Speaking UNIX: The Squirrel portable shell and scripting language

Write object-oriented shell scripts for multiple platforms

Skill Level: Intermediate

Martin Streicher (martin.streicher@gmail.com)
Software Developer
Pixel, Byte, and Comma

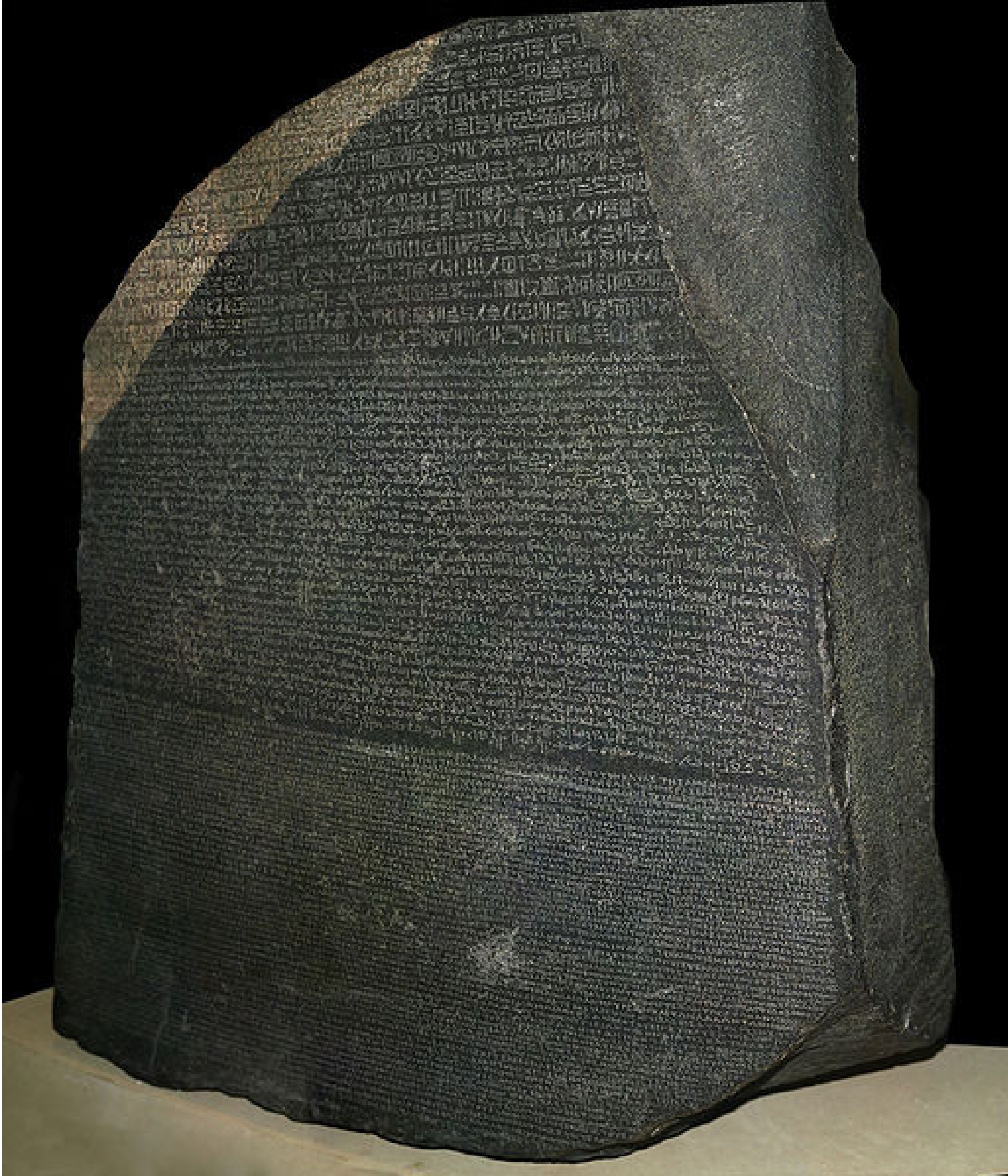
17 Mar 2009

If you don't want to commit to the idiosyncrasies of a specific shell running on a particular platform, try the Squirrel Shell. The Squirrel Shell provides an advanced, object-oriented scripting language that works equally well on UNIX®, Linux®, Mac OS X™, and Windows® systems. Write a script once, and run it anywhere.

In 1799, a French Army engineer made a remarkable find. No, it wasn't foie gras, camembert, pasteurization, or Sartre—in fact, it was the Rosetta stone, the key to deciphering much of Egypt's ancient hieroglyphics (see Figure 1).

