Title: The Perl Scripting Language

Author: David Stotts
Affiliation: Department of Computer Science
University of North Carolina at Chapel Hill
stotts@cs.unc.edu

Final Draft: December 20, 2002

Outline

Glossary

CGI Common Gateway Interface, standard for Web server scripting
CPAN Comprehensive Perl Archive Network, repository for useful Perl code and documentation
Hash Associate array, a collection of data items indexed by string (scalar) values
HTTP HyperText Transfer Protocol, encoding standard for Web transactions
HTML HyperText Markup Language, language for specifying Web page content
IPC Interprocess Communication
Linux Popular open source version of Unix for PCs
Pipe Unix name for a communication channel between processes
Regular expression formal language for specifiying complex patterns and strings of characters
Scalar Singular data value such as integer, string, boolean, real
Unix Popular operating system developed at Bell Labs, UC Berkeley in the late 70's

Abstract

This article gives an overview of the Perl scripting language and shows how it is used for programming Internet and Web applications. The history of the language is presented along with its Unix heritage. Basic language features are explained with examples of Perl scripts for string manipulation, pattern matching, text processing, and system interactions. The Perl community module repository *CPAN* is discussed, and several of the popular Perl modules are used to illustrate Web client, Web server, database, and network programming. Two newer scripting languages – Python and Ruby – are briefly discussed and compared to Perl. Finally, many of Perl's most advanced features are mentioned but not discussed, so references are provided for readers who wish further study.

1 Introduction

From its introduction to the programming community in 1987, Perl has become today one of the most widely known and used programming languages. Designed by Larry Wall, and originally thought of as a natural enhancement for the popular *csh* shell script notation of Unix, Perl was at first primarily used for text manipulation. Its maturity in the early 90's coincided with the rise of the Web, and it rapidly became the most popular programming language for HTML form processing and other Web development as well.

Perl has been called a "Swiss Army chainsaw" for its plethora of features coupled with its considerable programming power and flexibility. The common phrase among hardened Perl programmers is "there's more than one way to do it." Most programming goals can be achieved in Perl in at least three ways, depending on which language features and techniques the programmer prefers to use. It is not uncommon for an experienced Perl programmer to reach for the manual when reading code written by another programmer. Perl has also been called "duct tape for the Web" emphasizing its utility for producing applications, web sites, and general program fixes for a wide variety of problems and domains.

In this article we give a brief history of Perl, including major events preceding Perl that set the historical stage for it. We provide an overview of the language, including example code to show how its features are used in practice. We discuss Web site programming in Perl using the CGI (Common Gateway Interface) standard, and show several database interface methods in Perl. We discuss the community of programmers that has grown up around Perl, and conclude with a presentation of several technologies that are logical follow-ons to Perl.

2 A Brief History of Perl

Perl grew out of the Unix programming community. Though it did not formally appear until the late 80's, the technical components and motivations for Perl were developed in the two decades prior to that. Here are the main events in the "genealogy" of Perl:

1969 Unix is created at Bell Labs
1977 awk is invented by Aho, Weinberger, and Kernighan
1978 "sh" shell is developed for Unix
1987 Perl is created
1995 (March) Perl 5.001 released, the most recent major version;
as of this writing, Perl version 5.8.0 is the newest download at http://www.perl.com

The "Unix philosophy" of software construction, at least in the early days of that operating system, was to provide users with a large toolbox of useful "filters" – programs that could do one small task well – and then compose a larger program from the smaller ones. The shell script notations *sh* and *csh* were the means by which composition was done; *sed, awk, tr*, and other programs were some of the more commonly used filters. Perl was developed ostensibly to solve a problem in text processing that *awk* was not good at, and has continued to evolve from there.

To summarize Perl completely and succinctly, we probably cannot do much better than this excerpt from the original Unix help file:

> *Perl is (an) interpreted language optimized for scanning arbitrary text files, extracting information from those text files, and printing reports based on that information. It's also a good language for many system management tasks. The language is intended to be practical (easy to use, efficient,*

complete) rather than beautiful (tiny, elegant, minimal). It combines (in the author's opinion, anyway) some of the best features of C, sed, awk, and sh, so people familiar with those languages should have little difficulty with it. (Language historians will also note some vestiges of csh, Pascal, and even BASIC|PLUS.) Expression syntax corresponds quite closely to C expression syntax. If you have a problem that would ordinarily use sed or awk or sh, but it exceeds their capabilities or must run a little faster, and you don't want to write the silly thing in C, then perl may be for you. There are also translators to turn your sed and awk scripts into perl scripts. OK, enough hype.

Larry Wall is a trained linguist and this interest and expertise shows in Perl. Here he summarizes the nature and intent of his language, and his design rationale:

When they first built the University of California at Irvine campus, they just put the buildings in. They did not put any sidewalks, they just planted grass. The next year, they came back and built the sidewalks where the trails were in the grass. Perl is that kind of a language. It is not designed from first principles. Perl is those sidewalks in the grass. Those trails that were there before were the previous computer languages that Perl has borrowed ideas from. And Perl has unashamedly borrowed ideas from many, many different languages. Those paths can go diagonally. We want shortcuts. Sometimes we want to be able to do the orthogonal thing, so Perl generally allows the orthogonal approach also. But it also allows a certain number of shortcuts, and being able to insert those shortcuts is part of that evolutionary thing.

I don't want to claim that this is the only way to design a computer language, or that everyone is going to actually enjoy a computer language that is designed in this way. Obviously, some people speak other languages. But Perl was an experiment in trying to come up with not a large language -- not as large as English -- but a medium-sized language, and to try to see if, by adding certain kinds of complexity from natural language, the expressiveness of the language grew faster than the pain of using it. And, by and large, I think that experiment has been successful.

… Larry Wall, in Dr. Dobbs Journal, Feb. 1998

In its early versions, Perl was simple and much closer to the scripting notations from which it grew. In later versions, as with many languages, Perl began to accumulate features and facilities as advocates tried to make it more general purpose and keep it in step with object-oriented language developments.

3 Perl Language Overview

A discussion of the programming features and facilities of Perl is in order before we present the areas in which Perl can be applied. This will be an overview, not a tutorial, so no attempt is made to provide an exhaustive or in-depth treatment. Components of Perl that are unique or unusual will be emphasized at the expense of features common to many languages.

Perl is an interpreted language, meaning that a control program that understands the semantics of the language and its components (the *interpreter*) executes program components individually as they are encountered in the control flow. Today this usually is done by first translating the source code into an intermediate representation -- called *bytecode* -- and then interpreting the bytecode. Interpreted execution makes Perl flexible, convenient, and fast for programming, with some penalty paid in execution speed.

