13 Scripting Languages

Traditional programming languages are intended primarily for the construction of self-contained applications: programs that accept some sort of input, manipulate it in some well-understood way, and generate appropriate output. But most actual *uses* of computers require the coordination of multiple programs. A large institutional payroll system, for example, must process time-reporting data from card readers, scanned paper forms, and manual (keyboard) entry; execute thousands of database queries; enforce hundreds of legal and institutional rules; create an extensive "paper trail" for record-keeping, auditing, and tax preparation purposes; print paychecks; and communicate with servers around the world for on-line direct deposit, tax withholding, retirement accumulation, medical insurance, and so on. These tasks are likely to involve dozens or hundreds of separately executable programs. Coordination among these programs is certain to require tests and conditionals, loops, variables and types, subroutines and abstractions—the same sorts of logical tools that a conventional language provides *inside* an application.

On a much smaller scale, a graphic artist or photojournalist may routinely download pictures from a digital camera; convert them to a favorite format; rotate the pictures that were shot in vertical orientation; down-sample them to create browsable thumbnail versions; index them by date, subject, and color histogram; back them up to a remote archive; and then reinitialize the camera's memory. Performing these steps by hand is likely to be both tedious and error-prone. In a similar vein, the creation of a dynamic web page may require authentication and authorization, database lookup, image manipulation, remote communication, and the reading and writing of HTML text. All these scenarios suggest a need for programs that coordinate other programs.

It is of course possible to write coordination code in Java, C, or some other conventional language, but it isn't always easy. Conventional languages tend to stress efficiency, maintainability, portability, and the static detection of errors. Their type systems tend to be built around such hardware-level concepts as fixed-size integers, floating-point numbers, characters, and arrays. By contrast *scripting languages* tend to stress flexibility, rapid development, local customization, and

DOI: 10.1016/B978-0-12-374514-9.00023-9

dynamic (run-time) checking. Their type systems, likewise, tend to embrace such high-level concepts as tables, patterns, lists, and files.

General-purpose scripting languages like Perl and Python are sometimes called *glue languages*, because they were originally designed to "glue" existing programs together to build a larger system. With the growth of the World Wide Web, scripting languages have gained new prominence in the generation of dynamic content. They are also widely used as *extension languages*, which allow the user to customize or extend the functionality of "scriptable" tools.

We consider the history and nature of scripting in more detail in Section 13.1. We then turn in Section 13.2 to some of the problem domains in which scripting is widely used. These include command interpretation (shells), text processing and report generation, mathematics and statistics, general-purpose program coordination, and configuration and extension. In Section 13.3 we consider several forms of scripting used on the World Wide Web, including CGI scripts, server- and client-side processing of scripts embedded in web pages, Java applets, and (on the PLP CD) XSLT. Finally, in Section 13.4, we consider some of the more interesting language features, common to many scripting languages, that distinguish them from their more traditional "mainstream" cousins. We look in particular at naming, scoping, and typing; string and pattern manipulation; and high-level structured data. We will not provide a detailed introduction to any one scripting language, though we will consider concrete examples in several. As in most of this book, the emphasis will be on underlying concepts.

13.1 What Is a Scripting Language?

Modern scripting languages have two principal sets of ancestors. In one set are the command interpreters or "shells" of traditional batch and "terminal" (command-line) computing. In the other set are various tools for text processing and report generation. Examples in the first set include IBM's JCL, the MS-DOS `command` interpreter, and the Unix `sh` and `csh` shell families. Examples in the second set include IBM's RPG and Unix's `sed` and `awk`. From these evolved Rexx, IBM's "Restructured Extended Executor," which dates from 1979, and Perl, originally devised by Larry Wall in the late 1980s, and now the most widely used general-purpose scripting language. Other general-purpose scripting languages include Tcl ("tickle"), Python, Ruby, VBScript (for Windows) and AppleScript (for the Mac).

With the growth of the World Wide Web in the late 1990s, Perl was widely adopted for "server-side" web scripting, in which a web-server executes a program (on the server's machine) to generate the content of a page. One early web-scripting enthusiast was Rasmus Lerdorf, who created a collection of scripts to track access to his personal home page. Originally written in Perl but soon redesigned as a full-fledged and independent language, these scripts evolved into PHP, now the most popular platform for server-side web scripting. PHP competitors include

JSP (Java Server Pages), Ruby on Rails, and, on Microsoft platforms, VBScript. For scripting on the client computer, all major browsers implement JavaScript, a language developed by Netscape Corporation in the mid 1990s, and standardized by ECMA (the European standards body) in 1999 [ECM99].

In his classic paper on scripting [Ous98], John Ousterhout, the creator of Tcl, notes that "Scripting languages assume that a collection of useful components already exist in other languages. They are intended not for writing applications from scratch but rather for combining components." Ousterhout envisions a future in which programmers increasingly rely on scripting languages for the top-level structure of their systems, where clarity, reusability, and ease of development are crucial. Traditional "systems languages" like C, C++, or Java, he argues, will be used for self-contained, reusable system components, which emphasize complex algorithms or execution speed. As a general rule of thumb, he suggests that code

DESIGN & IMPLEMENTATION

Scripting on Microsoft platforms

As in several other aspects of computing, Microsoft tends to rely on internally developed technology in the area of scripting languages. Most of its scripting applications are based on VBScript, a dialect of Visual Basic. At the same time, Microsoft has developed a very general scripting interface (Windows Script) that is implemented uniformly by the operating system (Windows Script Host [WSH]), the web server (Active Server Pages [ASP]), and the Internet Explorer browser. A Windows Script implementation of JScript, the company's version of JavaScript, comes preinstalled on Windows machines, but languages like Perl and Python can be installed as well, and used to drive the same interface. Many other Microsoft applications, including the entire Office suite, use VBScript as an extension language, but for these the implementation framework (Visual Basic for Applications [VBA]) does not make it easy to use other languages instead.

Given Microsoft's share of the desktop computing market, VBScript is one of the most widely used scripting languages. It is almost never used on other platforms, however, while Perl, Tcl, Python, PHP, and others see significant use on Windows. For server-side web scripting, PHP currently predominates: as of August 2008, some 47% of the 175 million Internet web sites surveyed by Netcraft LTD were running the open-source Apache web server,[1] and most of the ones with active content were using PHP. Microsoft's Internet Information Server (IIS) was second to Apache, with 37% of the sites, and many of those had PHP installed as well. For client-side scripting, Internet Explorer controls barely over half of the browser market (52% as of August 2008),[2] and most website administrators need their content to be visible to the other 48%. Explorer supports JavaScript (JScript), but other browsers do not support VBScript.

[1] *news.netcraft.com/archives/web_server_survey.html*
[2] *www.w3schools.com/browsers/browsers_stats.asp*

can be developed 5 to 10 times faster in a scripting language, but will run 10 to 20 times faster in a traditional systems language.

Some authors reserve the term "scripting" for the glue languages used to coordinate multiple programs. In common usage, however, scripting is a broader and vaguer concept. It clearly includes web scripting. For most authors it also includes *extension languages.*

Many readers will be familiar with the Visual Basic "macros" of Microsoft Office and related applications. Others may be familiar with the Lisp-based extension language of the `emacs` text editor. Several languages, including Tcl, Rexx, Python, and the Guile and Elk dialects of Scheme, have implementations designed in such a way that they can be incorporated into a larger program and used to extend its features. Extension was in fact the original purpose of Tcl. In a similar vein, several widely used commercial applications provide their own proprietary extension languages. For graphical user interface (GUI) programming, the Tk toolkit, originally designed for use with Tcl, has been incorporated into several scripting languages, including Perl, Python, and Ruby.

One can also view XSLT (extensible stylesheet language transformations) as a scripting language, albeit somewhat different from the others considered in this chapter. XSLT is part of the growing family of XML (extensible markup language) tools. We consider it further in Section 13.3.5.

13.1.1 Common Characteristics

While it is difficult to define scripting languages precisely, there are several characteristics that they tend to have in common.

Both batch and interactive use. A few scripting languages (notably Perl) have a compiler that insists on reading the entire source program before it produces any output. Most other languages, however, are willing to compile or interpret their input line by line. Rexx, Python, Tcl, Guile, and (with a short helper script) Ruby will all accept commands from the keyboard.

Economy of expression. To support both rapid development and interactive use, scripting languages tend to require a minimum of "boilerplate." Some make heavy use of punctuation and very short identifiers (Perl is notorious for this), while others (e.g., Rexx, Tcl, and AppleScript) tend to be more "English-like," with lots of words and not much punctuation. All attempt to avoid the extensive declarations and top-level structure common to conventional languages. Where a trivial program looks like this in Java,

EXAMPLE 13.1
Trivial programs in conventional and scripting languages

```
class Hello {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

and like this in Ada,

```
with ada.text_IO; use ada.text_IO;
procedure hello is
begin
    put_line("Hello, world!");
end hello;
```

in Perl, Python, or Ruby it is simply

```
print "Hello, world!\n"
```

Lack of declarations; simple scoping rules. Most scripting languages dispense with declarations, and provide simple rules to govern the scope of names. In some languages (e.g., Perl) everything is global by default; optional declarations can be used to limit a variable to a nested scope. In other languages (e.g., PHP and Tcl), everything is local by default; globals must be explicitly imported. Python adopts the interesting rule that any variable that is assigned a value is local to the block in which the assignment appears. Special syntax is required to assign to a variable in a surrounding scope.

Flexible dynamic typing. In keeping with the lack of declarations, most scripting languages are dynamically typed. In some (e.g., PHP, Python, Ruby, and Scheme), the type of a variable is checked immediately prior to use. In others (e.g., Rexx, Perl, and Tcl), a variable will be interpreted differently in different contexts. In Perl, for example, the program

EXAMPLE 13.2
Coercion in Perl

```
$a = "4";
print $a . 3 . "\n";    # '.' is concatenation
print $a + 3 . "\n";    # '+' is addition
```

will print

```
43
7
```

DESIGN & IMPLEMENTATION

Compiling interpreted languages

Several times in this chapter we will make reference to "the compiler" for a scripting language. As we saw in Examples 1.9 and 1.10, interpreters almost never work with source code; a front-end translator first replaces that source with some sort of intermediate form. For most implementations of most of the languages described in this chapter, the front end is sufficiently complex to deserve the name "compiler." Intermediate forms are typically internal data structures (e.g., a syntax tree) or "byte-code" representations reminiscent of those of Java.

This contextual interpretation is similar to coercion, except that there isn't necessarily a notion of "natural" type from which an object must be converted; the various possible interpretations may all be equally "natural." We shall have more to say about context in Perl in Section 13.4.3. ■

Easy access to system facilities. Most programming languages provide a way to ask the underlying operating system to run another program, or to perform some operation directly. In scripting languages, however, these requests are much more fundamental, and have much more direct support. Perl, for one, provides well over 100 built-in commands that access operating system functions for input and output, file and directory manipulation, process management, database access, sockets, interprocess communication and synchronization, protection and authorization, time-of-day clock, and network communication. These built-in commands are generally a good bit easier to use than corresponding library calls in languages like C.

Sophisticated pattern-matching and string manipulation. In keeping with their text processing and report generation ancestry, and to facilitate the manipulation of textual input and output for external programs, scripting languages tend to have extraordinarily rich facilities for pattern matching, search, and string manipulation. Typically these are based on *extended regular expressions.* We discuss them further in Section 13.4.2.

High-level data types. High-level data types like sets, bags, dictionaries, lists, and tuples are increasingly common in the standard library packages of conventional programming languages. A few languages (notably C++) allow users to redefine standard infix operators to make these types as easy to use as more primitive, hardware-centric types. Scripting languages go one step further by

DESIGN & IMPLEMENTATION

Canonical implementations

Because they are implemented with interpreters, scripting languages tend to be easy to port from one machine to another—substantially easier than compilers for which one must write a new code generator. Given a native compiler for the language in which the interpreter is written, the only difficult part (and it may indeed be difficult) is to implement any necessary modifications to the part of the interpreter that provides the interface to the operating system.

At the same time, the ease of porting an interpreter means that several scripting languages, including Perl, Python, Tcl, and Ruby, have a single widely used implementation, which serves as the de facto language definition. Reading a book on Perl, it can be difficult to tell how a subtle program will behave. When in doubt, one may need to "try it out." Rexx and JavaScript appear to be unique among widely used scripting languages in having a formal definition codified by an international standards body and independent of any one implementation. (`Sed`, `awk`, and `sh` have also been standardized by POSIX [Int03b], but none of these has the complexity of Perl, Python, Tcl, or Ruby.)

building high-level types into the syntax and semantics of the language itself. In most scripting languages, for example, it is commonplace to have an "array" that is indexed by character strings, with an underlying implementation based on hash tables. Storage is invariably garbage collected.

Much of the most rapid change in programming languages today is occurring in scripting languages. This can be attributed to several causes, including the continued growth of the web, the dynamism of the open-source community, and the comparatively low investment required to create a new scripting language. Where a compiled, industrial-quality language like Java or C# requires a multiyear investment by a very large programming team, a single talented designer, working alone, can create a usable implementation of a new scripting language in only a year or two.

Due in part to this rapid change, newer scripting languages have been able to incorporate some of the most innovative concepts in language design. Ruby, for example, has a uniform object model (much like Smalltalk), true iterators (like Clu), array slices (like Fortran 90), structured exception handling, multiway assignment, and reflection. Python also provides several of these features, together with anonymous first-class functions and Haskell-like list comprehensions.

13.2 Problem Domains

Some general-purpose languages—Scheme and Visual Basic in particular—are widely used for scripting. Conversely, some scripting languages, including Perl, Python, and Ruby, are intended by their designers for general-purpose use, with features intended to support "programming in the large": modules, separate compilation, reflection, program development environments, and so on. For the most part, however, scripting languages tend to see their principal use in well-defined problem domains. We consider some of these in the following subsections.

13.2.1 Shell (Command) Languages

In the days of punch-card computing, simple command languages allowed the user to "script" the processing of a card deck. A control card at the front of the deck, for example, might indicate that the upcoming cards represented a program to be compiled, or perhaps machine language for the compiler itself, or input for a program already compiled and stored on disk. A control card embedded later in the deck might test the exit status of the most recently executed program and choose what to do next based on whether that program completed successfully. Given the linear nature of a card deck, however (one can't in general back up), command languages for batch processing tend not to be very sophisticated. JCL, for example, has no iteration constructs.

With the development of interactive timesharing in the 1960s and early 1970s, command languages became much more sophisticated. Louis Pouzin wrote a simple command interpreter for CTSS, the Compatible Time Sharing System at MIT, in 1963 and 1964. When work began on the groundbreaking Multics system in 1964, Pouzin sketched the design of an extended command language, with quoting and argument-passing mechanisms, for which he coined the term "shell." The subsequent implementation served as inspiration for Ken Thompson in the design of the original Unix shell in 1973. In the mid-1970s, Stephen Bourne and John Mashey separately extended the Thompson shell with control flow and variables; Bourne's design was adopted as the Unix standard, taking the place (and the name) of the Thompson shell, `sh`.

In the late 1970s Bill Joy developed the so-called "C shell" (`csh`), inspired at least in part by Mashey's syntax, and introducing significant enhancements for interactive use, including history, aliases, and job control. The `tcsh` version of `csh` adds command-line editing and command completion. David Korn incorporated these mechanisms into a direct descendant of the Bourne shell, `ksh`, which is very similar to the standard POSIX shell [Int03b]. The popular "Bourne-again" shell, `bash`, is an open-source version of `ksh`. While `tcsh` is still popular in some quarters, `ksh`/`bash`/POSIX `sh` is substantially better for writing shell scripts, and comparable for interactive use.

In addition to features designed for interactive use, which we will not consider further here, shell languages provide a wealth of mechanisms to manipulate filenames, arguments, and commands, and to glue together other programs. Most of these features are retained by more general scripting languages. We consider a few of them here, using `bash` syntax. The discussion is of necessity heavily simplified; full details can be found in the `bash` `man` page, or in various on-line tutorials.

Filename and Variable Expansion

EXAMPLE 13.3
"Wildcards" and "globbing"

Most users of a Unix shell are familiar with "wildcard" expansion of filenames. The following command will list all files in the current directory whose names end in `.pdf`:

```
ls *.pdf
```

The shell expands the pattern `*.pdf` into a list of all matching names. If there are three of them (say `fig1.pdf`, `fig2.pdf`, and `fig3.pdf`), the result is equivalent to

```
ls fig1.pdf fig2.pdf fig3.pdf
```

Filename expansion is sometimes called "globbing," after the original Unix `glob` command that implemented it. In addition to `*` wildcards, one can usually specify "don't care" or alternative characters or substrings. The pattern `fig?.pdf` will match (expand to) any file(s) with a single character between the `g` and the dot. The pattern `fig[0-9].pdf` will require that character to be a digit. The pattern `fig3.{eps,pdf}` will match both `fig3.eps` and `fig3.pdf`. ■

EXAMPLE 13.4
`For` loops in the shell

Filename expansion is particularly useful in loops. Such loops may be typed directly from the keyboard, or embedded in scripts intended for later execution. Suppose, for example, that we wish to create PDF versions of all our EPS figures:[3]

```
for fig in *.eps
do
    ps2pdf $fig
done
```

The `for` construct arranges for the shell variable `fig` to take on the names in the expansion of `*.eps`, one at a time, in consecutive iterations of the loop. The dollar sign in line 3 causes the value of `fig` to be expanded into the `ps2pdf` command before it is executed. (Interestingly, `ps2pdf` is itself a shell script that calls the `gs` Postscript interpreter.) Optional braces can be used to separate a variable name from following characters, as in `cp $foo ${foo}_backup`. ■

EXAMPLE 13.5
A whole loop on one line

Multiple commands can be entered on a single line if they are separated by semicolons. The following, for example, is equivalent to the loop in the previous example:

```
for fig in *.eps; do ps2pdf $fig; done
```
■

Tests, Queries, and Conditions

EXAMPLE 13.6
Conditional tests in the shell

The loop above will execute `ps2pdf` for every EPS file in the current directory. Suppose, however, that we already have some PDF files, and only want to create the ones that are missing.