Figure 1. The Rosetta stone, an 1100-pound, trilingual tax policy. The inscription is a proclamation relaxing taxes on the priesthood.

© Copyright IBM Corporation 2008. All rights reserved.



The stone, created in 196 BCE, inscribes three translations of a single passage—one each in hieroglyphics, Demotic (an Egyptian script), and classical Greek. Through *comparative translation,* or mapping phrases from one translation to another, the Rosetta stone revealed the meaning of many once-undecipherable glyphs.

© Copyright IBM Corporation 2008. All rights reserved.

In other words, think of the Rosetta stone as a half-ton Babelfish. Even in 196 BCE, there was more than one way to say something.

Software developers face a similar problem some 2000 years later. There are many ways to say the same thing in so many programming languages. Even at the command line, there are many analogs to choose from, including a variety of shells and various combinations of commands.

In general, the variety is good, but it can also be daunting. Which solution do you choose? Will the technology keep pace with requirements? Will the investment of time and effort pay off? Or will those neatly written glyphs (or are those Perl sigils?) become obsolete? Worse, will everything have to be translated (*rewritten*) for other environments?

If you don't want to commit to the idiosyncrasies of the Fish shell, the Bash shell, the Z shell, the Windows operating system's cmd.exe, or some other shell scripting language, try the Squirrel Shell. The Squirrel Shell provides an advanced, object-oriented scripting language that works equally well in UNIX, Linux, Mac OS X, and Windows systems. You can write a script once and run it anywhere.

Even better, you don't have to put a half-ton rock in your ear to use it.

Catching the Squirrel

The Squirrel Shell is readily available and free to use per the terms of the GNU Public License version 3 (GPLv3). The latest release is 1.2.2, dated 11 October 2008. The founder and maintainer of Squirrel Shell is Constantin "Dinosaur" Makshin.

The Squirrel Shell's download page (see Resources for a link) provides source code and binaries for 32- and 64-bit Windows. If you use a flavor of UNIX or Linux, check your distribution's repository for suitable binaries, or build Squirrel Shell from scratch.

A build from scratch is straightforward. Download and extract the source tarball, change to the source directory, and incant the fairly typical build spell shown in Listing 1.

Listing 1. Build the Squirrel Shell from source

```
$ ./configure --with-pcre=system && make && sudo make
install
Checking CPU architecture...   x86
Checking for install...   /usr/bin/install
...
Configuration has been completed successfully.
   Build for x86 CPU architecture
   Installation prefix: /usr/local
   Allow debugging: no
```


© Copyright IBM Corporation 2008. All rights reserved.

```
    Build static libraries
    Use system PCRE 6.7 library
    Install MIME information: auto
    Create symbolic link: no
    Compile C code with 'gcc'
    Compile C++ code with 'g++'
    Create static libraries with 'ar rc'
    Create executables and shared libraries with 'g++'
    Install files with 'install'
```

To find the list of package-specific options for configure, type `./configure --help` at the command line.

For convenience, Squirrel Shell bundles the source code of the Perl Compatible Regular Expression (PCRE) library, which is used extensively in the program. If your system lacks PCRE, the bundled code makes the build quick and easy. However, if your system already has PCRE, you can choose to use it by specifying the `--with-pcre=system` option. Otherwise, specify `--with-pcre=auto` to link with the newer of either the system library or Squirrel Shell's copy.

The result of the build is a new binary, aptly named *squirrelsh.* Assuming that the binary was installed in a directory in your PATH variable, such as /usr/local/bin, type `squirrelsh` to launch the shell. At the command prompt, type the command `printl(getenv("HOME"));` to print the path of your home directory:

```
$ squirrelsh
> printl( getenv( "HOME" ) );
/home/strike
> exit();
```

The Squirrel Shell is based on the Squirrel programming language (see Resources for a link to more information). The language is `C++` like and offers features more akin to object-oriented scripting languages such as Python and Ruby. The Squirrel Shell incorporates all the features and data types found in Squirrel and adds a host of new functions written specifically for common shell scripting tasks, such as copying a file and reading an environment variable.

Although the syntax of Squirrel Shell is too verbose for everyday, command-line use—`echo $HOME` is the Bash equivalent of Squirrel Shell's `printl( "~")`—it shines in scripts. You write once, run everywhere, not write once in UNIX and again for Windows. As Dinosaur says of his work, "Squirrel Shell is primarily a script interpreter."

Scripting with Squirrel

Let's look at an example Squirrel Shell script. Listing 2 shows the file listing2.nut, a script to recursively list the contents of your home directory.