Perl programs are often called *scripts* because of its historical development as an extension of the Unix command-level command scripting notations. A Perl script consists of a series of *declarations* and *statements* with intersperced *comments*. A *declaration* gives the interpreter type information and reserves storage for data. Each *statement* is a command that is recognized by the Perl interpreter and executed. Every statement is usually terminated by a semicolon, and in keeping with most modern syntax, white space between words is not significant. A *comment* begins with the # character and can appear anywhere; everything from the # to the end of the line is ignored by the Perl interpreter. [1]

---

[1] Though the Perl language proper does not have multi-line (block) comments, the effect can be achieved using POD (Plain Old Documentation) directives that are ignored by the interpreter. POD directives begin with an "=" and can appear anywhere the interpreter expects a statement to start. They allow various forms of documentation and text markup to be embedded in Perl scripts, and are meant to be processed by separate POD tools. A block comment can be done by opening it with a directive like "`=comment`" and ending it with "`=cut`". All lines in between are ignored.

Perl is considered by some to have convoluted and confusing syntax; others consider this same syntax to be compact, flexible, and elegant. Though the language has most of the features one would expect in a "full service" programming notation, Perl has become well known for its capabilities in a few areas where it exceeds the capabilities of most other languages. These include

- String manipulation
- File handling
- Regular expressions and pattern matching
- Flexible arrays (hashes, or associative arrays)

In the following sections we present the basics of core language, with emphasis on the features that Perl does especially well.

3.1 Basic Data Types and Values

Perl provides three types of data for programmers to manipulate: *scalar, array*, and *hash* (*associative array*). Scalar types are well known to most programmers, as is the array type. The hash is less well known and one of the most powerful aspects of Perl (along with pattern matching, discussed later). Scalar values include *integer, real, string*, and *boolean*, with the expected operations available for each. Variables do not have to be declared before use; the first character indicates the type.

Scalars Scalar variables have a leading "$"; for example, these are all valid Perl scalar variables:

```
$n   $N   $var28   $hello_World   $_X_   $_
```

Case matters in Perl, so the first two examples are different variables. The final variable "$_" is special, one of many that the Perl interpreter will use by default in various operations if the programmer does not indicate otherwise. Any particular valid identifier can be used to designate a scalar, an array, *and* a hash. The leading character determines which of the three types the variable has. For example, here the identifier "name" is used to denote three different variables:

```
$name   @name   %name
```

The first is a scalar; the second is an array; the last is a hash. All three can be used concurrently, as they denote different storage areas. A variable of type scalar can contain any scalar value:

```
$v1 = "good morning";
$v1 = 127;
```

The first assignment places a string value in the variable. The second replaces the string with an integer value. This is different from many strongly typed languages (like C++ and Java) where types are finely divided into categories like integer, real, string, and boolean. In Perl these are values, but not types; Perl uses type distinction mainly to separate singular entities (scalar) from collective entities (arrays and hashes).

String values are delimited with either single or double quotes. Single quoted literals are used exactly as written, whereas double quoted literals are subject to escape character and variable interpolation before the final value is obtained. For example:

```
$numer = 2;
$st1 = 'one fine $numer day';
$st2 = "$numer fine day \n";
print $st2;
print "$st1\n";
```

The output from this script is

```
2 fine day
one fine $numer day
```

Four interpolations have taken place. In the second line, the `$numer` appears to be a use of a variable, but since the string is in single quotes the characters are included as they appear. In the third line the string literal is in double quotes, so the variable name is replaced with its current value (2) to produce the final string value; the escape sequence "\n" also puts in a *newline* character. The first *print* statement shows the result. The second *print* shows two interpolations as well, as the string being printed is in double quotes. The value in

`$st1` is interpolated into the output string and then a *newline* is added to the end. Even though `$st1` has '`$numer`' in its value, this is not recursively interpolated when those characters are put into the output string.

Context Many of the operators in Perl will work on all three types, with different results being produced depending on the form of the operands. This polymorphism is known in Perl as *context*. There are two major contexts: *scalar* and *list*. Scalar context is further classified as *numeric, string,* or *boolean*. Consider a scalar variable that is assigned an integer value. If the same variable is later used in a string context (meaning operated on by a string function), the integer value is automatically treated as a string of ASCII characters representing the digits of the integer. For example:

```
$v1 = 127;
$v1 =  $v1 . ", and more !!";
print $v1, "\n";
$v1 = 127;
print $v1 + " 151 ", "\n";
print $v1 + ", and more !!", "\n";
```

The output from these statements is

```
127, and more !!
278
127
```

The "." operator in the second assignment performs string concatenation, making the expression have string context; the interpreter therefore treats the value of $v as an ASCII string of digits, not as an integer. Since integers and strings are both scalars, we can store the resulting string value back into $v, which previously held an integer value. The "+" in the second *print* is an arithmetic operator, giving that expression numeric context; the interpreter converts the string " 151 " to the obvious integer value for addition. The final *print* is also a numeric context, but there is no obvious valid integer value for the string ", and more !!" so a zero is used for the addition.

Arrays Use of array variables in expressions can cause some confusion. In fact, Perl's somewhat convoluted syntax is one of the main complaints against the language[2]. When an array is manipulated collectively, the leading "@" notation is used:

```
@A = ("hi", "low", 17, 2.14159, "medium");
@A = @B;
print "$B[1] \n";
```

This code fragment outputs "`low`" on one line. The first statement creates an array A by assigning the members of the list to consecutive array elements; the second line then sets another array B to have all the same values. Finally, it prints the 2nd element from B (array indexes start at 0 in Perl). Arrays contain scalars as elements, so integer, real, boolean, and string values can be stored in the same array. Note also that when individual elements in an array are manipulated, the scalar notation "$" is used; the reasoning is that the element itself is not an array, but is a scalar. Using this notation, individual array elements can be given values via assignment.

Array references can be used anywhere a scalar can be, such as in a subscript expression. If an array subscript expression produces a scalar that is not an integer (such as string or real) Perl converts it to some reasonable integer interpretation:

```
$A[0] = 72;
$A[4] = "moderate exercise";
$A[$i] = $B[$j];
print "$A[$A[3]]\n";
```

Here the last line produces "17" on one line. The inner expression evaluates to 2,14159; to use this as a subscript Perl takes the integer part. Thus, it prints $A[2], which is the scalar 17.

Hashes Hashes (associative arrays) have elements that are indexed by any scalar (usually strings) rather than by integer value:

---

[2] "It's the magic that counts," quipped Larry Wall on one occasion, when this feature of the syntax was publicly noted.

```
$assoc{"first"} = 37;
$assoc{'second'} = 82;
$assoc{"third"} = "3_rd";
```

Syntactically, subscripts are delimited with curly braces rather than brackets, and string constants used for subscripts can be delimited with single or double quotes. Elements are retrieved from an associative array in similar fashion:

```
$str = "first";
$N = $assoc{$str};
print "$assoc{'second'} - $N \n";
print %assoc, "\n";
```

The output from this segment is

```
82 - 37
first37third3_rdsecond82
```

The last line shows that accessing the entire associative array is possible with the leading "%", and that when printed this way, the output shows up as (index,value) pairs in arbitrary order (determined by the interpreter). Note also in the next to last line that scalar variables can appear inside strings; they are *interpolated*, that is, replaced by their values in the value of the string. Input/output is discussed in more detail later.

