1

Getting Situated

The first order of business in learning to program in Lua is to acquire and install the necessary tools. For your initial steps, all you need is the Lua *interpreter*, a small program that enables you to type Lua commands and have them executed immediately. As you advance through this book, you will need additional tools such as a text editor and the Lua compiler.

If you want to write web applications, you'll need access to a web server such as Kepler (a versatile Lua-based web server) or Apache (an industry-wide standard). These and other web servers are freely available on the Internet.

If you want to extend Lua with low-level libraries or to embed Lua into your application, you'll need a *software development kit* (often referred to as SDK) with a compiler that is compatible with Lua's *application program interface* (referred to as API).

Lua is written in the C programming language, and a C compiler turns a program written in this language into something that can run on your computer. Most C compilers work fine, as do Delphi and the cross-platform Free Pascal Compiler.

This chapter is unlike the others in this book, because it has little to do with Lua and much to do with setting up programs on your system. Although Lua doesn't have a setup program that handles installation details, you'll find the steps are straightforward. In addition to guiding you through the process, this chapter briefly explores programming editors and revision control systems — tools that can enhance your productivity as you become proficient in Lua.

Choosing How to Install Lua

Lua can be installed on a wide variety of platforms and, after it is installed, it will function similarly on all of them. Unlike most of the material that follows in this book, this chapter necessarily delves into some platform-specific details. Basically, there are two categories that are covered here: Windows desktop systems (including Windows 95 and up) and Unix-type systems, including GNU/Linux, Mac OS X, AIX, BSD, and Solaris. (The many other operating systems and hardware platforms capable of running Lua are outside the scope of this book.)

Lua Versions

In the instructions that follow, you'll see references to version 5.1.1. A later version of Lua may be available as you read this. As Lua evolves, some improvements are made that require changes to existing scripts. Your decision is either to install a version of Lua later than 5.1.1 and encounter possible instances where you have to modify the scripts and libraries in this book, or to install version 5.1.1 and forgo any improvements that may have been made to Lua. The Lua manual includes a section at the end named "Incompatibilities with Previous Versions" that can help you decide. If you install a later version, you'll need to make corresponding changes to the commands and directory names used in this chapter.

There are two excellent open-source packages for Windows that blur the Windows-Unix distinction somewhat. One of them, the Cygwin system, provides many GNU and Linux tools (including various shells and development tools) for Windows platforms. It is available at `www.cygwin.com`. Applications that you build with this environment will run only on systems that have Cygwin installed. If you want to install Lua in this environment, follow the directions for building Lua on Unix-type systems.

The other package is the MinGW system, which enables you to use standard Unix-like tools to build applications that run on all 32-bit desktop versions of Windows without any extra support libraries. This first-rate system is available at `www.mingw.org`.

As you read this chapter, section headers will indicate whether a Unix-like system or Windows is being discussed. You can skip the sections that don't apply to your platform.

The Lua interpreter, typically named `lua.exe` in Windows and `lua` in Unix and friends, is a small command-line program that executes your scripts either interactively or noninteractively. You'll become familiar with the Lua interpreter and both of these modes in the first chapters of this book.

A note about these names: In this book, Lua refers to the Lua programming language or implementation, and `lua` refers to the Lua interpreter.

To install Lua, you can download a package that has been compiled for your particular operating system platform, or download the source code and compile it yourself. There are the advantages and disadvantages to each approach, as the following sections discuss.

Building Lua Yourself

Compiling Lua is straightforward. Lua, including the language processor and its core libraries, is written in plain vanilla C so that it can be built on a wide variety of platforms with any ANSI-compliant C compiler. The advantage is that the resulting libraries and interpreter program are compatible with the system on which it's built. This is one of the principal advantages of open-source software in general. As long as the target platform supports the tools and libraries needed to compile the source code — in Lua's case, this is a standard C compiler — the resulting binary program is compatible with the platform.

The disadvantage of compiling Lua is that the system you intend to build it on must have a complete C development environment. That is generally not a problem on Unix and Unix-like operating systems where such development tools are part of a long-standing tradition. However, on platforms such as Windows, a C compiler and its related tools and files are not installed by default. Such a development package can require a surprisingly large amount of disk space and present a bewildering number of options. One refreshingly small exception to this rule is the Tiny C Compiler (TCC), which runs on Linux and Windows on the 80 × 86 platform. Building Lua with TCC is described later in this chapter.

Selecting Prebuilt Lua

Besides being able to skip the compilation step, an advantage of selecting a prebuilt version of Lua is that it will be compatible with a number of libraries that conform to its dependency conventions. These conventions involve issues such as which runtime libraries are used and whether those libraries are safe to use with multiple threads of execution.

If you are a Windows user and don't have a C development environment set up on your system, installing the appropriate binary Lua package may be your best option. You'll see how to do that in the section on installing binary packages later in this chapter.

Finding Your System's Shell

Most computer systems have a *shell*, also known as a *command-line interface*. This is a program you can use to type commands to the computer. These commands can tell the computer to copy, move, delete, and otherwise manipulate files, and to start programs, including (of course) Lua programs and Lua itself. In several places in this book, you'll need to access your system's shell; in particular, everything in this chapter that you need to type, you need to type into the shell. You can perform some of the operations, such as creating directories and moving files, using visual tools (such as Explorer on the Windows platform). Only the shell commands are presented here, but feel free to use whatever tools you are most comfortable with to accomplish the task at hand.

