSVC_VERSION will be set automatically to $MSVS_VERSION. If both are set to different values, scons will raise an error.

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.

MT

The program used on Windows systems to embed manifests into DLLs and EXEs. See also $WINDOWS_EMBED_MANIFEST.

MTEXECOM

The Windows command line used to embed manifests into executables. See also $MTSHLIBCOM.

MTFLAGS

Flags passed to the $MT manifest embedding program (Windows only).

MTSHLIBCOM

The Windows command line used to embed manifests into shared libraries (DLLs). See also $MTEXECOM.

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)
```

POAUTOINIT

The $POAUTOINIT variable, if set to `True` (on non-zero numeric value), let the `msginit` tool to automatically initialize *missing* `PO` files with **msginit(1)**. This applies to both, `POInit` and `POUpdate` builders (and others that use any of them).

POCREATE_ALIAS

Common alias for all `PO` files created with `POInit` builder (default: `'po-create'`). See `msginit` tool and `POInit` builder.

POSUFFIX

Suffix used for `PO` files (default: `'.po'`) See `msginit` tool and `POInit` builder.

POTDOMAIN

The $POTDOMAIN defines default domain, used to generate `POT` filename as `$POTDOMAIN.pot` when no `POT` file name is provided by the user. This applies to `POTUpdate`, `POInit` and `POUpdate` builders (and builders, that use them, e.g. `Translate`). Normally (if $POTDOMAIN is not defined), the builders use `messages.pot` as default `POT` file name.

POTSUFFIX

Suffix used for PO Template files (default: `'.pot'`). See `xgettext` tool and `POTUpdate` builder.

POTUPDATE_ALIAS

Name of the common phony target for all PO Templates created with `POUpdate` (default: `'pot-update'`). See `xgettext` tool and `POTUpdate` builder.

POUPDATE_ALIAS

Common alias for all `PO` files being defined with `POUpdate` builder (default: `'po-update'`). See `msgmerge` tool and `POUpdate` builder.

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.

PROGEMITTER

TODO

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 explicitly 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 `Moc`(); builder method.

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 $S
```

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 through the C preprocessor. If this is not set, then $SHFORTRANPPCOM (the command line) is displayed.

SHLIBEMITTER

TODO

SHLIBPREFIX

The prefix used for shared library file names.

SHLIBSUFFIX

The suffix used for shared library file names.

SHLIBVERSION

When this construction variable is defined, a versioned shared library is created. This modifies the $SHLINKFLAGS as required, adds the version number to the library name, and creates the symlinks that are needed. $SHLIBVERSION needs to be of the form X.Y.Z, where X and Y are numbers, and Z is a number but can also contain letters to designate alpha, beta, or release candidate patch levels.

SHLINK

The linker for programs that use shared libraries.

SHLINKCOM

The command line used to link programs using shared libraries.

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.

SUBST_DICT

The dictionary used by the `Substfile` or `Textfile` builders for substitution values. It can be anything acceptable to the dict() constructor, so in addition to a dictionary, lists of tuples are also acceptable.

SUBSTFILEPREFIX

The prefix used for `Substfile` file names, the null string by default.

SUBSTFILESUFFIX

The suffix used for `Substfile` file names, the null string by default.

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.

SWIGDIRECTORSUFFIX

The suffix that will be used for intermediate C++ header files generated by the scripting language wrapper and interface generator. These are only generated for C++ code when the SWIG 'directors' feature is turned on. The default value is `_wrap.h`.

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")
```

SWIGVERSION

The version number of the SWIG tool.

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.)

TARGET_ARCH

Sets the target architecture for Visual Studio compiler (i.e. the arch of the binaries generated by the compiler). If not set, default to $HOST_ARCH, or, if that is unset, to the architecture of the running machine's OS (note that the python build or architecture has no effect). This variable must be passed as an argument to the Environment() constructor; setting it later has no effect. This is currently only used on Windows, but in the future it will be used on other OSes as well.

Valid values for Windows are `x86`, `i386` (for 32 bits); `amd64`, `emt64`, `x86_64` (for 64 bits); and `ia64` (Itanium). For example, if you want to compile 64-bit binaries, you would set `TARGET_ARCH='x86_64'` in your SCons environment.

TARGET_OS

The name of the target operating system for the compiled objects created by this Environment. This defaults to the value of HOST_OS, and the user can override it. Currently only set for Win32.

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 program will search for include directories. The LaTeX implicit dependency scanner will search these directories for \include and \import files.

TEXTFILEPREFIX

The prefix used for `Textfile` file names, the null string by default.

TEXTFILESUFFIX

The suffix used for `Textfile` file names; `.txt` by default.

TOOLS

A list of the names of the Tool specifications that are part of this construction environment.

UNCHANGED_SOURCES

A reserved variable name that may not be set or used in a construction environment. (See "Variable Substitution," below.)

UNCHANGED_TARGETS

A reserved variable name that may not be set or used in a construction environment. (See "Variable Substitution," below.)

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_EMBED_MANIFEST

Set this variable to True or 1 to embed the compiler-generated manifest (normally `${TARGET}.manifest`) into all Windows exes and DLLs built with this environment, as a resource during their link step. This is done using $MT and $MTEXECOM and $MTSHLIBCOM.

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 `$SUMMARY\n$DESCRIPTION`

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 \\par.

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.

XGETTEXT

Path to **xgettext(1)** program (found via `Detect()`). See `xgettext` tool and `POTUpdate` builder.

XGETTEXTCOM

Complete xgettext command line. See `xgettext` tool and `POTUpdate` builder.

XGETTEXTCOMSTR

A string that is shown when **xgettext(1)** command is invoked (default: `''`, which means "print $XGETTEXTCOM"). See `xgettext` tool and `POTUpdate` builder.

_XGETTEXTDOMAIN

Internal "macro". Generates **xgettext** domain name form source and target (default: `'${TARGET.filebase}'`).

XGETTEXTFLAGS

Additional flags to **xgettext(1)**. See `xgettext` tool and `POTUpdate` builder.

XGETTEXTFROM

Name of file containing list of **xgettext(1)**'s source files. Autotools' users know this as `POTFILES.in` so they will in most cases set `XGETTEXTFROM="POTFILES.in"` here. The $XGETTEXTFROM files have same syntax and semantics as the well known GNU `POTFILES.in`. See `xgettext` tool and `POTUpdate` builder.

_XGETTEXTFROMFLAGS

Internal "macro". Genrates list of `-D<dir>` flags from the $XGETTEXTPATH list.

XGETTEXTFROMPREFIX

This flag is used to add single $XGETTEXTFROM file to **xgettext(1)**'s commandline (default: `'-f'`).

XGETTEXTFROMSUFFIX

(default: '')

XGETTEXTPATH

List of directories, there **xgettext(1)** will look for source files (default: `[]`).

Note: This variable works only together with $XGETTEXTFROM

See also `xgettext` tool and `POTUpdate` builder.

_XGETTEXTPATHFLAGS

Internal "macro". Generates list of `-f<file>` flags from $XGETTEXTFROM.

XGETTEXTPATHPREFIX

This flag is used to add single search path to **xgettext(1)**'s commandline (default: `'-D'`).

XGETTEXTPATHSUFFIX

(default: '')

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 if the `zipfile` module is 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.

Appendix B. Builders

This appendix contains descriptions of all of the Builders that are *potentially* available "out of the box" in this version of SCons.

`CFile()`

`env.CFile()`

Builds a C source file given a lex (`.l`) or yacc (`.y`) input file. The suffix specified by the $CFILESUFFIX 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')
```

`Command()`

`env.Command()`

The `Command` "Builder" is actually implemented as a function that looks like a Builder, but actually takes an additional argument of the action from which the Builder should be made. See the `Command` function description for the calling syntax and details.

`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 `\documentclass` or `\documentstyle` 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. The sources may be given as a string or as a node returned by a builder.

```
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.

`InstallVersionedLib()`

`env.InstallVersionedLib()`

Installs a versioned shared library. The $SHLIBVERSION construction variable should be defined in the environment to confirm the version number in the library name. The symlinks appropriate to the architecture will be generated.

```
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.

