SCons

February 10, 2017

1 Building and Installing SCons

SCons is written in Python, one must obviously have Python installed on your system to use SCons. 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).

SCons will work with any 2.x version of Python from 2.7 on; 3.0 and later are not yet supported.

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, type:

$ python setup.py install - -prefix = $H

2 Simple Build

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

Enter the following into a file named SConstruct:

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

Program is a builder_method, a Python call that tells SCons that you want to build an executable program. Run the scons command to build the program. Here 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.

2.1 Building Object Files

The Object builder method tells SCons to build an object file from the specified source file:

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

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

2.2 Simple Java Builds

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')

2.3 Cleaning Up After a Build

Use the scons -c or scons –clean option to remove the appropriate built files.

2.4 The SConstruct File

The SConstruct file is the input file that SCons reads to control the build.

There is, however, an important difference between an SConstruct file and a Makefile: the SConstruct file is actually a Python script.

Everything between a '#' and the end of the line will be ignored :

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 (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). 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.

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.

All of the configuration files (generically referred to as SConscript files) are read and executed first, and only then are the target files built. These messages help to distinguish between errors that occur while the configuration files are read, and errors that occur while targets are being built.

-Q option :

3 Less Simple Things to Do With Builds

3.1 Specifying the Name of the Target (Output) File

If you want to build a program with a different name than the base of the source file name, 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".)

3.2 Compiling Multiple Source Files

put the source files in a Python list (enclosed in square brackets)

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

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 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”.)

Program(‘program’, [‘prog.c’, ‘file1.c’, ‘file2.c’])

3.3 Making a list of files with Glob

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.

Program(‘program’, Glob(‘*.c’))

3.4 Specifying Single Files Vs. Lists of Files

Program(‘hello’, ‘hello.c’)

Program(‘hello’, [‘hello.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).

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.

Program(‘program’, Split(‘main.c file1.c file2.c’))

(Unlike the split() member function of strings in Python, 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.)

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)

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.

src_files = Split(" " "main.c

file1.c

file2.c" " ")

Program('program', src_files)

The Python "triple-quote" syntax allows a string to contain multiple lines. The three quotes can be either single or double quotes.

3.6 Keyword Arguments

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)

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')

3.7 Compiling Multiple Programs

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

Program('foo.c')

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

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.

3.8 Sharing Source Files Between Multiple Programs

To share source files between multiple programs is 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.

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. Create 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)
```

4 Building and Linking with Libraries

4.1 Building Libraries

Build own libraries by specifyingLibrary

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

SCons uses the appropriate library prefix and suffix for your system.

```
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
```

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.

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 they are object files. 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.

The Library function builds a traditional static library.

StaticLibrary function is to build static libraries explicitly

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

SharedLibrary function is to to build a shared library (on POSIX systems) or a DLL file (on Windows systems).

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

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

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 = '.')

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. SCons has taken care of constructing the correct command lines to link with the specified library on each system.

If you only have a single library to link with, you can specify the library name in single string, instead of a Python list,

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

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

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,

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

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.

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. Assign the lists of targets returned by the calls to the Object builder to variables, 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.

5.2 Explicitly Creating File and Directory Nodes

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.

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 need to refer to an entry in a file system without knowing in advance whether it's a file or a directory. Entry function 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

Use Node to print the file name that the node represents.

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.

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]

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

builtin Python str function

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

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!"

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 Nodes and/or strings, and returns the list of paths. If passed a single Node, the result is the same as calling str(node). 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"])

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.

6 Dependencies

```
% scons -Q

cc -o hello.o -c hello.c cc -o hello hello.o

% scons -Q

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

Name 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.
```

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 modifi- cation 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 avoids unnecessary rebuilds when, for example, someone rewrites the contents of a file without making a change. specify this default behavior (MD5 signatures) explicitly using the Decider function as follows:

Program('hello.c')

Decider('MD5')

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. 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. 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

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.

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.

Object('hello.c')

Decider('timestamp-newer')

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. 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.

Object('hello.c')

Decider('timestamp-match')

SCons will rebuild a target whenever a source file's modification time has changed.

6.1.3 Deciding If a File Has Changed Using Both MD Sig- natures and Time Stamps

SCons provides a way to use MD5 checksums of file contents but to read those contents only when the file's timestamp has changed. Call the Decider function with 'MD5-

timestamp’ argument.

Program(‘hello.c’)

Decider(‘MD5-timestamp’)

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. 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

6.1.5 Mixing Different Ways of Deciding If a File Has Changed

To configure different decision-making for different targets. Use the env.Decider method to affect only the configuration decisions for targets built with a specific construction environment.

env1 = Environment(CPPPATH = [‘.’])

env2 = env1.Clone()

env2.Decider(‘timestamp-match’)

env1.Program(‘prog-MD5’, ‘program1.c’)

env2.Program(‘prog-timestamp’, ‘program2.c’)

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.

6.2.1 The SourceSignatures Function

6.2.2 The TargetSignatures Function

6.3 Implicit Dependencies: The $CPPPATH Construction Variable

Suppose that "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);
```

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, the SConstruct file needs to be modified:

```
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).