Windows Shells

To access your shell on Windows XP or Windows 2000, select Start➪Run, type `cmd`, and press Enter. On Windows Me, Windows 98, or Windows 95, select Start➪Run, type `command`, and press Enter (Return on some keyboards — just substitute "Return" for "Enter" whenever it's mentioned in this book).

Shells on Unix and Unix-Like systems

On Mac OS X, open your Applications folder (on your startup disk). Inside it, open the Utilities folder; inside that, open Terminal. On other systems with a graphical user interface (GUI), look in the menu that you start programs from for a program with a name like `xterm`, `Konsole`, or `Terminal`. On systems without a graphical user interface, you are already at the shell, as you are if you use a program such as `ssh`, `telnet`, or `PuTTY` to connect to a remote Unix(-like) server.

Shell Features

Shells vary greatly in appearance and functionality, but each of them presents some form of a prompt to let you know that it's waiting for you to issue a command. In general, you type a command following the prompt and press the Enter key to submit the command to the shell. When you are working in a shell, there is always one directory that is considered your current working directory. Most shell prompts contain that directory to make it easier for you to keep your bearings as you move from directory to directory. For example, a typical prompt in Windows shells may look something like the following:

```
C:\Program Files>
```

and in Unix-type shells, something like the following:

```
mozart maryann /usr/local/bin>
```

To exit the shell, type `exit` and press Enter.

The Environment

Each shell also has a pool of variables, known as the *environment*, available to programs. The environment typically holds information about where the system should look for programs and libraries, what the shell prompt should look like, and so forth. You can view this information by issuing the following command at the shell prompt:

```
set
```

Regardless of the platform you use, you will want to modify the shell environment to let Lua know where to find extension modules. Additionally, if you intend to compile Lua or libraries, you'll need to set up environment variables that your SDK will look for.

Environment Variables on Unix-Like Systems

On Unix-like systems, you generally modify the shell environment in one of the shell startup scripts. For example, if you use the `bash` shell, it will process `/etc/bashrc` and, in your home directory, `.bashrc` when it starts. The first file is used for system-wide settings, and the second is used for your own private settings. You'll need root privileges to modify the first. Within these files, you set an environment variable by including a line that looks like the following:

```
export LUA_DIR=/usr/local/lib/lua/5.1
```

When you reference environment variables in shell scripts, you precede the name with `$`, as in `echo $LUA_DIR`.

In this book, the following environment variables are recommended for Unix-like systems:

```
LUA_DIR=/usr/local/lib/lua/5.1
LUA_CPATH=?.so;$LUA_DIR/?.so
LUA_PATH=?.lua;$LUA_DIR/?.lua
```

Restart the shell for changes to take effect.

Now create the actual directory that `LUA_DIR` identifies. Do this, as root, with the following command:

```
mkdir -p /usr/local/lib/lua/5.1
```

Environment Variables on Windows

Depending on which version of Windows you use, you modify the shell environment either through the `autoexec.bat` file (Window 95, 98 and ME) or, for later versions, through a dedicated dialog box that you get to through the System Properties dialog box. If you use `autoexec.bat`, you set environment variables with lines that look like the following:

```
SET LUA_DIR="c:\program files\lua\5.1"
```

If you use the dedicated dialog box, you'll need to choose between system variables and user variables. In this window, you can add a new variable, edit an existing variable, or delete an existing variable. When you add or edit a variable, there are separate input fields for the variable name and its value.

Within a shell script, surround an environment variable name with the `%` character, as in — `echo %LUA_DIR%`.

The Windows Search Path

On a Windows system, whether you compile Lua or acquire a precompiled package, you'll want to put the Lua interpreter, compiler, and dynamic link library in a location that makes them easy to use. From a shell prompt, the system should launch the interpreter when you execute the `lua` command. There are two practical approaches you can take: using aliases or using the search path.

When you're at the shell prompt, Windows enables you to use a simple alias — `lua`, for example — as a replacement for a more complicated command, such as `c:\program files\utility\lua.exe`. It implements aliases like these, in addition to command-line editing and history, using `doskey`. This method may locate the aliased program slightly faster, but you cannot use the alias in a batch script. Consult the output of the following to read more about this utility:

```
doskey /?
```

You can also use the Windows search path mechanism. When a command is invoked that is not internal (such as `dir` or `del`) and is not qualified with path information, Windows examines the search path, looking for a matching executable. To see the current search path from the shell, execute this command:

```
path
```

In the following steps, you work with files and directories so you can use Windows Explorer if you like. Complete these steps to move the Lua executables to a directory that is included in the Windows search path:

1. If your current search path does not include a directory where you store utilities, create one now (the directory `c:\program files\utility` is assumed for this example, but the choice is yours). Note that quotes are necessary when specifying names with spaces:

```
mkdir "c:\program files\utility"
```

2. Add this directory to the Windows search path. On older versions of Windows, use the `autoexec.bat` file in the root directory of the boot drive. (More recent versions of Windows still support this, but they also provide a graphical environment editor that you by opening the System applet from the Control Panel.)

   If the text field containing the path variable is too small to see the entire value, cut and paste the value to your text editor, make the appropriate change, and then cut and paste the modified value back to the path edit field.

