67 Scripting Languages

Robert E. Noonan
College of William and Mary

67.1 Introduction

According to WebMonkey [**15**]:

> A scripting language is a simple programming language used to write an executable list of commands, called a script. A scripting language is a high-level command language that is interpreted rather than compiled, and is translated on the fly rather than first translated entirely. JavaScript, Perl, VBscript, and AppleScript are scripting languages rather than general-purpose programming languages.

A major characteristic of scripting languages is that they can serve as *glue* for connecting existing components or applications together. Scripting languages usually have powerful string processing operations since text strings are a fairly universal communication medium.

67.2 History

Scripting languages originated in the early 1960s. At that time, they were used as a device for defining the control sequence for running a series of programs one after the other. Such a series was often known as *batch processing*, and it was the predominant means for accomplishing computing tasks in commercial data processing environments at that time.

67.2.1 Shell Languages

The earliest scripting languages were the job control languages of the 1960s. Their primary use was to instruct the operating system on the sequence of steps required to run a job.

For example, a job to develop a summary sales report for the past month might have the following steps:

1. A program to select all sales records for the previous month from a larger file and copy them to a temporary file
2. A program to sort the temporary file into sequence by sales district, usually using a general-purpose utility
3. A program to calculate subtotals and present the information in a way that is easy for the end-users to read
4. A program to format and display selected pages of the end-user information on a printer or terminal.

However, these languages lacked almost all of the features you might expect, including variables, if and loop statements, expression evaluation, etc. The first true job control languages capable of scripting appeared with the emergence of Unix in the 1970s. The command interpreter in Unix was known as a *command shell*; a file of such shell commands was termed a *shell script*.

The earliest shell was known as the Bourne shell, named after its creator, Steve Bourne. Although later shells, such as the C shell created by Bill Joy, became more popular as a command interpreter, the Bourne shell maintained its popularity for writing shell scripts.

Figure 67.1 shows a typical grading script. This application assumes that all the files needed to grade a class assignment have been collected into a single directory. Each student has his/her own subdirectory. The files in the directory include input test data files, various log files, various shell scripts, etc.

The purpose of the script is to compile each student's program files, and, if successful, execute the resulting program on all of the test cases, producing an output file for each input test case.

Assuming the name of the script was `testem.sh`, it would be invoked as

```
testem.sh Freecell *
```

The first command line argument is the name of the file that contains the main program, minus its suffix.

```
#! /bin/sh
if [$# -lt 2]; then
    echo Usage: testem. sh mainprog dir1 …
    exit
fi

main=$1; shift
# for each student
for d in $*; do
    if [ -d $d -a ! -f $d/DONE. log ]; then
         cd $d
         echo $d

         if [ ! - f $main.class ]; then
              cp ../*.java .
             echo "compiling $d"
             javac $main.java &> compile.log
    fi
         touch DONE.log
    cp ../*. $data.
    for f in ../*. $data; do
             java $main $f < $f & > $f.log
        done
        cd ..
    fi
done
```

FIGURE 67.1 A student test script.

When the script is invoked, it determines whether it has fewer than two arguments: the first is the filename of the file containing the main program. The script must also be invoked with at least one student subdirectory name. If the test fails, the script prints a usage message and then quits.

Otherwise, the first command argument is stored in the variable `$main`, and then deleted from the command line. A for loop is then executed on the remaining command line arguments.

Since each student corresponds to a subdirectory, line 10 tests to see if the argument is a directory and if it is not done. If both conditions are met, then the argument represents an untested student. The next two lines change directories into the student's directory (note the `cd` back to the parent on line??) and echos the student id to the screen.

The script next checks if a class file exists for the java source; if not, it copies any instructor supplied code into the student directory and compiles the result with output diverted to the file `compile.log`.

Next we record the fact that this student has been tested by creating the file `DONE.log` in the student directory. The next four lines test the student program on a set of test data supplied by the instructor, with the output diverted to a separate file for each input test case.

For many class assignments, this is the only script we need to test the student-written programs. It handles a wide variety of cases, including the following:

- Assignments that consist of a single source file, as well as those that include both student and instructor written source code. No assumption is made about who supplied the main program file.
- The student program can obtain the command line argument and then open and read that file. Alternatively, the program can read its input from the keyboard.

Note the use of variables in the script representing the current student and the current test.

67.2.2 Text Processing Languages

As part of the development of Unix, a number of text processing utilities were developed, as well as the troff word processor for writing documentation. The next step in the evolution of scripting languages was the development of specialized languages for processing text, specifically, *grep*, *sed*, and *awk*. The latter was an acronym made up of the first letters of the languages developers: Aho, Weinberger, and Kernighan. *awk* appeared around 1978.