3.2 Basic operations

Scalar Scalar values can be manipulated by the common operators you would expect of a modern programming language. Numbers have arithmetic operations and logical comparisons, autoincrement and decrement (++ and --), and operator assignments (+=, -=, *=). Strings have lexical comparison operations, as well as concatenation, truncation, substring extraction and indexing operations. Booleans have the expected logical operators, and the conjunction (&&) and disjunction (||) are evaluated clause-by-clause and will short-circuit as soon as the final expression value can be determined.

Array Perl has the expected assignment and referencing operators for arrays; it also provides subrange operators to use part of an array. $#arr3 will give you the scalar value that is the last index used in array @arr3; since Perl indexes arrays from 0, $#arr3 + 1 will give the array length. Several predefined functions allow a programmer to use arrays as implementations of other data abstractions. For example, *push(@ar7, $elt)* and *pop(@arr7)* will treat the array @arr7 as a stack; *reverse, sort, shift, join, splice,* and *map* are other predefined functions on arrays.

Hash (associative array) Hashes have assignment and multi-assignment to create attribute/value pairs, and have array referencing via scalar subscripts (usually strings) to retrieve the value associated with an attribute. Most other operations are provided as functions on the array name. For example, *keys (%aa2)* will return a list of the subscript strings for which there are values in the hash *aa2*. Other such operations are *value*s, *each*, and *delete*.

3.3 Control Flow

Aside from syntactic differences, Perl has much the same *while*, *until*, and *for* loop structures most programming languages have. In keeping with the stated goal of being flexible and not limiting, however, Perl allows several forms of limited jumps within the context of loops that many other languages do not.

If / elsif / else The traditional *if / then / else* conditional statement is altered a bit in Perl. There is no *then* keyword required on the true clause, and following that may be nothing, an *else* clause, or an *elsif* clauses. An *elsif* clause flattens the decision tree that would otherwise be formed by having another *if/else* as the body of an *else* clause. Perl lacks a *case* statement, so the *elsif* functions in this capacity, as in this example:

```
if ($thresh < 10) {
     # … the 'then' block of the conditional
} elsif ($thresh < 20) {
     # the next block in the decision tree
} elsif ($thresh < 40) {
     # and the next…
} else {
```

```
        # the final clause catches what falls through
}
```

The negation shorthand *unless(exp)* can be used for *if (!exp)* in all contexts where the *if* keyword is valid.

Expressions and do blocks In Perl, statements are viewed as expressions, and executing a statement produces a value for that expression. Every value can also, by convention, be interpreted as a truth value. Any empty string, the number 0, and the string "0" are all treated as "false"; other values are treated as "true" (with a few exceptions). For example, executing the assignment *$a = 27* has the effect of setting the value of variable $a, but it also produces the value 27 as the result of the expression. If this expression were used in a context where a Boolean was needed, then the 27 is interpreted as "true".

```
$a = $b = 27;  # assigns 27 to both variables,
               # since the first assignment to $b produces 27 as its value
print "val: ", ($a = $b = 27), "\n" ;
if ($a = 27) { # assignment to $a... illustration only, not good style
   print "it was true \n" ;
} else {
   print "it was false \n" ;
}
if ($a = 0) { # another assignment to $a
   print "it was true \n" ;
} else {
   print "it was false \n" ;
}
```

This code fragment produces this output:

```
val: 27
It was true
It was false
```

A *do { BLOCK }* statement simply executes the code within the statement block and returns the value of the last expression in the block. We can use this feature combined with statement values to produce an alternate form of conditional. The following two statements are equivalent:

```
( $thresh < 125 ) && do { print "it passed \n" ; } ;
if ($thresh < 125) { print "it passed \n" ; } ;
```

In the first form we also make use of the fact that Perl will evaluate the clauses of a logical conjunction one at a time, left to right, and stop if one should evaluate to false. In this case, should the boolean comparison fail, the second clause of the conjunction (the one with the printing) will not be attempted.

Loop structures Looping in Perl is done with variants of the *while*, the *do,* and the *for* structures. The *while* structure is equivalent to that of Java, C, or C++. The loop body block executes as long as the controlling expression remains true.. The *until (expnB)* structure is functionally equivalent to *while (! expnB)* :

```
while ($d < 37 ) { $d++; $sum += $d; }
until ($d >= 37) { $d++; $sum += $d; }
```

The *do/while* and *do/until* structures work similarly to the *while* structure, except that the code is executed at least once before the condition is checked.

```
do { $d++; $sum += $d; } while ($d < 37);
do { $d++; $sum += $d; } until ($d >= 37) ;
```

The *for* structure works similarly to that of C, C++ or Java, and is really syntactic sugar for a specific type of *while* statement. More interesting is the *foreach* loop, which is specifically designed for systematic processing of Perl's native data types. The *foreach* structure takes a scalar, a list and a block, and executes the block of code, setting the scalar to each value in the list, one at a time. Thus the *foreach* loop is a form of *iterator*, giving access to every element of some controlling collection. Consider this example:

```
my @collection = ("first", "second", "third", "fourth");
foreach $item (@collection) { print "$item\n"; }
```

This will print out each item in collection on a line by itself. You are permitted to declare the scalar variable directly within the *foreach*, and its scope is the extent of the loop. Perl programmers find the *foreach* loop to be one of the most useful structures in the language.

last operator The *last* operator, as well as the *next* and *redo* operators that follow, apply only to loop control structures. They cause execution to jump from where they occur to some other position, defined with respect to the block structure of the encompassing control structure. Thus, they function as limited forms of *goto* statements. Last causes control to jump from where it occurs to the first statement following the enclosing block. For example:

```
$d = 2;
while ( $d++ ) {
   if ($d >= 37) { last ; }
   $sum += $d ;
}
# last jumps to here
```

Jumps can be made from inner nested loops to points in outer loops by labeling the loops, and using the appropriate label after the *last* (as well as *next* and *redo).* We can now combine several of these features to give another way to "fake" the case statement shown previously as a decision tree with *if/elseif/else*:

```
CASE: {
   ($thresh < 10) && do {
      # the 'then' block of the conditional
      last CASE ; }
   ($thresh < 20) && do {
      # the next block in the decision tree
      last CASE ; }
   ($thresh < 40) && do {
      # and the next ...
      last CASE ; }
   # the final clause here catches what falls through
} # end of CASE block
```

As we mentioned earlier, there's *always* more than one way to do things in Perl.