```
for fig in *.eps
do
    target=${fig%.eps}.pdf
    if [ $fig -nt $target ]
    then
        ps2pdf $fig
    fi
done
```

The third line of this script is a variable assignment. The expression `${fig%.eps}` within the right-hand side expands to the value of `fig` with any trailing `.eps` removed. Similar special expansions can be used to test or modify the value of a

3 Postscript is a programming language developed at Adobe Systems, Inc. for the description of images and documents (we consider it again in the sidebar on page 738). Encapsulated Postscript (EPS) is a restricted form of Postscript intended for figures that are to be embedded in other documents. Portable Document Format (PDF, also by Adobe) is a self-contained file format that combines a subset of Postscript with font embedding and compression mechanisms. It is strictly less powerful than Postscript from a computational perspective, but much more portable, and faster and easier to render.

variable in many different ways. The square brackets in line 4 delimit a conditional test. The `-nt` operator checks to see whether the file named by its left operand is newer than the file named by its right operand (or if the left operand exists but the right does not). Similar *file query* operators can be used to check many other properties of files. Additional operators can be used for arithmetic or string comparisons. ■

Pipes and Redirection

EXAMPLE 13.7
Pipes

One of the principal innovations of Unix was the ability to chain commands together, "piping" the output of one to the input of the next. Like most shells, `bash` uses the vertical bar character (|) to indicate a pipe. To count the number of figures in our directory, without distinguishing between EPS and PDF versions, we might type

```
for fig in *; do echo ${fig%.*}; done | sort -u | wc -l
```

Here the first command, a `for` loop, prints the names of all files with extensions (dot-suffixes) removed. The `echo` command inside the loop simply prints its arguments. The `sort -u` command after the loop removes duplicates, and the `wc -l` command counts lines. ■

EXAMPLE 13.8
Output redirection

Like most shells, `bash` also allows output to be directed to a file, or input read from a file. To create a list of figures, we might type

```
for fig in *; do echo ${fig%.*}; done | sort -u > all_figs
```

DESIGN & IMPLEMENTATION

Built-in commands in the shell

Commands in the shell generally take the form of a sequence of *words*, the first of which is the name of the command. Most commands are executable programs, found in directories on the shell's *search path*. A large number, however (about 50 in `bash`), are *built-ins*—commands that the shell recognizes and executes itself, rather than starting an external program. Interestingly, several commands that are available as separate programs are duplicated as built-ins, either for the sake of efficiency or to provide additional semantics. Conditional tests, for example, were originally supported by the external `test` command (for which square brackets are syntactic sugar), but these occur sufficiently often in scripts that execution speed improved significantly when a built-in version was added. By contrast, while the `kill` command is not used very often, the built-in version allows processes to be identified by small integer or symbolic names from the shell's *job control* mechanism. The external version supports only the longer and comparatively unintuitive process identifiers supplied by the operating system.

The “greater than” sign indicates output redirection. If doubled (`sort -u >> all_figs`) it causes output to be appended to the specified file, rather than overwriting the previous contents.

In a similar vein, the “less than” sign indicates input redirection. Suppose we want to print our list of figures all on one line, separated by spaces, instead of on multiple lines. On a Unix system we can type

```
tr '\n' ' ' < all_figs
```

This invocation of the standard `tr` command converts all newline characters to spaces. Because `tr` was written as a simple filter, it does not accept a list of files on the command line; it only reads standard input. ■

For any executing Unix program, the operating system keeps track of a list of open files. By convention, *standard input* and *standard output* (`stdin` and `stdout`) are files numbers 0 and 1. File number 2 is by convention *standard error* (`stderr`), to which programs are supposed to print diagnostic error messages. One of the advantages of the `sh` family of shells over the `csh` family is the ability to redirect `stderr` and other open files independent of `stdin` and `stdout`.

EXAMPLE 13.9
Redirection of `stderr` and `stdout`

Consider, for example, the `ps2pdf` script. Under normal circumstances this script works silently. If it encounters an error, however, it prints a message to `stdout` and quits. This violation of convention (the message should go to `stderr`) is harmless when the command is invoked from the keyboard. If it is embedded in a script, however, and the output of the script is directed to a file, the error message may end up in the file instead of on the screen, and go unnoticed by the user. With `bash` we can type

```
ps2pdf my_fig.eps 1>&2
```

Here `1>&2` means “make `ps2pdf` send file 1 (`stdout`) to the same place that the surrounding context would normally send file 2 (`stderr`).” ■

EXAMPLE 13.10
Heredocs (in-line input)

Finally, like most shells, `bash` allows the user to provide the input to a command in-line:

```
tr '\n' ' ' <<END
list
of
input
lines
END
```

The `<<END` indicates that subsequent input lines, up to a line containing only `END`, are to be supplied as input to `tr`. Such in-line input (traditionally called a “here document”) is seldom used interactively, but is highly useful in shell scripts. ■

Quoting and Expansion

EXAMPLE 13.11

Single and double quotes

Shells typically provide several *quoting* mechanisms to group words together into strings. Single (forward) quotes inhibit filename and variable expansion in the quoted text, and cause it to be treated as a single word, even if it contains white space. Double quotes also cause the contents to be treated as a single word, but do not inhibit expansion. Thus

```
foo=bar
single='$foo'
double="$foo"
echo $single $double
```

will print "`$foo bar`". ■

EXAMPLE 13.12

Subshells

Several other bracketing constructs in `bash` group the text inside, for various purposes. Command lists enclosed in parentheses are passed to a subshell for evaluation. If the opening parenthesis is preceded by a dollar sign, the output of the nested command list is expanded into the surrounding context:

```
for fig in $(cat my_figs); do ps2pdf ${fig}.eps; done
```

Here `cat` is the standard command to print the content of a file. Most shells use backward single quotes for the same purpose (`` `cat my_figs` ``); `bash` supports this syntax as well, for backward compatibility. ■

EXAMPLE 13.13

Brace-quoted blocks in the shell

Command lists enclosed in braces are treated by `bash` as a single unit. They can be used, for example, to redirect the output of a sequence of commands:

```
{ date; ls; } >> file_list
```

Unlike parenthesized lists, commands enclosed in braces are executed by the current shell. From a programming languages perspective, parentheses and braces behave "backward" from the way they do in C: parentheses introduce a nested dynamic scope in `bash`, while braces are purely for grouping. In particular, variables that are assigned new values within a parenthesized command list will revert to their previous values once the list has completed execution. ■

EXAMPLE 13.14

Pattern-based list generation

When not surrounded by white space, braces perform pattern-based list generation, in a manner similar to filename expansion, but without the connection to the file system. For example, `echo abc{12,34,56} xyz` prints `abc12xyz abc34xyz abc56xyz`. Also, as we have seen, braces serve to delimit variable names when the opening brace is preceded by a dollar sign. ■

In Example 13.6 we used square brackets to enclose a conditional expression. Double square brackets serve a similar purpose, but with more C-like expression syntax, and without filename expansion. Double parentheses are used to enclose arithmetic computations, again with C-like syntax.

The interpolation of commands in `$()` or backquotes, patterns in `{}`, and arithmetic expressions in `(())` are all considered forms of expansion, analogous to filename expansion and variable expansion. The splitting of strings into words is also considered a form of expansion, as is the replacement, in certain contexts, of tilde (`~`) characters with the name of the user's home directory. All told, these give us seven different kinds of expansion in `bash`.

All of the various bracketing constructs have rules governing which kinds of expansion are performed within. The rules are intended to be as intuitive as possible, but they are not uniform across constructs. Filename expansion, for example, does not occur within `[[]]`-bracketed conditions. Similarly, a double quote character may appear inside a double-quoted string if escaped with a backslash, but a single-quote character may not appear inside a single-quoted string.

Functions

EXAMPLE 13.15
User-defined shell functions

Users can define functions in `bash` that then work like built-in commands. Many users, for example, define `ll` as a shortcut for `ls -l`, which lists files in the current directory in "long format."

```
function ll () {
    ls -l "$@"
}
```

Within the function, `$1` represents the first parameter, `$2` represents the second, and so on. In the definition of `ll`, `$@` represents the entire parameter list. Functions can be arbitrarily complex. In particular, `bash` supports both local variables and recursion. Shells in the `csh` family provide a more primitive `alias` mechanism that works via macro expansion. ■

The `#!` Convention

EXAMPLE 13.16
The `#!` convention in script files

As noted above, shell commands can be read from a *script* file. To execute them in the current shell, one uses the "dot" command:

```
. my_script
```

where `my_script` is the name of the file. Many operating systems, including most versions of Unix, allow one to make a script function as an executable program, so that users can simply type

```
my_script
```

Two steps are required. First, the file must be marked *executable* in the eyes of the operating system. On Unix one types `chmod +x my_script`. Second, the file must be self-descriptive in a way that allows the operating system to tell which shell (or other interpreter) will understand the contents. Under Unix, the file must begin with the characters `#!`, followed by the name of the shell. The typical `bash` script thus begins with

```
#!/bin/bash
```

Specifying the full path name is a safety feature: it anticipates the possibility that the user may have a search path for commands on which some other program named `bash` appears before the shell. (Unfortunately, the requirement for full path names makes `#!` lines nonportable, since shells and other interpreters may be installed in different places on different machines.) ■

✓ CHECK YOUR UNDERSTANDING

1. Give a plausible one-sentence definition of "scripting language."
2. List the principal ways in which scripting languages differ from conventional "systems" languages.
3. From what two principal sets of ancestors are modern scripting languages descended?
4. What IBM creation is generally considered the first general-purpose scripting language?
5. What is the most popular language for server-side web scripting?
6. How does the notion of *context* in Perl differ from coercion?
7. What is *globbing*? What is a *wildcard*?
8. What is a *pipe* in Unix? What is *redirection*?
9. Describe the three standard I/O streams provided to every Unix process.
10. Explain the significance of the `#!` convention in Unix shell scripts.

DESIGN & IMPLEMENTATION

Magic numbers

When the Unix kernel is asked to execute a file (via the `execve` system call), it checks the first few bytes of the file for a "magic number" that indicates the file's type. Some values correspond to directly executable object file formats. Under Linux, for example, the first four bytes of an object file are `0x7f45_4c46` (⟨`del`⟩`ELF` in ASCII). Under Mac OS X they are `0xfeed_face`. If the first two bytes are `0x2321` (`#!` in ASCII), the kernel assumes that the file is a script, and reads subsequent characters to find the name of the interpreter.

The `#!` convention in Unix is the main reason that most scripting languages use `#` as the opening comment delimiter. Early versions of `sh` used the no-op command (`:`) as a way to introduce comments. Joy's C shell introduced `#`, whereupon some versions of `sh` were modified to launch `csh` when asked to execute a script that appeared to begin with a C shell comment. This mechanism evolved into the more general mechanism used in many (though not all) variants of Unix today.

```
# label (target for branch):
:top
/<[hH][123]>.*<\/[hH][123]>/ {      ;# match whole heading
    h                               ;# save copy of pattern space
    s/\(<\/[hH][123]>\).*$/\1/      ;# delete text after closing tag
    s/^.*\(<[hH][123]>\)/\1/        ;# delete text before opening tag
    p                               ;# print what remains
    g                               ;# retrieve saved pattern space
    s/<\/[hH][123]>//               ;# delete closing tag
    b top
}                                   ;# and branch to top of script
/<[hH][123]>/ {                     ;# match opening tag (only)
    N                               ;# extend search to next line
    b top
}                                   ;# and branch to top of script
d                                   ;# if no match at all, delete
```

Figure 13.1 Script in sed to extract headers from an HTML file. The script assumes that opening and closing tags are properly matched, and that headers do not nest.

13.2.2 Text Processing and Report Generation

Shell languages tend to be heavily string-oriented. Commands are strings, parsed into lists of words. Variables are string-valued. Variable expansion mechanisms allow the user to extract prefixes, suffixes, or arbitrary substrings. Concatenation is indicated by simple juxtaposition. There are elaborate quoting conventions. Few more conventional languages have similar support for strings.

At the same time, shell languages are clearly not intended for the sort of text manipulation commonly performed in editors like emacs or vi. Search and substitution, in particular, are missing, and many other tasks that editors accomplish with a single keystroke—insertion, deletion, replacement, bracket matching, forward and backward motion—would be awkward to implement, or simply make no sense, in the context of the shell. For repetitive text manipulation it is natural to want to automate the editing process. Tools to accomplish this task constitute the second principal class of ancestors for modern scripting languages.

Sed

EXAMPLE 13.17
Extracting HTML headers with sed

As a simple text processing example, consider the problem of extracting all headers from a web page (an HTML file). These are strings delimited by <H1> ... </H1>, <H2> ... </H2>, and <H3> ... </H3> tags. Accomplishing this task in an editor like emacs, vi, or even Microsoft Word is straightforward but tedious: one must search for an opening tag, delete preceding text, search for a closing tag, mark the current position (as the starting point for the next deletion), and repeat. A program to perform these tasks in sed, the Unix "stream editor," appears in Figure 13.1. The code consists of a label and three commands, the first two of which are compound. The first compound command prints the first header, if any, found in the portion of the input currently being examined (what sed calls the *pattern space*). The second compound command appends a new line to the pattern space whenever it already contains a header-opening tag. Both compound commands, and several of the subcommands, use regular expression patterns, delimited by slashes. We will discuss these patterns further in Section 13.4.2. The third command (the lone d) simply deletes the current line. Because each compound command ends with a branch back to the top of the script, the second will execute only if the first does not, and the delete will execute only if neither compound does. ■

The editor heritage of sed is clear in this example. Commands are generally one character long, and there are no variables—no state of any kind beyond the program counter and text that is being edited. These limitations make sed best suited to "one-line programs," typically entered verbatim from the keyboard with the -e command-line switch. The following, for example, will read from standard input, delete blank lines, and (implicitly) print the nonblank lines to standard output:

EXAMPLE 13.18
One-line scripts in sed

```
sed -e'/^[[:space:]]*$/d'
```

Here ^ represents the beginning of the line and $ represents the end. The [[:space:]] expression matches any white-space character in the local character set, to be repeated an arbitrary number of times, as indicated by the Kleene star (*). The d indicates deletion. Nondeleted lines are printed by default. ■

Awk

In an attempt to address the limitations of sed, Alfred Aho, Peter Weinberger, and Brian Kernighan designed awk in 1977 (the name is based on the initial letters of their last names). Awk is in some sense an evolutionary link between stream editors like sed and full-fledged scripting languages. It retains sed's line-at-a-time filter model of computation, but allows the user to escape this model when desired, and replaces single-character editing commands with syntax reminiscent of C. Awk provides (typeless) variables and a variety of control-flow constructs, including subroutines.

An awk program consists of a sequence of *patterns*, each of which has an associated *action*. For every line of input, the interpreter executes, in order, the actions whose patterns evaluate to true. An example with a single pattern-action pair appears in Figure 13.2. It performs essentially the same task as the sed script of Figure 13.1. Lines that contain no opening tag are ignored. In a line with an opening tag, we delete any text that precedes the header. We then print lines until we find the closing tag, and repeat if there is another opening tag on the same line. We fall back into the interpreter's main loop when we're cleanly outside any header.

EXAMPLE 13.19
Extracting HTML headers with awk

Several conventions can be seen in this example. The current input line is available in the pseudovariable $0. The getline function reads into this variable by

```
/<[hH][123]>/ {
    # execute this block if line contains an opening tag
    do {
        open_tag = match($0, /<[hH][123]>/)
        $0 = substr($0, open_tag)      # delete text before opening tag
                                       # $0 is the current input line
        while (!/<\/[hH][123]>/) {     # print interior lines
            print                      #    in their entirety
            if (getline != 1) exit
        }
        close_tag = match($0, /<\/[hH][123]>/) + 4

        print substr($0, 0, close_tag) # print through closing tag
        $0 = substr($0, close_tag + 1) # delete through closing tag
    } while (/<[hH][123]>/)            # repeat if more opening tags
}
```

Figure 13.2 **Script in awk to extract headers from an HTML file.** Unlike the sed script, this version prints interior lines incrementally. It again assumes that the input is well formed.

default. The substr(s, a, b) function extracts the portion of string s starting at position a and with length b. If b is omitted, the extracted portion runs to the end of s. Conditions, like patterns, can use regular expressions; we can see an example in the do ... while loop. By default, regular expressions match against $0. ■

Perhaps the two most important innovations of awk are *fields* and *associative arrays*, neither of which appears in Figure 13.2. Like the shell, awk parses each input line into a series of words (fields). By default these are delimited by white space, though the user can change this behavior dynamically by assigning a regular expression to the built-in variable FS (field separator). The fields of the current input line are available in the pseudovariables $1, $2, ... The built-in variable NR gives the total number of fields. Awk is frequently used for field-based one-line programs. The following, for example, will print the second word of every line of standard input:

EXAMPLE 13.20
Fields in awk

```
awk '{print $2}'
```
■

EXAMPLE 13.21
Capitalizing a title in awk

Associative arrays will be considered in more detail in Section 13.4.3. Briefly, they combine the functionality of hash tables with the syntax of arrays. We can illustrate both fields and associative arrays with an example script (Figure 13.3) that capitalizes each line of its input as if it were a title. The script declines to modify "noise" words (articles, conjunctions, and short prepositions) unless they are the first word of the title or of a subtitle, where a subtitle follows a word ending with a colon or a dash. The script also declines to modify words in which any letter other than the first is already capitalized. ■

```
BEGIN {                          # "noise" words
    nw["a"] = 1;    nw["an"] = 1;   nw["and"] = 1;  nw["but"] = 1
    nw["by"] = 1;   nw["for"] = 1;  nw["from"] = 1; nw["in"] = 1
    nw["into"] = 1; nw["nor"] = 1;  nw["of"] = 1;   nw["on"] = 1
    nw["or"] = 1;   nw["over"] = 1; nw["the"] = 1;  nw["to"] = 1
    nw["via"] = 1;  nw["with"] = 1
}
{
    for (i=1; i <= NF; i++) {
        if ((!nw[$i] || i == 0 || $(i-1)~/[:-]$/) && ($i !~/.+[A-Z]/)){
            # capitalize
            $i = toupper(substr($i, 1, 1)) substr($i, 2)
        }
        printf $i " ";      # don't add trailing line feed
    }
    printf "\n";
}
```

Figure 13.3 Script in `awk` to capitalize a title. The `BEGIN` block is executed before reading any input lines. The main block has no explicit pattern, so it is applied to every input line.

Perl

Perl was originally developed by Larry Wall in 1987, while he was working at the National Security Agency. The original version was, to first approximation, an attempt to combine the best features of `sed`, `awk`, and `sh`. It was a Unix-only tool, meant primarily for text processing (the name stands for "practical extraction and report language"). Over the years Perl has grown into a large and complex language, with an enormous user community. Though it is hard to judge such things, Perl is almost certainly the most popular and widely used scripting language. It is also fast enough for much general-purpose use, and includes separate compilation, modularization, and dynamic library mechanisms appropriate for large-scale projects. It has been ported to almost every known operating system.

Perl consists of a relatively simple language core, augmented with an enormous number of built-in library functions and an equally enormous number of shortcuts and special cases. A hint at this richness of expression can be found in the standard language reference [WCO00, p. 622], which lists (only) the 97 built-in functions "whose behavior varies the most across platforms." The cover of the book is emblazoned with the language motto: "There's more than one way to do it."

EXAMPLE 13.22
Extracting HTML headers with Perl

We will return to Perl several times in this chapter, notably in Sections 13.2.4 and 13.4. For the moment we content ourselves with a simple text-processing example, again to extract headers from an HTML file (Figure 13.4). We can see several Perl shortcuts in this figure, most of which help to make the code shorter than the equivalent programs in `sed` (Figure 13.1) and `awk` (Figure 13.2). Angle brackets (`<>`) are the "readline" operator, used for text file input. Normally they surround a *file handle* variable name, but as a special case, empty angle brackets

```
while (<>) {                                    # iterate over lines of input
    next if !/<[hH][123]>/;                     # jump to next iteration
    while (!/<\/[hH][123]>/) { $_ .= <>; }      # append next line to $_
    s/.*?([hH][123]>.*?<\/[hH][123]>)//s;
         # perform minimal matching; capture parenthesized expression in $1
    print $1, "\n";
    redo unless eof;        # continue without reading next line of input
}
```

Figure 13.4 Script in Perl to extract headers from an HTML file. For simplicity we have again adopted the strategy of buffering entire headers, rather than printing them incrementally.

generate as input the concatenation of all files specified on the command line when the script was first invoked (or standard input, if there were no such files). When a readline operator appears by itself in the control expression of a `while` loop (but nowhere else in the language), it generates its input a line at a time into the pseudovariable `$_`. Several other operators work on `$_` by default. Regular expressions, for example, can be used to search within arbitrary strings, but when none is specified, `$_` is assumed.

The `next` statement is similar to `continue` in C or Fortran: it jumps to the bottom of the innermost loop and begins the next iteration. The `redo` statement also skips the remainder of the current iteration, but returns to the top of the loop, *without* reevaluating the control expression. In our example program, `redo` allows us to append additional input to the current line, rather than reading a new line. Because end-of-file is normally detected by an undefined return value from `<>`, and because that failure will happen only once per file, we must explicitly test for `eof` when using `redo` here. Note that `if` and its symmetric opposite, `unless`, can be used as either a prefix or a postfix test.

Readers familiar with Perl may have noticed two subtle but key innovations in the substitution command of line 4 of the script. First, where the expression `.*` (in `sed`, `awk`, and Perl) matches the longest possible string of characters that permits subsequent portions of the match to succeed, the expression `.*?` in Perl matches the *shortest* possible such string. This distinction allows us to easily isolate the first header in a given line. Second, much as `sed` allows later portions of a regular expression to refer back to earlier, parenthesized portions (line 4 of Figure 13.1), Perl allows such *captured* strings to be used *outside* the regular expression. We have leveraged this feature to print matched headers in line 6 of Figure 13.4. In general, the regular expressions of Perl are significantly more powerful than those of `sed` and `awk`; we will return to this subject in more detail in Section 13.4.2. ■

13.2.3 Mathematics and Statistics

As we noted in our discussions of `sed` and `awk`, one of the distinguishing characteristics of text processing and report generation is the frequent use of "one-line programs" and other simple scripts. Anyone who owns a programmable calculator realizes that similar needs arise in mathematics and statistics. And just as shell and report generation tools have evolved into powerful languages for general-purpose computing, so too have notations and tools for mathematical and statistical computing.

In Section 7.4.1 we mentioned APL, one of the more unusual languages of the 1960s. Originally conceived as a pen-and-paper notation for teaching applied mathematics, APL retained its emphasis on the concise, elegant expression of mathematical algorithms when it evolved into a programming language. Though it lacks both easy access to other programs and sophisticated string manipulation, APL displays all the other characteristics of scripting described in Section 13.1.1, and one sometimes finds it listed as a scripting language.

The modern successors to APL include a trio of commercial packages for mathematical computing: Maple, Mathematica, and Matlab. Though their design philosophies differ, each provides extensive support for numerical methods, symbolic mathematics (formula manipulation), data visualization, and mathematical modeling. All three provide powerful scripting languages, with a heavy orientation toward scientific and engineering applications.

As the "3 Ms" are to mathematical computing, so the S and R languages are to statistical computing. Originally developed at Bell Labs by John Chambers and colleagues in the late 1970s, S is a commercial package widely used in the statistics community and in quantitative branches of the social and behavioral sciences. R is an open-source alternative to S that is largely though not entirely compatible with its commercial cousin. Among other things, R supports multidimensional array and list types, array slice operations, user-defined infix operators, call-by-need parameters, first-class functions, and unlimited extent.

13.2.4 "Glue" Languages and General-Purpose Scripting

From their text-processing ancestors, scripting languages inherit a rich set of pattern matching and string manipulation mechanisms. From command interpreter shells they inherit a wide variety of additional features including simple syntax; flexible typing; easy creation and management of subprograms, with I/O redirection and access to completion status; file queries; easy interactive and file-based I/O; easy access to command-line arguments, environment strings, process identifiers, time-of-day clock, and so on; and automatic interpreter start-up (the `#!` convention). As noted in Section 13.1.1, many scripting languages have interpreters that will accept commands interactively.

EXAMPLE 13.23
"Force quit" script in Perl

The combination of shell- and text-processing mechanisms allows a scripting language to prepare input to, and parse output from, subsidiary processes. As a simple example, consider the (Unix-specific) "force quit" Perl script shown in Figure 13.5. Invoked with a regular expression as argument, the script identifies all of the user's currently running processes whose name, process id, or command-line arguments match that regular expression. It prints the information for each, and prompts the user for an indication of whether the process should be killed.