In this section, we present two *awk* scripts: the first is a very small, typical script, while the second is considerably more complex. The first is so small and useful that such scripts are still common today. Scripts like the second served as the impetus for the development of general purpose scripting languages.

The first script was written back in the late 1980s when one of the authors was the systems manager for a network of Unix computers. In the fall semester of each academic year, there were many new students and a few new professors who needed computer accounts. At that time, creating a new user account was a multistep, manual process carried out by student help.

One fall day a new student came to see me because they were unable to login to their new Unix account. My investigation quickly revealed that their record in the user table had one too few fields. It was a simple matter to edit the user table and add the missing field. This quick resolution made the user happy, but not I.

One solution to prevent this problem from recurring was to create a written procedure with a checklist to create a new user account. Although this approach had a certain appeal. I decided to take the extra step of implementing the procedure as a combination of a *Makefile*, a few shell scripts, and a few simple *awk* scripts.

Since editing the user table was a manual process, I decide to implement the checks on this step as a sequence of (largely *awk* scripts). The first of these was to check that each line had the same number of fields:

```
#! /usr/bin/awk -f
NR == 1 { ct = NF }
NF != ct { print NR, $0 }
```

An *awk* program consists of a *guard* followed by a block of statements, repeated as many times as desired; the first line is a pseudo comment used in Unix/Linux computers that identifies the interpreter to use. The *awk* processing cycle consists of reading an input line, then testing each guard in the program. For any guard that is true, its corresponding block is executed, after which the testing of guards resumes. Only after all the guards have been tested and their blocks executed, if applicable, is the next input line read and processed.

Based on a field separator (the default is whitespace), the input line is broken up into fields. The *awk* variable `NR` is the current line (record) number and `NF` the number of fields on the current input line. The variables `$1, $2`, … hold the contents of each field, whereas the variable `$0` refers to the entire line.

So the aforementioned program a consists of two guarded blocks. The first guard is true only for the first line read; its block stores the number of fields on the first input line into the variable `ct`. The last guarded block prints any lines whose number of fields disagrees with `ct`.

Although the author is no longer a systems manager, this first *awk* script still finds occasional use even today.

The second script is considerably more complex. It was developed to solve the following problem. The authors were writing a text book. In addition to the text, a number of nontrivial programs were developed. The authors needed to include portions of these programs as examples. Most importantly, they wanted the code in the textbook to be identical to the code used in the programs.

So they used a `Makefile` and a set of text filters for extracting the code of interest into a file, which was then automatically included in the book. The *awk* script presented here was given the task of extracting a single function or method in Java. A typical script parameter was the return type and name of the method, which made the starting point easy to find. Having located the starting point, the end point was determined by basically counting braces, based on the authors' coding style:

```
#! /bin/sh
awk "BEGIN { ct = 0; prt = 0 }
/$1/ { prt = 1 }
prt > 0 { print \$0 }
prt > 0 && /{/ { ct++ }
prt > 0 && /}/ { ct-- }
prt >0 && ct == 0 { prt = 0 }
"
```

Before the first input line is read, the script sets a `prt` switch to off (0). It then searches through the input file until it finds the function it is looking for, namely, the second guard, whose block turns the switch on. The third guard prints the input line if the `prt` switch is on. The next two statements increment/decrement the brace count whenever the `prt` switch is on and they detect a left/right brace. The final guarded block turns the switch off hen the brace count goes to zero.

There is a very subtle point in this script. The observant reader will have noted that, unlike the previous script, this one is a shell script which then invokes the *awk* interpreter. The parameter to the script is substituted into the second guarded block, while in the statement of the third guarded block, the dollar sign is escaped so that to the *awk* interpreter the statement appears as

```
print $0
```

Quirks like this, together with the limited processing ability of text processing languages, led to the development of more general purpose scripting languages, which are discussed in the next section.

67.2.3 Glue Languages

> Larry Wall … created Perl when he was trying to produce some reports from a Usenet-news-like hierarchy of files for a bug-reporting system, and *awk* ran out of steam. Larry, being the lazy programmer that he is, decided to over-kill the problem with a general purpose tool that he could use in at least one other place. The result was the first version of Perl. [12]

In the 1980s, scripting languages began to move away from simple text processing to more complex tasks with the development of the languages Perl and Tcl/Tk. A hallmark of a *glue* language is its ability to take data output from one application and reformat it so that it could be input to another application. Initiallly, scripting languages were considered much too slow and inefficient to implement an entire application. With the development of faster interpreters and faster computers, this assumption disappeared.