next operator The next operator is similar to last except that execution jumps to the end of the block, but remains *inside* the block, rather than exiting the block. Thus, iteration continues normally. For example:

```
while ($d < 37) {
   $d++ ;
   if ( ($d%5)==1 ) { next };
   $sum += $d ;
   # next jumps to here
}
```

redo operator The *redo* operator is similar to *next* except that execution jumps to the top of the block without re-evaluating the control expression. For example:

```
while ($d < 37) {
   # redo jumps to here
   $d++ ;
   $sum += $d;
   if (($d%3)==0) { redo; }
   $prod *= $d ;
}
```

3.4 Subroutines

A subprogram in Perl is often called a *function*, but we shall use the term *subroutine* here to distinguish programmer-defined structures from the built-in functions of Perl. A subroutine is invoked within the context of some expression. In early versions of Perl, an ampersand (&) was placed before the subroutine name to

denote invocation; current versions allow invocation without as well. If the subroutine takes arguments, they are placed within parentheses following the name of the subroutine.

```
&aSubProg() ;
bSubProg() ;
cSubProg($ar3, $temp5, @ARY) ;
```

Control is transferred to the code of the subroutine definition, and transfers back either when the end of the subroutine code is reached, or an explicit return()statement is executed in the subroutine body.

The subroutine *definition* is marked by the keyword *sub* followed by the name of the subroutine, without an ampersand prefix. A block of code for the subroutine body follows, enclosed in curly braces; this is executed when the subroutine is called.

```
sub aSubProg {
   stmt_1;
   stmt_2;
   $a = $b + $c;
}
```

The value returned by a Perl subroutine is the value of the last expression evaluated in the subroutine. In this example, *aSubProg* will return the value $a has at the time when the subroutine ends. Functions such as *print* return values of 0 or 1, indicating failure or success.

Arguments are enclosed in parentheses following the name of the subroutine during invocation; thus, they constitute a *list*. They are available within the subroutine definition block through @_ the predefined (list) variable:

```
aSubProg ($a, "Literal_string", $b);

sub aSubProg {
   foreach $temp(@_) { print "$temp \n"; }
}
```

Any variables defined within the body of a Perl program are available inside a Perl subroutine as global variables. Consequently, Perl provides an explicit scope operator (*my*) that can be used to limit the visibility of variables and protect globals from inadvertent side effects. Similarly, these locals will not be visible outside the subroutine. Local variables are, by convention, defined at the top of a Perl subroutine:

```
aFunc ($a, $b);

sub aFunc {
   my ($aLocal, $bLocal);
   $aLocal = $_[0]; # @_ is used $_[i] for individual arguments
   $bLocal = $_[1];
}
```

`$aLocal` and `$bLocal` will have the same values inside the subroutine as `$a` and `$b` have at the time it is invoked. Changes to either local variable inside the function, however, will not affect the values of `$a` or `$b`.

Built-in functions and system operations Perl offers a rich selection of built-in functions as part of the standard interpreter. These include mathematical operations (such as *abs, sin, sqrt*, *log*); list manipulation operations (such as *join, reverse, sort*); array manipulation operations (such as *push, pop, shift*); string manipulation operations (such as *chop, index, length, substr, pack, reverse*); and myriad operating system functions reflecting Perl's Unix birthright.

Since one of the reasons for the creation of Perl was to give Unix programmers more expressive power and convenience, the language provides several mechanisms for invocating operating system services from executing scripts. The most general method is the *system* function:

```
$retVal = system("pwd") ;
```

In this example, the Perl interpreter uses the system command to get the underlying operating system to execute the Unix "pwd" command. The result of the command appears on STDOUT just as it would if it were done from the command line; the return value, in this case, is an indicator of success or failure. Often programmers want to capture the output of a system command for inclusion in the executing script. This is accomplished by enclosing the command in backward single quotes, often called "backticks":

```
$dir = `pwd` ;
print "the current directory is $dir \n" ;
```

Many other operating system (specifically, Unix) manipulations are available in Perl via built-in functions. The *chdir* function allows a Perl script to alter the default directory in which it finds its files while executing; the *opendir, readdir*, and *closedir* functions allow a Perl script to obtain directory listings; *mkdir* and *rmdir* allow a script to create and delete directories; *rename* and *chmod* allow a script to rename a file and change its access permissions. All these capabilities exist because Perl was originally designed to make it easy for system managers to write programs to manipulate the operating system and user file spaces.

Functions *exec, fork, wait*, and *exit* allow scripts to create and manage child processes. Perl provides a means of connecting a running process with a file handle, allowing information to be sent to the process as input using print statements, or allowing the process to generate information to be read as if it were coming from a file. We illustrate these features in the section "*Network Programming in Perl*".

3.5 Regular Expressions and Pattern Matching

Perhaps the most useful, powerful, and recognizably Perl-ish aspect of Perl is its pattern matching facilities and the resulting rich and succinct text manipulations they make possible. Given a pattern and a string in which to search for that pattern, several operators in Perl will determine whether -- and if so, where -- the pattern occurs. The pattern descriptions themselves are called *regular expressions*. In addition to providing a general mechanism for evaluating regular expressions, Perl provides several operators that perform various manipulations on strings based upon the results of a pattern match.

Regular expression syntax Patterns in Perl are expressed as regular expressions, and they come to the language through its Unix *awk* heritage. Since regular expressions are well understood from many areas of computing, we will not give an involved introduction to them here. Rather, we will simply use Perl examples to give an idea of the text processing power they give the language.

By default, regular expressions are strings that are delimited by slashes, *e.g.*, /rooster/. This delimiter can be changed, but we will use it for the examples. By default, the string that will be searched is in the variable *$_*. One can apply the expression to other strings and string variables, as will be explained below.

The simplest form of pattern is a *literal string*. For example:

```
if (/chicken/) { print "chicken found in $_\n"; }
```

The "/" delimiters appearing alone denote a default application of the match operator. Thus this code fragment searches in the default variable $_ for a match to the literal "chicken", returning true if found. In addition to including literal characters, expressions can contain categories of characters. They can specify specific sequences with arbitrary intervening strings; they can specify matches at the beginning or end; they can specify exact matches, or matches that ignore character case. Examples of these uses include:

```
/.at/           # matches "cat," "bat", but not "at"
/[aeiou]/       # matches a single character from the set of vowels
/[0-9]/         # matches any single numeric digit
/\d/            # digits, a shorthand for the previous pattern
/[0-9a-zA-Z]*/  # matches a string of alphanumeric characters, or length zero
or more
/\w/            # words, a shorthand for the previous pattern
/[^0-9]/        # not a digit
/c*mp/          # any number of c's followed by mp
/a+t/           # one or more a's followed by t
/a?t/           # zero or one a followed by t
/a{2,4}t/       # between 2 and 4 a's followed by t
```

```
/k{43}/          # exactly 43 occurrence of "k"
/(pi)+(sq)*/     # strings with one or more "pi" pairs followed by zero or more
"sq" pairs
/^on/            # match at start: "on the corner" but not "Meet Jon"
/on$/            # match at end: "Meet Jon" but not "on the corner"
/cat/i           # ignore case, matches "cat", "CAT", "Cat", etc.
$A =~ /pong/       # does the content of string variable $A contain "pong"?
<STDIN> =~ /b.r+/  # does the next line of input contain this pattern
                   # which matches bar, bnr, bor, brrr, burrrrrr, etc.
```

Pattern matching is *greedy*, meaning that if a pattern can be found at more than one place in the string, the leftmost instance is returned; if there are overlapping leftmost instances, the longest match will be identified, thereby affecting the outcome of patterned-based operators such as *substitution*. String substituiton, and the comparison operations shown in the last two lines above, are exemplified in more detail with the longer programs in the section "*Putting it All Together*".