© Copyright IBM Corporation 2008. All rights reserved.

Listing 2. listing2.nut

```
#!/usr/bin/env squirrelsh

function reveal( filedir ) {
  if ( !exist( filedir ) ) {
    return;
  }

  if ( filename( filedir ) == ".." || filename( filedir )
== "." ) {
    return;
  }

  if ( filetype( filedir ) == FILE ) {
          printl( filename( filedir, true ) );
          return;
  }

  printl("directory: " + filename( filedir, true) );
  local names = readdir( filedir );

  foreach( index, name in names ) {
    reveal( name );
  }
}

local previous = getcwd();

chdir( "~" );

reveal( getcwd() );

chdir( previous );

exit( 0 );
```

Per convention, the first line of every shell script tells the operating system what program to launch to interpret the script. Typically, you see `#! /usr/bin/bash` or `#! /bin/zsh` on that line to launch a specific shell or interpreter from a specific location.

A`#!/usr/bin/env squirrelsh` is a bit different. It launches a special program, *env,* that in turn launches the first instance of `squirrelsh` found in your PATH variable. Hence, you can change your PATH variable to favor a local version of some program—say your own, modified copy of squirrelsh in $HOME/bin/squirrelsh—and not change the content of the shell script.

Note: This trick works with all kinds of interpreters. For instance, `#!/usr/bin/env ruby` would invoke your preferred version of Ruby, as dictated by your PATH setting. In general, if you plan to distribute any shell script you write, use the `#!/usr/bin/env` *`application`* form in the first line, as it's more "portable": It runs the version of the application the *user* has configured in his or her PATH variable.

The rest of Listing 2 should seem familiar, at least in approach. The function

© Copyright IBM Corporation 2008. All rights reserved.

`reveal()` is recursive:

- If you pass `reveal()` an invalid path or the perennial "dot" (`.`, the present directory) or "dot dot" (`..`, the parent directory), the recursion ends.
- Otherwise, if the argument, `filedir`, is a file, the code prints its name and returns, again halting further recursion. The function `filename()` can make one argument or two. With one argument or if the second argument is `false`, the extension of the file name is omitted. If you provide `true` as the second argument, the entire file name is returned.
- If the argument is a directory, the code prints its name, then scans its contents. (The processing is not necessarily depth-first, because the contents of the directory are not ordered in a special sequence. The next example improves the output.)

One item of interest: Because the call to `reveal()` is the last statement in that very same function, the Squirrel virtual machine (VM)—the engine running the script code—can permute the recursion into iteration through a technique called *tail recursion.* Essentially, tail recursion eliminates the use of the call stack for recursion; thus, arbitrarily deep recursion is possible and avoids stack overflow.

The syntax of Squirrel is pretty sparse, so writing code in the language comes quickly, especially if you have written code in `C`, `C++`, or any higher-level language.

Best of all, this shell code is portable. Transfer it to a Windows machine, install Squirrel Shell for Windows, and run your code.

Tinkering with tables

One of the nice features of Squirrel is its rich set of data structures relative to typical shells. Complex problems can often be solved quickly if data can simply be organized well. Squirrel has true objects, heterogeneous arrays, and associative arrays (or *tables,* in Squirrel speak).

A Squirrel table is composed of *slots,* or *(key-value)* pairs. Any value except Null can serve as a key; any value can be assigned to a slot. You create a new slot with the "arrow" operator (`<-`).

Let's improve the code in Listing 2 to show the contents of a directory before descending further into any subdirectories. The approach? Use a local table to accumulate files and subdirectories in separate slots, then process both categories accordingly. Listing 3 shows the new version of the code.

Listing 3. An enhanced version of Listing 2 to print a directory's contents first,

© Copyright IBM Corporation 2008. All rights reserved.

then recurse into subdirectories

```
#!/usr/bin/env squirrelsh

function reveal( filedir ) {
  local tally = {};
  tally[FILE] <- [];
  tally[DIR] <- [];

  if ( !exist( filedir ) ) {
    return;
  }

  if ( filename( filedir ) == ".." || filename( filedir )
== "." ) {
    return;
  }

  local names = readdir( filedir );

  foreach( index, name in names ) {
    tally[ filetype( name ) ].append( name ) ;
  }

  foreach( index, file in tally[FILE] ) {
    printl( file );
  }

  foreach( index, dir in tally[DIR] ) {
    printl( filename( dir ) + "/" );
  }

  foreach( index, dir in tally[DIR] ) {
    reveal( dir );
  }

}

local entries = readdir( (__argc >= 2) ? __argv[1] : "."
);

exit( 0 );
```

A table is an ideal data structure to use here. The table in `reveal()` has two slots: one for files and one for directories. The return value of the function `filetype( name )`—either the constant `FILE` or the constant `DIR`—collates each item in the file system into its appropriate slot.