Examples:

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

Java source files can use the native encoding for the underlying OS. Since SCons compiles in simple ASCII mode by default, the compiler will generate warnings about unmappable characters, which may lead to errors as the file is processed further. In this case, the user must specify the `LANG` environment variable to tell the compiler what encoding is used. For portibility, it's best if the encoding is hard-coded so that the compile will work if it is done on a system with a different encoding.

```
env = Environment()
env['ENV']['LANG'] = 'en_GB.UTF-8'
```

`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
```

`MOFiles()`

`env.MOFiles()`

This builder belongs to `msgfmt` tool. The builder compiles `PO` files to `MO` files.

Example 1. Create `pl.mo` and `en.mo` by compiling `pl.po` and `en.po`:

```
# ...
env.MOFiles(['pl', 'en'])
```

Example 2. Compile files for languages defined in `LINGUAS` file:

```
# ...
env.MOFiles(LINGUAS_FILE = 1)
```

Example 3. Create `pl.mo` and `en.mo` by compiling `pl.po` and `en.po` plus files for languages defined in `LINGUAS` file:

```
# ...
env.MOFiles(['pl', 'en'], LINGUAS_FILE = 1)
```

Example 4. Compile files for languages defined in `LINGUAS` file (another version):

```
# ...
env['LINGUAS_FILE'] = 1
env.MOFiles()
```

`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).

The `MSVSProject` builder 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.

Note that because SCons always executes its build commands from the directory in which the `SConstruct` file is located, if you generate a project file in a different directory than the `SConstruct` directory, users will not be able to double-click on the file name in compilation error messages displayed in the Visual Studio console output window. This can be remedied by adding the Visual C/C++ `/FC` compiler option to the $CCFLAGS variable so that the compiler will print the full path name of any files that cause compilation errors.

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.

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 `target` 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')
```

`POInit()`

`env.POInit()`

This builder belongs to `msginit` tool. The builder initializes missing `PO` file(s) if $POAUTOINIT is set. If $POAUTOINIT is not set (default), `POInit` prints instruction for user (that is supposed to be a translator), telling how the `PO` file should be initialized. In normal projects *you should not use `POInit` and use `POUpdate` instead*. `POUpdate` chooses intelligently between **msgmerge(1)** and **msginit(1)**. `POInit` always uses **msginit(1)** and should be regarded as builder for special purposes or for temporary use (e.g. for quick, one time initialization of a bunch of `PO` files) or for tests.

Target nodes defined through `POInit` are not built by default (they're `Ignore`d from `'.'` node) but are added to special `Alias` (`'po-create'` by default). The alias name may be changed through the $POCREATE_ALIAS construction variable. All `PO` files defined through `POInit` may be easily initialized by **scons po-create**.

Example 1. Initialize `en.po` and `pl.po` from `messages.pot`:

```
# ...
env.POInit(['en', 'pl']) # messages.pot --> [en.po, pl.po]
```

Example 2. Initialize `en.po` and `pl.po` from `foo.pot`:

```
# ...
env.POInit(['en', 'pl'], ['foo']) # foo.pot --> [en.po, pl.po]
```

Example 3. Initialize `en.po` and `pl.po` from `foo.pot` but using $POTDOMAIN construction variable:

```
# ...
env.POInit(['en', 'pl'], POTDOMAIN='foo') # foo.pot --> [en.po, pl.po]
```

Example 4. Initialize `PO` files for languages defined in `LINGUAS` file. The files will be initialized from template `messages.pot`:

```
# ...
env.POInit(LINGUAS_FILE = 1) # needs 'LINGUAS' file
```

Example 5. Initialize `en.po` and `pl.pl` `PO` files plus files for languages defined in `LINGUAS` file. The files will be initialized from template `messages.pot`:

```
# ...
env.POInit(['en', 'pl'], LINGUAS_FILE = 1)
```

Example 6. You may preconfigure your environment first, and then initialize `PO` files:

```
# ...
env['POAUTOINIT'] = 1
env['LINGUAS_FILE'] = 1
env['POTDOMAIN'] = 'foo'
env.POInit()
```

which has same efect as:

```
# ...
env.POInit(POAUTOINIT = 1, LINGUAS_FILE = 1, POTDOMAIN = 'foo')
```

`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')
```

`POTUpdate()`

`env.POTUpdate()`

The builder belongs to `xgettext` tool. The builder updates target `POT` file if exists or creates one if it doesn't. The node is not built by default (i.e. it is `Ignore`d from `'.'`), but only on demand (i.e. when given `POT` file is required or when special alias is invoked). This builder adds its targe node (`messages.pot`, say) to a special alias (`pot-update` by default, see $POTUPDATE_ALIAS) so you can update/create them easily with **scons pot-update**. The file is not written until there is no real change in internationalized messages (or in comments that enter `POT` file).

> **Note:** You may see **xgettext(1)** being invoked by the `xgettext` tool even if there is no real change in internationalized messages (so the `POT` file is not being updated). This happens every time a source file has changed. In such case we invoke **xgettext(1)** and compare its output with the content of `POT` file to decide whether the file should be updated or not.

Example 1. Let's create `po/` directory and place following `SConstruct` script there:

```
# SConstruct in 'po/' subdir
env = Environment( tools = ['default', 'xgettext'] )
env.POTUpdate(['foo'], ['../a.cpp', '../b.cpp'])
env.POTUpdate(['bar'], ['../c.cpp', '../d.cpp'])
```

Then invoke scons few times:

```
user@host:$ scons             # Does not create foo.pot nor bar.pot
user@host:$ scons foo.pot     # Updates or creates foo.pot
user@host:$ scons pot-update  # Updates or creates foo.pot and bar.pot
user@host:$ scons -c          # Does not clean foo.pot nor bar.pot.
```

the results shall be as the comments above say.

Example 2. The `POTUpdate` builder may be used with no target specified, in which case default target `messages.pot` will be used. The default target may also be overriden by setting $POTDOMAIN construction variable or providing it as an override to `POTUpdate` builder:

```
# SConstruct script
env = Environment( tools = ['default', 'xgettext'] )
env['POTDOMAIN'] = "foo"
env.POTUpdate(source = ["a.cpp", "b.cpp"]) # Creates foo.pot ...
env.POTUpdate(POTDOMAIN = "bar", source = ["c.cpp", "d.cpp"]) # and bar.pot
```

Example 3. The sources may be specified within separate file, for example `POTFILES.in`:

```
# POTFILES.in in 'po/' subdirectory
../a.cpp
../b.cpp
# end of file
```

The name of the file (`POTFILES.in`) containing the list of sources is provided via $XGETTEXTFROM:

```
# SConstruct file in 'po/' subdirectory
env = Environment( tools = ['default', 'xgettext'] )
env.POTUpdate(XGETTEXTFROM = 'POTFILES.in')
```

Example 4. You may use $XGETTEXTPATH to define source search path. Assume, for example, that you have files `a.cpp`, `b.cpp`, `po/SConstruct`, `po/POTFILES.in`. Then your `POT`-related files could look as below:

```
# POTFILES.in in 'po/' subdirectory
a.cpp
b.cpp
# end of file
```

```
# SConstruct file in 'po/' subdirectory
env = Environment( tools = ['default', 'xgettext'] )
env.POTUpdate(XGETTEXTFROM = 'POTFILES.in', XGETTEXTPATH='../')
```

Example 5. Multiple search directories may be defined within a list, i.e. `XGETTEXTPATH = ['dir1', 'dir2', ...]`. The order in the list determines the search order of source files. The path to the first file found is used.

Let's create `0/1/po/SConstruct` script:

```
# SConstruct file in '0/1/po/' subdirectory
env = Environment( tools = ['default', 'xgettext'] )
env.POTUpdate(XGETTEXTFROM = 'POTFILES.in', XGETTEXTPATH=['../', '../../'])
```

and `0/1/po/POTFILES.in`:

```
# POTFILES.in in '0/1/po/' subdirectory
a.cpp
# end of file
```

Write two `*.cpp` files, the first one is `0/a.cpp`:

```
/* 0/a.cpp */
gettext("Hello from ../../a.cpp")
```

and the second is `0/1/a.cpp`:

```
/* 0/1/a.cpp */
gettext("Hello from ../a.cpp")
```

then run scons. You'll obtain `0/1/po/messages.pot` with the message `"Hello from ../a.cpp"`. When you reverse order in `$XGETTEXTFOM`, i.e. when you write SConscript as

```
# SConstruct file in '0/1/po/' subdirectory
env = Environment( tools = ['default', 'xgettext'] )
env.POTUpdate(XGETTEXTFROM = 'POTFILES.in', XGETTEXTPATH=['../../', '../'])
```

then the `messages.pot` will contain `msgid "Hello from ../../a.cpp"` line and not `msgid "Hello from ../a.cpp"`.

`POUpdate()`

`env.POUpdate()`

The builder belongs to `msgmerge` tool. The builder updates `PO` files with **msgmerge(1)**, or initializes missing `PO` files as described in documentation of `msginit` tool and `POInit` builder (see also $POAUTOINIT). Note, that `POUpdate` *does not add its targets to* `po-create` *alias* as `POInit` does.

Target nodes defined through `POUpdate` are not built by default (they're `Ignore`d from `'.'` node). Instead, they are added automatically to special `Alias` (`'po-update'` by default). The alias name may be changed through the $POUPDATE_ALIAS construction variable. You can easilly update `PO` files in your project by **scons po-update**.

Example 1. Update `en.po` and `pl.po` from `messages.pot` template (see also $POTDOMAIN), assuming that the later one exists or there is rule to build it (see `POTUpdate`):