3. The new search path applies only to shells that are opened after the change, so exit and restart your shell.

Recommended Settings for Windows

For this book, the following environment variables are recommended on Windows systems:

```
UTIL_DIR=c:\program files\utility
LUA_DIR=c:\program files\lua\5.1
LUA_CPATH=?.dll;%LUA_DIR%\?.dll
LUA_PATH=?.lua;%LUA_DIR%\?.lua
```

The `UTIL_DIR` variable identifies the utility directory you created in the preceding section. Additionally, if you have a software development kit and intend to compile Lua and possibly libraries for Lua, set the following environment variables:

```
SDK_DIR=c:\program files\msc
INCLUDE=%SDK_DIR%\include;%SDK_DIR%\include\usr
LIB=%SDK_DIR%\lib;%SDK_DIR%\lib\usr
```

The `SDK_DIR` depends on where you installed your SDK.

Restart your shell for environment changes to take effect. Then use Windows Explorer or the command shell to create the various directories that these environment variables identify.

Dealing with Tarballs and Zip Files

Whether you install Lua using precompiled packages or compile it from a source code package, you will be dealing with a packaging form colloquially known as a *tarball*. Files of this type have the extension `.tar.gz` or `.tgz`. A tarball bundles a group of files that can be distributed over one or more directories. The contents, owners, permissions, and timestamps of the bundled files are preserved using the `tar` utility, whose name derives from its original purpose of transferring files to and from a tape archive. The amalgamated file is then compressed using the `gzip` utility or, for tarballs with the `.tar.bz2` extension, using the slower and more aggressive `bzip2` compression utility. Although tarballs are part of the Unix tradition, tools for managing them on Windows are freely available. In particular, one versatile open-source utility for Windows that handles any type of package you are likely to encounter is `7z`. Both graphical and shell-oriented versions are available from `www.7-zip.org`. Whichever version you use, make sure the directory in which you install 7-zip is included in your system search path. Extracting the contents of a tarball in Windows is a two-step process. Here's how to do it from the shell. First, unzip the embedded tarball using a command like the following:

```
7z x somefile.tar.gz
```

In a standard package, this creates the file `somefile.tar`. Extract the contents of this tarball with a command like the following:

```
7z x somefile.tar
```

Another packaging format, more common for Windows-based projects, is the zip file, which has a `.zip` extension. The `zip` and `unzip` utilities on Unix-style systems manage files of this type. On Windows, you can extract the contents of a zip file using `7z` with a command like the following:

```
7z x somefile.zip
```

Compiling Lua

In the general sense, compiling an application refers to the process of building an executable program from source components. The executable program comprises processor instructions that a computer can follow. It may also contain certain resources such as embedded icons and copyright notices. Source code is the text that is created in a programming language by a programmer. Strictly speaking, compilation is the intricate step that translates a source-code text file to a machine-code object file. Object files usually contain references to other object files that may or may not be part of the same package. A *linker* generates an executable program by combining all required object files and resolving their references to one another. Sometimes the compilation and linking steps are combined by a wrapper program.

The Lua Source Tarball

The contents of the Lua tarball are organized as follows:

```
lua-5.1.1
  doc
  etc
  src
  test
```

In the first level of this directory, you can read various text documents prepared by the Lua authors. The `README` file explains what Lua is, the terms by which it is available, how to install it, and its origin. The `HISTORY` file tracks the changes to the Lua language, its application programming interface (API), and its implementation. Read the relevant portion of that file when upgrading to a new version of Lua to understand the changes that have been made. The `INSTALL` file has information about building Lua.

You can read these files using a text viewer or editor. If you are using a Unix-type system, the `less` command is convenient for scanning text files. The `lynx` character-mode web browser is great for exploring the distribution: use the arrow keys to drill into and out of directories and the contents of text and HTML files. Press **Q** to exit `less` or `lynx` when you are ready to return to the shell itself. If you are a Windows user, note that the files have Unix-style line endings that will not be properly handled by Windows Notepad, so you should use a programmer's editor or a web browser to read them.

With your web browser, you can explore the hyperlinked Lua manual in the `doc` subdirectory. The `src` subdirectory contains all of the source code for the Lua interpreter, compiler, and core libraries. The `etc` subdirectory contains miscellaneous files such as the Lua icon and the source code for an especially small interpreter with reduced functionality. The `test` subdirectory contains a number of Lua scripts that provide an excellent survey of Lua's capabilities. Some of these scripts push at the far reaches of Lua's domain, so don't be dismayed if some appear rather dense at first.

Compiling Lua on Linux and Other Unix-Like Systems

One of the first things you should do at your shell is to check whether you have a functioning C compiler. To do so, execute the following command:

```
cc -v
```

If version and configuration information is displayed, then it's likely that you've got everything you need to successfully build Lua. If you receive a message indicating that the command `cc` could not be found, try executing `gcc -v` instead. If neither of these commands work, you need to find out if either the C development tools have not been installed or some configuration setting is simply keeping them from being available to you.

Where you build Lua depends on your purposes. If you intend for Lua to be used by other users on your system, and you have the appropriate privileges, you'll want to select a standard location for source code such as `/usr/local/src`. Otherwise, your home directory is a logical choice. In the shell, change your default working directory with the following command:

```
cd /usr/local/src
```

Alternatively, simply use `cd` to go to your home directory.