Further, each slot is an array, created by the two statements `tally[FILE] <- []` and `tally[DIR] <- [];`. (`[]` is an empty array.) Because `tally` is a local variable within the function, it is created anew with each call and falls out of scope and is destroyed automatically when each call returns.

The array function `append( arg )` adds *`arg`* to the end of the array, thus accumulating a list in the process. After the loop `foreach( index, name in names )`, every item has been put in one slot's list or the other. The rest of the code found in the function prints the files, then the directories, then recurses.

Of course, a shell script would be somewhat worthless without command-line

© Copyright IBM Corporation 2008. All rights reserved.

arguments. The special Squirrel Shell variables `__argc` and `__argv` contain the count of command-line arguments and the list of arguments as an array of strings, respectively. Again, per convention, `__argv[0]` is always the name of the shell script; hence, if `__argc` is at least 2, then additional arguments were provided. For brevity, this script processes only the first extra argument, `argv[1]`.

For reference, Listing 4 shows a Ruby script (written by Mr. Makshin) that is functionally identical to Listing 3. As terse as Ruby can be, it still isn't as succinct as the Squirrel Shell code.

Listing 4. A reimplementation of Listing 3 in Ruby

```
!/usr/bin/ruby

# List directory contents.

path = ARGV[0] == nil ? "." : ARGV[0].dup

# Remove trailing slashes
while path =~ /\/$/
  path.chop!
end

entries = Dir.open(path)
for entry in entries
  unless entry == "." || entry == ".."
    filePath    = "#{path}/#{entry}"
    fileStat = File.stat(filePath)
    if fileStat.directory?
      puts "dir : #{filePath}"
    elsif fileStat.file?
      puts "file: #{filePath}"
    end
  end
end

entries.close()
```

For more information about the Squirrel language, see the *Squirrel Programming Language Reference* (see Resources for a link).

Smartly, virtually all of the functions in the Squirrel Shell abstract away the specifics of the underlying operating system, so your code can be as generic as possible. For example, the `filename()` function (used in the first two listings above) strips the leading path from the file path name— reducing /home/example/some/directory/file.txt to file.txt, for example—no matter what platform your are on. Similarly, `readdir()` and `filetype()` allow you to remain blissfully ignorant of the machinations and trappings of the real, underlying operating and file systems. Typically, a regular shell does not offer such abstractions. (The more advanced scripting languages do.)

Other helpful, platform-independent functions include `convpath()` to convert a path name to the native path name format and `run()` to invoke another executable. The

© Copyright IBM Corporation 2008. All rights reserved.

`convpath()` function is bidirectional and very helpful when writing cross-platform scripts.

Mostly regular expressions

Shell scripts are used commonly to automate systems administration and maintenance work. The power behind much of this automation is the *regular expression*, a veritable set of hieroglyphics to find, match, and decompose strings. As mentioned at the outset, Squirrel Shell requires the PCRE library, which is also found in Perl, PHP, Ruby, and many other interpreters and programs. PCRE is the Samurai sword of data slicers and dicers.

Although complete, the Squirrel Shell's implementation of regular expressions is a bit different and may remind you of PHP's implementation. To use a regular expression in Squirrel Shell, you define the regular expression, compile it, make a comparison, then iterate over the results, if any.

Listing 5 shows a sample program that demonstrates regular expressions in Squirrel Shell (the code was written by Mr. Makshin and is used with permission).

Listing 5. A demonstration of regular expressions in Squirrel Shell

```
#!/usr/bin/env squirrelsh

// Match a regular expression against text

print("Text: ");
local text = scan();

print("Pattern: ");
local pattern = scan();

local re = regcompile(pattern);
if (!re)
{
        printl("Failed to compile regular expression - " +
regerror());
        exit(1);
}

local matches = regmatch(re, text);
if (!matches)
{
        printl("Failed to match regular expression - " +
regerror());
        regfree(re);
        exit(1);
}

regfree(re);
printl("Matches found:");
foreach (match in matches)
        printl("\t\"" + substr(text, match[0], match[1]) +
"\"");
```


© Copyright IBM Corporation 2008. All rights reserved.

Here, `scan()` reads some text and a pattern from standard input, albeit without the leading and trailing slash (`/`) characters that typically delimit the start and end of a regular expression.