String manipulation Regular expression operators include a regular expression as an argument but instead of just looking for the pattern and returning a truth value, as in the examples above, they perform some action on the string, such as replacing the matched portion with a specified substring (like the well-known "find and replace" commands in word processing programs). The simplest is the "m" operator, the explicit match. In the following example, a string is searched for the substring "now" (ignoring character case); the match operator return value is interpreted as a Boolean for control of the conditional:

```
my($text) = "Now is the time, now seize the day";
if ($text =~ m/now/i) { print "yep, got it\n"; }
if ($text =~ /now/i) { print "yep, got it\n"; } # equivalent form, no "m"
```

In general, when invoking the match operator the "m" is usually omitted, as illustrated in the 3rd line above. If a pattern is given with no explicit leading operator, the match operator is employed by default. Though we do not extract or use the matching substring in this example, the operator actually matches on the first 3 characters `"Now"` because of the ignore case option.

The substitution operator "s" looks for the specified pattern and replaces it with the specified string. By default, it does this for only the first occurrence found in the string. Appending a "g" to the end of the expression causes global replacement of all occurrences.

```
s/cat/dog/   # replaces first "cat" with "dog" in the default variable $_
s/cat/dog/gi # same thing, but applies to "CAT", "Cat" everywhere in $_
$A =~ s/cat/dog/  # substitution on the string in $A rather than the default $_
```

The *split* function searches for all occurrences of a pattern in a specified string and returns the pieces that were separated by the pattern occurrences as a list. If no string is specified, the operator is applied to $_.

```
$aStr = "All category";
@a = split(/cat/, $aStr);  # a[1] is "All " and a[2] is "egory"
@a = split(/cat/);         # this split happens on the string in default $_
```

The *join* function performs the opposite of a *split*, assembling the strings of a list into a single string with a separator (the first argument) placed between each part:

```
$a = join(":", "cat", "bird", "dog"); # returns "cat:bird:dog"
$a = join("", "con", "catenate");     # returns "concatentate"
$a = "con" . "catenate" ; # $a gets the value "concatentate"
@ar = ("now", "is", "the", "time");
$a = join "", @ar ;        # $a gets the value "nowisthetime"
```

In the second line above, where the separator is no character at all, the effect of the *join* is the same as using Perl's concatentation operator, as shown in the third line. The added power of *join* is that it will operate on all elements of a list without them being explicitly enumerated, as illustrated in the fourth and fifth lines.

Pattern memory The portion of the string that matches a pattern can be assigned to a variable for use later in the statement or in subsequent statements. This feature is triggered by placing the portions of a pattern to be *remembered* in parentheses. When used in the same statement or pattern, the matched segment will be available in the variables \1, \2, \3, etc. in the order their targets occur. Beyond the scope of the statement,

these stored segments are available in the variables, *$1, $2, $3*, etc. as well as contextually. Other matching information available in variables include *$&*, the sequence that matched; *$`*, everything in the string up to the match; and *$'*, everything in the string beyond the match.

For example, the following program separates the file name from the directory path in a Unix-style path name. It works by exploiting Perl's greedy matching, along with the pattern memories:

```
my($text) = "/tmp/subsysA/user5/fyle-zzz";
my($directory, $filename) = $text =~ m/(.*\/)(.*)$/;
print "D=$directory, F=$filename\n";
```

The pattern finds the last occurrence of "/" in the target string so the Unix directory can be split out from the file name. The first set of parentheses saves this directory substring, and the second set captures the file name. The assignment after the match on `$text` stores both pattern memories by positional order into the variable `$directory` and `$filename`. Here is another example using the \1 and $1 memory notations:

```
$A = "crave cravats" ;
$A =~s/c(.*)v(a.)*s/b\1\2e/ ;   # \1 is "rave cra" and \2 is "at"
print "$A\n";
print "$1\n" ;
print "$2\n" ;
```

The output from this code fragment is

```
brave craate
rave cra
at
```

The substitute operator in the second line performs the match by first finding the longest string of characters between the "c" and "v" and saving it in the \1 memory. It then finds the longest string of "a" followed by any single character in the rest, and saves that in the \2 memory. Once matched, the replacement string is formed by concatenating the memories, adding a "b" at the front, and an "e" at the end. The last two lines show that the string parts that matched the pattern parts are still available after the match for as long as the variables $1 and $2 are not overwritten.

3.6 Input/Output and File Handling

File handling is another area where Perl makes life easy for the programmer. The basic file manipulation operators, coupled with array capabilities, make creating internal structures out of text input succinct and efficient. Files are accessed within a Perl program through *filehandle*s, which are bound to a specific file through an *open* statement. By convention, Perl filehandle names are written in all uppercase, to differentiate them from keywords and function names. For example:

```
open (INPUT, "index.html");
```

associates the file named "index.html" with the filehandle INPUT. In this case, the file is opened for read access. It may also be opened for write access and for update (appending) by preceding the filename with appropriate symbols:

```
open (INPUT, ">index.html");  # opens for write
open (INPUT, ">>index.html"); # opens for appending
```

Since Perl will continue operating regardless of whether a file open is successful or not, we need to test the success of an open statement. Like other Perl constructs, the open statement returns a true or false value. Thus, one common way to test the success of the open and take appropriate action is to combine the lazy evaluation of logical *or* with a *die* clause, which prints a message to STDERR and terminates execution:

```
open (INPUT, "index.html") || die "Error opening file index.html ";
```

Files are closed implicitly when a script ends, but they also may be closed explicitly:

```
close (INPUT);
```

Perl provides default access to the keyboard, terminal screen, and error log with predefined filehandles STDIN, STDOUT, and STDERR; these handles will be automatically available whenever a script is executed. Once

opened and associated with a filehandle, a file can be read with the diamond operator (<>), which can appear in a variety of constructs. STDIN is most commonly accessed this way. When placed in a scalar context, the diamond operator returns the next line; when place in an array context, it returns the entire file, one line per item in the array. For example:

```
$a = <STDIN>; # returns next line in file
@a = <STDIN>; # returns entire file
```

STDOUT is the default file accessed through a *print* statement. STDERR is the file used by the system to which it writes error messages; it is usually mapped to the terminal display.