```
% 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.

% [CHANGE THE CONTENTS OF hello.h]

% scons -Q hello

cc -o hello.o -c -I. hello.c

cc -o hello hello.o
```

SCons added the -I. argument from the $CPPPATH variable so that the compilation would find the hello.h file in the local directory. 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'])
```

6.4 Caching Implicit Dependencies

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.

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. ––implicit-cache instructs SCons to not rebuild "correctly" in the following cases:

1. 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.

2. 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. Update them by running SCons with the ––implicit-deps-changed option scons -Q −−implicit-deps-changed hello

6.4.2 The −−implicit-deps-unchanged Option

When caching dependencies, SCons notices when a file has been modified and re-scans the file for any updated implicit dependency information. 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. Use the −−implicit-deps-unchanged option

scons -Q −−implicit-deps-unchanged hello

6.5 Explicit Dependencies: the Depends Function

Sometimes a file depends on another file that is not detected by an SCons scanner. Specific explicitly that one file depends on another file, and must be rebuilt whenever that file changes. Use Depends method.

hello = Program('hello.c')

Depends(hello, 'other_file')

The dependency (the second argument to Depends) may also be a list of Node objects.

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).

6.6 Dependencies From External Files: the ParseDepends Function

Sometimes these scanners fail to extract certain implicit dependencies due to limitations of the scanner implementation.

ParseDepends can parse the contents of the compiler output in the style of Make, and explicitly establish all of the listed dependencies.

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

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

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

6.8 Order-Only Dependencies: the Requires Function

6.9 The AlwaysBuild Function

7 Environments

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 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.

Construction Environment : 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.

Execution Environment : 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.

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). 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.

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

Use the os.environ dictionary in your SConscript files to initialize construction environments with values from the user's external environment.

7.2 Construction Environments

SCons accommodates the 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.)

7.2.1 Creating a Construction Environment: the Environment Function

A construction environment is created by the Environment method.

```
env = Environment()
```

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.

```
env = Environment(CC = 'gcc',
CCFLAGS = '-O2')
env.Program('foo.c')
```

The explicit initializations of $CC and $CCFLAGS override the default values in the newly-created construction environment.

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

7.2.2 Fetching Values From a Construction Environment

Fetch individual construction variables using the normal syntax for accessing individual named items in a Python dictionary

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

A construction environment, however, is 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])

print the specified dictionary items for us on POSIX systems

% scons -Q

key = OBJSUFFIX, value = .o

key = LIBSUFFIX, value = .a

key = PROGSUFFIX, value =

scons: ‘.’ is up to date.

print the values of all of the construction variables in a construction environment

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

Use the subst method on a string containing $ expansions of construction variable names

to get information from a construction environment

env = Environment()

print “CC is:”, env.subst(‘$CC’)

env = Environment(CCFLAGS = ‘-DFOO’)

print “CCCOM is:”, env.subst(’$CCCOM’)

Will recursively expand all of the construction variables prefixed with $ (dollar signs). 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.

8 Automatically Putting Command-line Options into their Construction Variables

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.

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’]
```

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’]
```

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’]
```

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.

ParseFlags method that takes a set of typical command-line options and distributes 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’)
```

If the options are limited to generic types like those above, they will be correctly translated for other platform types. 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’)
```

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’)
```

If a string begins with a “!” (an exclamation mark, often called a bang), the string is passed to the shell for execution.

```
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’)
```

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. Various utilities 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 to provide the set of strings for his particular package. ParseConfig method 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.

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']

9 Controlling Build Output

10 Controlling a Build From the Command Line

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. 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)

12 Platform-Independent File System Manipulation

13 Controlling Removal of Targets

14 Hierarchical Builds

15 Separating Source and Build Directories

16 Variant Builds

17 Internationalization and localization with gettext

18 Writing Your Own Builders

19 Not Writing a Builder: the Command Builder

20 Pseudo-Builders: the AddMethod function

21 Writing Scanners

22 Building From Code Repositories

23 Multi-Platform Configuration (Autoconf Functionality)

23.1 Configure Contexts