Assuming you are connected to the Internet, acquire the source package as follows:

```
wget http://www.lua.org/ftp/lua-5.1.1.tar.gz
```

The program `wget` is a standard tool for retrieving Internet resources such as web pages and other files. If your system doesn't have it, you can try `curl` or an interactive web browser. The file you download will be in the form of a tarball that has the extension `.tar.gz`. Extract the contents as follows:

```
tar xzvf lua-5.1.1.tar.gz
```

The `tar` utility will recreate the same directory structure that the Lua authors used to package the source materials. Drop into the newly created directory by executing the following command:

```
cd lua-5.1.1
```

You'll use the make utility to control the build process. This utility reads a script, typically named makefile, that describes the dependency relationships between the various source files and intermediate and final targets. For example, one of the lines in Lua's makefile is as follows:

```
lua.o: lua.c lua.h luaconf.h lauxlib.h lualib.h
```

Here, lua.o is the object file that corresponds to lua.c, the main source file of the Lua interpreter. The other files that have the extension .h are header files that contain definitions and prototypes. This line is interpreted as: If lua.o is missing or if its timestamp is older than any of the timestamps of lua.c, lua.h, luaconf.h, lauxlib.h, or lualib.h, then invoke the appropriate rule to generate lua.o. The make utility is indispensable for keeping track of this kind of dependency and for automating the build process.

If you type make by itself, as shown in the following line, you get a list of the various platforms on which you can build Lua:

```
make
```

The output looks like this:

```
  Please do
    make PLATFORM
  where PLATFORM is one of these:
    aix ansi bsd generic linux macosx mingw posix solaris
```

Select the platform that you're on. For example, if you are compiling on Linux, execute the following command:

```
make linux
```

You'll see the commands displayed on the console as they are executed.

If you are familiar with building other open-source packages, you'll notice that Lua's approach is a little different. There is no configuration stage to identify the characteristics of the system and create a tailor-made makefile. Instead, the platform argument you provide is all the information make needs to select the correct commands and parameters.

Because Lua has such standard requirements, it is unlikely that you will encounter any problems building it. If errors do occur at this stage, they are likely to be related to an incomplete installation of the C development tools or incorrectly configured search paths for header files or libraries. If that happens, read the documentation for your operating system distribution to install and configure the development tools properly.

With the default settings used here, make will generate the Lua interpreter (lua) and the Lua byte-code compiler (luac). These will be created in the src subdirectory. No shared libraries will be created — all of the Lua internals required by each of these executables will be statically linked. For example, components such as the parser will be embedded into both lua and luac, but the byte-code interpreter will be embedded only into lua.

After make completes, your shell prompt is displayed. If no error messages were encountered during the building process, you can test the Lua interpreter by executing the following command:

```
make test
```

That should result in output that looks like

```
Hello world, from Lua 5.1!
```

In the unlikely event that you don't get this response, the probable culprit is the unavailability of one or more runtime libraries. Although the Lua internals are statically linked into lua, other libraries are expected to be available for dynamic linking when lua is actually run. These runtime dependencies include the math library, the general purpose C library, and, depending on your platform, libraries that support screen formatting and text input editing and recall. List the shared library dependencies with the following ldd command:

```
ldd src/lua
```

The result is a series of lines such as this:

```
libm.so.6 => /lib/libm.so.6 (0x40024000
libdl.so.2 => /lib/libdl.so.2 (0x40047000)
```

If, in the output, you see one or more "not found" lines such as the following, you'll know that the referenced library is not present on your system or that the dynamic loader is unable to find it:

```
libncurses.so.5 => not found
```

In this case, consult your operating system documentation to guide you through the process of installing the missing library. Alternatively, you can rebuild lua with fewer dependencies with the command make clean generic.

If you have root privileges on your system and would like lua and luac to be available for other users, you should become root at this point. Do this with the following command:

```
su -
```

Alternatively, you can use the sudo command to elevate your privilege level for particular commands. In this case, prefix the command requiring root authority with sudo. In general, using sudo requires some configuration using the visudo command.

The hyphen tells the command that you want the root's environment, including its search paths, to be loaded. Doing so will likely change your default working directory, so you may need to return to the lua directory by using the cd command. For example:

```
cd /usr/local/src/lua-5.1.1
```

You're now ready to install the Lua executables, static library, header files, and manual pages. To do so, execute the following command:

```
make install
```

Then execute the following command to return to your nonroot session:

```
exit
```

The following command should present the `lua` manual page:

```
man lua
```

You can scroll up and down through the document by using the standard navigation keys. Press **Q** to return to the shell itself.

Now enter following command:

```
lua -v
```

This should present you with a version statement like the following:

```
Lua 5.1.1 Copyright (C) 1994-2006 Lua.org, PUC-Rio
```

Any problem at this point indicates an issue that you can clarify with the following `which` command:

```
which lua
```

The response should be a line like this:

```
/usr/local/bin/lua
```

A response like the following means that `lua` was installed into a directory that is not in your *search path*, which is an ordered list of directories that the operating system examines to resolve external commands:

```
which: no lua in (/usr/local/bin:/usr/bin)
```

You can remedy this by editing the `PATH` variable in your shell startup script to include the directory in which `lua` and `luac` were installed. You need to exit and restart the shell for these changes to take effect.

If you don't have root privileges or simply want to test the installation locally, you can execute the following command:

```
make local
```

This creates several additional directories such as bin and man beneath the current directory.