Here is an example that reads an entire file from STDIN, line-by-line, and echos each line to STDOUT with line numbering:

```
$lnum = 0;
while (<STDIN>) {  # read one line at a time until EOF
                   # in this case, the default variable $_ receives the line
  chomp;           # remove line-ending character (newline here)
                   # again, it operates on $_ automatically
  $lnum++;         # auto-increment operator on line counter
  print "$lnum: $_\n";  # print the line read, using default $_
}
```

In this example, we illustrate one of the many Perl conveniences. In many contexts, if no scalar variable is indicated, an operation will give a value to a variable named '$_ ', the default scalar variable. This is in keeping with Perl's design philosophy of making it very easy to do common tasks. We could also have omitted the filehandle STDIN and simply have written "`while (<>)`"; the diamond operator will operate on STDIN by default if given no filehandle explicitly.

Once a file has been opened for either *write* or *update* access, data can be sent to that file through the print operator. For example:

```
print OUTPUT "$next \n";  # outputs $next followed by newline
print "this statement works on STDOUT by default\n";
```

The second example illustrates a common and useful shorthand in Perl; a *print* statement with no filehandle operates on STDOUT by default.

Finally, there are a number of circumstances where the actions taken by the Perl program should take into account attributes of the file, such as whether or not the file currently exists, or whether it has content. A number of tests can be performed on files through *file test* operators. For example, to check for file existence use the *–e* test:

```
if (-e "someFile.txt") {
   open (AFYLE, "someFile.txt") || die "not able to open file" ;
}
```

Using different characters, many other attributes can be tested including if a file is readable, writable, executable, or owned by certain users; if it is text or binary; if it is a directory or symbolic link; or if it is empty to name a few.

There's more than one way to do it In concluding this section we again illustrate the famous Perl adage, this time with file *open* statements. Here are several examples of conditional expressions for safely opening files and trapping errors. In the following, the last four lines all do the same thing:

```
$aFile = "foo.txt";
if (!open(fh, $aFile)) { die "(a) Can't open $aFile: $!"; }
die "(b) Can't open $aFile: $!" unless open(fh,$aFile);
open(fh,$aFile) || die "(c) Can't open $aFile: $!";
open(fh,$aFile) ? '' : die "(d) Can't open $aFile: $!";
```

3.7 Other Perl Features

Perl has several other features and capabilities that have found their way into the language as it evolved. These later features tend to be capabilities that programmers found useful in other languages and desired to have in Perl. In particular, Perl version 5 introduced *classes, objects* and *references* (or pointers) into a language that was previously a more traditional Unix scripting notation "on steroids". Since they do not greatly enhance Perl's capabilities in the areas for which it has proven especially superior (text processing, file handling, string matching, OS interactions) we will not go into them in detail. Some programmers even consider these additions to aggravate the already difficult task of reading Perl code. These later features are not *unimportant* aspects of the language; they are simply well beyond the original domains of expertise and applicability for which Perl was developed. As such, they represent the natural end to which languages tend to evolve as they gain popularity – something of everything for everyone.

Perl has many more sophisticated capabilities. Access to the interpreter is available to an executing script through the *eval* function, allowing a program to create and then run new code dynamically. Symbol tables can be accessed and manipulated directly with Perl *typeglobs*. Function *closures* can be created (as in many functional languages) allowing subroutines to be packaged dynamically with their data and passed back from a function call as a reference for execution later. *Packages* and *modules* provide encapsulation and namespace control. The later versions of Perl even support concurrent computations with a *thread* model.

We refer the reader to the texts cited in *For More Information* for thorough presentations of all these topics.

4 Putting it All Together: Sample Programs

4.1 First example: text processing

Here is a Perl script that will take as input a file called "foo.txt", produce as output a file called "bar.txt"; lines in input will be copied to output, except for the following transformations:

- any line with the string "IgNore" in it will *not* go to output
- any line with the string "#" in it will have that character and all characters after it, to end of line, removed
- any string "*DATE*" will be replaced by the current date in output

One program to do this is as follows:

```
#!/usr/local/bin/perl
$infile = "foo.txt" ;
$outfile = "bar.txt" ;
$scrapfile = "baz.txt" ;
open(INF,"<$infile") || die "Can't open $infile for reading" ;
open(OUTF,">$outfile") || die "Can't open $outfile for writing" ;
open(SCRAPS,">$scrapfile") || die "Can't open $scrapfile for writing" ;
chop($date = `date`) ; # run system command, remove the newline at the end
foreach $line (<INF>) {
   if ($line =~ /IgNore/) {
      print SCRAPS $line ;
      next;
   }
   $line =~ s/\*DATE\*/$date/g ;
   if ($line =~ /\#/) {
      @parts = split ("#", $line);
      print OUTF "$parts[0]\n" ;
      print SCRAPS "#" . @parts[1..$#parts] ; # range of elements
   } else {
      print OUTF $line ;
   }
}
close INF ; close OUTF ; close SCRAPS ;
```

In keeping with the Perl adage that there's more than one way to do things, here is an alternative way to write the *foreach* loop; this one uses the implicit $_ variable for pattern matching:

```
# this version uses the implicitly defines $_ variable
foreach (<INF>) {
   if ( /IgNore/ ) {
      print SCRAPS ;
      next;
      }
   s/\*DATE\*/$date/g ;
   if ( /\#/ ) {
      @parts = split ("#");
      print OUTF "$parts[0]\n" ;
      print SCRAPS "#" . @parts[1..$#parts] ; # range of elements
   } else {
      print OUTF ;
      }
}
```

And finally, a third version, using boolean and conditional expressions in place of *if-else* statements:

```
# this version uses boolean interpretation of expressions as
# substitution for if clauses in previous versions
foreach (<INF>) {
   /IgNore/ && do { print SCRAPS; next } ;
   s/\*DATE\*/$date/g ;
   /#/ ? do {
      @parts = split ("#");
      print OUTF "$parts[0]\n" ;
      print SCRAPS "#" . @parts[1..$#parts] ; # range of elements
   }
   : do {
      print OUTF ;
   }
}
```

4.2 A Simpler, More Sophisticated Example

Consider this problem: take an input file and produce an output file which is a copy of the input with any duplicate input lines removed. Here is a first solution:

```
#!/usr/local/bin/perl
foreach ( <STDIN> ) { print unless $seen{$_}++ ; }
```

This is, of course, exactly why so many like Perl so fervently. A task that would take many lines of C code can be done in Perl with a few lines, thanks to the sophisticated text handling facilities built in to the language. In this solution, we are reading and writing standard input and output; in Unix we supply specific file names for these streams when the program it is invoked from the command line, like this:

```
second.pl <foo.txt >bar.txt
```