Although Perl and Tcl/Tk had their roots as a Unix scripting languages, all major scripting languages are now available for all major platforms including Unix, Linux, Macintosh, and Windows. These languages include Perl, Tcl/Tk, Python, and Ruby.

For example, the first author has used these scripting languages for the following tasks:

- Systems administration tasks on a network of computers including: setting up new user accounts, running backups, reconfiguring servers, and installing software.
- Class management, including converting electronic class rolls from one form to another, managing email for each course, grading support, and an interactive, GUI-based grading system.
- A textbook support system, which consisted primarily of scripts for including source code in a book.
- Newer versions of programs previously written in conventional programming languages such as C++ and Java. Such programs fall into a variety of subject domains including simple utility programs, computer science research, etc.

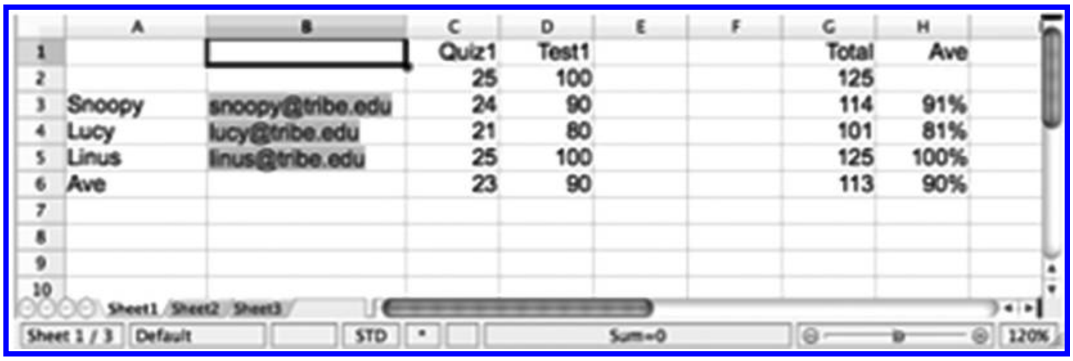
To illustrate the robustness of scripting languages, let us consider the academic task of maintaining a grade book of numeric grades earned by students in a class for their assignments, quizzes, tests, and exams. Each grade may have a different weight; for example, an exam may be more heavily weighted than a quiz. The script presented here provides a secure way to deliver grades to students.*

A spreadsheet is used to do the grade management itself, one spreadsheet per class. An example spreadsheet is given in Figure 67.2. In this example, column one is used for student names and column two is used for email addresses. The rightmost two columns used are weighted totals and averages. The intervening columns are used for grades on assignments and tests; at any point in time, any number of grade columns may be blank (empty).

Also, note that the first row is a descriptive title for each grade, and the second row for the maximum points possible for each grade. The last row is used for the average for each assignment or test. The intervening rows contain the students records, one row per student.

The scripting problem here is to move the grade values from the spreadsheet to the email system. This process consists of exporting the data in textual form from the spreadsheet and then using a Python script to generate the email. The spreadsheet output used is comma-separated-values. We have found over the years that each spreadsheet application has its own, often unique definition of the format of

* Specialized web software developed for education, such as Blackboard and WebCt, provide equivalent functionality to instructors through secure login to a dedicated Web server. However, the system described here was developed long before such software became available; it relied on the use of email for conveying grades back to students.



	A	B	C	D	E	F	G	H
1			Quiz1	Test1			Total	Ave
2			25	100			125	
3	Snoopy	snoopy@tribe.edu	24	90			114	91%
4	Lucy	lucy@tribe.edu	21	80			101	81%
5	Linus	linus@tribe.edu	25	100			125	100%
6	Ave		23	90			113	90%
7								
8								
9								
10								

FIGURE 67.2 A spreadsheet gradebook.

comma-separated-values. Fortunately, Python provides a module (or library) which deals with that. For our current spreadsheet the third row would appear as

```
“Snoopy”,“snoopy@wm.edu”,24,90,,,114,91.2
```

In the remainder of this section, we describe the Python script given in Figure 67.3 as a means of examining some of the features of glue languages in general, and Python in particular.

As noted in the previous section, line 1 of the script is a pseudo comment that gives the full path name of the language interpreter. Line 2 imports any needed modules (or libraries).