```
$#ARGV == 0 || die "usage: $0 pattern\n";
open(PS, "ps -w -w -x -o'pid,command' |");  # 'process status' command
<PS>;                                        # discard header line
while (PS>) {
    @words = split;             # parse line into space-separated words
    if (/$ARGV[0]/i && $words[0] ne $$) {
        chomp;                  # delete trailing newline
        print;
        do {
            print "? ";
            $answer = <STDIN>;
        } until $answer =~ /^[yn]/i;
        if ($answer =~ /^y/i) {
            kill 9, $words[0];  # signal 9 in Unix is always fatal
            sleep 1;            # wait for 'kill' to take effect
            die "unsuccessful; sorry\n" if kill 0, $words[0];
        }                       # kill 0 tests for process existence
    }
}
```

Figure 13.5 Script in Perl to "force quit" errant processes. Perl's text processing features allow us to parse the output of `ps`, rather than filtering it through an external tool like `sed` or `awk`.

The second line of the code starts a subsidiary process to execute the Unix ps command. The command-line arguments cause ps to print the process id and name of all processes owned by the current user, together with their full command-line arguments. The pipe symbol (|) at the end of the command indicates that the output of ps is to be fed to the script through the `PS` file handle. The main `while` loop then iterates over the lines of this output. Within the loop, the `if` condition matches each line against `$ARGV[0]`, the regular expression provided on the script's command line. It also compares the first word of the line (the process id) against `$$`, the id of the Perl interpreter currently running the script.

Scalar variables (which in Perl include strings) begin with a dollar sign (`$`). Arrays begin with an at sign (`@`). In the first line of the `while` loop in Figure 13.5, the input line (`$_`, implicitly) is split into space-separated words, which are then assigned into the array `@words`. In the following line, `$words[0]` refers to the first element of this array, a scalar. A single variable name may have different values when interpreted as a scalar, an array, a hash table, a subroutine, or a file handle. The choice of interpretation depends on the leading punctuation mark and on the *context* in which the name appears. We shall have more to say about context in Perl in Section 13.4.3. ■

Beyond the combination of shell- and text-processing mechanisms, the typical glue language provides an extensive library of built-in operations to access features of the underlying operating system, including files, directories, and I/O; processes and process groups; protection and authorization; interprocess communication and synchronization; timing and signals; and sockets, name service, and network communication. Just as text-processing mechanisms minimize the need to employ external tools like `sed`, `awk`, and `grep`, operating system built-ins minimize the need for other external tools.

At the same time, scripting languages have, over time, developed a rich set of features for internal computation. Most have significantly better support for mathematics than is typically found in a shell. Several, including Scheme, Python, and Ruby, support arbitrary precision arithmetic. Most provide extensive support for higher-level types, including arrays, strings, tuples, lists, and hashes (associative arrays). Several support classes and object orientation. Some support iterators, continuations, threads, reflection, and first-class and higher-order functions. Some, including Perl, Tcl, Python, and Ruby, support modules and dynamic loading, for "programming in the large." These features serve to maximize the amount of code that can be written in the scripting language itself, and to minimize the need to escape to a more traditional, compiled language.

In summary, the philosophy of general-purpose scripting is make it as easy as possible to construct the overall framework of a program, escaping to external tools only for special-purpose tasks, and to compiled languages only when performance is at a premium.

Tcl

Tcl was originally developed in the late 1980s by Prof. John Ousterhout of the University of California, Berkeley. Over the previous several years his group had developed a suite of VLSI design automation tools, each of which had its own idiosyncratic command language. The initial motivation for Tcl ("tool command language") was the desire for an *extension language* that could be embedded in all the tools, providing them with uniform command syntax and reducing the complexity of development and maintenance. Tk, a set of extensions for graphical user interface programming, was added to Tcl early in its development, and both Tcl and Tk were made available to other researchers starting in 1990. The user community grew rapidly in the 1990s, and Tcl quickly evolved beyond its emphasis on command extension to encompass "glue" applications as well. Ousterhout joined Sun Microsystems in 1994, where for three years he led a multiperson team devoted to Tcl development. In 1997 he launched a start-up company specializing in Tcl applications and tools.

In comparison to Perl, Tcl is somewhat more verbose. It makes less use of punctuation, and has fewer special cases. Everything in the language, including control-flow constructs, takes the form of a (possibly quoted) *command* (an identifier) followed by a series of arguments. In the spirit of Unix command-line invocation, the first few, optional arguments typically begin with a minus sign (`-`) and are known as "switches."

EXAMPLE 13.24
"Force quit" script in Tcl

A simple Tcl script, equivalent to the Perl script of Figure 13.5, appears in Figure 13.6. The `set` command is an assignment; it copies the value of its second argument into the variable named by the first argument. In most other contexts a variable name needs to be preceded by a dollar sign (`$`); as in shell languages, this indicates that the value of the variable should be expanded in-line. (Note the

```
if {$argc != 1} {puts stderr "usage: $argv0 pattern"; exit 1}
set PS [open "|/bin/ps -w -w -x -opid,command" r]

gets $PS                        ;# discard header line
while {! [eof $PS]} {
    set line [gets $PS]         ;# returns blank line at eof
    regexp {[0-9]+} $line proc
    if {[regexp [lindex $argv 0] $line] && [expr $proc != [pid]]} {
        puts -nonewline "$line? "
        flush stdout            ;# force prompt out to screen
        set answer [gets stdin]
        while {! [regexp -nocase {^[yn]} $answer]} {
            puts -nonewline "? "
            flush stdout
            set answer [gets stdin]
        }
        if {[regexp -nocase {^y} $answer]} {
            set stat [catch {exec kill -9 $proc}]
            exec sleep 1
            if {$stat || ![catch {exec ps -p $proc}]} {
                puts stderr "unsuccessful; sorry"; exit 1
            }
        }
    }
}
```

Figure 13.6 **Script in Tcl to "force quit" errant processes.** Compare to the Perl script of Figure 13.5.

contrast to Perl, in which the dollar sign indicates scalar type, and must appear even when the variable is used as an l-value.) As in most scripting languages, variables in Tcl need not be declared.

Double quote marks (as in `"$line? "`) behave in the familiar way: variable references inside are expanded before the string is used. Braces (`{ }`) work much as the single quotes of shell languages or Perl: they inhibit internal expansion. Brackets (`[ ]`) are a bit like traditional backquotes, but instead of interpreting the enclosed string as a program name and arguments, they interpret that string as a Tcl script, whose output should be expanded in place of the bracketed string. In the header of the `while` loop of Figure 13.6, the `eof` command returns a `1` or a `0`, which is then interpreted as true or false. Like `$`-prefixed variable names, bracketed expressions are expanded inside double quotes and brackets, but not inside braces.

In the third line of the `while` loop there are two pairs of nested brackets. The expression `[lindex $argv 0]` returns the first element of the list `$argv` (the one with index zero). This is the pattern specified on the command line of the script. It is passed as the first argument to the `regexp` command, along with the current line of output from the `ps` program. The `regexp` command in turn returns a

`1` or a `0`, depending on whether the pattern could be found within the line. The `expr` command interprets its remaining arguments as an arithmetic/logical expression with infix operators. The `pid` command returns the process id of the Tcl interpreter currently running the script. To facilitate the use of infix notation in conditions, the first argument to the `if` and `while` commands is automatically passed to `expr`.

Multiple Tcl commands can be written on a single line, so long as they are separated by semicolons. A newline character terminates the current command unless it is escaped with a backslash (`\`) or appears within a brace-quoted string. Control structures like `if` and `while` can thus span multiple lines so long as the nested commands are enclosed in braces, and the opening brace appears on the same line as the condition. All variables and arguments, including nested bracketed scripts, are represented internally as character strings. Moreover arguments are expanded and evaluated lazily, so `if` and `while` behave as one would expect. The sharp character (`#`) introduces a comment, but as in `sed` (and in contrast to most programming languages) this is permitted only where a command might otherwise appear. In particular, a comment that follows a command on the same line of the script must be separated from the command by a semicolon.

The `exec` command interprets its remaining arguments as the name and arguments of an external program; it executes that program and returns its output. Many functions that are built into Perl must be invoked as external programs in Tcl; the `kill` and `sleep` functions of Figures 13.5 and 13.6 are two examples. The `catch` command executes the nested `exec` in a protected environment that produces no error messages, but returns a status code than can be inspected later (nonzero indicates error). The second use of `catch` (for `ps -p $proc`) checks for process existence.

■

Python

As noted in Section 13.1, Rexx is generally considered the first of the general-purpose scripting languages, predating Perl and Tcl by almost a decade. Perl and Tcl are roughly contemporaneous: both were initially developed in the late 1980s. Perl was originally intended for glue and text-processing applications. Tcl was originally an extension language, but soon grew into glue applications as well. As the popularity of scripting grew in the 1990s, users were motivated to develop additional languages, to provide additional features, address the needs of specific application domains (more on this in subsequent sections), or support a style of programming more in keeping with the personal taste of their designers.

Python was originally developed by Guido van Rossum at CWI in Amsterdam, the Netherlands, in the early 1990s. He continued his work at CNRI in Reston, Virginia, beginning in 1995. After a a series of subsequent moves, he joined Google in 2005. Recent versions of the language are owned by the Python Software Foundation. All releases are open source.

EXAMPLE 13.25
"Force quit" script in Python

Figure 13.7 presents a Python version of our "force quit" program. Reflecting the maturation of programming language design, Python was from the beginning

```
import sys, os, re, time
if len(sys.argv) != 2:
    sys.stderr.write('usage: ' + sys.argv[0] + ' pattern\n')
    sys.exit(1)

PS = os.popen("/bin/ps -w -w -x -o'pid,command'")
line = PS.readline()                # discard header line
line = PS.readline().rstrip()       # prime pump
while line != "":
    proc = int(re.search('\S+', line).group())
    if re.search(sys.argv[1], line) and proc != os.getpid():
        print line + '? ',
        answer = sys.stdin.readline()
        while not re.search('^[yn]', answer, re.I):
            print '? ',             # trailing comma inhibits newline
            answer = sys.stdin.readline()
        if re.search('^y', answer, re.I):
            os.kill(proc, 9)
            time.sleep(1)
            try:                    # expect exception if process
                os.kill(proc, 0)    # no longer exists
                sys.stderr.write("unsuccessful; sorry\n"); sys.exit(1)
            except: pass        # do nothing
        sys.stdout.write('')    # inhibit prepended blank on next print
    line = PS.readline().rstrip()
```

Figure 13.7 **Script in Python to "force quit" errant processes.** Compare to Figures 13.5 and 13.6.

an object-oriented language.[4] It includes a standard library as rich as that of Perl, but partitioned into a collection of namespaces reminiscent of those of C++, Java, or C#. The first line of our script imports symbols from the `sys`, `os`, `re`, and `time` library modules. The fifth line launches `ps` as an external program, and ties its output to the file object `PS`. In standard object-oriented style, `readline` is then invoked as a method of this object.

Perhaps the most distinctive feature of Python, though hardly the most important, is its reliance on indentation for syntactic grouping. We have already seen that Tcl uses linebreaks to separate commands. Python does so also, and further specifies that the body of a structured statement consists of precisely those subsequent statements that are indented one more tab stop. Like the "more than one way to do it" philosophy of Perl, Python's use of indentation tends to arouse strong feelings among users: some strongly positive, some strongly negative.

4 Rexx and Tcl have object-oriented extensions, named Object Rexx and Incr Tcl, respectively. Perl 5 includes some (rather awkward) object-oriented features; Perl 6 will have more uniform object support.

The regular expression (`re`) library has all of the power available in Perl, but employs the somewhat more verbose syntax of method calls, rather than the built-in notation of Perl. The `search` routine returns a "match object" that captures, lazily, the places in the string at which the pattern appears. If no match is found, `search` returns `None`, the empty object, instead. In a condition, `None` is interpreted as false, while a true match object is interpreted as true. The match object in turn supports a variety of methods, including `group`, which returns the substring corresponding to the first match. The `re.I` flag to `search` indicates case insensitivity. Note that `group` returns a string. Unlike Perl and Tcl, Python will not coerce this to an integer—hence the need for the explicit type conversion on the first line of the body of the `while` loop.

As in Perl (and in contrast to Tcl), the `readline` method does not remove the newline character at the end of an input line; we use the `rstrip` method to do this. The `print` routine adds a newline to the end of its argument list unless that list ends with a trailing comma. The `print` routine also prepends a space to its output unless a set of well-defined heuristics indicate that the output will appear at the beginning of a line. The `write` of a null string at the bottom of the `while` loop serves to defeat these heuristics in the wake of the user's input, avoiding a spurious blank at the beginning of the next process prompt.

The `sleep` and `kill` routines are built into Python, much as they are in Perl. When given a signal number of 0, `kill` tests for process existence. Instead of returning a status code, however, as it does in Perl, the Python `kill` throws an exception if the process does not exist. We use a `try` block to catch this exception in the expected case. ■

While our "force quit" program may convey, at least in part, the "feel" of various languages, it cannot capture the breadth of their features. Python includes many of the more interesting features discussed in earlier chapters, including nested functions with static scoping, lambda expressions and higher-order functions, true iterators, list comprehensions, array slice operations, reflection, structured exception handling, multiple inheritance, and modules and dynamic loading.

Ruby

Ruby is the newest of the widely used glue languages. It was developed in Japan in the early 1990s by Yukihiro "Matz" Matsumoto. Matz writes that he "wanted a language more powerful than Perl, and more object-oriented than Python" [TFH04, Foreword]. The first public release was made available in 1995, and quickly gained widespread popularity in Japan. With the publication in 2001 of English-language documentation [TFH04, 1st ed.], Ruby spread rapidly elsewhere as well. Much of its recent success can be credited to the Ruby on Rails web-development framework. Originally released by David Heinemeier Hansson in 2004, Rails has been adopted by several major players—notably Apple, which included it in the 10.5 "Leopard" release of the Mac OS.

EXAMPLE 13.26
Method call syntax in Ruby

In keeping with Matz's original motivation, Ruby is a pure object-oriented language, in the sense of Smalltalk: everything—even instances of built-in

```
ARGV.length() == 1 or begin
    $stderr.print("usage: #{$0} pattern\n"); exit(1)
end

pat = Regexp.new(ARGV[0])
IO.popen("ps -w -w -x -o'pid,command'") {|PS|
    PS.gets                 # discard header line
    PS.each {|line|
        proc = line.split[0].to_i
        if line =~ pat and proc != Process.pid then
            print line.chomp
            begin
                print "? "
                answer = $stdin.gets
            end until answer =~ /^[yn]/i
            if answer =~ /^y/i then
                Process.kill(9, proc)
                sleep(1)
                begin       # expect exception (process gone)
                    Process.kill(0, proc)
                    $stderr.print("unsuccessful; sorry\n"); exit(1)
                rescue      # handler -- do nothing
                end
            end
        end
    }
}
```

Figure 13.8 **Script in Ruby to "force quit" errant processes.** Compare to Figures 13.5, 13.6, and 13.7.

types—is an object. Integers have more than 25 built-in methods. Strings have more than 75. Smalltalk-like syntax is even supported: `2 * 4 + 5` is syntactic sugar for `(2.*(4)).+(5)`, which is in turn equivalent to `(2.send('*', 4)).send('+', 5)`.[5] ■

EXAMPLE 13.27 "Force quit" script in Ruby

Figure 13.8 presents a Ruby version of our "force quit" program. As in Tcl, a newline character serves to end the current statement, but indentation is not significant. A dollar sign (`$`) at the beginning of an identifier indicates a global name. Though it doesn't appear in this example, an at sign (`@`) indicates an instance variable of the current object. Double at signs (`@@`) indicate an instance variable of the current *class*.

5 Parentheses here are significant. Infix arithmetic follows conventional precedence rules, but method invocation proceeds from left to right. Likewise, parentheses can be omitted around argument lists, but the method-selecting dot (`.`) groups more tightly than the argument-separating comma (`,`), so `2.send '*', 4.send '+', 5` evaluates to 18, not 13.

Probably the most distinctive feature of Figure 13.8 is its use of *blocks* and iterators. The `IO.popen` class method takes as argument a string that specifies the name and arguments of an external program. The method also accepts, in a manner reminiscent of Smalltalk, an *associated block*, specified as a multiline fragment of Ruby code delimited with curly braces. This block is invoked by `popen`, passing as parameter a file handle (an object of class `IO`) that represents the output of the external command. The `|PS|` at the beginning of the block specifies the name by which this handle is known within the block. In a similar vein, the `each` method of object `PS` is an iterator that invokes the associated block (the code in braces beginning with `|line|`) once for every line of data. For those more comfortable with traditional `for` loop syntax, the iterator can also be written