Here is a second solution:

```
#!/usr/local/bin/perl
# this version prints out the unique lines in a file, but the order
# is not guaranteed to be the same as they appear in the file
foreach ( <> ) { $unique{$_} = 1 ; }
print keys(%unique);  # values(%unique) is the other half
```

And a third solution:

```
#!/usr/local/bin/perl
# this version eliminates duplicate lines
```

```
# and prints them out in arbitrary order
# also tells how many time each line was seen
# oh, and it sorts the lines in alpha order
foreach ( <> ) { $unique{$_} += 1 ; }
foreach (sort keys(%unique)) {
  print "($unique{$_}):$_" ;
}
```

This last example shows the considerable power and terseness of Perl. In essentially 4 lines of code, we filter a file to remove duplicate lines, reports a count of how many times each unique line appeared in the original input, and prints the unique lines sorted in alphabetic order. All the facilities used in this program are part of the standard Perl language definition. It does not depend on any user-supplied routines or libraries.

4.3 Directory information processing

In this example, the script takes input from a Linux *dir* command (directory); we assume the command has been piped into the Perl script, so the script reads standard input and writes to standard output. The output produced is itself an executable script (in Linux *csh* notation) that copies every file (not directory) older than 12/22/97 to a directory called `\ancient`. The output of *dir* (and so the input to the script) is this:

```
.              <DIR>        12-18-97 11:14a .
..             <DIR>        12-18-97 11:14a ..
INDEX    HTM         3,214  02-06-98  3:12p index.htm
CONTACT  HTM         7,658  12-24-97  5:13p contact.htm
PIX            <DIR>        12-18-97 11:14a pix
RANGE    HTM         9,339  12-24-97  5:13p range.htm
FIG12    GIF           898  06-02-97  3:14p fig12.gif
README   TXT         2,113  12-24-97  5:13p readme.txt
ACCESS   LOG        12,715  12-24-97  5:24p ACCESS.LOG
ORDER    EXE        77,339  12-24-97  5:13p order.exe
INVNTRY        <DIR>        02-06-98  1:58p invntry
         7 file(s)        113,276 bytes
         4 dir(s)      27,318,120 bytes free
```

In C or C++ this program would be long and involved. In Perl, we get a compact handful of lines, making good use of the regular expressions, pattern matching, and pattern memories:

```
my $totByte = 0;
while(<>){
   my($line) = $_;
   chomp($line);
   if($line !~ /<DIR>/) {   # we don't want to process directory lines
      # dates is column 28 and the filename is column 44
      if ($line =~ /.{28}(\d\d)-(\d\d)-(\d\d).{8}(.+)$/) {
         my($filename) = $4;
         my($yymmdd) = "$3$1$2";
         if($yymmdd lt "971222") {
            print "copy $filename \\ancient\n"; } }
      if ($line =~ /.{12}((\d| |,){14})  \d\d-\d\d-\d\d/) {
         my($bytecount) = $1;
         $bytecount =~ s/,//;  # delete commas
         $totByte += $bytecount;
      }
   }
   print STDERR "$totByte bytes are in this directory.\n";
}
```

In the first match, the variables $1, $2, $3 and $4 are the pattern memories corresponding to the parenthesis sets. The first three are re-assembled into a *yymmdd* date string which can be compared with the constant "971222". The fourth holds the filename that will be copied to the `\ancient` directory. As a side effect of

processing the directory listing, we set up an accumulator and extract a cumulative byte count. This is done with a second match on the same input line, as well as a substitution operation to remove commas from the numbers found.

5 Network Programming in Perl

The World Wide Web is the most widely known Internet application. Many Web sites provide more than static HTML pages. Instead, they collect and process data, or provide some sort of computational service to browsers. For example, several companies operate Web sites that allow a user to enter personal income and expense information, and will then not only compute income tax returns online but will also electronically file them with the IRS. There are numerous technical ways to provide this processing horsepower to a Web site (e.g., Microsoft's Active Server Pages, JavaScript, C/C++/C# programs, etc.) but Perl is the most widespread and popular of these options. In this section we look at how Perl scripts can provide communications between Web browsers and servers, and how they can make use of databases for persistent storage. We also discuss some of Perl's capabilities for interprocess communication and network computations.

Much of this Web and network programming is not usually done from scratch, but rather by re-using excellent Perl modules written by other programmers to encapsulate details and provide abstractions of the various domain entities and services. We begin with a discussion of CPAN, the Perl community's repository for these freely shared modules.

5.1 Packages, Modules and CPAN

CPAN is a large collection of Perl code and documentation that has been donated by developers to the greater Perl programming community. It is accessed on the Web at `http://www.cpan.org` and contains many modules that have become *de facto* standards for common Perl scripting tasks. In addition to programmer-contributed modules, the source code for the standard Perl distribution can be found at CPAN. In the words of Larry Wall in *Programming Perl*, "If it's written in Perl, and it's helpful and free, it's probably on CPAN."

Packages CPAN is best known for its collection of *modules*. These are groups of Perl variables and subroutines that are encapsulated to provide namespace protection, making the code more easily reusable. Modules are based on the *package,* the unit of namespace control in Perl. By default, a Perl program produces names in a *main* package, and many useful programs never use any other. However, for programs intended to be reused in contexts where the programmer cannot anticipate the variable names that will exist, a separate or protected namespace is needed. A *package* declaration establishes this new namespace. Variable references are always evaluated in the current package, unless fully qualified with a specific package name:

```
$total = 17;
package PackOne ;
  $total = 0 ;
  $name = "Mr. Jones" ;
  @arr = (12, 34, 45.7) ;
package main;
  print $total;
  print "(b)$arr[1]";
  print "(c)$PackOne::arr[1]";
```

This program prints "`(a)17(b)(c)34`". The first declaration of the variable `$total` goes into the symbol table for the default *main* namespace. The *package* declaration in the second line establishes a symbol table that is separate from the default *main* and the three variables declared after that are placed in this new namespace. A package declaration stays in effect until the end of the enclosing scope, or until another package declaration is encountered, as in line six here. The final three print statements take place again in the context of *main*, so the value of "17" is found for `$total`. No values exist for any elements of `@arr` in main, so the second print produces only the "(b)". The last line shows how to qualify a variable name to indicate explicitly what package namespace to look into; here we find the array values established at line fine, in the `PackOne` package.

As this example shows, Perl allows multiple packages to be declared in one code file; it also allows a single package declaration to span multiple files. However, for simplicity many programmers follow the convention that each package declaration gets its own file, and the file name is the same as the package name.