Alternatively, you can specify some other location for the installation. For example, if you want to install in /tmp/lua-5.1.1, execute the following command:

```
make INSTALL_TOP=/tmp/lua-5.1.1 install
```

In these last cases, you'll need to specify a full or relative path to lua, luac, and the man pages because the default search path won't include them. For example, to read the man page installed with the last example, execute the following:

```
man /tmp/lua-5.1.1/man/man1/lua.1
```

Then, to test `lua`, execute the following command:

```
/tmp/lua-5.1.1/bin/lua -v
```

This section led you through a basic build of Lua on Unix-like systems. Many options exist for configuring Lua in different ways. For example, you can configure Lua to use a primitive data type other than `double` for numbers. The curious can examine `src/luaconf.h` to see the configurable options. Your best approach at this point is to leave them at their default settings. This book assumes that you are using Lua with its default options.

Compiling Lua on Windows

Although Lua compiles cleanly on Windows, the makefile that comes with the Lua source package is oriented toward Unix-style systems. Unless you are using a Unix emulation layer such as Cygwin, you'll need to craft your own approach to the task of creating the Lua interpreter and byte-code compiler.

A lot of C/C++ compilers exist for Windows, and many of them are available for free. The tools in these SDKs generally have their own particular set of command line switches, configuration conventions, and system dependencies, making it impractical to cover more than one Windows SDK in this book. The instructions in this book assume you are using the Microsoft Visual C++ 6.0 SDK. Later versions of this kit have been released, but Visual C++ 6.0 is still widely used in the industry because of its support for a broad range of Windows versions. You can configure it to generate programs that use `MSVCRT.DLL`, a system library that is available on all 32-bit desktop versions of Windows. An effort has been made in this book to provide instructions that will generate applications and libraries that are compatible with the Lua binaries for Windows available on LuaForge. These binaries are easy to deploy, because they depend only on libraries that come with Windows.

If you have a later version of Visual C++ or are using another SDK (such as a product from Pelles C or Borland) you will need to consult the appropriate documentation to find how to build and deploy applications and libraries. Recent versions of the Microsoft C++ SDK require manifest files to be deployed along with applications and libraries.

The directions that follow create the Lua interpreter and compiler. Three approaches are shown. The first works well with Visual C++ 6.0. If you are some other large-scale C development system, follow these directions with the understanding that you may need to make toolkit-specific changes. The second approach is suitable for users of TCC, and the third applies to the MinGW SDK.

Building Lua with Microsoft Visual C++

First, make sure that the SDK's `bin` directory has been included in your system's search path. Under the SDK's `include` directory, make a subdirectory named `usr`. Similarly, under the SDK's `lib` directory, make a subdirectory named `usr`. Additionally, as covered in the environment section, you should set the `LIB` environment variable to the SDK's `lib` and `lib\usr` directories, and the `INCLUDE` environment variable to the SDK's include and `include\usr` directories. Your SDK may include a batch file named `vcvars32.bat` that can help with this. If you modify these environment variables, exit and restart the shell to have your changes take effect.

Create a directory in which to build Lua. The following lines assume that this will be `c:\dev`:

```
c:
cd \
mkdir dev
cd dev
```

Download `lua-5.1.1.tar.gz` and place it in this directory. Extract the contents. Here are the commands you would use with 7-zip:

```
7z x lua-5.1.1.tar.gz
7z x lua-5.1.1.tar
```

This creates a subdirectory named `lua-5.1.1`. Delete `lua-5.1.1.tar` at this point, like this:

```
del lua-5.1.1.tar
```

Drop into the `src` subdirectory like this:

```
cd lua-5.1.1\src
```

Create a new text file and add the following lines:

```
cl /MD /O2 /W3 /c /DLUA_BUILD_AS_DLL *.c
del *.o
ren lua.obj lua.o
ren luac.obj luac.o
ren print.obj print.o
link /DLL /IMPLIB:lua5.1.lib /OUT:lua5.1.dll *.obj
link /OUT:lua.exe lua.o lua5.1.lib
lib /out:lua5.1-static.lib *.obj
link /OUT:luac.exe luac.o print.o lua5.1-static.lib
```

Save the file as `build.bat` in the current directory (`c:\dev\lua-5.1.1\src`).

While you're still in the `src` subdirectory, run the newly created batch script from the shell:

```
build
```

This compiles each of the source files into a corresponding object file. Prior to linking these object files into a dynamic link library, three object files are renamed to keep them from being included in the library. These are the interpreter, the compiler, and a support file for the compiler. Finally, the interpreter and compiler executables are created. The Lua interpreter is quite small, because its main functionality comes from the dynamic link library. To test the Lua interpreter, execute the following command:

```
.\lua ..\test\hello.lua
```

This should result in the following output:

```
Hello world, from Lua 5.1!
```

Copy the import library and header files associated with the dynamic-link library to standard development directories where your compiler and linker can find them. The approach taken here is to place them in the `usr` subdirectory beneath the SDK's `lib` and `include` directories. These subdirectories can then hold third-party files where they won't be confused with toolkit files.