```
for line in PS
    ...
end
```

In addition to (true) iterators, Ruby provides continuations, first-class and higher-order functions, and closures with unlimited extent. Its *module* mechanism supports an extended form of mix-in inheritance. Though a class cannot inherit data members from a module, it *can* inherit code. Run-time type checking makes such inheritance more or less straightforward. Methods of modules that have not been explicitly included into the current class can be accessed as qualified names; `Process.kill` is an example in Figure 13.8. Methods `sleep` and `exit` belong to module `Kernel`, which is included by class `Object`, and is thus available everywhere without qualification. Like `popen`, they are class methods, rather than instance methods; they have no notion of "current object." Variables `stdin` and `stderr` refer to global objects of class `IO`.

Regular expression operations in Ruby are methods of class `RegExp`, and can be invoked with standard object-oriented syntax. For convenience, Perl-like notation is also supported as syntactic sugar; we have used this notation in Figure 13.8.

The `rescue` clause of the innermost `begin ... end` block is an exception handler. As in the Python code of Figure 13.7, it allows us to determine whether the `kill` operation has succeeded by catching the (expected) exception that arises when we attempt to refer to a process after it has died. ■

13.2.5 Extension Languages

Most applications accept some sort of *commands*, which tell them what to do. Sometimes these commands are entered textually; more often they are triggered by user interface events such as mouse clicks, menu selections, and keystrokes. Commands in a graphical drawing program might save or load a drawing; select, insert, delete, or modify its parts; choose a line style, weight, or color; zoom or rotate the display; or modify user preferences.

An *extension language* serves to increase the usefulness of an application by allowing the user to create new commands, generally using the existing commands as primitives. Extension languages are increasingly seen as an essential feature of sophisticated tools. Adobe's graphics suite (Illustrator, Photoshop, InDesign, etc.) can be extended (scripted) using JavaScript, Visual Basic (on Windows), or AppleScript (on the Mac). AOLserver, an open source web server from America Online, can be scripted using Tcl. Disney and Industrial Light & Magic use Python to extend their internal (proprietary) tools. Many commercially available packages, including AutoCAD, Maya, Director, and Flash have their own unique scripting languages. This list barely scratches the surface.

To admit extension, a tool must

- incorporate, or communicate with, an interpreter for a scripting language.
- provide hooks that allow scripts to call the tool's existing commands.
- allow the user to tie newly defined commands to user interface events.

With care, these mechanisms can be made independent of any particular scripting language. As we noted in the sidebar on page 651, Microsoft's Windows Script interface allows arbitrary languages to be used to script the operating system, web server, and browser. GIMP, the widely used GNU Image Manipulation Program, has a comparably general interface, and can be scripted in Scheme, Tcl, Python, and Perl, among others. There is a tendency, of course, for user communities to converge on a favorite language, to facilitate sharing of code. Microsoft tools are usually scripted with Visual Basic. GIMP is usually scripted with the SIOD dialect of Scheme. Adobe tools are usually scripted with Visual Basic on the PC, or AppleScript on the Mac.

One of the oldest existing extension mechanisms is that of the `emacs` text editor, used to write this book. An enormous number of extension packages have been created for `emacs`; many of them are installed by default in the standard distribution. In fact much of what users consider the editor's core functionality is actually provided by extensions; the truly built-in parts are comparatively small.

EXAMPLE 13.28
Numbering lines with Emacs Lisp

The extension language for `emacs` is a dialect of Lisp called Emacs Lisp. An example script appears in Figure 13.9. It assumes that the user has used the standard *marking* mechanism to select a region of text. It then inserts a line number at the beginning of every line in the region. The first line is numbered 1 by default, but an alternative starting number can be specified with an optional parameter. Line numbers are bracketed with a prefix and suffix that are " " (empty) and "`) `" by default, but can be changed by the user if desired. To maintain existing alignment, small numbers are padded on the left with enough spaces to match the width of the number on the final line.

Many features of Emacs Lisp can be seen in this example. The `setq-default` command is an assignment that is visible in the current buffer (editing session) and in any concurrent buffers that haven't explicitly overridden the previous value. The `defun` command defines a new command. Its arguments are, in order, the command name, formal parameter list, documentation string, interactive specification, and body. The argument list for `number-region` includes the start and

```
(setq-default line-number-prefix "")
(setq-default line-number-suffix ") ")
(defun number-region (start end &optional initial)
  "Add line numbers to all lines in region.
With optional prefix argument, start numbering at num.
Line number is bracketed by strings line-number-prefix
and line-number-suffix (default \"\" and \") \")."
  (interactive "*r\np")  ; how to parse args when invoked from keyboard
  (let* ((i (or initial 1))
         (num-lines (+ -1 initial (count-lines start end)))
         (fmt (format "%%%dd" (length (number-to-string num-lines))))
                           ; yields "%1d", "%2d", etc. as appropriate
         (finish (set-marker (make-marker) end)))
    (save-excursion
      (goto-char start)
      (beginning-of-line)
      (while (< (point) finish)
        (insert line-number-prefix (format fmt i) line-number-suffix)
        (setq i (1+ i))
        (forward-line 1))
      (set-marker finish nil))))
```

Figure 13.9 Emacs Lisp function to number the lines in a selected region of text.

end locations of the currently marked region, and the optional initial line number. The documentation string is automatically incorporated into the on-line help system. The interactive specification controls how arguments are passed when the command is invoked through the user interface. (The command can also be called from other scripts, in which case arguments are passed in the conventional way.) The "`*`" raises an exception if the buffer is read-only. The "`r`" represents the beginning and end of the currently marked region. The "`\n`" separates the "`r`" from the following "`p`," which indicates an optional numeric *prefix argument*. When the command is bound to a keystroke, a prefix argument of, say, 10 can be specified by preceding the keystroke with "`C-u 10`" (control-U 10).

As usual in Lisp, the `let*` command introduces a set of local variables in which later entries in the list (`fmt`) can refer to earlier entries (`num-lines`). A *marker* is an index into the buffer that is automatically updated to maintain its position when text is inserted in front of it. We create the `finish` marker so that newly inserted line numbers do not alter our notion of where the to-be-numbered region ends. We set `finish` to `nil` at the end of the script to relieve `emacs` of the need to keep updating the marker between now and whenever the garbage collector gets around to reclaiming it.

The `format` command is similar to `sprintf` in C. We have used it, once in the declaration of `fmt` and again in the call to `insert`, to pad all line numbers out to an appropriate length. The `save-excursion` command is roughly equivalent to an exception handler (e.g., a Java `try` block) with a `finally` clause that restores the current focus of attention (`(point)`) and the borders of the marked region.

Our script can be supplied to `emacs` by including it in a personal start-up file (usually `~/.emacs`), by using the interactive `load-file` command to read some other file in which it resides, or by loading it into a buffer, placing the focus of attention immediately after it, and executing the interactive `eval-last-sexp` command. Once any of these has been done, we can invoke our command interactively by typing `M-x number-region <RET>` (meta-X, followed by the command name and the return key). Alternatively, we can *bind* our command to a keyboard shortcut:

```
(define-key global-map [?\C-#] 'number-region)
```

This one-line script, executed in any of the ways described above, binds our `number-region` command to the key combination "control-number-sign". ■

✓ CHECK YOUR UNDERSTANDING

11. What is the most widely used scripting language?
12. List the principal limitations of `sed`.
13. What is meant by the *pattern space* in `sed`?
14. Briefly describe the *fields* and *associative arrays* of `awk`.
15. What is the Perl motto?
16. Explain the special relationship between `while` loops and file handles in Perl. What is the meaning of the empty file handle, `<>`?
17. Name three widely used commercial packages for mathematical computing.
18. List several distinctive features of the R statistical scripting language.
19. Explain the meaning of the `$` and `@` characters at the beginning of variable names in Perl. Explain the different meaning for the `$` sign in Tcl, and the still different meanings of `$`, `@`, and `@@` in Ruby.
20. Describe the semantics of braces (`{ }`) and square brackets (`[ ]`) in Tcl.
21. Which of the languages described in Section 13.2.4 uses indentation to control syntactic grouping?
22. List several distinctive features of Python.
23. Describe, briefly, how Ruby uses *blocks* and *iterators*.
24. What capabilities must a scripting language provide in order to be used for extension?
25. Name several commercial tools that use extension languages.

13.3 Scripting the World Wide Web

Much of the content of the World Wide Web—particularly the content that is visible to search engines—is static: pages that seldom, if ever, change. But hypertext, the abstract notion on which the Web is based, was always conceived as a way to represent "the complex, the changing, and the indeterminate" [Nel65]. Much of the power of the Web today lies in its ability to deliver pages that move, play sounds, respond to user actions, or—perhaps most important—contain information created or formatted on demand, in response to the page-fetch request.

From a programming languages point of view, simple playback of recorded audio or video is not particularly interesting. We therefore focus our attention here on content that is generated on the fly by a program—a script—associated with an Internet URI (uniform resource identifier).[6] Suppose we type a URI into a browser on a client machine, and the browser sends a request to the appropriate web server. If the content is dynamically created, an obvious first question is: does the script that creates it run on the server or the client machine? These options are known as *server-side* and *client-side* web scripting, respectively.

Server-side scripts are typically used when the service provider wants to retain complete control over the content of the page, but can't (or doesn't want to) create the content in advance. Examples include the pages returned by search engines, Internet retailers, auction sites, and any organization that provides its clients with on-line access to personal accounts. Client-side scripts are typically used for tasks that don't need access to proprietary information, and are more efficient if executed on the client's machine. Examples include interactive animation, error-checking of fill-in forms, and a wide variety of other self-contained calculations.

13.3.1 CGI Scripts

The original mechanism for server-side web scripting is the Common Gateway Interface (CGI). A CGI script is an executable program residing in a special directory known to the web server program. When a client requests the URI corresponding to such a program, the server executes the program and sends its output back to the client. Naturally, this output needs to be something that the browser will understand—typically HTML.

CGI scripts may be written in any language available on the server's machine, though Perl is particularly popular: its string-handling and "glue" mechanisms are ideally suited to generating HTML, and it was already widely available during

6 The term "URI" is often used interchangably with "URL" (uniform resource locator), but the World Wide Web Consortium distinguishes between the two. All URIs are hierarchical (multipart) names. URLs are one kind of URIs; they use a naming scheme that indicates where to find the resource. Other URIs can use other naming schemes.

```
#!/usr/bin/perl

print "Content-type: text/html\n\n";

$host = `hostname`; chop $host;
print "<HTML>\n<HEAD>\n<TITLE>Status of ", $host,
      "</TITLE>\n</HEAD>\n<BODY>\n";
print "<H1>", $host, "</H1>\n";
print "<PRE>\n", `uptime`, "\n", `who`;
print "</PRE>\n</BODY>\n</HTML>\n";
```

Figure 13.10 **A simple CGI script in Perl.** If this script is named `status.perl`, and is installed in the server's `cgi-bin` directory, then a user anywhere on the Internet can obtain summary statistics and a list of users currently logged in to the server by typing *hostname/cgi-bin/status.perl* into a browser window.

EXAMPLE 13.29
Remote monitoring with a CGI script

the early years of the Web. As a simple if somewhat artificial example, suppose we would like to be able to monitor the status of a server machine shared by some community of users. The Perl script in Figure 13.10 creates a web page titled by the name of the server machine, and containing the output of the `uptime` and `who` commands (two simple sources of status information). The script's initial `print` command produces an HTTP message header, indicating that what follows is HTML. Sample output from executing the script appears in Figure 13.11. ■

EXAMPLE 13.30
Adder web form with a CGI script

CGI scripts are commonly used to process on-line forms. A simple example appears in Figure 13.12. The `FORM` element in the HTML file specifies the URI of the CGI script, which is invoked when the user hits the Submit button. Values previously entered into the `INPUT` fields are passed to the script either as a trailing part of the URI (for a `get`-type form) or on the standard input stream (for a `post`-type form, shown here).[7] With either method, we can access the values using the `param` routine of the standard `CGI` Perl library, loaded at the beginning of our script. ■

13.3.2 Embedded Server-Side Scripts

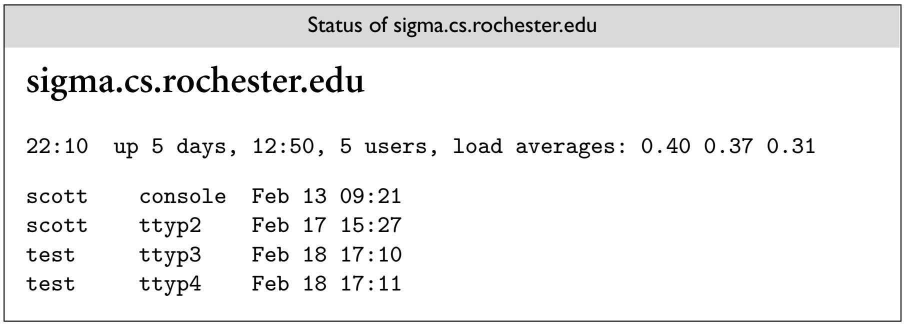
Though widely used, CGI scripts have several disadvantages:

- The web server must launch each script as a separate program, with potentially significant overhead (though a CGI script compiled to native code can be very fast once running).
- Because the server has little control over the behavior of a script, scripts must generally be installed in a trusted directory by trusted system administrators; they cannot reside in arbitrary locations as ordinary pages do.

7 One typically uses `post` type forms for one-time requests. A `get` type form appears a little clumsier, because arguments are visibly embedded in the URI, but this gives it the advantage of repeatability: it can be "bookmarked" by client browsers.

```
<HTML>
<HEAD>
<TITLE>Status of sigma.cs.rochester.edu</TITLE>
</HEAD>
<BODY>
<H1>sigma.cs.rochester.edu</H1>
<PRE>
22:10  up 5 days, 12:50, 5 users, load averages: 0.40 0.37 0.31

scott    console  Feb 13 09:21
scott    ttyp2    Feb 17 15:27
test     ttyp3    Feb 18 17:10
test     ttyp4    Feb 18 17:11
</PRE>
</BODY>
</HTML>
```


Status of sigma.cs.rochester.edu

sigma.cs.rochester.edu

```
22:10  up 5 days, 12:50, 5 users, load averages: 0.40 0.37 0.31

scott    console  Feb 13 09:21
scott    ttyp2    Feb 17 15:27
test     ttyp3    Feb 18 17:10
test     ttyp4    Feb 18 17:11
```

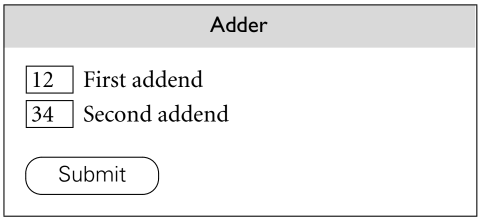
Figure 13.11 Sample output from the script of Figure 13.10. HTML source appears at top; the rendered page is below.

- The name of the script appears in the URI, typically prefixed with the name of the trusted directory, so static and dynamic pages look different to end users.
- Each script must generate not only dynamic content, but also the HTML tags that are needed to format and display it. This extra "boilerplate" makes scripts more difficult to write.

To address these disadvantages, most web servers now provide a "module-loading" mechanism that allows interpreters for one or more scripting languages to be incorporated into the server itself. Scripts in the supported language(s) can then be embedded in "ordinary" web pages. The web server interprets such scripts directly, without launching an external program. It then replaces the scripts with the output they produce, before sending the page to the client. Clients have no way to even know that the scripts exist.

Embedable server-side scripting languages include PHP, Visual Basic (in Microsoft Active Server Pages), Ruby, Cold Fusion (from Macromedia Corp.), and Java (via "Servlets" in Java Server Pages). The most common of these is

```
<HTML>
<HEAD>
<TITLE>Adder</TITLE>
</HEAD>
<BODY>
<FORM action="/cgi-bin/add.perl" method="post">
<P><INPUT name="argA" size=3>First addend<BR>
   <INPUT name="argB" size=3>Second addend
<P><INPUT type="submit">
</FORM>
</BODY>
</HTML>
```



```
#!/usr/bin/perl

use CGI qw(:standard);      # provides access to CGI input fields
$argA = param("argA");  $argB = param("argB");  $sum = $argA + $argB;

print "Content-type: text/html\n\n";

print "<HTML>\n<HEAD>\n<TITLE>Sum</TITLE>\n</HEAD>\n<BODY>\n";
print "<P>$argA plus $argB is $sum";
print "</BODY>\n</HTML>\n";
```

```
<HTML>
<HEAD>
<TITLE>Sum</TITLE>
</HEAD>
<BODY>
<P>12 plus 34 is 46</BODY>
</HTML>
```

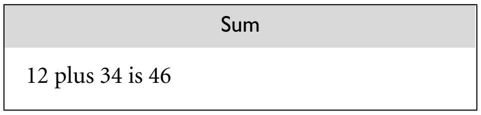


Figure 13.12 An interactive CGI form. Source for the original web page is shown at the upper left, with the rendered page to the right. The user has entered 12 and 34 in the text fields. When the Submit button is pressed, the client browser sends a request to the server for URI */cgi-bin/add.perl*. The values 12 and 13 are contained within the request. The Perl script, shown in the middle, uses these values to generate a new web page, shown in HTML at the bottom left, with the rendered page to the right.

PHP. Though descended from Perl, PHP has been extensively customized for its target domain, with built-in support for (among other things) email and MIME encoding, all the standard Internet communication protocols, authentication and security, HTML and URI manipulation, and interaction with dozens of database systems.

EXAMPLE 13.31
Remote monitoring with a PHP script

The PHP equivalent of Figure 13.10 appears in Figure 13.13. Most of the text in this figure is standard HTML. PHP code is embedded between `<?php` and `?>` delimiters. These delimiters are not themselves HTML; rather they indicate a

```
<HTML>
<HEAD>
<TITLE>Status of <?php echo $host = chop(‘hostname‘) ?></TITLE>
</HEAD>
<BODY>
<H1><?php echo $host ?></H1>
<PRE>
<?php echo ‘uptime‘, "\n", ‘who‘ ?>
</PRE>
</BODY>
</HTML>
```

Figure 13.13 A simple PHP script embedded in a web page. When served by a PHP-enabled host, this page performs the equivalent of the CGI script of Figure 13.10.

```
<HTML><BODY><P>
<?php
         for ($i = 0; $i < 20; $i++) {
             if ($i % 2) { ?>
<B><?php
                 echo " $i"; ?>
</B><?php
             } else echo " $i";
         }
 ?>
</BODY></HTML>
```

Figure 13.14 A fragmented PHP script. The `if` and `for` statements work as one might expect, despite the intervening raw HTML. When requested by a browser, this page displays the numbers from 0 to 19, with odd numbers written in bold.

processing instruction that needs to be executed by the PHP interpreter to generate replacement text. The "boilerplate" parts of the page can thus appear verbatim; they need not be generated by `print` (Perl) or `echo` (PHP) commands. Note that the separate script fragments are part of a single program. The `$host` variable, for example, is set in the first fragment and used again in the second. ■

EXAMPLE 13.32
A fragmented PHP script

PHP scripts can even be broken into fragments in the middle of structured statements. Figure 13.14 contains a script in which `if` and `for` statements span fragments. In effect, the HTML text between the end of one script fragment and the beginning of the next behaves as if it had been output by an `echo` command. Web designers are free to use whichever approach (`echo` or escape to raw HTML) seems most convenient for the task at hand. ■

Self-Posting Forms

EXAMPLE 13.33
Adder web form with a PHP script

By changing the `action` attribute of the `FORM` element, we can arrange for the Adder page of Figure 13.12 to invoke a PHP script instead of a CGI script:

```
<FORM action="add.php" method="post">
```

```
<HTML><HEAD><TITLE>Sum</TITLE></HEAD><BODY><P>
<?php
      $argA = $_REQUEST['argA']; $argB = $_REQUEST['argB'];
      $sum = $argA + $argB;
      echo "$argA plus $argB is $sum\n";
 ?>
</BODY></HTML>
```

```
<?php
      $argA = $_REQUEST['argA'];  $argB = $_REQUEST['argB'];
      if (!isset($_REQUEST['argA']) || $argA == "" || $argB == "") {
          # form has not been posted, or arguments are incomplete
 ?>
          <HTML><HEAD><TITLE>Adder</TITLE></HEAD><BODY>
          <FORM action="adder.php" method="post">
          <P>First addend: <INPUT name="argA" size=3>
            Second addend: <INPUT name="argB" size=3>
          <P><INPUT type="submit">
          </FORM></BODY></HTML>
<?php
      } else {  # form is complete; return results
 ?>
          <HTML><HEAD><TITLE>Sum</TITLE></HEAD><BODY><P>
<?php
          $sum = $argA + $argB;
          echo "$argA plus $argB is $sum\n";
 ?>
          </BODY></HTML>
<?php
      }
 ?>
```

Figure 13.15 An interactive PHP web page. The script at top could be used in place of the script in the middle of Figure 13.12. The lower script in the current figure replaces both the web page at the top and the script in the middle of Figure 13.12. It checks to see if it has received a full set of arguments. If it hasn't, it displays the fill-in form; if it has, it displays results.

The PHP script itself is shown in the top half of Figure 13.15. Form values are made available to the script in an associative array (hash table) named `_REQUEST`. No special library is required. ■

EXAMPLE 13.34
Self-posting Adder web form

Because our PHP script is executed directly by the web server, it can safely reside in an arbitrary web directory, including the one in which the Adder page resides. In fact, by checking to see how a page was requested, we can merge the form and the script into a single page, and let it service its own requests! We illustrate this option in the bottom half of Figure 13.15. ■

13.3.3 Client-Side Scripts

While embedded server-side scripts are generally faster than CGI scripts, at least when start-up cost predominates, communication across the Internet is still too slow for truly interactive pages. If we want the behavior or appearance of the page to change as the user moves the mouse, clicks, types, or hides or exposes windows, we really need to execute some sort of script on the client's machine.

Because they run on the web designer's site, CGI scripts and, to a lesser extent, embedable server-side scripts can be written in many different languages. All the client ever sees is standard HTML. Client-side scripts, by contrast, require an interpreter on the client's machine. As a result, there is a powerful incentive for convergence in client-side scripting languages: most designers want their pages to be viewable by as wide an audience as possible. While Visual Basic is widely used within specific organizations, where all the clients of interest are known to run Internet Explorer, pages intended for the general public almost always use JavaScript for interactive features.

EXAMPLE 13.35

Adder web form in JavaScript

Figure 13.16 shows a page with embedded JavaScript that imitates (on the client) the behavior of the Adder scripts of Figures 13.12 and 13.15. Function `doAdd` is defined in the header of the page so it is available throughout. In particular, it will be invoked when the user clicks on the Calculate button. By default the input values are character strings; we use the `parseInt` function to convert them to integers. The parentheses around `(argA + argB)` in the final assignment statement then force the use of integer addition. The other occurrences of `+` are string concatenation. To disable the usual mechanism whereby input data are submitted to the server when the user hits the enter or return key, we have specified a dummy behavior for the `onsubmit` attribute of the form.

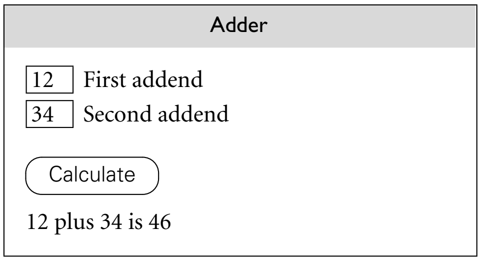
Rather than replace the page with output text, as our CGI and PHP scripts did, we have chosen in our JavaScript version to append the output at the bottom. The HTML `SPAN` element provides a named place in the document where this output can be inserted, and the `getElementById` JavaScript method provides us with a reference to this element. The HTML *Document Object Model (DOM)*, standardized by the World Wide Web Consortium, specifies a very large number of other elements, attributes, and user actions, all of which are accessible in JavaScript. Through them scripts can, at appropriate times, inspect or alter almost any aspect of the content, structure, or style of a page. ■

13.3.4 Java Applets

An applet is a program designed to run inside some other program. The term is most often used for Java programs that display their output in (a portion of) a web page. To support the execution of applets, most modern browsers contain a Java virtual machine.

Like JavaScript, Java applets can be used to create animated or interactive pages. Together with the similarity in language names, the fact that many tasks