Given a pattern, the function `reqgcompile()` compiles the pattern, which speeds repeated matches. You can enable or disable case sensitivity with a flag to the `reqgcompile()` function (equivalent to the PCRE `/i` modifier), and you can match against one line or many with another option (an equivalent for the PCRE `/m` option). If you do not compile the regular expression, all matches fail.

The `regmatch(re, text)` function compares the regular expression to the text, yielding either Null for no matches or an array of pairs of integers (a two-element array). The first integer in any pair is the start of the match; the second integer is the end of the match. This explains the use of `substr(text, match[0], match[1])` in the last line of code.

After you perform the comparison, you can iterate over the results. If at any time you no longer need the compiled regular expression, dispose of it with `regfree()`. There is also a `regfreeall()` function that disposes of all resources held by all compiled expressions.

Nuts for the Squirrel Shell

In a perfect world, the same programming logic would apply to UNIX, Linux, and Windows, and programmers would be happier if not more productive. Alas, operating systems differ, and there are times when you have to resort to custom code for a specific system.

In those cases where neither Squirrel Shell nor you can abstract the platform away, Squirrel Shell provides a handy function to probe the operating system so code can follow an appropriate branch.

Listing 6 shows how to use the `platform()` function to make decisions. This function always a returns a value, although it could be the value `unknown`.

Listing 6. The platform() function yields the type of operating system

```
print( "Made by ... ");

local platform = platform();

switch ( platform ) {
  case "linux":
    printl( "Linus." );
    break;

  case "macintosh":
    printl( "Steve." );
```


© Copyright IBM Corporation 2008. All rights reserved.

```
    break;

  case "win32":
  case "win64":
    printl( "Bill." );
    break;

  default:
        printl( "Unknown" );
}
```

You can also find the type of the current platform through the Squirrel Shell environment variable PLATFORM:

```
> printl( PLATFORM );
linux
```

The environment variable CPU_ARCH yields the processor for which the shell was compiled:

```
> printl( CPU_ARCH );
x86
```

Ah, nuts!

The remainder of the Squirrel Shell functions manage files, manipulate the environment, and perform mathematics. Indeed, there are nearly 20 built-in functions for trigonometry. Version 2.0 is being planned now and will include more classes, support for Unicode, an improved interactive mode, and a modular, plug-in architecture.

Squirrel Shell isn't much of an interactive shell, but that's okay. There are many alternatives in that category already. Squirrel Shell is far better as a script runner. Its data structures are more capable than a traditional shell, it's syntax is familiar, and its underlying virtual engine supports everything from enumerated types to threads. The Squirrel engine is also tiny, at less than 6000 lines of code. You can even embed all of Squirrel in another application.

Try the Squirrel Shell when you have to write code for two platforms. Its nuts will help you keep yours.

© Copyright IBM Corporation 2008. All rights reserved.

Resources

Learn

- Check out the Squirrel Shell.
- The Squirrel Shell is based on the Squirrel programming language. You can download and read the *Squirrel Programming Language Reference* to learn more about Squirrel and its many features. There is also a manual for the functions found in the Squirrel Shell.
- Be sure to check out PCRE.
- Speaking UNIX: Check out other parts in this series.
- Learn more about UNIX shells.
- The AIX and UNIX developerWorks zone provides a wealth of information relating to all aspects of AIX systems administration and expanding your UNIX skills.
- New to AIX and UNIX? Visit the New to AIX and UNIX page to learn more.
- Browse the technology bookstore for books on this and other technical topics.

Get products and technologies

- The Squirrel Shell is available for download at no cost.

Discuss

- Check out developerWorks blogs and get involved in the developerWorks community.
- Participate in the AIX and UNIX forums:
    - AIX Forum
    - AIX Forum for developers
    - Cluster Systems Management
    - IBM Support Assistant Forum
    - Performance Tools Forum
    - Virtualization Forum
    - More AIX and UNIX Forums

About the author

© Copyright IBM Corporation 2008. All rights reserved.

Martin Streicher
Martin Streicher is a freelance Ruby on Rails developer and the former Editor-in-Chief of *Linux Magazine*. Martin holds a Masters of Science degree in computer science from Purdue University and has programmed UNIX-like systems since 1986. He collects art and toys. You can reach Martin at martin.streicher@gmail.com.

Trademarks

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.
UNIX is a registered trademark of The Open Group in the United States and other countries.
Windows is a registered trademark of Microsoft Corporation in the United States, other countries, or both.
Mac OS X is a trademark of Apple Inc.

© Copyright IBM Corporation 2008. All rights reserved.