```
# ...
env.POUpdate(['en','pl']) # messages.pot --> [en.po, pl.po]
```

Example 2. Update `en.po` and `pl.po` from `foo.pot` template:

```
# ...
env.POUpdate(['en', 'pl'], ['foo']) # foo.pot -->  [en.po, pl.pl]
```

Example 3. Update `en.po` and `pl.po` from `foo.pot` (another version):

```
# ...
env.POUpdate(['en', 'pl'], POTDOMAIN='foo') # foo.pot -- > [en.po, pl.pl]
```

Example 4. Update files for languages defined in `LINGUAS` file. The files are updated from `messages.pot` template:

```
# ...
env.POUpdate(LINGUAS_FILE = 1) # needs 'LINGUAS' file
```

Example 5. Same as above, but update from `foo.pot` template:

```
# ...
env.POUpdate(LINGUAS_FILE = 1, source = ['foo'])
```

Example 6. Update `en.po` and `pl.po` plus files for languages defined in `LINGUAS` file. The files are updated from `messages.pot` template:

```
# produce 'en.po', 'pl.po' + files defined in 'LINGUAS':
env.POUpdate(['en', 'pl' ], LINGUAS_FILE = 1)
```

Example 7. Use $POAUTOINIT to automatically initialize `PO` file if it doesn't exist:

```
# ...
env.POUpdate(LINGUAS_FILE = 1, POAUTOINIT = 1)
```

Example 8. Update `PO` files for languages defined in `LINGUAS` file. The files are updated from `foo.pot` template. All necessary settings are pre-configured via environment.

```
# ...
env['POAUTOINIT'] = 1
env['LINGUAS_FILE'] = 1
env['POTDOMAIN'] = 'foo'
```

```
env.POUpdate()
```

`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 some platforms, there is a distinction between a shared library (loaded automatically by the system to resolve external references) and a loadable module (explicitly loaded by user action). For maximum portability, use the `LoadableModule` builder for the latter.

When the $SHLIBVERSION construction variable is defined a versioned shared library is created. This modifies the $SHLINKFLAGS as required, adds the version number to the library name, and creates the symlinks that are needed. $SHLIBVERSION needs to be of the form X.Y.Z, where X and Y are numbers, and Z is a number but can also contain letters to designate alpha, beta, or release candidate patch levels.

This builder may create multiple links to the library. On a POSIX system, for the shared library libbar.so.2.3.1, the links created would be libbar.so, libbar.so.2, and libbar.so.2.3; on a Darwin (OSX) system the library would be libbar.2.3.1.dylib and the link would be libbar.dylib.

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 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
        ARM: CodeSourcery Sourcery Lite
.sx     assembly language file + C pre-processor
        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 more information.

`Substfile()`

`env.Substfile()`

The `Substfile` builder generates a single text file by concatenating the source files. Nested lists of sources are flattened. $LINESEPARATOR is used to separate the source files; see the description of `Textfile` for details.

If a single source file is present with an `.in` suffix, the suffix is stripped and the remainder is used as the default target name.

The prefix and suffix specified by the $SUBSTFILEPREFIX and $SUBSTFILESUFFIX construction variables (the null string by default in both cases) are automatically added to the target if they are not already present.

If a construction variable named $SUBST_DICT is present, it may be either a Python dictionary or a sequence of (key,value) tuples. If the former, the dictionary is converted into a list of tuples in an arbitrary order, so if one key is a prefix of another key or if one substitution could be further expanded by another subsitition, it is unpredictible whether the expansion will occur.

Any occurences in the source of a key are replaced by the corresponding value, which may be a Python callable function or a string. If a value is a function, it is first called (with no arguments) to produce a string. The string is *subst*-expanded and the result replaces the key.

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

env['prefix'] = '/usr/bin'
script_dict = {'@prefix@': '/bin', @exec_prefix@: '$prefix'}
env.Substfile('script.in', SUBST_DICT = script_dict)

conf_dict = {'%VERSION%': '1.2.3', '%BASE%': 'MyProg'}
env.Substfile('config.h.in', conf_dict, SUBST_DICT = conf_dict)

# UNPREDICTABLE - one key is a prefix of another
bad_foo = {'$foo': '$foo', '$foobar': '$foobar'}
env.Substfile('foo.in', SUBST_DICT = bad_foo)

# PREDICTABLE - keys are applied longest first
good_foo = [('$foobar', '$foobar'), ('$foo', '$foo')]
env.Substfile('foo.in', SUBST_DICT = good_foo)

# UNPREDICTABLE - one substitution could be futher expanded
bad_bar = {'@bar@': '@soap@', '@soap@': 'lye'}
env.Substfile('bar.in', SUBST_DICT = bad_bar)

# PREDICTABLE - substitutions are expanded in order
good_bar = (('@bar@', '@soap@'), ('@soap@', 'lye'))
env.Substfile('bar.in', SUBST_DICT = good_bar)

# the SUBST_DICT may be in common (and not an override)
substutions = {}
subst = Environment(tools = ['textfile'], SUBST_DICT = substitutions)
substitutions['@foo@'] = 'foo'
subst['SUBST_DICT']['@bar@'] = 'bar'
subst.Substfile('pgm1.c', [Value('#include "@foo@.h"'),
                           Value('#include "@bar@.h"'),
                           "common.in",
                           "pgm1.in"
                          ])
subst.Substfile('pgm2.c', [Value('#include "@foo@.h"'),
                           Value('#include "@bar@.h"'),
                           "common.in",
                           "pgm2.in"
                          ])
```

`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')
```

`Textfile()`

`env.Textfile()`

The `Textfile` builder generates a single text file. The source strings constitute the lines; nested lists of sources are flattened. $LINESEPARATOR is used to separate the strings.

If present, the $SUBST_DICT construction variable is used to modify the strings before they are written; see the `Substfile` description for details.

The prefix and suffix specified by the $TEXTFILEPREFIX and $TEXTFILESUFFIX construction variables (the null string and `.txt` by default, respectively) are automatically added to the target if they are not already present. Examples:

```
# builds/writes foo.txt
env.Textfile(target = 'foo.txt', source = ['Goethe', 42, 'Schiller'])

# builds/writes bar.txt
env.Textfile(target = 'bar',
             source = ['lalala', 'tanteratei'],
             LINESEPARATOR='|*')

# nested lists are flattened automatically
env.Textfile(target = 'blob',
             source = ['lalala', ['Goethe', 42 'Schiller'], 'tanteratei'])

# files may be used as input by wraping them in File()
env.Textfile(target = 'concat',  # concatenate files with a marker between
             source = [File('concat1'), File('concat2')],
             LINESEPARATOR = '====================\n')

Results are:
foo.txt
  ....8<----
```

```
  Goethe
  42
  Schiller
  ....8<---- (no linefeed at the end)

bar.txt:
  ....8<----
  lalala|*tanteratei
  ....8<---- (no linefeed at the end)

blob.txt
  ....8<----
  lalala
  Goethe
  42
  Schiller
  tanteratei
  ....8<---- (no linefeed at the end)
```

`Translate()`

`env.Translate()`

This pseudo-builder belongs to `gettext` toolset. The builder extracts internationalized messages from source files, updates `POT` template (if necessary) and then updates `PO` translations (if necessary). If $POAUTOINIT is set, missing `PO` files will be automatically created (i.e. without translator person intervention). The variables $LINGUAS_FILE and $POTDOMAIN are taken into acount too. All other construction variables used by `POTUpdate`, and `POUpdate` work here too.

Example 1. The simplest way is to specify input files and output languages inline in a SCons script when invoking `Translate`

```
# SConscript in 'po/' directory
env = Environment( tools = ["default", "gettext"] )
env['POAUTOINIT'] = 1
env.Translate(['en','pl'], ['../a.cpp','../b.cpp'])
```

Example 2. If you wish, you may also stick to conventional style known from autotools, i.e. using `POTFILES.in` and `LINGUAS` files