In lines 4–6, we set any global constants that are likely to change. For this script, these are kept in an dictionary or associative array as a key-value pair. This allows each grade spreadsheet to have its own set of values, which can be read from a file, overriding the defaults. These parameters include the column number of the student's name, his/her email address, etc. For the sake of simplicity, the routine for reading (and checking) these parameters (amounting to 11 lines of code) is omitted here.

The main logic of the program is shown in lines 8–10. First, the CSV input file is read and stored in an array of students. All the student entries must be read because the averages (which are included in each mail message) are last. Then an email is constructed and sent to each student:

All that remains is to examine the functions `readGrades` and `emailGrades`. The first function is passed the name of the file to be read as the first command line argument; a simple GUI alternative would use a dialog box to ask for the file name. The first line of the function sets the variable `student` to be an empty, indexed array. The `filename` is opened and passed to a reader which deals with comma-separated format files. The `for` loop reads one row per iteration, which is appended to the end of the `student` array.

The function named `emailGrades` is passed the `student` array. Next, leading nonstudent entries are copied and trailing ones removed. Then a simple `for` loop that generates an email message per student entry.

Basically, this function iterates over the students, generating a string that contains the student's grades and then mailing them. It counts the lines as it goes so that it can skip the initial lines which contain header information. On a row containing an actual student, the function formats the student's grades in a manner familiar to any programmer, storing the message in the string `outline`.

The function *emailGrades* then creates a subprocess that executes a system command; the `mailx` command works here if you have a static IP address, otherwise it is somewhat more complicated. Note that the subprocess's file `stdin` is set to be a pipe, which is then filled by the `proc.communicate` method call. The function then waits for the subprocess to finish. One essential feature of a *glue* language is the ability to execute system commands (like `mailx`) and send them input or read their output or both.

```
#! /usr/bin/python
import sys, csv, process

param = {“name”:0, “email”:1, “gradecol”:2, “studentrow”:2,
        “titles”:0, “maxgrade”:1}
readPrefs(“ugrades.pref”)

student = readGrades(sys.argv[1])
ct = emailGrades(student)
print ct, “messages sent”

def readGrades(filename):
    student = [ ]
    for row in csv.reader(open(filename, “rU”)):
        student.append(row)
    return student

def emailGrades(student):
    titles = student[prefs[“titles”]]
    maxgrade = student[prefs[“email”]]
    ave = student.pop()
    ct = 0
    for s in student:
        ct += 1
        if ct <= prefs[“studentrow”]:
            continue
        outline = formatGrades(s, titles, maxgrade, ave[s])
        proc = subprocess.Popen(“mailx –sgrades %s@cs.wm.edu”
                %(student[prefs[“name”]],
                student[prefs[“email”]]),
                shell=True, stdin=subprocess.PIPE)
        out = proc.communicate(outline)
        proc.wait( )
    return len(student) – prefs[“studentrow”]
```

FIGURE 67.3 A glue script for emailing grades.

The function `formatGrades` (not shown here) generates a multiline email message; it amounts to 11 lines of code.

Python has some interesting features that are worth noting here. First, it uses the plus symbol as the string concatenation operator. Second, it provides no automatic conversion from a number to a string, even in string concatenation. Third, the percent operator with a string as the left operand and a tuple as the right operand is basically the `sprintf` function of C; the left operand serves as a formatting code for each operand in the tuple. After setting up some column headers, the `for` loop generates one grade per output line, each containing a grade name, a maximum grade, and the grade earned by the student.

As presented here, the entire program has fewer than 40 lines of code. It was written and debugged in less than a day to replace a similar, cryptic, less flexible Perl script. The script accommodates spreadsheet output generated by Excel, OpenOffice Calc, and GoogleDocs using a variety of conventions used by the author and his graders. It has proven to be a very valuable glue script, since it is used about 10 times per semester per class.

This typical Python glue script linking two disparate applications exposes many commonly used features of glue languages such as Python:

- Scripting languages do not require declaration of variables and supports dynamic typing.
- Scripting languages support a wide variety of string operations, including pattern matching.

- Scripting languages provide both dynamic arrays (lists) and dictionaries (also known as associative arrays or hash tables).
- Scripting languages provide a means of executing system commands with the ability to supply input to those commands or to read their output.
- Unlike many languages, Python has a simple syntax. A compound statement such as a function defi ition (`def`), a `for`, or an `if` end with a colon. The statements forming the loop body (for example) being indented. In Python if a statement looks like it is part of a function definition, loop body, or a then/else body, then it is interpreted as one.
- Python supports a large number of modules, which are very useful language extensions. In this program, we used the `sys` module to access command line arguments, the `csv` module to simplify reading CSV format files, and the `subprocess` module to create a subprocess capable of executing system commands and communicating with its parent.