```
<HTML>
<HEAD>
<TITLE>Adder</TITLE>
<SCRIPT type="text/javascript">
function doAdd() {
    argA = parseInt(document.adder.argA.value)
    argB = parseInt(document.adder.argB.value)
    x = document.getElementById('sum')
    while (x.hasChildNodes())
        x.removeChild(x.lastChild)  // delete old content
    t = document.createTextNode(argA + " plus "
        + argB + " is " + (argA + argB))
    x.appendChild(t)
}
</SCRIPT>
</HEAD>
<BODY>
<FORM name="adder" onsubmit="return false">
<P><INPUT name="argA" size=3> First addend<BR>
   <INPUT name="argB" size=3> Second addend
<P><INPUT type="button" onclick="doAdd()" value="Calculate">
</FORM>
<P><SPAN id="sum"></SPAN>
</BODY>
</HTML>
```



12 plus 34 is 46

Figure 13.16 An interactive JavaScript web page. Source appears at left. The rendered version on the right shows the appearance of the page after the user has entered two values and hit the Calculate button, causing the output message to appear. By entering new values and clicking again, the user can calculate as many sums as desired. Each new calculation will replace the output message.

can be accomplished with either mechanism has created a great deal of confusion between the two (see sidebar on page 688). In fact, however, they are very different.

EXAMPLE 13.36
Embedding an applet in a web page

To embed an applet in a web page, one would traditionally use an `APPLET` tag:

```
<APPLET width=150 height=150 code="Clock.class">
```

Seeing this element embedded in the page, the client browser would request the URI *Clock.class* from the server. Assuming the server returned an applet, it would run this applet and display the output on the page. ■

Unlike a JavaScript script, an applet does not produce HTML output for the browser to render. Rather it directly controls a portion of the page's real estate, in which it uses routines from one of Java's graphical user interface (GUI) libraries (typically `AWT` or `Swing`) to display whatever it wants. The `width` and `height` attributes of the `APPLET` element tell the browser how big the applet's portion of the page should be.

In effect, applets allow the web designer to escape from HTML entirely, and to create a very precise "look and feel," independent of any design choices embodied by the browser. Images, of course, provide another way to escape from HTML, with static or simple animated content, as do embedded objects of other kinds (movies in Flash or QuickTime format are popular examples). Most modern browsers provide a "plug-in" mechanism that allows the installation of interpreters for arbitrary formats. In support of these, the HTML 4.0 standard provides a generic `OBJECT` element that is meant to be used for any embedded content not rendered by the browser itself. The `APPLET` element is now officially deprecated:[8] one is supposed to use the following instead:

EXAMPLE 13.37
Embedding an object in a web page

```
<P><OBJECT codetype="application/java" classid="java:Clock.class"
    width=150 height=150>
```

Applets are subject to certain restrictions intended to prevent them from damaging the client's machine. For the most part, however, they can make use of the entire Java language, and it is usually a simple task to covert an applet to a standalone program or vice versa. The typical applet has no significant interaction with the browser or any other program. For this reason, applets are generally *not* considered a scripting mechanism.

DESIGN & IMPLEMENTATION

JavaScript and Java

Despite its name, JavaScript has no connection to Java beyond some superficial syntactic similarity. The language was originally developed by Brendan Eich at Netscape Corp. in 1995. Eich called his creation *LiveScript*, but the company chose to rename it as part of a joint marketing agreement with Sun Microsystems, prior to its public release. Trademark on the JavaScript name is actually owned by Sun.

Netscape's browser was still the market leader in 1995, and JavaScript usage grew extremely fast. To remain competitive, developers at Microsoft added JavaScript support to Internet Explorer, but they used the name *JScript* instead, and they introduced a number of incompatibilities with the Netscape version of the language. A common version was standardized as *ECMAScript* by the European standards body in 1997 (and subsequently by the ISO), but major incompatibilities remained in the Document Object Models provided by different browsers. These have been gradually resolved through a series of standards from the World Wide Web Consortium, but legacy pages and legacy browsers continue to plague web developers.

8 A deprecated feature is one whose use is officially discouraged, but permitted on a temporary basis, to ease the transition to new and presumably better alternatives.

13.3.5 XSLT

Most readers will undoubtedly have had the opportunity to write, or at least to read, the HTML (hypertext markup language) used to compose web pages. HTML has, for the most part, a nested structure in which fragments of documents (*elements*) are delimited by *tags* that indicate their purpose or appearance. We saw in Section 13.2.2, for example, that top-level headings are delimited with `<H1>` and `</H1>`. Unfortunately, as a result of the chaotic and informal way in which the web evolved, HTML ended up with many inconsistencies in its design, and incompatibilities among the versions implemented by different vendors.

XML (extensible markup language) is a more recent and general language in which to capture structured data. Compared to HTML, its syntax and semantics are more regular and consistent, and more consistently implemented across

DESIGN & IMPLEMENTATION

How far can you trust a script?

Security becomes an issue whenever code is executed using someone else's resources. Web servers are usually installed with very limited access rights, and with only a limited view of the file system of the server machine. This generally limits the set of pages they can serve to a well-defined subset of what would be visible to users logged in to the server machine directly. Because they are separate executable programs, CGI scripts can be designed to run with the privileges of whoever installed them. To prevent users on the server machine from accidentally or intentionally passing their privileges to arbitrary users on the Internet, most system administrators configure their servers so that CGI scripts must reside in a special directory, and be installed by a trusted user. Embedded server-side scripts can reside in any file because they are guaranteed to run with the (limited) rights of the server.

A larger risk is posed by code downloaded over the Internet and executed on a client machine. Because such code is in general untrusted, it must be executed in a carefully controlled environment, sometimes called a *sandbox*, to prevent it from doing any damage. As a general rule, JavaScript scripts cannot access the local file system, memory management system, or network, nor can they manipulate documents from other sites. Java applets, likewise, have only limited ability to access external resources. Reality is a bit more complicated, of course: Sometimes a script needs access to, say, a temporary file of limited size, or a network connection to a trusted server. Mechanisms exist to certify sites as *trusted*, or to allow a trusted site to certify the trustworthiness of pages from other sites. Scripts on pages obtained through a trusted mechanism may then be given extended rights. Such mechanisms must be used with care. Finding the right balance between security and functionality remains one of the central challenges of the Web, and of distributed computing in general. (More on this topic can be found in Section 15.2.4, and in Explorations 15.17 and 15.18.)

platforms. It is *extensible*, meaning that users can define their own tags. It also makes a clear distinction between the *content* of a document (the data it captures) and the *presentation* of that data. Presentation, in fact, is deferred to a companion standard known as XSL (extensible stylesheet language). XSLT is a portion of XSL devoted to *transforming* XML: selecting, reorganizing, and modifying tags and the elements they delimit—in effect, scripting the processing of data represented in XML.

IN MORE DEPTH

XML can be used to create specialized markup languages for a very wide range of application domains. XHTML is an almost (but not quite) backward compatible variant of HTML that conforms to the XML standard. Web tools are increasingly being designed to generate XHTML.

On the PLP CD we consider a variety of topics related to XML, with a particular emphasis on XSLT. We elaborate on the distinction between content and presentation, introduce the general notion of stylesheet languages, and describe the *document type definitions* (DTDs) and *schemas* used to define domain-specific applications of XML, using XHTML as an example.

Because tags are required to nest, an XML document has a natural tree-based structure. XSLT is designed to process these trees via recursive traversal. Though it can be used for almost any task that takes XML as input, perhaps its most common use is to transform XML into formatted output—often XHTML to be presented in a browser. As an extended example, we consider the formatting of an XML-based bibliographic database.

CHECK YOUR UNDERSTANDING

26. Explain the distinction between *server-side* and *client-side* web scripting.
27. List the tradeoffs between CGI scripts and embedded PHP.
28. Why are CGI scripts usually installed only in a special directory?
29. Explain how a PHP page can service its own requests.
30. Why might we prefer to execute a web script on the server rather than the client? Why might we sometimes prefer the client instead?
31. What is the HTML *Document Object Model*? What is its significance for client-side scripting?
32. What is the relationship between JavaScript and Java?
33. What is an *applet*? Why are applets usually not considered a form of scripting?
34. What is HTML? XML? XSLT? How are they related to one another?

13.4 Innovative Features

In Section 13.1.1 we listed several common characteristics of scripting languages:

1. Both batch and interactive use
2. Economy of expression
3. Lack of declarations; simple scoping rules
4. Flexible dynamic typing
5. Easy access to other programs
6. Sophisticated pattern matching and string manipulation
7. High-level data types

Several of these are discussed in more detail in the subsections below. Specifically, Section 13.4.1 considers naming and scoping in scripting languages; Section 13.4.2 discusses string and pattern manipulation; and Section 13.4.3 considers data types. Items (1), (2), and (5) in our list, while important, are not particularly difficult or subtle, and will not be considered further here.

13.4.1 Names and Scopes

Most scripting languages (Scheme is the obvious exception) do not require variables to be declared. A few languages, notably Perl and JavaScript, permit optional declarations, primarily as a sort of compiler-checked documentation. Perl can be run in a mode (`use strict 'vars'`) that requires declarations.

With or without declarations, most scripting languages use dynamic typing. Values are generally self-descriptive, so the interpreter can perform type checking at run time, or coerce values when appropriate. Tcl is unusual in that all values—even lists—are represented internally as strings, which are parsed as appropriate to support arithmetic, indexing, and so on.

Nesting and scoping conventions vary quite a bit. Scheme, Python, JavaScript, and R provide the classic combination of nested subroutines and static (lexical) scope. Tcl allows subroutines to nest, but uses dynamic scoping (more on this below). Named subroutines (methods) do not nest in PHP or Ruby, and they only sort of nest in Perl (more on this below as well), but Perl and Ruby join Scheme, Python, JavaScript, and R in providing first-class anonymous local subroutines. Nested blocks are statically scoped in Perl. In Ruby they are part of the named scope in which they appear. Scheme, Perl, Python, Ruby, JavaScript, and R all provide unlimited extent for variables captured in closures. PHP, R, and the major glue languages (Perl, Tcl, Python, Ruby) all have sophisticated namespace mechanisms for information hiding and the selective import of names from separate modules.

```
i = 1;  j = 3
def outer():
    def middle(k):
        def inner():
            global i        # from main program, not outer
            i = 4
        inner()
        return i, j, k      # 3-element tuple
    i = 2                   # new local i
    return middle(j)        # old (global) j

print outer()
print i, j
```

Figure 13.17 A program to illustrate scope rules in Python. There is one instance each of `j` and `k`, but two of `i`: one global and one local to `outer`. The scope of the latter is all of `outer`, not just the portion after the assignment. The `global` statement provides `inner` with access to the outermost `i`, so it can write it without defining a new instance.

What Is the Scope of an Undeclared Variable?

In languages with static scoping, the lack of declarations raises an interesting question: when we access a variable `x`, how do we know if it is local, global, or (if scopes can nest) something in-between? Existing languages take several different approaches. In Perl all variables are global unless explicitly declared. In PHP they are local unless explicitly imported (and all imports are global, since scopes do not nest). Ruby, too, has only two real levels of scoping, but as we saw in Section 13.2.4 it distinguishes between them using prefix characters on names: `foo` is a local variable; `$foo` is a global variable; `@foo` is an instance variable of the current object (the one whose method is currently executing); `@@foo` is an instance variable of the current object's *class* (shared by all sibling instances). (Note: as we shall see in Section 13.4.3, Perl uses similar prefix characters to indicate *type*. These very different uses are a potential source of confusion for programmers who switch between the two languages.)

Perhaps the most interesting scope-resolution rule is that of Python and R. In these languages a variable that is written is assumed to be local, unless it is explicitly imported. A variable that is only read in a given scope is found in the closest enclosing scope that contains a defining write. Consider, for example, the Python program of Figure 13.17. Here we have a set of nested subroutines, as indicated by indentation level. The main program calls `outer`, which calls `middle`, which in turn calls `inner`. Before its call, the main program writes both `i` and `j`. `Outer` reads `j` (to pass it to `middle`), but does not write it. It does, however, write `i`. Consequently, `outer` reads the global `j`, but has its own `i`, different from the global one. `Middle` reads both `i` and `j`, but it does not write either, so it must find them in surrounding scopes. It finds `i` in `outer`, and `j` at the global

EXAMPLE 13.38
Scoping rules in Python

```
proc bar { } {
    upvar i j      ;# j is local name for caller's i
    puts "$j"
    uplevel 2 { puts [expr $a + $b] }
        # execute 'puts' two scopes up the dynamic chain
}

proc foo { i } {
    bar
}

set a 1;  set b 2;  foo 5
```

Figure 13.18 A program to illustrate scope rules in Tcl. The `upvar` command allows `bar` to access variable `i` in its caller's scope, using the name `j`. The `uplevel` command allows `bar` to execute a nested Tcl script (the `puts` command) in its caller's caller's scope.

level. Inner, for its part, also writes the global `i`. When executed the program prints

```
(2, 3, 3)
4 3
```

Note that while the tuple returned from `middle` (forwarded on by `outer`, and printed by the main program) has a 2 as its first element, the global `i` still contains the 4 that was written by `inner`. Note also that while the write to `i` in `outer` appears textually after the read of `i` in `middle`, its scope extends over all of `outer`, including the body of `middle`. ■

EXAMPLE 13.39
Superassignment in R

Interestingly, there is no way in Python for a nested routine to write a variable that belongs to a surrounding but nonglobal scope. In Figure 13.17, `inner` could not be modified to write `outer`'s `i`. R provides an alternative mechanism that does provide this functionality. Rather than declare `i` to be `global`, R uses a "superassignment" operator. Where a normal assignment `i <- 4` assigns the value 4 into a local variable `i`, the superassignment `i <<- 4` assigns 4 into whatever `i` would be found under the normal rules of static (lexical) scoping. ■

EXAMPLE 13.40
Scoping rules in Tcl

In a completely different vein, Tcl not only makes the unusual choice of employing dynamic scoping, but also implements that choice in an unusual way. Variables in calling scopes are never accessed automatically. The programmer must ask for them explicitly, as shown in Figure 13.18. The `upvar` and `uplevel` commands take an optional first argument that specifies a frame on the dynamic chain, either as an absolute value prefaced with a sharp sign (`#`) or, as in the call to `uplevel` shown in our example, as a distance below the current frame. If omitted, as in our call to `upvar`, the argument defaults to 1. The `upvar` command accesses a variable in the specified frame, and gives it a local name. The `uplevel` command provides a nested Tcl script, which is executed in the context of the specified

```
sub outer($) {                    # must be called with scalar arg
    $sub_A = sub {
        print "sub_A  $lex, $dyn\n";
    };
    my $lex = $_[0];              # static local initialized to first arg
    local $dyn = $_[0];           # dynamic local initialized to first arg
    $sub_B = sub {
        print "sub_B  $lex, $dyn\n";
    };
    print "outer  $lex, $dyn\n";
    $sub_A->();
    $sub_B->();
}

$lex = 1; $dyn = 1;
print "main   $lex, $dyn\n";
outer(2);
print "main   $lex, $dyn\n";
```

Figure 13.19 A program to illustrate scope rules in Perl. The `my` operator creates a statically scoped local variable; the `local` operator creates a new dynamically scoped instance of a global variable. The static scope extends from the point of declaration to the lexical end of the block; the dynamic scope extends from elaboration to the end of the block's execution.

frame, in a manner reminiscent of call-by-name parameters. In our example we use `upvar` to obtain a local name for `foo`'s `i`, and `uplevel` to execute a command that uses the global `a` and `b`. The program prints a 5 and a 3. Note that the usual behavior of dynamic scoping, in which we automatically obtain the most recently created variable of a given name, regardless of the scope that created it, is not available in Tcl. ■

Scoping in Perl

Perl has evolved over the years. At first there were only global variables. Locals were soon added for the sake of modularity, so a subroutine with a variable named `i` wouldn't have to worry about modifying a global `i` that was needed elsewhere in the code. Unfortunately, locals were originally defined in terms of dynamic scoping, and the need for backward compatibility required that this behavior be retained when static scoping was added in Perl 5. Consequently, the language provides both mechanisms.

Any variable that is not declared is global in Perl by default. Variables declared with the `local` operator are dynamically scoped. Variables declared with the `my` operator are statically scoped. The difference can be seen in Figure 13.19, in which subroutine `outer` declares two local variables, `lex` and `dyn`. The former is statically scoped; the latter is dynamically scoped. Both are initialized to be a copy of `foo`'s first parameter. (Parameters are passed in the pseudovariable `@_`. The first element of this array is `$_[0]`.)

EXAMPLE 13.41
Static and dynamic scoping in Perl

Two lexically identical anonymous subroutines are nested inside `outer`, one before and one after the redeclarations of `$lex` and `$dyn`. References to these are stored in local variables sub_A and sub_B. Because static scopes in Perl extend from a declaration to the end of its block, sub_A sees the global $lex, while sub_B sees outer's $lex. In contrast, because the declaration of local $dyn occurs before either sub_A or sub_B is called, both see this local version. Our program prints

```
main   1, 1
outer  2, 2
sub_A  1, 2
sub_B  2, 2
main   1, 1
```

EXAMPLE 13.42
Accessing globals in Perl

In cases where static scoping would normally access a variable at an in-between level of nesting, Perl allows the programmer to force the use of a global variable with the our operator, whose name is intended to contrast with my:

```
($x, $y, $z) = (1, 1, 1);       # global scope
{                               # middle scope
    my ($x, $y) = (2, 2);
    local $z = 3;
    {                           # inner scope
        our ($x, $z);           # use globals
        print "$x, $y, $z\n";
    }
}
```

Here there is one lexical instance of z and two of x and y: one global, one in the middle scope. There is also a dynamic z in the middle scope. When it executes its

DESIGN & IMPLEMENTATION

Thinking about dynamic scoping

In Section 3.3.6 we described dynamic scope rules as introducing a new meaning for a name that remains visible, wherever we are in the program, until control leaves the scope in which the new meaning was created. This conceptual model mirrors the association list implementation described in Section ◎3.4.2 and, as described in the sidebar on page 141, probably accounts for the use of dynamic scoping in early dialects of Lisp.

Documentation for Perl suggests a semantically equivalent but conceptually different model. Rather than saying that a local declaration introduces a new variable whose name hides previous declarations, Perl says that there is a *single* variable, at the global level, whose previous *value* is saved when the new declaration is encountered, and then automatically restored when control leaves the new declaration's scope. This model mirrors the underlying implementation in Perl, which uses a central reference table (also described in Section ◎3.4.2). In keeping with this model and implementation, Perl does not allow a local operator to create a dynamic instance of a variable that is not global.

`print` statement, the inner scope finds the `y` from the middle scope. It finds the global `x`, however, because of the `our` operator on line 6. Now what about `z`? The rules require us to start with static scoping, ignoring `local` operators. According, then, to the `our` operator in the inner scope, we are using the global `z`. Once we know this, we look to see whether a dynamic (`local`) redeclaration of `z` is in effect. In this case indeed it is, and our program prints `1, 2, 3`. As it turns out, the `our` declaration in the inner scope had no effect on this program. If only `x` had been declared `our`, we would still have used the global `z`, and then found the dynamic instance from the middle scope. ■

13.4.2 String and Pattern Manipulation

When we first considered regular expressions, in Section 2.1.1, we noted that many scripting languages and related tools employ extended versions of the notation. Some extensions are simply a matter of convenience. Others increase the expressive power of the notation, allowing us to generate (match) nonregular sets of strings. Still other extensions serve to tie the notation to other language features.

We have already seen examples of extended regular expressions in `sed` (Figure 13.1), `awk` (Figures 13.2 and 13.3), Perl (Figures 13.4 and 13.5), Tcl (Figure 13.6), Python (Figure 13.7), and Ruby (Figure 13.8). Many readers will also be familiar with `grep`, the stand-alone Unix pattern-matching tool (see sidebar on page 697).

While there are many different implementations of extended regular expressions ("REs" for short), with slightly different syntax, most fall into two main groups. The first group includes `awk`, `egrep` (the most widely used of several different versions of `grep`), the `regex` library for C, and older versions of Tcl. These implement REs as defined in the POSIX standard [Int03b]. Languages in the second group follow the lead of Perl, which provides a large set of extensions, sometimes referred to as "advanced REs." Perl-like advanced REs appear in PHP, Python, Ruby, JavaScript, Emacs Lisp, Java, C#, and recent versions of Tcl. They can also be found in third-party packages for C++ and other languages. A few tools, including `sed`, classic `grep`, and older Unix editors, provide so-called "basic" REs, less capable than those of `egrep`.

In certain languages and tools—notably `sed`, `awk`, Perl, PHP, Ruby, and JavaScript—regular expressions are tightly integrated into the rest of the language, with special syntax and built-in operators. In these languages an RE is typically delimited with slash characters, though other delimiters may be accepted in some cases (and Perl in fact provides slightly different semantics for a few alternative delimiters). In most other languages, REs are expressed as ordinary character strings, and are manipulated by passing them to library routines. Over the next few pages we will consider POSIX and advanced REs in more detail. Following Perl, we will use slashes as delimiters. Our coverage will of necessity be incomplete. The chapter on REs in the Perl book [WCO00, Chap. 5] is nearly 80 pages long. The corresponding Unix `man` page runs to more than 20 pages.

POSIX Regular Expressions

EXAMPLE 13.43
Basic operations in POSIX REs

Like the "true" regular expressions of formal language theory, extended REs support concatenation, alternation, and Kleene closure. Parentheses are used for grouping.