```
# LINGUAS
en pl
#end


# POTFILES.in
a.cpp
b.cpp
# end


# SConscript
```

```
env = Environment( tools = ["default", "gettext"] )
env['POAUTOINIT'] = 1
env['XGETTEXTPATH'] = ['../']
env.Translate(LINGUAS_FILE = 1, XGETTEXTFROM = 'POTFILES.in')
```

The last approach is perhaps the recommended one. It allows easily split internationalization/localization onto separate SCons scripts, where a script in source tree is responsible for translations (from sources to `PO` files) and script(s) under variant directories are responsible for compilation of `PO` to `MO` files to and for installation of `MO` files. The "gluing factor" synchronizing these two scripts is then the content of `LINGUAS` file. Note, that the updated `POT` and `PO` files are usually going to be committed back to the repository, so they must be updated within the source directory (and not in variant directories). Additionaly, the file listing of `po/` directory contains `LINGUAS` file, so the source tree looks familiar to translators, and they may work with the project in their usual way.

Example 3. Let's prepare a development tree as below

```
project/
 + SConstruct
 + build/
 + src/
     + po/
         + SConscript
         + SConscript.i18n
         + POTFILES.in
         + LINGUAS
```

with `build` being variant directory. Write the top-level `SConstruct` script as follows

```
# SConstruct
env = Environment( tools = ["default", "gettext"] )
VariantDir('build', 'src', duplicate = 0)
env['POAUTOINIT'] = 1
SConscript('src/po/SConscript.i18n', exports = 'env')
SConscript('build/po/SConscript', exports = 'env')
```

the `src/po/SConscript.i18n` as

```
# src/po/SConscript.i18n
Import('env')
env.Translate(LINGUAS_FILE=1, XGETTEXTFROM='POTFILES.in', XGETTEXTPATH=['../'])
```

and the `src/po/SConscript`

```
# src/po/SConscript
Import('env')
env.MOFiles(LINGUAS_FILE = 1)
```

Such setup produces POT and PO files under source tree in src/po/ and binary MO files under variant tree in build/po/. This way the POT and PO files are separated from other output files, which must not be committed back to source repositories (e.g. MO files).

Note: In above example, the PO files are not updated, nor created automatically when you issue **scons '.'** command. The files must be updated (created) by hand via **scons po-update** and then MO files can be compiled by running **scons '.'**.

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')
```

Appendix C. Tools

This appendix contains descriptions of all of the Tools modules that are available "out of the box" in this version of SCons.

386asm

Sets construction variables for the 386ASM assembler for the Phar Lap ETS embedded operating system.

Sets: $AS, $ASCOM, $ASFLAGS, $ASPPCOM, $ASPPFLAGS.

Uses: $CC, $CPPFLAGS, $_CPPDEFFLAGS, $_CPPINCFLAGS.

aixc++

Sets construction variables for the IMB xlc / Visual Age C++ compiler.

Sets: $CXX, $CXXVERSION, $SHCXX, $SHOBJSUFFIX.

aixcc

Sets construction variables for the IBM xlc / Visual Age C compiler.

Sets: $CC, $CCVERSION, $SHCC.

aixf77

Sets construction variables for the IBM Visual Age f77 Fortran compiler.

Sets: $F77, $SHF77.

aixlink

Sets construction variables for the IBM Visual Age linker.

Sets: $LINKFLAGS, $SHLIBSUFFIX, $SHLINKFLAGS.

applelink

Sets construction variables for the Apple linker (similar to the GNU linker).

Sets: $FRAMEWORKPATHPREFIX, $LDMODULECOM, $LDMODULEFLAGS, $LDMODULEPREFIX, $LDMODULESUFFIX, $LINKCOM, $SHLINKCOM, $SHLINKFLAGS, $_FRAMEWORKPATH, $_FRAMEWORKS.

Uses: $FRAMEWORKSFLAGS.

ar

Sets construction variables for the ar library archiver.

Sets: $AR, $ARCOM, $ARFLAGS, $LIBPREFIX, $LIBSUFFIX, $RANLIB, $RANLIBCOM, $RANLIBFLAGS.

as

Sets construction variables for the as assembler.

Sets: $AS, $ASCOM, $ASFLAGS, $ASPPCOM, $ASPPFLAGS.

Uses: $CC, $CPPFLAGS, $_CPPDEFFLAGS, $_CPPINCFLAGS.

bcc32

Sets construction variables for the bcc32 compiler.

Sets: $CC, $CCCOM, $CCFLAGS, $CFILESUFFIX, $CFLAGS, $CPPDEFPREFIX, $CPPDEFSUFFIX, $INCPREFIX, $INCSUFFIX, $SHCC, $SHCCCOM, $SHCCFLAGS, $SHCFLAGS, $SHOBJSUFFIX.

Uses: $_CPPDEFFLAGS, $_CPPINCFLAGS.

BitKeeper

Sets construction variables for the BitKeeper source code control system.

Sets: $BITKEEPER, $BITKEEPERCOM, $BITKEEPERGET, $BITKEEPERGETFLAGS.

Uses: $BITKEEPERCOMSTR.

cc

Sets construction variables for generic POSIX C copmilers.

Sets: $CC, $CCCOM, $CCFLAGS, $CFILESUFFIX, $CFLAGS, $CPPDEFPREFIX, $CPPDEFSUFFIX, $FRAMEWORKPATH, $FRAMEWORKS, $INCPREFIX, $INCSUFFIX, $SHCC, $SHCCCOM, $SHCCFLAGS, $SHCFLAGS, $SHOBJSUFFIX.

Uses: $PLATFORM.

cvf

Sets construction variables for the Compaq Visual Fortran compiler.

Sets: $FORTRAN, $FORTRANCOM, $FORTRANMODDIR, $FORTRANMODDIRPREFIX, $FORTRANMODDIRSUFFIX, $FORTRANPPCOM, $OBJSUFFIX, $SHFORTRANCOM, $SHFORTRANPPCOM.

Uses: $CPPFLAGS, $FORTRANFLAGS, $SHFORTRANFLAGS, $_CPPDEFFLAGS, $_FORTRANINCFLAGS, $_FORTRANMODFLAG.

CVS

Sets construction variables for the CVS source code management system.

Sets: $CVS, $CVSCOFLAGS, $CVSCOM, $CVSFLAGS.

Uses: $CVSCOMSTR.

cXX

Sets construction variables for generic POSIX C++ compilers.

Sets: $CPPDEFPREFIX, $CPPDEFSUFFIX, $CXX, $CXXCOM, $CXXFILESUFFIX, $CXXFLAGS, $INCPREFIX, $INCSUFFIX, $OBJSUFFIX, $SHCXX, $SHCXXCOM, $SHCXXFLAGS, $SHOBJSUFFIX.

Uses: $CXXCOMSTR.

default

Sets variables by calling a default list of Tool modules for the platform on which SCons is running.

dmd

Sets construction variables for D language compilers (the Digital Mars D compiler, or GDC).

dvi

Attaches the `DVI` builder to the construction environment.

dvipdf

Sets construction variables for the dvipdf utility.

Sets: $DVIPDF, $DVIPDFCOM, $DVIPDFFLAGS.

Uses: $DVIPDFCOMSTR.

dvips

Sets construction variables for the dvips utility.

Sets: $DVIPS, $DVIPSFLAGS, $PSCOM, $PSPREFIX, $PSSUFFIX.

Uses: $PSCOMSTR.

f77

Set construction variables for generic POSIX Fortran 77 compilers.

Sets: $F77, $F77COM, $F77FILESUFFIXES, $F77FLAGS, $F77PPCOM, $F77PPFILESUFFIXES, $FORTRAN, $FORTRANCOM, $FORTRANFLAGS, $SHF77, $SHF77COM, $SHF77FLAGS, $SHF77PPCOM, $SHFORTRAN, $SHFORTRANCOM, $SHFORTRANFLAGS, $SHFORTRANPPCOM, $_F77INCFLAGS.

Uses: $F77COMSTR, $F77PPCOMSTR, $FORTRANCOMSTR, $FORTRANPPCOMSTR, $SHF77COMSTR, $SHF77PPCOMSTR, $SHFORTRANCOMSTR, $SHFORTRANPPCOMSTR.

f90

Set construction variables for generic POSIX Fortran 90 compilers.

Sets: $F90, $F90COM, $F90FLAGS, $F90PPCOM, $SHF90, $SHF90COM, $SHF90FLAGS, $SHF90PPCOM, $_F90INCFLAGS.

Uses: $F90COMSTR, $F90PPCOMSTR, $SHF90COMSTR, $SHF90PPCOMSTR.

f95

Set construction variables for generic POSIX Fortran 95 compilers.

Sets: $F95, $F95COM, $F95FLAGS, $F95PPCOM, $SHF95, $SHF95COM, $SHF95FLAGS, $SHF95PPCOM, $_F95INCFLAGS.