To install Lua, create a file with the following contents:

```
xcopy lua5.1.lib "%SDK_DIR%\lib\usr\*.*" /y
xcopy lua5.1-static.lib "%SDK_DIR%\lib\usr\*.*" /y
xcopy lua.exe "%UTIL_DIR%\*.*" /y
xcopy luac.exe "%UTIL_DIR%\*.*" /y
xcopy lua5.1.dll "%UTIL_DIR%\*.*" /y
xcopy lua.h "%SDK_DIR%\include\usr\*.*" /y
xcopy luaconf.h "%SDK_DIR%\include\usr\*.*" /y
xcopy lualib.h "%SDK_DIR%\include\usr\*.*" /y
xcopy lauxlib.h "%SDK_DIR%\include\usr\*.*" /y
```

Save this file as `install.bat` in the `src` directory. Copy the files by executing this batch script:

```
install
```

Building Lua with the Tiny C Compiler

The Tiny C Compiler (TCC) is a freely available C development system that you can use to build Lua on both Linux and Windows. It is discussed here because it is an excellent way for Windows users who don't have a C toolkit to familiarize themselves with developing programs in C. The TCC web site (`http://fabrice.bellard.free.fr/tcc`) contains a link to the Window binary distribution in the form of a zip file that includes everything you need to compile Lua.

TCC is perfectly suitable for building Lua itself, but you may want to consider a more full-featured SDK if you intend to build extension libraries for Lua.

Download the zip file and place it in the directory above the point where you want to install TCC. Assuming you are using the 7-zip utility and the version of the file is 0.9.23, extract the contents of the file as follows:

```
7z x tcc-0.9.23.zip
```

This creates the following subdirectory structure:

```
tcc-0.9.23
  doc
  examples
  include
    sys
    winapi
  lib
  tcc
```

With this particular version of TCC, you need to make two small adjustments before proceeding. In the `include/winapi` directory, open the `winnt.h` file with your text editor. On lines 1814 and 2288, change the occurrences of `Value` to `_Value`.

Change your working directory to Lua's source directory. For example, if you extracted the Lua tarball in `c:\dev`, use the following shell commands:

```
c:
cd \dev\lua-5.1.1\src
```

TCC requires a change to the file `ldo.c`. Open this file with your text editor and find line 487, which reads as follows:

```
static void f_parser (lua_State *L, void *ud) {
```

Just beneath it, add the following line:

```
typedef Proto* (* load_func) (lua_State*, ZIO*, Mbuffer*, const char*);
```

Several lines lower, find the line that includes

```
((c == LUA_SIGNATURE[0]) ? luaU_undump : luaY_parser)
```

and replace

```
luaU_undump
```

with

```
(load_func) luaU_undump
```

and

```
luaY_parser
```

with

```
(load_func) luaY_parser.
```

Use your text editor to prepare the following batch file. Adjust the first line if necessary to specify the directory in which you installed TCC. The second line begins with `SET TCCCMD` and ends with `-lkernel32` — make sure it is all on one line. (The ↩ symbol indicates that the code line is too long to print on one line in the book; the code that follows is a continuation of the first line. In other words, ↩ tells you to keep typing on the same line.)

```
SET TCCDIR=c:\program files\tcc-0.9.23
SET TCCCMD="%TCCDIR%\tcc\tcc" -D_WIN32 -I"%TCCDIR%\include" ↩
  -I"%TCCDIR%\include\winapi" -L"%TCCDIR%\lib" -lkernel32
ren luac.c luac.c0
ren print.c print.c0
%TCCCMD% -o lua.exe *.c
ren lua.c lua.c0
ren luac.c0 luac.c
ren print.c0 print.c
%TCCCMD% -o luac.exe *.c
ren lua.c0 lua.c
SET TCCDIR=
SET TCCCMD=
SET TCCIMP=
```

Save this file as `build.bat` in the current directory. Build Lua by running this batch script:

```
.\build
```

Although TCC can generate dynamic link libraries, the code that is shown here builds statically linked versions of `lua.exe` and `luac.exe`. Copy these to your utility directory as follows:

```
xcopy lua.exe "%UTIL_DIR%\*.*" /y
xcopy luac.exe "%UTIL_DIR%\*.*" /y
```

Building Lua with MinGW

The MinGW package provides you with all the command-line tools you need to develop Windows applications. An optional related package, MSYS, includes a Unix-like shell and, among other tools, `awk`, `bzip2`, `find`, `grep`, `sed`, `tar`, `vi`, and `which`. The instructions that follow cover only the use of MinGW, but MSYS is definitely worth investigating if you want the power and flexibility of working in a Unix-like development environment. The MinGW website has an excellent FAQ (Frequently Asked Questions) page as well as a comprehensive wiki to help you use the MinGW and MSYS systems to their fullest.

The following instructions show you how to install the MinGW system. You need about 50MB of space available on your disk. Obtain the following files (or more recent versions if they are available) from the current section of download page of the MinGW site, `www.mingw.org`:

```
binutils-2.15.91-20040904-1.tar.gz
w32api-3.6.tar.gz
mingw-utils-0.3.tar.gz
gcc-core-3.4.2-20040916-1.tar.gz
mingw-runtime-3.9.tar.gz
```

Create a directory for the MinGW files as follows:

These instructions assume you will install MinGW in `c:\mingw`*, but you can choose another location if you like. If you do, make the appropriate changes in the following lines.*

```
c:
mkdir \mingw
cd \mingw
```

Extract the contents of the tarballs as follows, changing all occurrences of `\path\to` to the directory where you placed the downloaded files:

The 7-zip tool is used in this example; remember that its directory must be in the Windows search path. If you have downloaded more recent versions of any of these files, be sure to make the appropriate file-name changes. .

```
7z x \path\to\binutils-2.15.91-20040904-1.tar.gz
7z x \path\to\w32api-3.6.tar.gz
7z x \path\to\mingw-utils-0.3.tar.gz
7z x \path\to\gcc-core-3.4.2-20040916-1.tar.gz
7z x \path\to\mingw-runtime-3.9.tar.gz
7z x -y *.tar
del *.tar
```

Place `c:\mingw\bin` in your Windows search path. (See "The Windows Search Path" earlier in this chapter for more details on setting this.)