Finally, we note that many universities now teach Python in their CS1 courses (the first course in a computer science major). One reason for this is that the course is rapidly becoming a service course, taken primarily by physical science and social science majors. Its popularity is partly due to its simple syntax and partly because of this value as a tool for solving computing problems in other fields of study.

67.2.4 Web Languages

According to its developer, Rasmus Lerdorf, the motivation for developing PHP [4] was the following:

> As the Web caught on, the number of non-coders creating Web content grew exponentially. … But soon they were asked to add dynamic content to their sites. … This is where PHP found its niche. … I had written all sorts of CGI [Common Gateway Interface] programs in C, and found that I was writing the same code over and over. What I needed was a simple wrapper that would enable me to separate the HTML portion of my CGI scripts from my C code … This concept became PHP.

Lerdorf initially developed PHP in 1994, but because its usage quickly grew beyond the abilities of a single developer, it became an open-source product. PHP is a server-side scripting language whose scripts can be directly embedded within the HTML code that defines the page layout. In this way, PHP is ideal for use with Web pages that have dynamic content, including forms processing and database access.

A major advance in the development of PHP was the release of PHP 5 in 2004, which added the following major features:

- A new OOP model
- Improved database support for MySQL and SQLite
- A host of new functions to simplify coding of common features, such as date and array handling

These features, along with its basic syntactic similarity with Java, C, and C++, tend to obscure the distinction between PHP and other general purpose languages. They also make PHP easier to learn for programmers who have prior experience with these other languages.

At many sites, PHP is installed on the main Web server. It is often installed together with MySQL, an open source database management language. A popular package used across many server varieties is known as *xAMP*, which combines PHP, MySQL, and an Apache server in an easily installed server configuration that can be run either locally or remotely.*

* The Windows variety is called WAMP, the Mac OS variety is called MAMP, and the Linux variety is called LAMP, appropriately. All three varieties of xAMP are open source and so can be freely downloaded and used by any software developers.

Typical examples of PHP usage on a Web page include

- Dynamic content such as today's date or a news item of the day
- Forms processing, including forms validation
- Database access

The PHP processor takes a document file as input and produces HTML as output. Like JavaScript, the input file consists of a mixture of HTML and PHP, which are marked by special HTML-like tags. The major difference from JavaScript is that a PHP script is executed on the server, not on the client. This provides a level of security that is unachievable with JavaScript, since the PHP script is never downloaded to the client's Web browser.

The PHP processor has two modes of operation. It begins in copy mode in which HTML tags and text are copied directly to the output. When it encounters the special tags:

```
<?php
// one or more lines of PHP code
?>
```

the PHP processor switches to script mode and interprets the script on the fly. The output from the script, whatever is written to stdout, is inserted directly into the HTML page. The double slashes (//) denote a PHP comment, which continues until the end of the line.

Figure 67.4 contains a simple example that illustrates some of the features of PHP. Familiarity with both HTML tags and C programming is presumed. In this example, we use a PHP script in conjunction with a database to produce department directories, one each for the faculty, the staff, and the graduate teaching assistants. The directory desired is specified as a parameter in the URL; for example, http://www.cs.wm.edu/people/index.php?id=Faculty will list the faculty directory.

```
<?php
$title = $_GET['id'];
include "header.inc";
if (! eregi(":Faculty:Staff:GradTA:", ":$title:")
    errorPage($title);
include "dbaccess.inc";
$result = mysql_query ("select * from $title");
print '<center> <table cellspacing=5 cellpadding=5 border=2>
    <tr align=left><th>Name</th><th>Office</th>
    <th>Phone</th><th>Email</th></tr>
';
while (list($name, $url, $office, $phone, $email) =
                mysql_fetch_array ($result)) {
    print "<tr align=left>\n";
    $lname = $name;
    if ($url !="")
        $lname = "<a href=$url>$name</a>";
    print "<td>$lname</td>\n";
    print "<td>$office</td>\n";
    print "<td>$phone</td>\n";
    print "<td><a href=mailto:$email>email</a></td>\n";
    print "</tr>\n";
}
print '</table> </center> ';
include "trailer.inc";
? >
```

FIGURE 67.4 A PHP script for a directory listing.