Uses: $F95COMSTR, $F95PPCOMSTR, $SHF95COMSTR, $SHF95PPCOMSTR.

fortran

Set construction variables for generic POSIX Fortran compilers.

Sets: $FORTRAN, $FORTRANCOM, $FORTRANFLAGS, $SHFORTRAN, $SHFORTRANCOM, $SHFORTRANFLAGS, $SHFORTRANPPCOM.

Uses: $FORTRANCOMSTR, $FORTRANPPCOMSTR, $SHFORTRANCOMSTR, $SHFORTRANPPCOMSTR.

g++

Set construction variables for the gXX C++ compiler.

Sets: $CXX, $CXXVERSION, $SHCXXFLAGS, $SHOBJSUFFIX.

g77

Set construction variables for the g77 Fortran compiler. Calls the `f77` Tool module to set variables.

gas

Sets construction variables for the gas assembler. Calls the `as` module.

Sets: $AS.

gcc

Set construction variables for the gcc C compiler.

Sets: $CC, $CCVERSION, $SHCCFLAGS.

gettext

This is actually a toolset, which supports internationalization and localization of sofware being constructed with SCons. The toolset loads following tools:

- `xgettext` - to extract internationalized messages from source code to `POT` file(s),

- `msginit` - may be optionally used to initialize `PO` files,
- `msgmerge` - to update `PO` files, that already contain translated messages,
- `msgfmt` - to compile textual `PO` file to binary installable `MO` file.

When you enable `gettext`, it internally loads all abovementioned tools, so you're encouraged to see their individual documentation.

Each of the above tools provides its own builder(s) which may be used to perform particular activities related to software internationalization. You may be however interested in *top-level* builder `Translate` described few paragraphs later.

To use `gettext` tools add `'gettext'` tool to your environment:

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

gfortran

Sets construction variables for the GNU F95/F2003 GNU compiler.

Sets: $F77, $F90, $F95, $FORTRAN, $SHF77, $SHF77FLAGS, $SHF90, $SHF90FLAGS, $SHF95, $SHF95FLAGS, $SHFORTRAN, $SHFORTRANFLAGS.

gnulink

Set construction variables for GNU linker/loader.

Sets: $RPATHPREFIX, $RPATHSUFFIX, $SHLINKFLAGS.

gs

Set construction variables for Ghostscript.

Sets: $GS, $GSCOM, $GSFLAGS.

Uses: $GSCOMSTR.

hpc++

Set construction variables for the compilers aCC on HP/UX systems.

hpcc

Set construction variables for the aCC on HP/UX systems. Calls the `cXX` tool for additional variables.

Sets: $CXX, $CXXVERSION, $SHCXXFLAGS.

hplink

Sets construction variables for the linker on HP/UX systems.

Sets: $LINKFLAGS, $SHLIBSUFFIX, $SHLINKFLAGS.

icc

Sets construction variables for the icc compiler on OS/2 systems.

Sets: $CC, $CCCOM, $CFILESUFFIX, $CPPDEFPREFIX, $CPPDEFSUFFIX, $CXXCOM, $CXXFILESUFFIX, $INCPREFIX, $INCSUFFIX.

Uses: $CCFLAGS, $CFLAGS, $CPPFLAGS, $_CPPDEFFLAGS, $_CPPINCFLAGS.

icl

Sets construction variables for the Intel C/C++ compiler. Calls the `intelc` Tool module to set its variables.

ifl

Sets construction variables for the Intel Fortran compiler.

Sets: $FORTRAN, $FORTRANCOM, $FORTRANPPCOM, $SHFORTRANCOM, $SHFORTRANPPCOM.

Uses: $CPPFLAGS, $FORTRANFLAGS, $_CPPDEFFLAGS, $_FORTRANINCFLAGS.

ifort

Sets construction variables for newer versions of the Intel Fortran compiler for Linux.

Sets: $F77, $F90, $F95, $FORTRAN, $SHF77, $SHF77FLAGS, $SHF90, $SHF90FLAGS, $SHF95, $SHF95FLAGS, $SHFORTRAN, $SHFORTRANFLAGS.

ilink

Sets construction variables for the ilink linker on OS/2 systems.

Sets: $LIBDIRPREFIX, $LIBDIRSUFFIX, $LIBLINKPREFIX, $LIBLINKSUFFIX, $LINK, $LINKCOM, $LINKFLAGS.

ilink32

Sets construction variables for the Borland ilink32 linker.

Sets: $LIBDIRPREFIX, $LIBDIRSUFFIX, $LIBLINKPREFIX, $LIBLINKSUFFIX, $LINK, $LINKCOM, $LINKFLAGS.

install

Sets construction variables for file and directory installation.

Sets: $INSTALL, $INSTALLSTR.

intelc

Sets construction variables for the Intel C/C++ compiler (Linux and Windows, version 7 and later). Calls the `gcc` or `msvc` (on Linux and Windows, respectively) to set underlying variables.

Sets: $AR, $CC, $CXX, $INTEL_C_COMPILER_VERSION, $LINK.

jar

Sets construction variables for the jar utility.

Sets: $JAR, $JARCOM, $JARFLAGS, $JARSUFFIX.

Uses: $JARCOMSTR.

javac

Sets construction variables for the javac compiler.

Sets: $JAVABOOTCLASSPATH, $JAVAC, $JAVACCOM, $JAVACFLAGS, $JAVACLASSPATH, $JAVACLASSSUFFIX, $JAVASOURCEPATH, $JAVASUFFIX.

Uses: $JAVACCOMSTR.

javah

Sets construction variables for the javah tool.

Sets: $JAVACLASSSUFFIX, $JAVAH, $JAVAHCOM, $JAVAHFLAGS.

Uses: $JAVACLASSPATH, $JAVAHCOMSTR.

latex

Sets construction variables for the latex utility.

Sets: $LATEX, $LATEXCOM, $LATEXFLAGS.

Uses: $LATEXCOMSTR.

lex

Sets construction variables for the lex lexical analyser.

Sets: $LEX, $LEXCOM, $LEXFLAGS.

Uses: $LEXCOMSTR.

link

Sets construction variables for generic POSIX linkers.

Sets: $LDMODULE, $LDMODULECOM, $LDMODULEFLAGS, $LDMODULEPREFIX, $LDMODULESUFFIX, $LIBDIRPREFIX, $LIBDIRSUFFIX, $LIBLINKPREFIX, $LIBLINKSUFFIX, $LINK, $LINKCOM, $LINKFLAGS, $SHLIBSUFFIX, $SHLINK, $SHLINKCOM, $SHLINKFLAGS.

Uses: $LDMODULECOMSTR, $LINKCOMSTR, $SHLINKCOMSTR.

linkloc

Sets construction variables for the LinkLoc linker for the Phar Lap ETS embedded operating system.

Sets: $LIBDIRPREFIX, $LIBDIRSUFFIX, $LIBLINKPREFIX, $LIBLINKSUFFIX, $LINK, $LINKCOM, $LINKFLAGS, $SHLINK, $SHLINKCOM, $SHLINKFLAGS.

Uses: $LINKCOMSTR, $SHLINKCOMSTR.

m4

Sets construction variables for the m4 macro processor.

Sets: $M4, $M4COM, $M4FLAGS.

Uses: $M4COMSTR.

masm

Sets construction variables for the Microsoft assembler.

Sets: $AS, $ASCOM, $ASFLAGS, $ASPPCOM, $ASPPFLAGS.

Uses: $ASCOMSTR, $ASPPCOMSTR, $CPPFLAGS, $_CPPDEFFLAGS, $_CPPINCFLAGS.

midl

Sets construction variables for the Microsoft IDL compiler.

Sets: $MIDL, $MIDLCOM, $MIDLFLAGS.

Uses: $MIDLCOMSTR.

mingw

Sets construction variables for MinGW (Minimal Gnu on Windows).

Sets: $AS, $CC, $CXX, $LDMODULECOM, $LIBPREFIX, $LIBSUFFIX, $OBJSUFFIX, $RC, $RCCOM, $RCFLAGS, $RCINCFLAGS, $RCINCPREFIX, $RCINCSUFFIX, $SHCCFLAGS, $SHCXXFLAGS, $SHLINKCOM, $SHLINKFLAGS, $SHOBJSUFFIX, $WINDOWSDEFPREFIX, $WINDOWSDEFSUFFIX.

Uses: $RCCOMSTR, $SHLINKCOMSTR.

msgfmt

This scons tool is a part of scons `gettext` toolset. It provides scons interface to **msgfmt(1)** command, which generates binary message catalog (`MO`) from a textual translation description (`PO`).

Sets: $MOSUFFIX, $MSGFMT, $MSGFMTCOM, $MSGFMTCOMSTR, $MSGFMTFLAGS, $POSUFFIX.