```
/ab(cd|ef)g*/     matches abcd, abcdg, abefg, abefgg, abcdggg, etc.
```

EXAMPLE 13.44
Extra quantifiers in POSIX REs

Several other *quantifiers* (generalizations of Kleene closure) are also available: `?` indicates zero or one repetitions, `+` indicates one or more repetitions, `{`*n*`}` indicates exactly *n* repetitions, `{`*n*`,}` indicates at least *n* repetitions, and `{`*n*`,` *m*`}` indicates *n*–*m* repetitions:

```
/a(bc)*/          matches a, abc, abcbc, abcbcbc, etc.
/a(bc)?/          matches a or abc
/a(bc)+/          matches abc, abcbc, abcbcbc, etc.
/a(bc){3}/        matches abcbcbc only
/a(bc){2,}/       matches abcbc, abcbcbc, etc.
/a(bc){1,3}/      matches abc, abcbc, and abcbcbc (only)
```

EXAMPLE 13.45
Zero-length assertions

Two *zero-length assertions*, `^` and `$`, match only at the beginning and end, respectively, of a target string. Thus while `/abe/` will match `abe`, `abet`, `babe`, and `label`, `/^abe/` will match only the first two of these, `/abe$/` will match only the first and the third, and `/^abe$/` will match only the first.

EXAMPLE 13.46
Character classes

As an abbreviation for `/a|b|c|d/`, extended REs permit *character classes* to be specified with square brackets:

```
/b[aeiou]d/       matches bad, bed, bid, bod, and bud
```

DESIGN & IMPLEMENTATION

The `grep` command and the birth of Unix tools

Historically, regular expression tools have their roots in the pattern matching mechanism of the `ed` line editor, which dates from the earliest days of Unix. In 1973, Doug McIlroy, head of the department where Unix was born, was working on a project in computerized voice synthesis. As part of this project he was using the editor to search for potentially challenging words in an on-line dictionary. The process was both tedious and slow. At McIlroy's request, Ken Thompson extracted the pattern matcher from `ed` and made it a stand-alone tool. He named his creation `grep`, after the `g/`*re*`/p` command sequence in the editor: `g` for "global"; `/ /` to search for a regular expression (*re*); `p` to print [HH97a, Chapter 9].

Thompson's creation was one of the first in a large suite of stream-based Unix tools. As described in Section 13.2.1 (page 658), such tools are frequently combined with pipes to perform a variety of filtering, transforming, and formatting operations.

Ranges are also permitted:

```
/0x[0-9a-fA-F]+/        matches any hexadecimal integer
```

■

EXAMPLE 13.47
The dot (.) character

Outside a character class, a dot (.) matches any character other than a newline. The expression `/b.d/`, for example, matches not only `bad`, `bbd`, `bcd`, and so on, but also `b:d`, `b7d`, and many, many others, including sequences in which the middle character isn't printable. In a Unicode-enabled version of Perl, there are tens of thousands of options. ■

EXAMPLE 13.48
Negation and quoting in character classes

A caret (`^`) at the beginning of a character class indicates negation: the class expression matches anything *other* than the characters inside. Thus `/b[^aq]d/` matches anything matched by `/b.d/` except for `bad` and `bqd`. A caret, right bracket, or hyphen can be specified inside a character class by preceding it with a backslash. A backslash will similarly protect any of the special characters `| ( ) [ ] { } $ . * + ?` outside a character class.[9] To match a literal backslash, use two of them in a row:

```
/a\\b/                  matches  a\b
```

■

EXAMPLE 13.49
Predefined POSIX character classes

Several character classes expressions are predefined in the POSIX standard. As we saw in Example 13.18, the expression `[:space:]` can be used to capture white space. For punctuation there is `[:punct:]`. The exact definition of these classes depends on the local character set and language. Note, too, that these expressions must be used *inside* a built-up character class; they aren't classes by themselves. A variable name in C, for example, might be matched by `/[[:alpha:]_][[:alpha:][:digit:]_]*/` or, a bit more simply, `/[[:alpha:]_][[:alnum:]_]*/`. Additional syntax, not described here, allows character classes to capture Unicode *collating elements* (multibyte sequences such as a character and associated accents) that collate (sort) as if they were single elements. Perl provides less cumbersome versions of most of these special classes. ■

Perl Extensions

EXAMPLE 13.50
RE matching in Perl

Extended REs are a central part of Perl. The built-in `=~` operator is used to test for matching:

```
$foo = "albatross";
if ($foo =~ /ba.*s+/) ...        # true
if ($foo =~ /^ba.*s+/) ...       # false (no match at start of string)
```

9 Strictly speaking, `]` and `}` don't require a protective backslash unless there is a preceding unmatched (and unprotected) `[` or `{`, respectively.

The string to be matched against can also be left unspecified, in which case Perl uses the pseudovariable `$_` by default:

```
$_ = "albatross";
if (/ba.*s+/) ...                      # true
if (/^ba.*s+/) ...                     # false
```

Recall that (as we noted in Section 13.2.2 [page 666]), `$_` is set automatically when iterating over the lines of a file. It is also the default index variable in `for` loops. ■

EXAMPLE 13.51
Negating a match in Perl

The `!~` operator returns true when a pattern *does not* match:

```
if ("albatross" !~ /^ba.*s+/) ...      # true
```
■

EXAMPLE 13.52
RE substitution in Perl

For substitution, the binary "mixfix" operator `s///` replaces whatever lies between the first and second slashes with whatever lies between the second and the third:

```
$foo = "albatross";
$foo =~ s/lbat/c/;                     # "across"
```

Again, if a left-hand side is not specified, `s///` matches and modifies `$_`. ■

Modifiers and Escape Sequences

Both matches and substitutions can be *modified* by adding one or more characters after the closing delimiter. A trailing `i`, for example, makes the match case-insensitive:

EXAMPLE 13.53
Trailing modifiers on RE matches

```
$foo = "Albatross";
if ($foo =~ /^al/i) ...                # true
```

DESIGN & IMPLEMENTATION

Automata for regular expressions

POSIX regular expressions are typically implemented using the constructions described in Section 2.2.1, which transform the RE into an NFA and then a DFA. Advanced REs of the sort provided by Perl are typically implemented via backtracking search in the obvious NFA. The NFA-to-DFA construction is usually not employed, because it fails to preserve some of the advanced RE extensions (notably the *capture* mechanism described in Examples 13.57–13.60) [WCO00, pp. 197–202]. Some implementations use a DFA first to determine whether there *is* a match, and then an NFA or backtracking search to actually effect the match. This strategy pays the price of the slower automaton only when it's sure to be worthwhile.

Escape	Meaning
`\0`	NUL character
`\a`	alarm (BEL) character
`\b`	backspace (within character class)
`\e`	escape (ESC) character
`\f`	form-feed (FF) character
`\n`	newline
`\r`	return
`\t`	tab
`\`*NNN*	character given by *NNN* in octal
`\x{`*abcd*`}`	character given by *abcd* in hexadecimal
`\b`	word boundary (outside character classes)
`\B`	not a word boundary
`\A`	beginning of string
`\z`	end of string
`\Z`	prior to final newline, or end of string if none
`\d`	digit (decimal)
`\D`	not a digit
`\s`	white space (space, tab, newline, return, form feed)
`\S`	not white space
`\w`	word character (letter, digit, underscore)
`\W`	not a word character

Figure 13.20 Regular expression escape sequences in Perl. Sequences in the top portion of the table represent individual characters. Sequences in the middle are zero-width assertions. Sequences at the bottom are built-in character classes.

A trailing g on a substitution replaces *all* occurrences of the regular expression:

```
$foo = "albatross";
$foo =~ s/[aeiou]/-/g;              # "-lb-tr-ss"
```

For matching in multiline strings, a trailing `s` allows a dot (`.`) to match an embedded newline (which it normally cannot). A trailing `m` allows `$` and `^` to match immediately before and after such a newline, respectively. A trailing `x` causes Perl to ignore both comments and embedded white space in the pattern, so that particularly complicated expressions can be broken across multiple lines, documented, and indented.

In the tradition of C and its relatives (Example 7.66, page 343), Perl allows nonprinting characters to be specified in REs using backslash *escape sequences*. These are summarized in the top portion of Figure 13.20. Perl also provides several zero-width assertions, in addition to the standard `^` and `$`. These are shown in the middle of the figure. The `\A` and `\Z` escapes differ from `^` and `$` in that they continue to match only at the beginning and end of the string, respectively, even in multiline searches that use the modifier `m`. Finally, Perl provides several built-in character classes, shown at the bottom of the figure. These can be used both inside and outside user-defined (i.e., bracket-delimited) classes. Note that `\b` has *different* meanings inside and outside such classes.

Greedy and Minimal Matches

EXAMPLE 13.54

Greedy and minimal matching

The usual rule for matching in REs is sometimes called "left-most longest": when a pattern can match at more than one place within a string, the chosen match will be the one that starts at the earliest possible position within the string, and then extends as far as possible. In the string `abcbcbcde`, for example, the pattern `/(bc)+/` can match in six different ways:

$\texttt{a}\underline{\texttt{bc}}\texttt{bcbcde}$

$\texttt{a}\underline{\texttt{bcbc}}\texttt{bcde}$

$\texttt{a}\underline{\texttt{bcbcbc}}\texttt{de}$

$\texttt{abc}\underline{\texttt{bc}}\texttt{bcde}$

$\texttt{abc}\underline{\texttt{bcbc}}\texttt{de}$

$\texttt{abcbc}\underline{\texttt{bc}}\texttt{de}$

The third of these is "left-most longest," also known as *greedy*. In some cases, however, it may be desirable to obtain a "left-most shortest" or *minimal* match. This corresponds to the first alternative above. ■

EXAMPLE 13.55

Minimal matching of HTML headers

We saw a more realistic example in Example 13.22 (Figure 13.4), which contains the following substitution:

```
s/.*?([hH][123]>.*?<\/[hH][123]>)//s;
```

Assuming that the HTML input is well formed, and that headers do not nest, this substitution deletes everything between the beginning of the string (implicitly `$_`) and the end of the first embedded header. It does so by using the `*?` quantifier instead of the usual `*`. Without the question marks, the pattern would match through (and the substitution would delete through) the end of the *last* header in the string. Recall that the trailing `s` modifier allows our headers to span lines.

In general, `*?` matches the smallest number of instances of the preceding subexpression that will allow the overall match to succeed. Similarly, `+?` matches at least one instance, but no more than necessary to allow the overall match to succeed, and `??` matches either zero or one instances, with a preference for zero. ■

Variable Interpolation and Capture

Like double-quoted strings, regular expressions in Perl support *variable interpolation.* Any dollar sign that does not immediately proceed a vertical bar, closing parenthesis, or end of string is assumed to introduce the name of a Perl variable,

EXAMPLE 13.56
Variable interpolation in extended REs

whose value as a string is expanded prior to passing the pattern to the regular expression evaluator. This allows us to write code that generates patterns at run time:

```
$prefix = ...
$suffix = ...
if ($foo =~ /^$prefix.*$suffix$/) ...
```

Note the two different roles played by `$` in this example. ■

EXAMPLE 13.57
Variable capture in extended REs

The flow of information can go the other way as well: we can pull the values of variables out of regular expressions. We saw a simple example in the `sed` script of Figure 13.1:

```
s/^.*\(<[hH][123]>\)/\1/        ;# delete text before opening tag
```

The equivalent in Perl would look something like this:

```
$line =~ s/^.*([hH][123]>)/\1/;
```

DESIGN & IMPLEMENTATION

Compiling regular expressions

Before it can be used as the basis of a search, a regular expression must be compiled into a deterministic or nondeterministic (backtracking) automaton. Patterns that are clearly constant can be compiled once, either when the program is loaded or when they are first encountered. Patterns that contain interpolated strings, however, must in the general case be recompiled whenever they are encountered, at potentially significant run-time cost. A programmer who knows that interpolated variables will never change can inhibit recompilation by attaching a trailing `o` modifier to the regular expression, in which case the expression will be compiled the first time it is encountered, and never thereafter. For expressions that must sometimes but not always be recompiled, the programmer can use the `qr` operator to force recompilation of a pattern, yielding a result that can be used repeatedly and efficiently:

```
for (@patterns) {             # iterate over patterns
    my $pat = qr($_);         # compile to automaton
    for (@strings) {          # iterate over strings
        if (/$pat/) {         # no recompilation required
            print;            # print all strings that match
            print "\n";
        }
    }
    print "\n";
}
```

Every parenthesized fragment of a Perl RE is said to *capture* the text that it matches. The captured strings may be referenced in the right-hand side of the substitution as \1, \2, and so on. Outside the expression they remain available (until the next substitution is executed) as $1, $2, and so on:

```
print "Opening tag: ", $1, "\n";
```

■

EXAMPLE 13.58
Backreferences in extended REs

One can even use a captured string later in the RE itself. Such a string is called a *backreference*:

```
if (/.*?([hH]([123])>.*?<\/[hH]\2>)/) {
    print "header: $1\n";
}
```

Here we have used \2 to insist that the closing tag of an HTML header match the opening tag. ■

EXAMPLE 13.59
Dissecting a floating-point literal

One can, of course capture multiple strings:

```
if (/^([+-]?)((\d+)\.|(\d*)\.(\d+))(e([+-]?\d+))?$/) {
    # floating-point number
    print "sign:     ", $1, "\n";
    print "integer:  ", $3, $4, "\n";
    print "fraction: ", $5, "\n";
    print "mantissa: ", $2, "\n";
    print "exponent: ", $7, "\n";
}
```

As in the previous example, the numbering corresponds to the occurrence of left parentheses, read from left to right. With input -123.45e-6 we see

```
sign:     -
integer:  123
fraction: 45
mantissa: 123.45
exponent: -6
```

Note that because of alternation, exactly one of $3 and $4 is guaranteed to be set. Note also that while we need the sixth set of parentheses for grouping (it has a ? quantifier), we don't really need it for capture. ■

EXAMPLE 13.60
Implicit capture of prefix, match, and suffix

For simple matches, Perl also provides pseudovariables named $‘, $&, and $’. These name the portions of the string before, in, and after the most recent match, respectively:

```
$line = <>;
chop $line;                     # delete trailing newline
$line =~ /is/;
print "prefix($‘) match($&) suffix($’)\n";
```

With input “now is the time”, this code prints

```
prefix(now ) match(is) suffix( the time)
```

■

✓ **CHECK YOUR UNDERSTANDING**

35. What popular scripting language uses dynamic scoping?
36. Summarize the strategies used in Perl, PHP, Ruby, and Python to determine the scope of variables that are not declared.
37. Describe the conceptual model for dynamically scoped variables in Perl.
38. List the principal features found in POSIX regular expressions, but not in the regular expressions of formal language theory (Section 2.1.1).
39. List the principal features found in Perl REs, but not in those of POSIX.
40. Explain the purpose of search *modifiers* (characters following the final delimiter) in Perl-type regular expressions.
41. Describe the three different categories of *escape sequences* in Perl-type regular expressions.
42. Explain the difference between *greedy* and *minimal* matches.
43. Describe the notion of *capture* in regular expressions.

13.4.3 Data Types

As we have seen, scripting languages don't generally require (or even permit) the declaration of types for variables. Most perform extensive run-time checks to make sure that values are never used in inappropriate ways. Some languages (e.g., Scheme, Python, and Ruby) are relatively strict about this checking; the programmer who wants to convert from one type to another must say so explicitly. If we type the following in Ruby,

EXAMPLE 13.61
Coercion in Ruby and Perl

```
a = "4"
print a + 3, "\n"
```

we get the following message at run time: "In '+': failed to convert Fixnum into String (TypeError)." Perl is much more forgiving. As we saw in Example 13.2, the program

```
$a = "4";
print $a . 3 . "\n";            # '.' is concatenation
print $a + 3 . "\n";            # '+' is addition
```

prints 43 and 7. ■

In general, Perl (and likewise Rexx and Tcl) takes the position that programmers should check for the errors they care about, and in the absence of such checks the program should do something reasonable. Perl is willing, for example, to accept the following (though it prints a warning if run with the -w compile-time switch):

EXAMPLE 13.62
Coercion and context in Perl

```
$a[3] = "1";                    # (array @a was previously undefined)
print $a[3] + $a[4], "\n";
```

Here `$a[4]` is uninitialized and hence has value `undef`. In a numeric context (as an operand of `+`) the string `"1"` evaluates to 1, and `undef` evaluates to 0. Added together, these yield 1, which is converted to a string and printed. ■

EXAMPLE 13.63
Explicit conversion in Ruby

A comparable code fragment in Ruby requires a bit more care. Before we can subscript `a` we must make sure that it refers to an array:

```
a = []                          # empty array assignment
a[3] = "1"
```

If the first line were not present (and `a` had not been initialized in any other way), the second line would have generated an "undefined local variable" error. After these assignments, `a[3]` is a string, but other elements of `a` are `nil`. We cannot concatenate a string and `nil`, nor can we add them (both operators are specified in Ruby using the operator `+`). If we want concatenation, and `a[4]` may be `nil`, we must say