With this parameter, the script accesses the appropriate database table, in this case the faculty table, and generates the appropriate HTML output. Because these directories are fairly static, it is fair to ask the question: Why not maintain the information as static HTML pages? Here are three reasons:

1. The staff people who maintain this information need only interact with a form that interfaces with the database—they do not need to learn HTML.
2. Using PHP allows the webmaster to more easily maintain a consistent look and feel to the web pages.
3. The information can be accessed from other portions of the web site.

A PHP page begins in HTML mode; this would be used to set up the page in a site-specific standard format, including the title, background color, navigation buttons, etc. Because HTML lacks an include facility, a common use of PHP is to set up the page via parameterized header (line 3) and trailer (line 25) files.

The global `$_GET` is an associative array of variable-value pairs that are passed to the script via the URL parameters. In this case, the parameter `id`, whose value is the string `Faculty`, is passed to the PHP variable `$title`. The `header.inc` script expects the title variable to be set.

Before we begin setting up the body of the page, we want to check to see that the `id` parameter has a valid value. Because this value is used to access a database table, a malicious user could attempt to attack the database by providing an unexpected value. In this case, we test the value supplied for the id parameter using a pattern match against the three legal values. If the pattern match fails, an error routine is called, which generates an error page and exits with no further processing.

Otherwise, database access code is included, which sets the name of the database along with security information such as userid and password. The select SQL statement is shown, which in this example accesses the faculty table in the database. At the current time, each of the three tables contains the following information for each person:

- The person's name
- A URL, if they have a homepage
- Their office address
- Their phone number
- Their email address

The information is to be generated as an HTML table with one row per person. Each column should have an appropriate heading (lines 7–10).

Next comes the main logic of the script. A while loop (line 11 ff.) is used to fetch the rows of the result of the SQL query one person at a time. That result is returned as an array, so the list function is used to assign the array values to conveniently named variables. The body of the loop consists mostly of print statements to write the appropriate columns: As with Perl, variable references may be freely embedded in double-quoted strings. The name field is made into a link if the person has a non-empty URL field.

All that remains is to close the table and center HTML tags and invoke the standard web site trailer:

This script is a typical example of server-side scripting and exposes some of the commonly used features of PHP:

PHP scripts freely alternate between pure HTML and PHP.

PHP does not require the declaration of variables and supports dynamic typing. The same value can be treated both as a string and as a number (provided it can be interpreted as a valid number).

PHP supports a wide variety of string operations. It also supports pattern matching as found in the Unix utilities grep and sed.

PHP provides both dynamically sized arrays and associative arrays (hashtables).

PHP directly supports accessing information from a database.

Besides PHP, all of the glue scripting languages can be used for dynamic web programming, including Perl, Python, and Ruby.

67.2.5 Embedded Languages

Embedded languages provide functionality that can be activated by the user by the user to perform specific tasks, like text editing. One of the earliest examples of an embedded language is GNU Emacs, developed in the 1980s.

Like other versions of Emacs available at the time, GNU Emacs had an embedded Lisp interpreter. Everything from setting startup options to the development of modes was done in Lisp. This made Gnu Emacs customizable by the end user. Most of the customizations themselves were developed by end users writing in Lisp.

For example, a typical customization of Emacs was developed for editing Python programs. This customization automatically indents the program as it is typed, colorizes reserved words, identifiers, strings and other literals, etc. By selecting a region, the user can comment it out, indent it left or right, wrap lines as needed, etc.

The development of computer games, whether on a dedicated console or on a personal computer, has rekindled interest in embedded languages. Many books on writing game programs discuss the use of an embedded scripting language. The choice of language can vary from Lisp/Scheme to Lua to languages similar to Javascript. Factors that are considered in choosing an embedded scripting language include its suitability to the task, familiarity of the development team with the language, and the size and speed of the interpreter for the language.

Tasks performed by the embedded language may include loading options and configuration files at startup, saving and reloading state information, implementing and interpreting finite-state machines, moving objects in the game, etc.

An important consideration in choosing an embedded scripting language is the size and speed of the interpreter. Consider Lua as an example. Compared to Python, Lua has a paucity of data structures and types. Python offers both infinite precision integers, as well as floating point numbers, while Lua offers only the latter.

67.3 Principles

Until recently, scripting languages were designed primarily for programming in the small, as opposed to application programming languages, which were designed for programming in the large. The languages used dynamic typing and were interpreted rather than compiled. The primary application for scripting languages was to glue existing applications and utilities together to make a new application.

The advent of languages like Java and C# began to muddy these waters. Like scripting languages, they were compiled to byte code for a virtual machine and then interpreted. Other than the fact that they used static rather than dynamic typing, there seems little to distinguish them from scripting languages such as Perl and Python.