Uses: $LINGUAS_FILE.

msginit

This scons tool is a part of scons `gettext` toolset. It provides scons interface to **msginit(1)** program, which creates new `PO` file, initializing the meta information with values from user's environment (or options).

Sets: $MSGINIT, $MSGINITCOM, $MSGINITCOMSTR, $MSGINITFLAGS, $POAUTOINIT, $POCREATE_ALIAS, $POSUFFIX, $POTSUFFIX, $_MSGINITLOCALE.

Uses: $LINGUAS_FILE, $POAUTOINIT, $POTDOMAIN.

msgmerge

This scons tool is a part of scons `gettext` toolset. It provides scons interface to **msgmerge(1)** command, which merges two Uniform style `.po` files together.

Sets: $MSGMERGE, $MSGMERGECOM, $MSGMERGECOMSTR, $MSGMERGEFLAGS, $POSUFFIX, $POTSUFFIX, $POUPDATE_ALIAS.

Uses: $LINGUAS_FILE, $POAUTOINIT, $POTDOMAIN.

mslib

Sets construction variables for the Microsoft mslib library archiver.

Sets: $AR, $ARCOM, $ARFLAGS, $LIBPREFIX, $LIBSUFFIX.

Uses: $ARCOMSTR.

mslink

Sets construction variables for the Microsoft linker.

Sets: $LDMODULE, $LDMODULECOM, $LDMODULEFLAGS, $LDMODULEPREFIX, $LDMODULESUFFIX, $LIBDIRPREFIX, $LIBDIRSUFFIX, $LIBLINKPREFIX, $LIBLINKSUFFIX, $LINK, $LINKCOM, $LINKFLAGS, $REGSVR, $REGSVRCOM, $REGSVRFLAGS, $SHLINK, $SHLINKCOM, $SHLINKFLAGS, $WIN32DEFPREFIX, $WIN32DEFSUFFIX, $WIN32EXPPREFIX, $WIN32EXPSUFFIX, $WINDOWSDEFPREFIX, $WINDOWSDEFSUFFIX, $WINDOWSEXPPREFIX, $WINDOWSEXPSUFFIX, $WINDOWSPROGMANIFESTPREFIX, $WINDOWSPROGMANIFESTSUFFIX, $WINDOWSSHLIBMANIFESTPREFIX, $WINDOWSSHLIBMANIFESTSUFFIX, $WINDOWS_INSERT_DEF.

Uses: $LDMODULECOMSTR, $LINKCOMSTR, $REGSVRCOMSTR, $SHLINKCOMSTR.

mssdk

Sets variables for Microsoft Platform SDK and/or Windows SDK. Note that unlike most other Tool modules, mssdk does not set construction variables, but sets the *environment variables* in the environment SCons uses to execute the Microsoft toolchain: `%INCLUDE%`, `%LIB%`, `%LIBPATH%` and `%PATH%`.

Uses: $MSSDK_DIR, $MSSDK_VERSION, $MSVS_VERSION.

msvc

Sets construction variables for the Microsoft Visual C/C++ compiler.

Sets: $BUILDERS, $CC, $CCCOM, $CCFLAGS, $CCPCHFLAGS, $CCPDBFLAGS, $CFILESUFFIX, $CFLAGS, $CPPDEFPREFIX, $CPPDEFSUFFIX, $CXX, $CXXCOM, $CXXFILESUFFIX, $CXXFLAGS, $INCPREFIX, $INCSUFFIX, $OBJPREFIX, $OBJSUFFIX, $PCHCOM, $PCHPDBFLAGS, $RC, $RCCOM, $RCFLAGS, $SHCC, $SHCCCOM, $SHCCFLAGS, $SHCFLAGS, $SHCXX, $SHCXXCOM, $SHCXXFLAGS, $SHOBJPREFIX, $SHOBJSUFFIX.

Uses: $CCCOMSTR, $CXXCOMSTR, $PCH, $PCHSTOP, $PDB, $SHCCCOMSTR, $SHCXXCOMSTR.

msvs

Sets construction variables for Microsoft Visual Studio.

Sets: $MSVSBUILDCOM, $MSVSCLEANCOM, $MSVSENCODING, $MSVSPROJECTCOM, $MSVSREBUILDCOM, $MSVSSCONS, $MSVSSCONSCOM, $MSVSSCONSCRIPT, $MSVSSCONSFLAGS, $MSVSSOLUTIONCOM.

mwcc

Sets construction variables for the Metrowerks CodeWarrior compiler.

Sets: $CC, $CCCOM, $CFILESUFFIX, $CPPDEFPREFIX, $CPPDEFSUFFIX, $CXX, $CXXCOM, $CXXFILESUFFIX, $INCPREFIX, $INCSUFFIX, $MWCW_VERSION, $MWCW_VERSIONS, $SHCC, $SHCCCOM, $SHCCFLAGS, $SHCFLAGS, $SHCXX, $SHCXXCOM, $SHCXXFLAGS.

Uses: $CCCOMSTR, $CXXCOMSTR, $SHCCCOMSTR, $SHCXXCOMSTR.

mwld

Sets construction variables for the Metrowerks CodeWarrior linker.

Sets: $AR, $ARCOM, $LIBDIRPREFIX, $LIBDIRSUFFIX, $LIBLINKPREFIX, $LIBLINKSUFFIX, $LINK, $LINKCOM, $SHLINK, $SHLINKCOM, $SHLINKFLAGS.

nasm

Sets construction variables for the nasm Netwide Assembler.

Sets: $AS, $ASCOM, $ASFLAGS, $ASPPCOM, $ASPPFLAGS.

Uses: $ASCOMSTR, $ASPPCOMSTR.

Packaging

Sets construction variables for the `Package` Builder.

packaging

A framework for building binary and source packages.

pdf

Sets construction variables for the Portable Document Format builder.

Sets: $PDFPREFIX, $PDFSUFFIX.

pdflatex

Sets construction variables for the pdflatex utility.

Sets: $LATEXRETRIES, $PDFLATEX, $PDFLATEXCOM, $PDFLATEXFLAGS.

Uses: $PDFLATEXCOMSTR.

pdftex

Sets construction variables for the pdftex utility.

Sets: $LATEXRETRIES, $PDFLATEX, $PDFLATEXCOM, $PDFLATEXFLAGS, $PDFTEX, $PDFTEXCOM, $PDFTEXFLAGS.

Uses: $PDFLATEXCOMSTR, $PDFTEXCOMSTR.

Perforce

Sets construction variables for interacting with the Perforce source code management system.

Sets: $P4, $P4COM, $P4FLAGS.

Uses: $P4COMSTR.

qt

Sets construction variables for building Qt applications.

Sets: $QTDIR, $QT_AUTOSCAN, $QT_BINPATH, $QT_CPPPATH, $QT_LIB, $QT_LIBPATH, $QT_MOC, $QT_MOCCXXPREFIX, $QT_MOCCXXSUFFIX, $QT_MOCFROMCXXCOM, $QT_MOCFROMCXXFLAGS, $QT_MOCFROMHCOM, $QT_MOCFROMHFLAGS, $QT_MOCHPREFIX, $QT_MOCHSUFFIX, $QT_UIC, $QT_UICCOM, $QT_UICDECLFLAGS, $QT_UICDECLPREFIX, $QT_UICDECLSUFFIX, $QT_UICIMPLFLAGS, $QT_UICIMPLPREFIX, $QT_UICIMPLSUFFIX, $QT_UISUFFIX.

RCS

Sets construction variables for the interaction with the Revision Control System.

Sets: $RCS, $RCS_CO, $RCS_COCOM, $RCS_COFLAGS.

Uses: $RCS_COCOMSTR.

rmic

Sets construction variables for the rmic utility.

Sets: $JAVACLASSSUFFIX, $RMIC, $RMICCOM, $RMICFLAGS.

Uses: $RMICCOMSTR.

rpcgen

Sets construction variables for building with RPCGEN.

Sets: $RPCGEN, $RPCGENCLIENTFLAGS, $RPCGENFLAGS, $RPCGENHEADERFLAGS, $RPCGENSERVICEFLAGS, $RPCGENXDRFLAGS.

SCCS

Sets construction variables for interacting with the Source Code Control System.

Sets: $SCCS, $SCCSCOM, $SCCSFLAGS, $SCCSGETFLAGS.

Uses: $SCCSCOMSTR.

sgiar

Sets construction variables for the SGI library archiver.

Sets: $AR, $ARCOMSTR, $ARFLAGS, $LIBPREFIX, $LIBSUFFIX, $SHLINK, $SHLINKFLAGS.