Create a directory in which to build Lua. The following lines assume that this will be `c:\dev`:

```
c:
mkdir \dev
cd \dev
```

Extract the contents. Here's how to do this if you are using the 7-zip tool:

```
7z x \path\to\lua-5.1.1.tar.gz
7z x lua-5.1.1.tar
del lua-5.1.1.tar
```

Drop into the `src` subdirectory as follows:

```
cd lua-5.1.1\src
```

With your Windows text editor, create a new file with the following contents:

```
gcc -O2 -Wall -c *.c
ren lua.o lua.obj
ren luac.o luac.obj
ren print.o print.obj
gcc -shared -Wl,--export-all-symbols -o lua5.1.dll *.o
strip --strip-unneeded lua5.1.dll
gcc -o lua.exe -s lua.obj lua5.1.dll -lm
gcc -o luac.exe -s -static luac.obj print.obj *.o -lm
```

Save the file as `build.bat` in the `c:\dev\lua-5.1.1\src` directory. While you're still in Lua's src directory, invoke this batch file to build Lua:

```
.\build
```

The batch script generates three files: `lua.exe`, `luac.exe`, and `lua5.1.dll`. You can verify that they use only standard Windows libraries as follows:

If you have installed the MSYS tools, replace `find` *with* `grep`.

```
objdump -x lua.exe | find "DLL Name"
```

This will print the following import references:

```
DLL Name: KERNEL32.dll
DLL Name: msvcrt.dll
DLL Name: lua5.1.dll
```

You can repeat this for `luac.exe` and `lua5.1.dll`. Notice that `luac.exe` does not depend on `lua5.1.dll`.

Install the three files in your utility directory as follows:

```
xcopy lua.exe "%UTIL_DIR%\*.*" /y
xcopy luac.exe "%UTIL_DIR%\*.*" /y
xcopy lua5.1.dll "%UTIL_DIR%\*.*" /y
```

Binary Packages

Unlike source code that can conform to a wide spectrum of environments, binary applications (that is, those that have already been compiled) function only in a particular niche. Binary packages need to distinguish a number of factors, including the following:

- ❑ Operating system (such as AIX, Solaris, Linux, or Windows)
- ❑ Hardware architecture (such as 32-bit versus 64-bit or Intel versus PowerPC)
- ❑ Required C runtime library (such as the various versions of the Microsoft Visual C runtime library)

These are mostly issues that you don't have to be concerned with when compiling Lua from source code, but you do need to pay attention to when the packages are precompiled. Despite the plethora of different platforms, there is a good chance that you can find a binary package for your particular environment.

Selecting a Prebuilt Binary Package

To acquire a Lua package precompiled for your platform, visit LuaForge (`http://luaforge.net/`), a web site devoted to open-source projects created and maintained by members of the growing Lua community.

One of the most popular packages maintained at LuaForge is LuaBinaries, a set of ready-made Lua packages (`http://luaforge.net/projects/luabinaries/`). In the file download section of the LuaBinaries site is a list of files that includes entries like these:

```
        lua5_1_Win32_bin.tar.gz      82060  1,376  Intel x86 .gz
        lua5_1_Linux26_bin.tar.gz   127132     80  Intel x86 .gz
```

The names of these files include abbreviated information about their contents. For more detailed information about each of the packages, read the packaging file (`apackaging_lua5.1.html`) located in the same directory. It will help you select the appropriate package for your platform.

Installing a Prebuilt Binary Package on a Unix-Type System

After you have selected the appropriate tarball from the LuaBinaries site, download it and place it in the `/tmp` directory. The following commands assume you have selected `lua5_1_Linux24g3_bin.tar.gz` — make the appropriate changes if necessary. Unpack it as follows:

```
cd /tmp
tar xzvf lua5_1_Linux24g3_bin.tar.gz
```

This creates the following directory structure:

```
lua5.1
  bin
    Linux24g3
```

In the `Linux24g3` subdirectory are three files: the interpreter (`lua5.1`), the byte-code compiler (`luac5.1`), and a utility that helps with embedding Lua scripts into C programs (`bin2c5.1`). Make sure the interpreter works with your system, as follows:

```
lua5.1/bin/Linux24g3/lua5.1 -v
```

This should result in the following output:

```
Lua 5.1 Copyright (C) 1994-2006 Lua.org, PUC-Rio
```

You should get the same response when invoking the compiler with the `-v` switch. If you don't get this, you've likely selected a binary package that is incompatible with your operating system.

If the interpreter and compiler work with your system, move them to a location in your search path. If you have root privileges and want to make Lua available to all users of your system, the location `/usr/local/bin` is traditional. If you lack sufficient privileges or are interested only in using Lua yourself, the bin directory beneath your home directory is a good location. In this last case, you may need to create the `bin` subdirectory and set it as part of your PATH environment variable.