```
print a[3] + String(a[4]), "\n"
```

If we want addition, we must say

```
print Integer(a[3]) + Integer(a[4]), "\n"
```

■

As these examples suggest, Perl (and likewise Tcl) uses a value model of variables. Scheme, Python, and Ruby use a reference model. PHP and JavaScript, like Java, use a value model for variables of primitive type and a reference model for variables of object type. The distinction is less important in PHP and JavaScript than it is in Java, because the same variable can hold a primitive value at one point in time and an object reference at another.

Numeric Types

As we have seen in Section 13.4.2, scripting languages generally provide a very rich set of mechanisms for string and pattern manipulation. Syntax and interpolation conventions vary, but the underlying functionality is remarkably consistent, and heavily influenced by Perl. The underlying support for numeric types shows a bit more variation across languages, but the programming model is again remarkably consistent: users are, to first approximation, encouraged to think of numeric values as "simply numbers," and not to worry about the distinction between fixed and floating point, or about the limits of available precision.

Internally, numbers in JavaScript are always double-precision floating point. In Tcl they are strings, converted to integers or floating-point numbers (and back again) when arithmetic is needed. PHP uses integers (guaranteed to be at least 32 bits wide), plus double-precision floating point. To these Perl and Ruby add arbitrary precision (multiword) integers, sometimes known as *bignums*. Python has bignums, too, plus support for complex numbers. Scheme has all of the above, plus precise rationals, maintained as ⟨numerator, denominator⟩ pairs. In all cases the interpreter "up-converts" as necessary when doing arithmetic on values with different representations, or when overflow would otherwise occur.

Perl is scrupulous about hiding the distinctions among different numeric representations. Most other languages allow the user to determine which is being used, though this is seldom necessary. Ruby is perhaps the most explicit about the existence of different representations: classes `Fixnum`, `Bignum`, and `Float` (double-precision floating point) have overlapping but not identical sets of built-in methods. In particular, integers have iterator methods, which floating-point numbers do not, and floating-point numbers have rounding and error checking methods, which integers do not. `Fixnum` and `Bignum` are both descendants of `Integer`.

Composite Types

The type constructors of compiled languages like C, Fortran, and Ada were chosen largely for the sake of efficient implementation. Arrays and records, in particular, have straightforward time- and space-efficient implementations, which we studied in Chapter 7. Efficiency, however, is less important in scripting languages. Designers have felt free to choose type constructors oriented more toward ease of understanding than pure run-time performance. In particular, most scripting languages place a heavy emphasis on *mappings*, sometimes called *dictionaries*, *hashes*, or *associative arrays*. As might be guessed from the third of these names, a mapping is typically implemented with a hash table. Access time for a hash remains $O(1)$, but with a significantly higher constant than is typical for a compiled array or record.

Perl, the oldest of the widely used scripting languages, inherits its principal composite types—the array and the hash—from `awk`. It also uses prefix characters on variable names as an indication of type: `$foo` is a scalar (a number, Boolean, string, or pointer [which Perl calls a "reference"]); `@foo` is an array; `%foo` is a hash; `&foo` is a subroutine; and plain `foo` is a filehandle or an I/O format, depending on context.

EXAMPLE 13.64
Perl arrays

Ordinary arrays in Perl are indexed using square brackets and integers starting with 0:

```
@colors = ("red", "green", blue");       # initializer syntax
print $colors[1];                        # green
```

Note that we use the `@` prefix when referring to the array as a whole, and the `$` prefix when referring to one of its (scalar) elements. Arrays are self-expanding: assignment to an out-of-bounds element simply makes the array larger (at the cost of dynamic memory allocation and copying). Uninitialized elements have the value `undef` by default. ■

EXAMPLE 13.65
Perl hashes

Hashes are indexed using curly braces and character string names:

```
%complements = ("red" => "cyan",
                "green" => "magenta", "blue" => "yellow");
print $complements{"blue"};             # yellow
```

These, too, are self-expanding.

Records and objects are typically built from hashes. Where the C programmer would write `fred.age = 19`, the Perl programmer writes `$fred{"age"} = 19`. In object-oriented code, `$fred` is more likely to be a reference, in which case we have `$fred->{"age"} = 19`. ■

EXAMPLE 13.66
Arrays and hashes in Python and Ruby

Python and Ruby, like Perl, provide both conventional arrays and hashes. They use square brackets for indexing in both cases, and distinguish between array and hash initializers (aggregates) using bracket and brace delimiters, respectively:

```
colors = ["red", "green", "blue"]
complements = {"red" => "cyan",
               "green" => "magenta", "blue" => "yellow"}
print colors[2], complements["blue"]
```

(This is Ruby syntax; Python uses `:` in place of `=>`.) ■

EXAMPLE 13.67
Array access methods in Ruby

As a purely object-oriented language, Ruby defines subscripting as syntactic sugar for invocations of the `[]` (get) and `[]=` (put) methods:

DESIGN & IMPLEMENTATION

Typeglobs in Perl

It turns out that a global name in Perl can have multiple independent meanings. It is possible, for example, to use `$foo`, `@foo`, `%foo`, `&foo` and two different meanings of `foo`, all in the same program. To keep track of these multiple meanings, Perl interposes a level of indirection between the symbol table entry for `foo` and the various values `foo` may have. The intermediate structure is called a *typeglob*. It has one slot for each of `foo`'s meanings. It also has a name of its own: `*foo`. By manipulating typeglobs, the expert Perl programmer can actually modify the table used by the interpreter to look up names at run time. The simplest use is to create an alias:

```
*a = *b;
```

After executing this statement, `a` and `b` are indistinguishable; they both refer to the same typeglob, and changes made to (any meaning of) one of them will be visible through the other. Perl also supports *selective* aliasing, in which *one slot* of a typeglob is made to point to a value from a different typeglob:

```
*a = \&b;
```

The backslash operator (`\`) in Perl is used to create a pointer. After executing this statement, `&a` (the meaning of `a` as a function) will be the same as `&b`, but all other meanings of `a` will remain the same. Selective aliasing is used, among other things, to implement the mechanism that imports names from libraries in Perl.

```
c = colors[2]                 # same as  c = colors.[](2)
colors[2] = c                 # same as  colors.[]=(2, c)
```

■

EXAMPLE 13.68
Tuples in Python

In addition to arrays (which it calls *lists*) and hashes (which it calls *dictionaries*), Python provides two other composite types: tuples and sets. A tuple is essentially an immutable list (array). The initializer syntax uses parentheses rather than brackets:

```
crimson = (0xdc, 0x14, 0x3c)    # R,G,B components
```

Tuples are more efficient to access than arrays: their immutability eliminates the need for most bounds and resizing checks. They also form the basis of multiway assignment:

```
a, b = b, a                     # swap
```

Parentheses can be omitted in this example: the comma groups more tightly than the assignment operator. ■

EXAMPLE 13.69
Sets in Python

Python sets are like dictionaries that don't map to anything of interest, but simply serve to indicate whether elements are present or absent. Unlike dictionaries, they also support union, intersection, and difference operations:

```
X = set(['a', 'b', 'c', 'd'])   # set constructor
Y = set(['c', 'd', 'e', 'f'])   #     takes array as parameter
U = X | Y                       # (['a', 'b', 'c', 'd', 'e', 'f'])
I = X & Y                       # (['c', 'd'])
D = X - Y                       # (['a', 'b'])
O = X ^ Y                       # (['a', 'b', 'e', 'f'])
'c' in I                        # True
```

■

EXAMPLE 13.70
Conflated types in PHP, Tcl, and JavaScript

PHP and Tcl have simpler composite types: they eliminate the distinction between arrays and hashes. An array is simply a hash for which the programmer chooses to use numeric keys. JavaScript employs a similar simplification, unifying arrays, hashes, and objects. The usual `obj.attr` notation to access a member of an object (what JavaScript calls a *property*) is simply syntactic sugar for `obj["attr"]`. So objects are hashes, and arrays are objects with integer property names. ■

EXAMPLE 13.71
Multidimensional arrays in Python and other languages

Higher-dimensional types are straightforward to create in most scripting languages: one can define arrays of (references to) hashes, hashes of (references to) arrays, and so on. Alternatively, one can create a "flattened" implementation by using composite objects as keys in a hash. Tuples in Python work particularly well:

```
matrix = {}                     # empty dictionary (hash)
matrix[2, 3] = 4                # key is (2, 3)
```

This idiom provides the appearance and functionality of multidimensional arrays, though not their efficiency. There exist extension libraries for Python that provide more efficient homogeneous arrays, with only slightly more awkward syntax. Numeric and statistical scripting languages, such as Maple, Mathematica, Matlab, and R, have much more extensive support for multidimensional arrays. ■

Context

In Section 7.2.2 we defined the notion of *type compatibility*, which determines, in a statically typed language, which types can be used in which *contexts*. In this definition the term "context" refers to information about how a value will be used. In C, for example, one might say that in the declaration

```
double d = 3;
```

the `3` on the right-hand side occurs in a context that expects a floating-point number. The C compiler *coerces* the `3` to make it a `double` instead of an `int`.

In Section 7.2.3 we went on to define the notion of *type inference*, which allows a compiler to determine the type of an expression based on the types of its constituent parts and, in some cases, the context in which it appears. We saw an extreme example in ML and its descendants, which use a sophisticated form of inference to determine types for most objects without the need for declarations.

In both of these cases—compatibility and inference—contextual information is used at compile time only. Perl extends the notion of context to drive decisions made at run time. More specifically, each operator in Perl determines, at compile time, and for each of its arguments, whether that argument should be interpreted as a *scalar* or a *list*. Conversely each argument (which may itself be a nested operator) is able to tell, at run time, which kind of context it occupies, and can consequently exhibit different behavior.

EXAMPLE 13.72
Scalar and list context in Perl

As a simple example, the assignment operator (`=`) provides a scalar or list context to its right-hand side based on the type of its left-hand side. This type is always known at compile time, and is usually obvious to the casual reader, because the left-hand side is a name and its prefix character is either a dollar sign (`$`), implying a scalar context, or an at (`@`) or percent (`%`) sign, implying a list context. If we write

```
$time = gmtime();
```

Perl's standard `gmtime()` library function will return the time as a character string, along the lines of `"Sun Aug 17 15:10:32 2008"`. On the other hand, if we write

```
@time_arry = gmtime();
```

the same function will return `(39, 09, 21, 15, 2, 105, 2, 73)`, an 8-element array indicating seconds, minutes, hours, day of month, month of year (with

January = 0), year (counting from 1900), day of week (with Sunday = 0), and day of year. ■

EXAMPLE 13.73
Using wantarray to determine calling context

So how does `gmtime` know what to do? By calling the built-in function `wantarray`. This returns true if the current function was called in a list context, and false if it was called in a scalar context. By convention, functions typically indicate an error by returning the empty array when called in a list context, and the undefined value (`undef`) when called in a scalar context:

```
if ( something went wrong ) {
    return wantarray ? () : undef;
}
```
■

13.4.4 Object Orientation

Though not an object-oriented language, Perl 5 has features that allow one to program in an object-oriented style.[10] PHP and JavaScript have cleaner, more conventional-looking object-oriented features, but both allow the programmer to use a more traditional imperative style as well. Python and Ruby are explicitly and uniformly object-oriented.

Perl uses a value model for variables; objects are always accessed via pointers. In PHP and JavaScript, a variable can hold either a value of a primitive type or a reference to an object of composite type. In contrast to Perl, however, these languages provide no way to speak of the reference itself, only the object to which it refers. Python and Ruby use a uniform reference model.

Classes are themselves objects in Python and Ruby, much as they are in Smalltalk. They are merely types in PHP, much as they are in C++, Java, or C#. Classes in Perl are simply an alternative way of looking at packages (namespaces). JavaScript, remarkably, has objects but no classes; its inheritance is based on a concept known as *prototypes*, initially introduced by the Self programming language.

Perl 5

Object support in Perl 5 boils down to two main things: (1) a *blessing* mechanism that associates a reference with a package, and (2) special syntax for method calls that automatically passes an object reference or package name as the initial argument to a function. While any reference can in principle be blessed, the usual convention is to use a hash, so that fields can be named as shown in Example 13.65.

EXAMPLE 13.74
A simple class in Perl

As a very simple example, consider the Perl code of Figure 13.21. Here we have defined a package, `Integer`, that plays the role of a class. It has three functions, one of which (`new`) is intended to be used as a constructor, and two of which (`set` and `get`) are intended to be used as accessors. Given this definition we can write

10 More extensive features, currently under design for Perl 6, will not be covered here.

```
{   package Integer;

    sub new {
        my $class = shift;      # probably "Integer"
        my $self = {};          # reference to new hash
        bless($self, $class);
        $self->{val} = (shift || 0);
        return $self;
    }
    sub set {
        my $self = shift;
        $self->{val} = shift;
    }
    sub get {
        my $self = shift;
        return $self->{val};
    }
}
```

Figure 13.21 **Object-oriented programming in Perl.** Blessing a reference (object) into package `Integer` allows `Integer`'s functions to serve as the object's methods.

```
$c1 = Integer->new(2);          # Integer::new("Integer", 2)
$c2 = new Integer(3);           # alternative syntax
$c3 = new Integer;              # no initial value specified
```

Both `Integer->new` and `new Integer` are syntactic sugar for calls to `Integer::new` with an additional first argument that contains the name of the package (class) as a character string. In the first line of function `new` we assign this string into the variable `$class`. (The `shift` operator returns the first element of pseudovariable `@_` [the function's arguments], and shifts the remaining arguments, if any, so they will be seen if `shift` is used again.) We then create a reference to a new hash, store it in local variable `$self`, and invoke the `bless` operator to associate it with the appropriate class. With a second call to `shift` we retrieve the initial value for our integer, if any. (The "or" expression [`||`] allows us to use 0 instead if no explicit argument was present.) We assign this initial value into the `val` field of `$self` using the usual Perl syntax to dereference a pointer and subscript a hash. Finally we return a reference to the newly created object. ■

EXAMPLE 13.75
Invoking methods in Perl

Once a reference has been blessed, Perl allows it to be used with method invocation syntax: `c1->get()` and `get c1()` are syntactic sugar for `Integer::get($c1)`. Note that this call passes a reference as the additional first parameter, rather than the name of a package. Given the declarations of `$c1`, `$c2`, and `$c3` above, the following code

```
print $c1->get, " ", $c2->get, " ", $c3->get, " ", "\n";
$c1->set(4);  $c2->set(5);  $c3->set(6);
print $c1->get, " ", $c2->get, " ", $c3->get, " ", "\n";
```

will print

```
2 3 0
4 5 6
```

As usual in Perl, if an argument list is empty, the parentheses can be omitted. ■

EXAMPLE 13.76
Inheritance in Perl

Inheritance in Perl is obtained by means of the `@ISA` array, initialized at the global level of a package. Extending the previous example, we might define a `Tally` class that inherits from `Integer`:

```
{   package Tally;
    @ISA = ("Integer");

    sub inc {
        my $self = shift;
        $self->{val}++;
    }
}
...
$t1 = new Tally(3);
$t1->inc;
$t1->inc;
print $t1->get, "\n";           # prints 5
```

The `inc` method of `t1` works as one might expect. However when Perl sees a call to `Tally::new` or `Tally::get` (neither of which is actually in the package), it uses the `@ISA` array to locate additional package(s) in which these methods may be found. We can list as many packages as we like in the `@ISA` array; Perl supports multiple inheritance. The possibility that `new` may be called through `Tally` rather than `Integer` explains the use of `shift` to obtain the class name in Figure 13.21. If we had used `"Integer"` explicitly we would not have obtained the desired behavior when creating a `Tally` object. ■

EXAMPLE 13.77
Inheritance via `use base`

Most often packages (and thus classes) in Perl are declared in separate modules (files). In this case, one must import the module corresponding to a superclass in addition to modifying `@ISA`. The standard `base` module provides convenient syntax for this combined operation, and is the preferred way to specify inheritance relationships:

```
{   package Tally;
    use base ("Integer");
    ...
```
■

PHP and JavaScript

While Perl's mechanisms suffice to create object-oriented programs, dynamic lookup makes them slower than equivalent imperative programs, and it seems fair to characterize the syntax as less than elegant. Objects are more fundamental to PHP and JavaScript.

PHP 4 provided a variety of object-oriented features, which were heavily revised in PHP 5. The newer version of the language provides a reference model of (class-typed) variables, interfaces and mix-in inheritance, abstract methods and classes, final methods and classes, static and constant members, and access control specifiers (`public`, `protected`, and `private`) reminiscent of those of Java, C#, and C++. In contrast to all other languages discussed in this subsection, class declarations in PHP must include declarations of all members (fields and methods), and the set of members in a given class cannot subsequently change (though one can of course declare derived classes with additional members).

JavaScript takes the unusual approach of providing objects—with inheritance and dynamic method dispatch—without providing classes. Such a language is said to be *object-based*, as opposed to object-oriented. Functions are first-class entities in JavaScript—objects, in fact. A method is simply a function that is referred to by a *property* (member) of an object. When we call `o.m`, the keyword `this` will refer to `o` during the execution of the function referred to by `m`. Likewise when we call `new f`, `this` will refer to a newly created (initially empty) object during the execution of `f`. A constructor in JavaScript is thus a function whose purpose is to assign values into properties (fields and methods) of a newly created object.

Associated with every constructor `f` is an object `f.prototype`. If object `o` was constructed by `f`, then JavaScript will look in `f.prototype` whenever we attempt to use a property of `o` that `o` itself does not provide. In effect, `o` inherits from `f.prototype` anything that it does not override. Prototype properties are commonly used to hold methods. They can also be used for constants or for what other languages would call "class variables."

EXAMPLE 13.78
Prototypes in JavaScript

Figure 13.22 illustrates the use of prototypes. It is roughly equivalent to the Perl code of Figure 13.21. Function `Integer` serves as a constructor. Assignments to properties of `Integer.prototype` serve to establish methods for objects constructed by `Integer`. Using the code in the figure, we can write

```
c2 = new Integer(3);
c3 = new Integer;

document.write(c2.get() + "&nbsp;&nbsp;" + c3.get() + "<BR>");
c2.set(4);  c3.set(5);
document.write(c2.get() + "&nbsp;&nbsp;" + c3.get() + "<BR>");
```

This code will print

```
3  0
4  5
```

EXAMPLE 13.79
Overriding instance methods in JavaScript

Interestingly, the lack of a formal notion of class means that we can override methods and fields on an object-by-object basis:

```
c2.set = new Function("n", "this.val = n * n;");
    // anonymous function constructor
c2.set(3);  c3.set(4);      // these call different methods!
document.write(c2.get() + "&nbsp;&nbsp;" + c3.get() + "<BR>");
```

```
function Integer(n) {
    this.val = n || 0;      // use 0 if n is missing (undefined)
}
function Integer_set(n) {
    this.val = n;
}
function Integer_get() {
    return this.val;
}
Integer.prototype.set = Integer_set;
Integer.prototype.get = Integer_get;
```

Figure 13.22 Object-oriented programming in JavaScript. The `Integer` function is used as a constructor. Assignments to members of its prototype object serve to establish methods. These will be available to any object created by `Integer` that doesn't have corresponding members of its own.

If nothing else has changed since the previous example, this code will print

```
9  4
```

■

EXAMPLE 13.80
Inheritance in JavaScript

To obtain the effect of inheritance, we can write

```
function Tally(n) {
    this.base(n);                 // call to base constructor
}
function Tally_inc() {
    this.val++;
}
Tally.prototype = new Integer;     // inherit methods
Tally.prototype.base = Integer;    // make base constructor available
Tally.prototype.inc = Tally_inc;   // new method
...
t1 = new Tally(3);
t1.inc();  t1.inc();
document.write(t1.get() + "<br>");
```

This code will print a 5. ■

Python and Ruby

As we have noted, both Python and Ruby are explicitly object-oriented. Both employ a uniform reference model for variables. Like Smalltalk, both incorporate an object hierarchy in which classes themselves are represented by objects. The root class in Python is called `object`; in Ruby it is `Object`.

EXAMPLE 13.81
Constructors in Python and Ruby

In both Python and Ruby, each class has a single distinguished constructor, which cannot be overloaded. In Python it is `__init__`; in Ruby it is `initialize`. To create a new object in Python one says `my_object = My_class(`*args*`)`; in

Ruby one says `my_object = My_class.new(`*args*`)`. In each case the *args* are passed to the constructor. To achieve the effect of overloading, with different numbers or types of arguments, one must arrange for the single constructor to inspect its arguments explicitly. We employed a similar idiom in Perl (in the `new` routine of Figure 13.21) and JavaScript (in the `Integer` function of Figure 13.22). ■

Both Python and Ruby are more flexible than PHP or more traditional object-oriented languages regarding the contents (members) of a class. New fields can be added to a Python object simply by assigning to them: `my_object.new_field = value`. The set of methods, however, is fixed when the class is first defined. In Ruby only methods are visible outside a class ("put" and "get" methods must be used to access fields), and all methods must be explicitly declared. It is possible, however, to modify an existing class declaration, adding or overriding methods. One can even do this on an object-by-object basis. As a result, two objects of the same class may not display the same behavior.

EXAMPLE 13.82
Naming class members in Python and Ruby

Python and Ruby differ in many other ways. The initial parameter to methods is explicit in Python; by convention it is usually named `self`. In Ruby `self` is a keyword, and the parameter it represents is invisible. Any variable beginning with

DESIGN & IMPLEMENTATION

Executable class declarations

Both Python and Ruby take the interesting position that class declarations are executable code. Elaboration of a declaration executes the code inside. Among other things, we can use this mechanism to achieve the effect of conditional compilation:

```
class My_class                    # Ruby code
    def initialize(a, b)
        @a = a;  @b = b;
    end
    if expensive_function()
        def get()
            return @a
        end
    else
        def get()
            return @b
        end
    end
end
```

Instead of computing the expensive function inside `get`, on every invocation, we compute it once, ahead of time, and define an appropriate specialized version of `get`.

a single `@` sign in Ruby is a field of the current object. Within a Python method, uses of object members must name the object explicitly. One must, for example, write `self.print()`; just `print()` will not suffice. ■

Ruby methods may be `public`, `protected`, or `private`.[11] Access control in Python is purely a matter of convention; both methods and fields are universally accessible. Finally, Python has multiple inheritance. Ruby has mix-in inheritance: a class cannot obtain data from more than one ancestor. Unlike most other languages, however, Ruby allows an interface (mix-in) to define not only the signatures of methods, but also their implementation (code).

✓ CHECK YOUR UNDERSTANDING

44. Contrast the philosophies of Perl and Ruby with regard to error checking and reporting.
45. Compare the numeric types of popular scripting languages to those of compiled languages like C or Fortran.
46. What are *bignums*? Which languages support them?
47. What are *associative arrays*? By what other names are they sometimes known?
48. Why don't most scripting languages provide direct support for records?
49. What is a *typeglob* in Perl? What purpose does it serve?
50. Describe the *tuple* and *set* types of Python.
51. Explain the unification of arrays and hashes in PHP and Tcl.
52. Explain the unification of arrays and objects in JavaScript.
53. Explain how tuples and hashes can be used to emulate multidimensional arrays in Python.
54. Explain the concept of *context* in Perl. How is it related to type compatibility and type inference? What are the two principal contexts defined by the language's operators?
55. Compare the approaches to object orientation taken by Perl 5, PHP 5, JavaScript, Python, and Ruby.
56. What is meant by the *blessing* of a reference in Perl?
57. What are *prototypes* in JavaScript? What purpose do they serve?

11 The meanings of `private` and `protected` in Ruby are different from those in C++, Java, or C#: `private` methods in Ruby are available only to the current instance of an object; `protected` methods are available to any instance of the current class or its descendants.

13.5 Summary and Concluding Remarks

Scripting languages serve primarily to control and coordinate other software components. Though their roots go back to interpreted languages of the 1960s, they have received relatively little attention from academic computer science. With an increasing emphasis on programmer productivity, however, and with the birth of the World Wide Web, scripting languages have seen enormous growth in interest and popularity, both in industry and in academia. Many significant advances have been made by commercial developers and by the open-source community. Scripting languages may well come to dominate programming in the 21st century, with traditional compiled languages more and more seen as special-purpose tools.

In comparison to their traditional cousins, scripting languages emphasize flexibility and richness of expression over sheer run-time performance. Common characteristics include both batch and interactive use, economy of expression, lack of declarations, simple scoping rules, flexible dynamic typing, easy access to other programs, sophisticated pattern matching and string manipulation, and high-level data types.

We began our chapter by tracing the historical development of scripting, starting with the command interpreter, or *shell* programs of the mid-1970s, and the text processing and report generation tools that followed soon thereafter. We looked in particular at the "Bourne-again" shell, `bash`, and the Unix tools `sed` and `awk`. We also mentioned such special-purpose domains as mathematics and

DESIGN & IMPLEMENTATION

Worse Is Better

Any discussion of the relative merits of scripting and "systems" languages invariably ends up addressing the tradeoffs between expressiveness and flexibility on the one hand and compile-time safety and performance on the other. It may also digress into questions of "quick-and-dirty" versus "polished" applications. An interesting take on this debate can be found in the widely circulated essays of Richard Gabriel (*www.dreamsongs.com/WorseIsBetter.html*). While working for Lucid Corp. in 1989, Gabriel found himself asking why Unix and C had been so successful at attracting users, while Common Lisp (Lucid's principal focus) had not. His explanation contrasts "The Right Thing," as exemplified by Common Lisp, with a "Worse Is Better" philosophy, as exemplified by C and Unix. "The Right Thing" emphasizes complete, correct, consistent, and elegant design. "Worse Is Better" emphasizes the rapid development of software that does most of what users need most of the time, and can be tuned and improved incrementally, based on field experience. Much of scripting, and Perl in particular, fits the "Worse Is Better" philosophy (Ruby and Scheme enthusiasts might beg to disagree). Gabriel, for his part, says he still hasn't made up his mind; his essays argue both points of view.

statistics, where scripting languages are widely used for data analysis, visualization, modeling, and simulation. We then turned to the three domains that dominate scripting today: "glue" (coordination) applications, configuration and extension, and scripting of the World Wide Web.

In terms of "market share," Perl is almost certainly the most popular of the general-purpose scripting languages, widely used for report generation, glue, and server-side (CGI) web scripting. Python and Ruby both appear to be growing in popularity, and Tcl retains a strong core of support. Several scripting languages, including Scheme, Python, and Tcl, are widely used to extend the functionality of complex applications. In addition, many commercial packages have their own proprietary extension languages. Visual Basic has historically been the language of choice for scripting on Microsoft platforms, but will probably give way over time to C# and the various cross-platform options.

Web scripting comes in many forms. On the server side of an HTTP connection, the Common Gateway Interface (CGI) standard allows a URI to name a program that will be used to generate dynamic content. Alternatively, web-page–embedded scripts, often written in PHP, can be used to create dynamic content in a way that is invisible to users. To reduce the load on servers, and to improve interactive responsiveness, scripts can also be executed within the client browser. JavaScript is the dominant notation in this domain; it uses the HTML Document Object Model (DOM) to manipulate web-page elements. For more demanding tasks, most browsers can be directed to run a Java *applet*, which takes full responsibility for some portion of the "screen real estate." With the continued evolution of the Web, XML is likely to become the standard vehicle for storing and transmitting structured data. XSL, the Extensible Stylesheet Language, will then play a major role in transforming and formatting dynamic content.

Because of their rapid evolution, scripting languages have been able to take advantage of many of the most powerful and elegant mechanisms described in previous chapters, including first-class and higher-order functions, garbage collection, unlimited extent, iterators, list comprehensions, and object orientation—not to mention extended regular expressions and such high-level data types as dictionaries, sets, and tuples. Given current technological trends, scripting languages are likely to become increasingly ubiquitous, and to remain a principal focus of language innovation.

13.6 Exercises

13.1 Does filename "globbing" provide the expressive power of standard regular expressions? Explain.

13.2 Write shell scripts to

(a) Replace blanks with underscores in the names of all files in the current directory.

(b) Rename every file in the current directory by prepending to its name a textual representation of its modification date.

(c) Find all `eps` files in the file hierarchy below the current directory, and create any corresponding `pdf` files that are missing or out of date.

(d) Print the names of all files in the file hierarchy below the current directory for which a given predicate evaluates to true. Your (quoted) predicate should be specified on the command line using the syntax of the Unix `test` command, with one or more at signs (`@`) standing in for the name of the candidate file.

13.3 In Example 13.15 we used `"$@"` to refer to the parameters passed to `ll`. What would happen if we removed the quote marks? (Hint: try this for files whose names contain spaces!) Read the `man` page for `bash` and learn the difference between `$@` and `$*`. Create versions of `ll` that use `$*` or `"$*"` instead of `"$@"`. Explain what's going on.

13.4 (a) Extend the code in Figure 13.5, 13.6, 13.7, or 13.8 to try to kill processes more gently. You'll want to read the `man` page for the standard `kill` command. Use a `TERM` signal first. If that doesn't work, ask the user if you should resort to `KILL`.

(b) Extend your solution to part (a) so that the script accepts an optional argument specifying the signal to be used. Alternatives to `TERM` and `KILL` include `HUP`, `INT`, `QUIT`, and `ABRT`.

13.5 Write a Perl, Python, or Ruby script that creates a simple *concordance*: a sorted list of significant words appearing in an input document, with a sublist for each that indicates the lines on which the word occurs, with up to six words of surrounding context. Exclude from your list all common articles, conjunctions, prepositions, and pronouns.

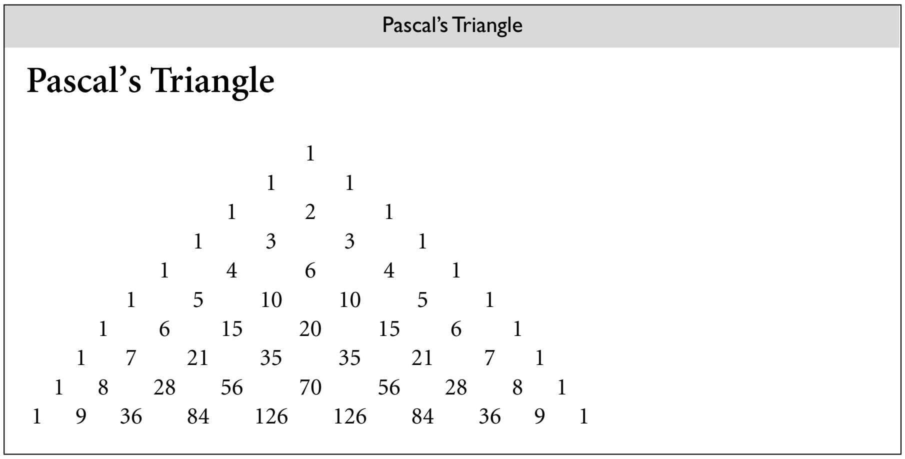
13.6 Write Emacs Lisp scripts to

(a) Insert today's date into the current buffer at the insertion point (current cursor location).

(b) Place quote marks (`" "`) around the word surrounding the insertion point.

(c) Fix end-of-sentence spaces in the current buffer. Use the following heuristic: if a period, question mark, or exclamation point is followed by a single space (possibly with closing quote marks, parentheses, brackets, or braces in-between), then add an extra space, unless the character preceding the period, question mark, or exclamation point is a capital letter (in which case we assume it is an abbreviation).

(d) Run the contents of the current buffer through your favorite spell checker, and create a new buffer containing a list of misspelled words.



Pascal's Triangle

Pascal's Triangle

```
                                  1
                              1       1
                          1       2       1
                      1       3       3       1
                  1       4       6       4       1
              1       5      10      10       5       1
          1       6      15      20      15       6       1
      1       7      21      35      35      21       7       1
   1      8      28      56      70      56      28       8      1
1     9     36      84     126     126      84      36      9     1
```

Figure 13.23 Pascal's triangle rendered in a web page (Exercise 13.8).

(e) Delete one misspelled word from the buffer created in (d), and place the cursor (insertion point) on top of the first occurrence of that misspelled word in the current buffer.

13.7 Explain the circumstances under which it makes sense to realize an interactive task on the Web as a CGI script, an embedded server-side script, or a client-side script. For each of these implementation choices, give three examples of tasks for which it is clearly the preferred approach.

13.8 (a) Write a web page with embedded PHP to print the first 10 rows of Pascal's triangle (see Example ◎16.10 if you don't know what this is). When rendered, your output should look like Figure 13.23.

(b) Modify your page to create a self-posting form that accepts the number of desired rows in an input field.

(c) Rewrite your page in JavaScript.

13.9 Create a fill-in web form that uses a JavaScript implementation of the Luhn formula (Exercise 4.10) to check for typos in credit card numbers. (But don't use real credit card numbers; homework exercises don't tend to be very secure!)

13.10 (a) Modify the code of Figure 13.16 (Example 13.35) so that it replaces the form with its output, as the CGI and PHP versions of Figures 13.12 and 13.15 do.

(b) Modify the CGI and PHP scripts of Figures 13.12 and 13.15 (Examples 13.30 and 13.34) so they appear to append their output to the bottom of the form, as the JavaScript version of Figure 13.16 does.

13.11 Run the following program in Perl:

```
sub foo {
    my $lex = $_[0];
    sub bar {
        print "$lex\n";
    }
    bar();
}

foo(2);  foo(3);
```

You may be surprised by the output. Perl 5 allows named subroutines to nest, but does not create closures for them properly. Rewrite the code above to create a reference to an anonymous local subroutine and verify that it does create closures correctly. Add the line `use diagnostics;` to the beginning of the original version and run it again. Based on the explanation this will give you, speculate as to how nested named subroutines are implemented in Perl 5.

13.12 Write a program that will map the web pages stored in the file hierarchy below the current directory. Your output should itself be a web page, containing the names of all directories and `.html` files, printed at levels of indentation corresponding to their level in the file hierarchy. Each `.html` filename should be a live link to its file. Use whatever language(s) seem most appropriate to the task.

13.13 In Section 13.4.1 we claimed that nested blocks in Ruby were part of the named scope in which they appear. Verify this claim by running the following Ruby script and explaining its output:

```
def foo(x)
    y = 2
    bar = proc {
        print x, "\n"
        y = 3
    }
    bar.call()
    print y, "\n"
end

foo(3)
```

Now comment out the second line (`y = 2`) and run the script again. Explain what happens. Restate our claim about scoping more carefully and precisely.

13.14 Write a Perl script to translate English measurements (in, ft, yd, mi) into metric equivalents (cm, m, km). You may want to learn about the `e` modifier on regular expressions, which allows the right-hand side of an `s///e` expression to contain executable code.

13.15 Write a Perl script to find, for each input line, the longest substring that appears at least twice within the line, without overlapping. (*Warning*: This is harder than it sounds. Remember that by default Perl searches for a *left-most longest* match.)

13.16 Perl provides an alternative `(?:...)` form of parentheses that supports grouping in regular expressions without performing capture. Using this syntax, Example 13.59 could have been written as follows:

```
if (/^([+-]?)((\d+)\.|(\d*)\.(\d+))(?:e([+-]?\d+))?$/) {
    # floating-point number
    print "sign:     ", $1, "\n";
    print "integer:  ", $3, $4, "\n";
    print "fraction: ", $5, "\n";
    print "mantissa: ", $2, "\n";
    print "exponent: ", $6, "\n";            # not $7
}
```

What purpose does this extra notation serve? Why might the code here be preferable to that of Example 13.59?

13.17 Consider again the `sed` code of Figure 13.1. It is tempting to write the first of the compound statements as follows (note the differences in the three substitution commands):

```
/<[hH][123]>.*<\/[hH][123]>/ {  ;# match whole heading
    h                           ;# save copy of pattern space
    s/^.*\(<[hH][123]>\)/\1/    ;# delete text before opening tag
    s/\(<\/[hH][123]>\).*$/\1/  ;# delete text after closing tag
    p                           ;# print what remains
    g                           ;# retrieve saved pattern space
    s/^.*<\/[hH][123]>//        ;# delete through closing tag
    b top
```

Explain why this doesn't work. (Hint: remember the difference between *greedy* and *minimal* matches [Example 13.55]. `Sed` lacks the latter.)

13.18 Consider the following regular expression in Perl: `/^(?:((?:ab)+)|a((?:ba)*))$/`. Describe, in English, the set of strings it will match. Show a natural NFA for this set, together with the minimal DFA. Describe the substrings that should be captured in each matching string. Based on this example, discuss the practicality of using DFAs to match strings in Perl.

13.19–13.21 In More Depth.

13.7 Explorations

13.22 Learn about the Scheme shell, `scsh`. Compare it to `sh/bash`. Which would you rather use from the keyboard? Which would you rather use for scripting?

13.23 Learn about TEX [Knu86] and LATEX [Lam94], the typesetting system used to create this book. Explore the ways in which its specialized target domain—professional typesetting—influenced its design. Features you might wish to consider include dynamic scoping, the relatively impoverished arithmetic and control-flow facilities, and the use of macros as the fundamental control abstraction.

13.24 Research the security mechanisms of JavaScript and/or Java applets. What exactly are programs allowed to do and why? What potentially useful features have not been provided because they can't be made secure? What potential security holes remain in the features that *are* provided?

13.25 Learn about *web crawlers*—programs that explore the World Wide Web. Build a crawler that searches for something of interest. What language features or tools seem most useful for the task? *Warning*: Automated web-crawling is a public activity, subject to strict rules of etiquette. Before creating a crawler, do a web search and learn the rules, and test your code *very* carefully before letting it outside your local subnet (or even your own machine). In particular, be aware that rapid-fire requests to the same server constitute a *denial of service attack*, a potentially criminal offense.

13.26 In the sidebar on page 699 we noted that the "extended" REs of `awk` and `egrep` are typically implemented by translating first to an NFA and then to a DFA, while those of Perl et al. are typically implemented via backtracking search. Some tools, including GNU `ggrep`, use a variant of the Boyer-Moore-Gosper algorithm [BM77, KMP77] for faster deterministic search. Find out how this algorithm works. What are its advantages? Could it be used in languages like Perl?

13.27 In the sidebar on page 702 we noted that nonconstant patterns must generally be recompiled whenever they are used. Perl programmers who wish to reduce the resulting overhead can inhibit recompilation using the `o` trailing modifier or the `qr` quoting operator. Investigate the impact of these mechanisms on performance. Also speculate as to the extent to which it might be possible for the language implementation to determine, automatically and efficiently, when recompilation should occur.

13.28 Our coverage of Perl REs in Section 13.4.2 was incomplete. Features not covered include look-ahead and look-behind (context) assertions, comments, incremental enabling and disabling of modifiers, embedded code, conditionals, Unicode support, nonslash delimiters, and the transliteration (`tr///`) operator. Learn how these work. Explain if (and how) they extend the expressive power of the notation. How could each be emulated (possibly with surrounding Perl code) if it were not available?

13.29 Investigate the details of RE support in PHP, Tcl, Python, Ruby, JavaScript, Emacs Lisp, Java, and C#. Write a paper that documents, as concisely as possible, the differences among these, using Perl as a reference for comparison.

13.30 Do a web search for Perl 6, which seems to be nearing completion as of summer 2008. Write a report that summarizes the changes with respect to Perl 5. What do you think of these changes? If you were in charge of the revision, what would you do differently?

13.31–13.33 In More Depth.

13.8 Bibliographic Notes

Most of the major scripting languages are described in books by the language designers or their close associates: `awk` [AKW88], Perl [WCO00], PHP [LT02], Tcl [Ous94, WJH03], Python [vRD03], and Ruby [TFH04]. Several of these have versions available on-line. Most of the languages are also described in a variety of other texts, and most have dedicated web sites: *perl.com*, *php.net*, *tcl.tk*, *python.org*, *ruby-lang.org*. Extensive documentation for Perl is available on-line at many sites; type `man perl` for an index.

Rexx [Ame96a] has been standardized by ANSI, the American National Standards Institute. JavaScript [ECM99] has been standardized by ECMA, the European standards body. Scheme implementations intended for scripting include Elk (*www-rn.informatik.uni-bremen.de/software/elk/* and *http://sam.zoy.org/projects/elk/*), Guile (*gnu.org/software/guile/*), and SIOD (Scheme in One Defun) (*people.delphiforums.com/gjc/siod.html*). Standards for the World Wide Web, including HTML, XML, XSL, XPath, and XHTML, are promulgated by the World Wide Web Consortium: *www.w3.org*. For those experimenting with the conversion to XHTML, the validation service at *validator.w3.org* is particularly useful. High-quality tutorials on many web-related topics can be found at *w3schools.com*.

Hauben and Hauben [HH97a] describe the historical roots of the Internet, including early work on Unix. Original articles on the various Unix shell languages include those of Mashey [Mas76], Bourne [Bou78], and Korn [Kor94]. Information on the Scheme shell, `scsh`, is available at *scsh.net*. The original reference on APL is by Iverson [Ive62]. Ousterhout [Ous98] makes the case for scripting languages in general, and Tcl in particular. Chonacky and Winch [CW05] compare and contrast Maple, Mathematica, and Matlab. Richard Gabriel's collection of "Worse Is Better" papers can be found at *www.dreamsongs.com/WorseIsBetter.html*. A similar comparison of Tcl and Scheme can be found in the introductory chapter of Abelson, Greenspun, and Sandon's on-line *Tcl for Web Nerds* guide (*philip.greenspun.com/tcl/index.adp*).