Entire applications, such as content management systems for web applications, are now being written in scripting languages. Indeed, except for embedded languages, such as Lua, the distinction between scripting languages and application programming languages may soon disappear. In such a case, scripting languages should be held to the same principles as application programming languages. Such principles include

1. A well-defined syntax, expressible in a content-free grammar
2. A clear, well-defined semantics, which is expressible in an efficient implementation
3. A well-defined type system
4. The syntax, semantics and type systems should work together to minimize programming errors

5. The language should support its intended application area with a rich set of constructs that encompass needed functions and idioms
6. The language should be as simple as possible, both to learn and to use
7. The language should support future extensions while maintaining backward compatibility

In the remainder of this section, we will consider each of these in turn. In cases where principles are in conflict with one another, the language designer must make appropriate tradeoffs to achieve a workable implementation for the language.

Further complicating the issue is that, with the exception of Ruby and Lua, most scripting languages began as imperative languages, to which object-oriented features were later added. In some cases this addition resulted in breaking earlier features.

67.3.1 Syntax

A well-designed language starts with a simple syntax that is expressible by a grammar. Most application language definitions present their syntax as a context-free grammar; the size and complexity can vary among languages. For example, see Table 67.1 [13], which shows the number of pages of text required to present the complete grammar for each of four prominent programming languages.

From this table, it can be inferred that the languages Pascal and C are syntactically simpler than Java and C++.

However, a review of the websites for Perl, for example, reveals that Perl may not have a syntactic grammar in the traditional sense.

Moreover, its informal syntactic definition reveals that Perl distinguishes between constant strings using either single or double quotes [14]:

> Besides the backslash escapes listed above, double-quoted strings are subject to *variable interpolation* of scalar and list values. … Variable interpolation may only be done for scalar variables, entire arrays (but not hashes), single elements from an array or hash, or slices (multiple subscripts) of an array or hash.

But "variable interpolation" as described earlier is never given a precise syntax, so beginners often avoid its use altogether.

Secondly, in order to distinguish among scalar variables, arrays, and hashes, Perl requires that each of these start with a unique character. Omitting this character leads to the so-called bare word compile error, which (in our experience) is the most common error in Perl. Languages that have copied this convention, namely, PHP and Ruby, inherit the same weakness.

Since Python does not support variable interpretation of arbitrary strings, variables are not required to start with a special character. This supports Python's reputation as a simpler language than Perl.

Since these languages support the `sprintf` function of C, as well as string concatenation, *variable interpretation* of double quoted strings leads to more problems than it solves.

TABLE 67.1 Grammars for Various Languages

Language	Approximate Pages
Pascal	5
C	6
C++	22
Java	14

67.3.2 Semantics

Languages should have a well-defined semantics based on a formal semantic definition to the extent possible. However, if formal semantic definitions of scripting languages exist, we could not find them. A formal semantic definition allows alternative implementations of language, since it can be proved that the abstract models of various implementations are equivalent. These formal definitions can be based on either operational semantics or denotational semantics.

Again using Perl as an example, the implied subject of most operations is the special variable $_. A common error in Perl is to assume that each function has its own copy of this variable but, in fact, there is only a single copy in the entire program. One implication of this rule is that modules or library functions that a programmer develops must avoid using the same special variable.

67.3.3 Type System

Scripting languages are all dynamically typed. A language is defined to be *dynamically typed* if types are associated with values rather than variables; thus, a variable can have a different type at any time during program execution, depending on the value currently assigned to the variable at that time.

Since scripting languages were originally intended for programming in the small, it was felt that a script with only a screenful of code (or two) did not need the extra complexity of having to declare all its variables.

The question then becomes what do you do if one or more operands of an operator has the wrong type. With respect to automatic coercion from one type to another, a permissive language attempts to convert a value from one type to another. A language is more strict with respect to automatic coercions if it allows only those which are guaranteed to be safe.

Permissive languages such as Perl and PHP try to keep the program running if possible. Because of this they tend to allow automatic coercions of values, even when such conversions are more likely to be incorrect rather than correct. In the case of PHP, this attitude eliminates many error messages being written to the browser window. Such a practice can be defended on security grounds.

Because of this permissiveness, scripting languages try to avoid overloading operators based on the types of their operands. Perl and PHP use the dot to denote string concatenation, whereas Python overloads the plus operator for concatenation. Perl even has unique operators for string comparisons, instead of using the mathematical operators already used for numeric types.