Uses: $ARCOMSTR, $SHLINKCOMSTR.

sgic++

Sets construction variables for the SGI C++ compiler.

Sets: $CXX, $CXXFLAGS, $SHCXX, $SHOBJSUFFIX.

sgicc

Sets construction variables for the SGI C compiler.

Sets: $CXX, $SHOBJSUFFIX.

sgilink

Sets construction variables for the SGI linker.

Sets: $LINK, $RPATHPREFIX, $RPATHSUFFIX, $SHLINKFLAGS.

sunar

Sets construction variables for the Sun library archiver.

Sets: $AR, $ARCOM, $ARFLAGS, $LIBPREFIX, $LIBSUFFIX, $SHLINK, $SHLINKCOM, $SHLINKFLAGS.

Uses: $ARCOMSTR, $SHLINKCOMSTR.

sunc++

Sets construction variables for the Sun C++ compiler.

Sets: $CXX, $CXXVERSION, $SHCXX, $SHCXXFLAGS, $SHOBJPREFIX, $SHOBJSUFFIX.

suncc

Sets construction variables for the Sun C compiler.

Sets: $CXX, $SHCCFLAGS, $SHOBJPREFIX, $SHOBJSUFFIX.

sunf77

Set construction variables for the Sun f77 Fortran compiler.

Sets: $F77, $FORTRAN, $SHF77, $SHF77FLAGS, $SHFORTRAN, $SHFORTRANFLAGS.

sunf90

Set construction variables for the Sun f90 Fortran compiler.

Sets: $F90, $FORTRAN, $SHF90, $SHF90FLAGS, $SHFORTRAN, $SHFORTRANFLAGS.

sunf95

Set construction variables for the Sun f95 Fortran compiler.

Sets: $F95, $FORTRAN, $SHF95, $SHF95FLAGS, $SHFORTRAN, $SHFORTRANFLAGS.

sunlink

Sets construction variables for the Sun linker.

Sets: $RPATHPREFIX, $RPATHSUFFIX, $SHLINKFLAGS.

swig

Sets construction variables for the SWIG interface generator.

Sets: $SWIG, $SWIGCFILESUFFIX, $SWIGCOM, $SWIGCXXFILESUFFIX, $SWIGDIRECTORSUFFIX, $SWIGFLAGS, $SWIGINCPREFIX, $SWIGINCSUFFIX, $SWIGPATH, $SWIGVERSION, $_SWIGINCFLAGS.

Uses: $SWIGCOMSTR.

tar

Sets construction variables for the tar archiver.

Sets: $TAR, $TARCOM, $TARFLAGS, $TARSUFFIX.

Uses: $TARCOMSTR.

tex

Sets construction variables for the TeX formatter and typesetter.

Sets: $BIBTEX, $BIBTEXCOM, $BIBTEXFLAGS, $LATEX, $LATEXCOM, $LATEXFLAGS, $MAKEINDEX, $MAKEINDEXCOM, $MAKEINDEXFLAGS, $TEX, $TEXCOM, $TEXFLAGS.

Uses: $BIBTEXCOMSTR, $LATEXCOMSTR, $MAKEINDEXCOMSTR, $TEXCOMSTR.

textfile

Set construction variables for the `Textfile` and `Substfile` builders.

Sets: $LINESEPARATOR, $SUBSTFILEPREFIX, $SUBSTFILESUFFIX, $TEXTFILEPREFIX, $TEXTFILESUFFIX.

Uses: $SUBST_DICT.

tlib

Sets construction variables for the Borlan tib library archiver.

Sets: $AR, $ARCOM, $ARFLAGS, $LIBPREFIX, $LIBSUFFIX.

Uses: $ARCOMSTR.

xgettext

This scons tool is a part of scons `gettext` toolset. It provides scons interface to **xgettext(1)** program, which extracts internationalized messages from source code. The tool provides `POTUpdate` builder to make `PO` *Template* files.

Sets: $POTSUFFIX, $POTUPDATE_ALIAS, $XGETTEXTCOM, $XGETTEXTCOMSTR, $XGETTEXTFLAGS, $XGETTEXTFROM, $XGETTEXTFROMPREFIX, $XGETTEXTFROMSUFFIX, $XGETTEXTPATH, $XGETTEXTPATHPREFIX, $XGETTEXTPATHSUFFIX, $_XGETTEXTDOMAIN, $_XGETTEXTFROMFLAGS, $_XGETTEXTPATHFLAGS.

Uses: $POTDOMAIN.

yacc

Sets construction variables for the yacc parse generator.

Sets: $YACC, $YACCCOM, $YACCFLAGS, $YACCHFILESUFFIX, $YACCHXXFILESUFFIX, $YACCVCGFILESUFFIX.

Uses: $YACCCOMSTR.

zip

Sets construction variables for the zip archiver.

Sets: $ZIP, $ZIPCOM, $ZIPCOMPRESSION, $ZIPFLAGS, $ZIPSUFFIX.

Uses: $ZIPCOMSTR.

Appendix D. Functions and Environment Methods

This appendix contains descriptions of all of the function and construction environment methods in this version of SCons

```
Action(action, [cmd/str/fun, [var, ...]] [option=value, ...])
env.Action(action, [cmd/str/fun, [var, ...]] [option=value, ...])
```

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 argument 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.

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).

When multiple targets are supplied, the action may be called multiple times, once after each action that generates one or more targets in the list.

`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).

When multiple targets are specified, the action(s) may be called multiple times, once before each action that generates one or more targets in the list.

Note that if any of the targets are built in multiple steps, the action will be invoked just before the "final" action that specifically generates the specified target(s). For example, when building an executable program from a specified source `.c` file via an intermediate object file:

```
foo = Program('foo.c')
AddPreAction(foo, 'pre_action')
```

The specified `pre_action` would be executed before `scons` calls the link command that actually generates the executable program binary `foo`, not before compiling the `foo.c` file into an object file.

`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, delete_existing])`

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.

If `delete_existing` is 0, then adding a path that already exists will not move it to the end; it will stay where it is in the list.

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, [...], delete_existing=0)`

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. However, if delete_existing is 1, existing matching values are removed first, so existing values in the arg list move to the end of 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.

This function is deprecated. For details, see the entry for the `SourceCode` function.

Example:

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

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

`env.BuildDir(build_dir, src_dir, [duplicate])`

Deprecated synonyms for `VariantDir` and `env.VariantDir()`. The `build_dir` argument becomes the `variant_dir` argument of `VariantDir` or `env.VariantDir()`.

`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')
```

In this example, installing the project creates a subdirectory for the documentation. This statement causes the subdirectory to be removed if the project is deinstalled.

```
Clean(docdir, os.path.join(docdir, projectname))
```

`env.Clone([key=val, ...])`

Returns 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')
```

`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.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.

This function is deprecated. For details, see the entry for 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` will be rebuilt whenever the `dependency` has changed. Both the specified `target` and `dependency` can be a string (usually the path name of a file or directory) or Node objects, or a list of strings or Node objects (such as returned by a Builder call). 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')

mylib = env.Library('mylib.c')
installed_lib = env.Install('lib', mylib)
bar = env.Program('bar.c')

# Arrange for the library to be copied into the installation
# directory before trying to build the "bar" program.
# (Note that this is for example only.  A "real" library
# dependency would normally be configured through the $LIBS
# and $LIBPATH variables, not using an env.Depends() call.)

env.Depends(bar, installed_lib)
```

`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:

```
'$CC -c -o $TARGET $CCFLAGS $CPPFLAGS $_CPPDEFFLAGS $_CPPINCFLAGS $SOURCES'
```

While this SConstruct:

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

will print:

```
{ 'AR': 'ar',
  'ARCOM': '$AR $ARFLAGS $TARGET $SOURCES\n$RANLIB $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.

Note that `scons` will print an error message if the executed `action` fails--that is, exits with or returns a non-zero value. `scons` will *not*, however, automatically terminate the build if the specified `action` fails. If you want the build to stop in response to a failed `Execute` call, you must explicitly check for a non-zero return value:

```
Execute(Copy('file.out', 'file.in'))

if Execute("mkdir sub/dir/ectory"):
    # The mkdir failed, don't try to build.
    Exit(1)
```

```
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. Keyword arguments can be used to provide names and their values. 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)

# 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`. `dirs` may be a list of directory names or a single directory name. In addition to searching for files that exist in the filesystem, 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()
```

`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'))
```

`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.

`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
```

If the first character of a filename is a dot, it must be matched explicitly. 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.)

Examples:

```
Program('foo', Glob('*.c'))
Zip('/tmp/everything', Glob('.??*') + Glob('*'))
```

`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.