Assuming you have root privileges and want to make Lua available to all users of your system, execute these commands:

```
su -
cd /tmp/lua5.1/bin/Linux24g3
mv lua5.1 /usr/local/bin/lua
mv luac5.1 /usr/local/bin/luac
chown root.root /usr/local/bin/{lua,luac}
chmod u=rwx,go=rx /usr/local/bin/{lua,luac}
cd /tmp
rm -fr lua5.1
exit
```

Now you should be able to execute `lua` without any path qualification, like this:

```
lua -v
```

Installing a Prebuilt Binary Package on Windows

If you are installing onto a 32-bit version of Windows, a safe bet is to select the package named `lua5_1_Win32_bin.tar.gz`. This package uses the C runtime library `MSVCRT.DLL`, which is available on all recent versions of Windows.

After you download one of the tarballs, you need to extract the package contents to a suitable location using a utility such as Winzip or 7-zip. The following instructions assume you have downloaded the package `lua5_1_Win32_bin.tar.gz` and are using the 7-zip command-line utility — make the appropriate changes if you have selected a different package or are using a different extraction utility. Place the package in the directory of your choice, and then execute the following commands at a shell prompt:

```
7z x lua5_1_Win32_bin.tar.gz
7z x lua5_1_Win32_bin.tar
del lua5_1_Win32_bin.tar
```

The following directory structure will be created:

```
lua5.1
  bin
    Win32
```

Four files are included in the Win32 subdirectory: the interpreter (`lua5.1.exe`), the byte-code compiler (`luac5.1.exe`), the Lua core in dynamic link library form (`lua5.1.dll`), and an embedding tool (`bin2c5.1.exe`). Change your default working directory to Win32 with the following command:

```
cd lua5.1\bin\Win32
```

Install the dynamic link library and executables as follows:

```
copy /b lua5.1.exe "%UTIL_DIR%\lua.exe"
copy /b luac5.1.exe "%UTIL_DIR%\luac.exe"
copy /b lua5.1.dll "%UTIL_DIR%\"
copy /b bin2c5.1.exe "%UTIL_DIR%\bin2c.exe"
```

You do not rename the library, `lua5.1.dll`, because references to it are embedded in the interpreter and compiler.

Type **lua -v** and **luac -v** from any working directory to make sure the two programs are accessible on the search path.

An additional nicety is to add the Lua icon to this directory. You can copy the icon named `lua.ico` in the `etc` directory of the Lua source package to the utility directory.

Additional Tools

While on the topic of getting situated with Lua, a few notes about programming editors and script management are appropriate. Although these tools and methods are not required to create programs, their use definitely makes you more productive.

Programmer's Editor

One tool you want to choose carefully is your text editor. You'll use it not only to create Lua scripts, but also to read and search through them. A vast number of free and commercial text editors are available for all mainstream platforms. Wikipedia has a comprehensive summary of their availability and features at `http://en.wikipedia.org/wiki/Comparison_of_text_editors`. Programmer's editors provide features well beyond the basic editing of text including features such as the following:

- ❑ Advanced search and replace using regular expressions, a powerful form of recognizing patterns in text
- ❑ Syntax highlighting that gives you visual confirmation that your script is properly structured
- ❑ Multiple undoing and redoing
- ❑ Bracket matching
- ❑ The capability to automatically hide portions of text and code
- ❑ Macros to automatically repeat time-consuming or error-prone operations
- ❑ The capability to filter selected portions of text through an external program (that is, one written in Lua) to process text in ways that might be awkward or complicated using the editor's own commands

Most programmer's editors enable you to configure syntax highlighting for various programming languages, and syntax highlighting for Lua is provided as part of many editor packages. An editor's highlighting mechanism might have trouble with Lua's novel way of handling multiline comments and strings, but you can usually configure it to handle those constructions at least gracefully if not perfectly.

Revision Control System

Organizing your Lua scripts in a directory or directory tree makes it easier to reuse code that you have written. It also simplifies your managing those scripts with a revision control system such as CVS or Subversion. A revision control system enables you to do the following:

- ❑ Recover an earlier revision of a script
- ❑ Review the history of a script's progress in the form of commit log entries
- ❑ Safely develop scripts from more than one machine
- ❑ Back up your work easily because only the repository needs to be backed up
- ❑ Create and merge multiple sets of files — known as *branches* — to distinguish between the installed and developmental versions of your scripts

One advanced system, Monotone, uses distributed repositories with Lua as a scripting language. Revision control is usually associated with collaborative team efforts, but it has many benefits for the independent programmer as well. The usual cycle is to edit and test your script until you reach some sort of milestone, and then commit the script with a note to the revision control system's repository.

Many well-written books and how-to guides exist for setting up and using open-source revision control systems.

Summary

You now have a working Lua interpreter and compiler on your system.

In this chapter, you learned about the following:

- ❑ Lua's package structure
- ❑ How to build Lua from scratch
- ❑ How to install a precompiled Lua package
- ❑ The advantages of a programmer's editor
- ❑ The advantages of a revision control system for your source code

You'll use the various shell operations you learned about in the chapters ahead. When you extend Lua's functionality with libraries, you'll use the same techniques to obtain and install the libraries as you did in this chapter for Lua itself.