Strict languages such as Python do not permit automatic coercions unless they are safe. For example, converting an integer to a string is guaranteed to work, although Python does not permit an automatic conversion. Converting a string to a number would be disallowed, since the string may contain arbitrary text not representing a number. For a glue language, reporting errors tends to be preferable to generating incorrect output.

67.3.4 Other Properties

A good scripting language should support the needed functions, data structures, and idioms of its intended applications. We have seen how PHP supports an HTML mode, which is quite useful in web applications.

Most of the languages we have examined support unbounded indexed lists and dictionaries. They also support iterators for traversing these structures. Whether these features lead to dramatically simpler applications is a subject of debate.

The one area where all of these languages have had problems is in maintaining backward compatibility. As the size and number of scripts written in these languages continues to grow, breaking existing scripts is likely to cause programmers to seek a more stable language.

Consider the language Python as an example. Through versions 1 and 2, Python remained largely backward compatible. However, the design goals for version 3 was that "there should be one—and preferably only one—obvious way to do it." The backward-compatibility problems relative to earlier versions are as follows:

- The `print` statement became a function, necessitating parentheses surrounding the arguments. This broke almost all scripts written for earlier versions of Python.
- The division operator was redefined to always produce a floating point number. A new operator was introduced which produced an integer when one integer was divided by another.
- Removing previous backward-compatibility features, including old-style classes, integer-truncating division, string exceptions, and implicit relative imports.
- Many changes were made to key libraries. For example, the glue script in Figure 67.3 will not run under version 3 due to changes in the `process` module. Many user libraries would not work under version 3.

Although Python version 3 came out in December 2008, only in late 2011 and early 2012 have Python textbooks started to appear in large numbers.

To summarize, as the distinction between traditional application programming languages and scripting languages continues to shrink, the principles and qualities of traditional languages should be applied more carefully to scripting languages as well.

67.4 Conclusions

Scripting languages began in the 1960s as a simple device for connecting individual tasks in a sequence of programs that ran in batches.

Throughout the 1970s and 1980s, especially with the rapid emergence of Unix and time sharing systems, scripting languages grew in richness and versatility to accommodate the needs of these systems.

With the widespread emergence of the Internet in the 1990s and recent years, newer scripting languages embodied still more features and functionality to support web-based programming requirements.

In fact, scripting languages of today, like PHP and Python, are not easily distinguishable from other modern programming languages like Java and C++. For that reason, it appears that tomorrow's scripting languages will need to adhere to a more rigorous definitional standard, in order that they maintain such important characteristics as backward compatibility, interoperability, and type safety.

References

Bynum, B. and T. Camp. After you, Alfonse: A mutual exclusion toolkit. *Proceedings of the 28th SIGCSE Technical Symposium on Computer Science Education*, Philadelphia, PA, 1996, pp. 170–174.

Bynum, W. L., R. E. Noonan, and R.H. Prosl. Using a project submission tool across the curriculum. *The Journal of Computing in Small Colleges*, 15 (5), 2000, 96–104.

C Shell, http://en.wikipedia.org/wiki/C_shell, June, 2012.

Dougherty, D. and A. Robbins. *sed & awk*. O'Reilly, 1990.

Hughes, S. *PHP Developer's Cookbook*. SAMS, 2001.

IBM, *Introduction to the New Mainframe: z/OS Basics*. IBM RedBooks, 2012.

Joy, W. *An Introduction to the C Shell*. University of California, Berkeley, CA, 1982.

Kernighan, B. W. and R. Pike. *The UNIX Programming Environment*. Prentice Hall, Englewood Cliffs, NJ, 1984.

Osterhout, J. K. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

Osterhout, J. K. Scripting: Higher level programming for the 21st century. *IEEE Computer*, 31 (3), March 1998, 23–30.

Rosenblatt, B. *Learning the Korn Shell*. O'Reilly & Associates, Sebastopol, CA, 1993.
Schwartz, R. L. *Learning Perl*. O'Reilly & Associates, Sebastopol, CA, 1993.
Tucker, A. and R. Noonan. *Programming Languages*, 2nd edn. McGraw-Hill, 2007.
Wall, L., T. Christiansen, and R. L. Schwartz. *Programming Perl*, 2nd edn. O'Reilly, Sebastopol, CA, 1996.
Web Monkey, http://www.webmonkey.com/2010/02/scripting_language.
WikiBooks, *C Shell Scripting*. http://en.wikibooks.org/wiki/C_Shell_Scripting, 2011.