You can also use `Ignore` to remove a target from the default build. In order to do this you must specify the directory the target will be built in as the target, and the file you want to skip building as the dependency.

Note that this will only remove the dependencies listed from the files built by default. It will still be built if that dependency is needed by another object being built. See the third and forth examples below.

Examples:

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

`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 environment'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 this 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.

This function is deprecated. For details, see the entry for 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)`

The `Platform` form returns a callable object that can be used to initialize a construction environment using the platform keyword of the `Environment` function.

Example:

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

The `env.Platform` form 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.

`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, delete_existing])`

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.

If `delete_existing` is 0, then adding a path that already exists will not move it to the beginning; it will stay where it is in the list.

Example:

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

The above example will print:

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

`env.PrependUnique(key=val, delete_existing=0, [...])`

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. However, if

delete_existing is 1, existing matching values are removed first, so existing values in the arg list move to the front of the list.

Example:

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

```
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(object):
    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)
```

`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:

This function is deprecated. For details, see the entry for 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, recursiv
env.Scanner(function, [argument, keys, path_function, node_class, node_factory, scan_check, recu
```

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

This tells `scons` to execute one or more subsidiary SConscript (configuration) files. Any variables returned by a called script using `Return` will be returned by the call to `SConscript`. 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`). Examples:

```
SConscript('SConscript')      # run SConscript in the current directory
SConscript('src/SConscript')  # run SConscript in the src directory
SConscript(['src/SConscript', 'doc/SConscript'])
config = SConscript('MyConfig.py')
```

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 first three examples below have the same effect as the first three examples above:

```
SConscript(dirs='.')      # run SConscript in the current directory
SConscript(dirs='src')    # run SConscript in the src directory
SConscript(dirs=['src', 'doc'])
SConscript(dirs=['sub1', 'sub2'], name='MySConscript')
```

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. Examples:

```
foo = SConscript('sub/SConscript', exports='env')
SConscript('dir/SConscript', exports=['env', 'variable'])
SConscript(dirs='subdir', exports='env variable')
SConscript(dirs=['one', 'two', 'three'], exports='shared_info')
```

If the optional `variant_dir` argument is present, it causes an effect equivalent to the `VariantDir` method described below. (If `variant_dir` is not present, the `duplicate` argument is ignored.) The `variant_dir` argument is interpreted relative to the directory of the calling `SConscript` file. See the description of the `VariantDir` function below for additional details and restrictions.

If `variant_dir` is present, the source directory is the directory in which the `SConscript` file resides and the `SConscript` file is evaluated as if it were in the `variant_dir` directory:

```
SConscript('src/SConscript', variant_dir = 'build')
```

is equivalent to

```
VariantDir('build', 'src')
SConscript('build/SConscript')
```

This later paradigm is often used when the sources are in the same directory as the `SConstruct`:

```
SConscript('SConscript', variant_dir = 'build')
```

is equivalent to

```
VariantDir('build', '.')
SConscript('build/SConscript')
```

Here are some composite examples:

```
# collect the configuration information and use it to build src and doc
shared_info = SConscript('MyConfig.py')
SConscript('src/SConscript', exports='shared_info')
SConscript('doc/SConscript', exports='shared_info')


# build debugging and production versions.  SConscript
# can use Dir('.').path to determine variant.
SConscript('SConscript', variant_dir='debug', duplicate=0)
SConscript('SConscript', variant_dir='prod', duplicate=0)


# build debugging and production versions.  SConscript
# is passed flags to use.
```

```
opts = { 'CPPDEFINES' : ['DEBUG'], 'CCFLAGS' : '-pgdb' }
SConscript('SConscript', variant_dir='debug', duplicate=0, exports=opts)
opts = { 'CPPDEFINES' : ['NODEBUG'], 'CCFLAGS' : '-O' }
SConscript('SConscript', variant_dir='prod', duplicate=0, exports=opts)


# build common documentation and compile for different architectures
SConscript('doc/SConscript', variant_dir='build/doc', duplicate=0)
SConscript('src/SConscript', variant_dir='build/x86', duplicate=0)
SConscript('src/SConscript', variant_dir='build/ppc', duplicate=0)
```

`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.

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 'FOO' not in env: 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 file that is created or updated 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, and various log files are created updated as side effects of various TeX commands. 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.

Because multiple build commands may update the same side effect file, by default the `side_effect` target is *not* automatically removed when the `target` is removed by the `-c` option. (Note, however, that the `side_effect` might be removed as part of cleaning the directory in which it lives.) If you want to make sure the `side_effect` is cleaned whenever a specific `target` is cleaned, you must specify this explicitly with the `Clean` or `env.Clean` function.

`SourceCode(entries, builder)`

`env.SourceCode(entries, builder)`

This function and its associate factory functions are deprecated. There is no replacement. The intended use was to keep a local tree in sync with an archive, but in actuality the function only causes the archive to be fetched on the first run. Synchronizing with the archive is best done external to SCons.

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)
```

`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
""")
```

`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 λ 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)
```

`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 )
```

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

The `Tool` form of the function 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
```

The `env.Tool` form of the function 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])`

Use the `VariantDir` function to create a copy of your sources in another location: if a name under `variant_dir` is not found but exists under `src_dir`, the file or directory is copied to `variant_dir`. Target files can be built in a different directory than the original sources by simply refering to the sources (and targets) within the variant tree.

`VariantDir` can be called multiple times with the same `src_dir` to set up multiple builds with different options (`variants`). The `src_dir` location must be in or underneath the SConstruct file's directory, and `variant_dir` may not be underneath `src_dir`.

The default behavior is for `scons` to physically duplicate the source files in the variant tree. Thus, a build performed in the variant tree is guaranteed to be identical to a build performed in the source tree even if intermediate source files are generated during the build, or preprocessors or other scanners search for included files relative to the source file, or 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 the `duplicate` argument to `0` (zero). 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:

```
# run src/SConscript in two variant directories
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.

Examples:

```
# use names in the build directory, not the source directory
VariantDir('build', 'src', duplicate=0)
Program('build/prog', 'build/source.c')


# this builds both the source and docs in a separate subtree
VariantDir('build', '.', duplicate=0)
SConscript(dirs=['build/src','build/doc'])


# same as previous example, but only uses SConscript
SConscript(dirs='src', variant_dir='build/src', duplicate=0)
SConscript(dirs='doc', variant_dir='build/doc', duplicate=0)
```

`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.

Appendix E. Handling Common Tasks

There is a common set of simple tasks that many build configurations rely on as they become more complex. Most build tools have special purpose constructs for performing these tasks, but since `SConscript` files are Python scripts, you can use more flexible built-in Python services to perform these tasks. This appendix lists a number of these tasks and how to implement them in Python and SCons.

Example E-1. Wildcard globbing to create a list of filenames

```
files = Glob(wildcard)
```

Example E-2. Filename extension substitution

```
import os.path
filename = os.path.splitext(filename)[0]+extension
```

Example E-3. Appending a path prefix to a list of filenames

```
import os.path
filenames = [os.path.join(prefix, x) for x in filenames]
```

Example E-4. Substituting a path prefix with another one

```
if filename.find(old_prefix) == 0:
    filename = filename.replace(old_prefix, new_prefix)
```

Example E-5. Filtering a filename list to exclude/retain only a specific set of extensions

```
import os.path
filenames = [x for x in filenames if os.path.splitext(x)[1] in extensions]
```

Example E-6. The "backtick function": run a shell command and capture the output

```
import os
output = os.popen(command).read()
```

Example E-7. Generating source code: how code can be generated and used by SCons

The Copy builders here could be any arbitrary shell or python function that produces one or more files. This example shows how to create those files and use them in SCons.

```
#### SConstruct
env = Environment()
env.Append(CPPPATH = "#")

## Header example
```

```
env.Append(BUILDERS =
   {'Copy1' : Builder(action = 'cat < $SOURCE > $TARGET',
                      suffix='.h', src_suffix='.bar')})
env.Copy1('test.bar') # produces test.h from test.bar.
env.Program('app','main.cpp') # indirectly depends on test.bar

## Source file example
env.Append(BUILDERS =
  {'Copy2' : Builder(action = 'cat < $SOURCE > $TARGET',
                     suffix='.cpp', src_suffix='.bar2')})
foo = env.Copy2('foo.bar2') # produces foo.cpp from foo.bar2.
env.Program('app2',['main2.cpp'] + foo) # compiles main2.cpp and foo.cpp into app2.
```

Where main.cpp looks like this:

```
#include "test.h"
```

produces this:

```
% scons -Q
cc -o app main.cpp
cat < foo.bar2 > foo.cpp
cc -o app2 main2.cpp foo.cpp
cat < test.bar > test.h
```