Modules The *module* is the main mechanism for code reuse in Perl. A module is a package declared in a file that has ".pm" as its filename extension; the package, module, and file have the same name. The author of a module further defines what names within the module are to be made available to outside Perl programs. This is done through the *Export* module. To incorporate the variables, subroutines, and objects of a modules into a program, the *use* statement is employed:

```
use JacksCode ; # in which a variable $jackrabbit is declared
print "$jackrabbit \n";
print "$JacksCode::jackrabbit \n";
```

In this example, the *use* statement requests access to all names that are exported from the module "JacksCode", which the interpreter will expect to find in a file named "JacksCode.pm" someplace on its search path. If this module declares a variable named "$jackrabbit" then the last two lines do the same thing. A variable name imported from a module need no longer be fully qualified with the module name. There are several alternate forms of the *use* statement that give finer-grained control over which of the exported names are imported.

Many of the modules most commonly used by programmers come as part of the standard Perl distribution. CPAN contains dozens of other heavily used modules. These include

- *CGI, HTML, HTTP, LWP, Apache* module families for Web server scripts
- *POSIX* for Unix-programming compatibility
- *Socket* for network structures and programming
- *Net::FTP, Net::DNS, Net::TCP, Net::SMTP, Net::IMAP,* and many other for dozens of protocols
- *RPC::plserver, RPC::plclient, RCP::simple* for remote procedure call support
- *Math::BigInt, Math::Trig, Math::Polynomial* and dozens more supporting various forms of mathematical structures and functions
- *List, Set, Heap, Graph* module families giving common abstract data types
- *Statistics* module family for various common tests and distributions
- *Date, Time:, Calendar* module families
- *DBI, DBD, Oracle, Sybase* and other database interfaces
- *Java* for communication between Perl and the JVM
- *Language::ML, Language::Prolog, C::DynaLib, Python*, and other language interfaces
- *String, Text, Lingua* module families for text and language processing
- *PostScript, Font, PDF, XML, RTF, Tex, SQL* module families for documents
- *PGP, DES, Crypt, Authen* module families for encryption and security

As enjoyable as Perl programmers find their craft to be, no one wants to spend time re-writing code someone else has already done well. CPAN is the result of an enthusiastic community effort to leverage success.

5.2 Web Server Scripts with CGI

When a Web page contains fill-out forms, or has some other computational behavior required, there are several ways to provide the processing needed on the Web server side of the transaction. One way is via scripts that adhere to the data formatting standards of the CGI web interface. CGI scripts can be written in any programming language that the server will support. A separate article in this compilation covers the CGI standard in detail, so we concentrate here on the specifics of how Perl allows programmers to take advantage of this standard for Web development.

In any interaction between a Web browser and a Web server, there is data being exchanged. The browser sends information to the server requesting some action be taken, and the server sends a reply back, usually in the form of an HTML Web page. This interaction often happens by the user filling out the fields of a form in the browser window, and clicking on some "submit" button. Submitting the form entails the browser collecting the data from the form fields, encoding it as a string according to the CGI standard, and passing it to the Web server specified in the URL associated with the submit button in the form. This URL not only contains the location of the server that will process the form data, but the name of the Perl script that should be executed as well on the server side.

The following script fragments show portions of such a Perl server-side CGI script. The data in the form from the browser is made available to Perl via *environment variable*s. Usually the Web site programmer will either write a collection of Perl subroutines to process this incoming information in various ways, or will choose to use one of the many such packages out there that others have already written and published. In this example, the first line specifies the inclusion of a set of CGI processing subroutines ("*cgi.pl*") and then immediately invokes several of them to process the information being sent from the Web browser:

```
require "cgi.pl";
&cgi_receive;  # get the URL information from the environment
&cgi_decode;   # parse the CGI formatted string and build @FORM
&cgi_head;     # begin forming the CGI-compliant HTML page to return
# produce the title etc. for the HTML page being returned…
print <<EndOfStuff ;
<html>
  <head>
    <title>Company Evaluation</title>
  </head>
  <body background="/images/backgrnd.gif" >
    <H1>$FORM{'uid'}: Submit your report</H1>
    <hr>
EndOfStuff
# form data processing proceeds . . .
# remove blanks from what the user typed into the form
$FORM{'measid'} =~ s/ //g;
if ($FORM{'uid'} eq "") {
  print "<H2>Missing User ID</H2>\n";
  exit;
}
if ($FORM{'passwd'} ne "onThe.Road") {
  print "<H2>Bad password given\n";
  exit;
}
```

The data that was put into the form fields on the browser side is now available to the Perl CGI script in the associative array %FORM. This array is built by the subroutine "cgi_decode"; the keys of the hash are the names of the fields from the form. The Perl subroutines in this CGI package look like this:

```
sub cgi_receive {
# Procedure to get the information sent from a form. Works with both
# POST and GET methods.
    if ($ENV{'REQUEST_METHOD'} eq "POST") {
        read(STDIN, $incoming, $ENV{'CONTENT_LENGTH'});
    } else {
        $incoming = $ENV{'QUERY_STRING'};
    }
}

sub cgi_head { print "Content-type: text/html\n\n"; }

sub cgi_decode {
# Procedure to process the information sent from a form.
# creates two arrays: @fields and %FORM.
# elements of @fields are the subscripts for %FORM.
# elements of %FORM are the form values.
    my(@pairs) = split(/&/, $incoming);
    foreach (@pairs) {
        ($name, $value) = split(/=/, $_);
        $name  =~ tr/+/ /;
        $value =~ tr/+/ /;
        $name  =~ s/%([A-F0-9][A-F0-9])/pack("C", hex($1))/gie;
        $value =~ s/%([A-F0-9][A-F0-9])/pack("C", hex($1))/gie;
        # Strip out semicolons unless for special character
        $value =~ s/;/$$/g;
```

```
            $value =~ s/&(\S{1,6})$$/&\1;/g;
            $value =~ s/$$/ /g;
            $value =~ s/\|/ /g;
            $value =~ s/^!/ /g; ## Allow exclamation points in sentences
            $value =~ s/\n/ /g; ## Remove embedded newlines
            # Skip blank text entry fields
            next if ($value eq "");
            # Check for "assign-dynamic" field names
            # Mainly for on-the-fly input names, especially checkboxes
            if ($name =~ /^assign-dynamic/) {
                $name = $value;
                $value = "on";
            }
            # Allow for multiple values of a single name
            $FORM{$name} .= " " if ($FORM{$name});
            $FORM{$name} .= $value;
            push (@fields, $name) unless ($name eq $fields[$#fields]);
        }
    }
```

Module "CGI" The previous example is a "roll-your-own" version of CGI processing, included here to illustrate how the CGI standard encodes web form data and to give a few more examples of Perl scripts manipulating string data. In the early days of the Web, such scripts were common; each Web site developer would produce her own collection of subroutines since standard code libraries were not available. Today, the task is made much easier through the standard Perl module "CGI". Using the CGI module, the previous server-side form processing script would look something like this (simplified):[3]

```
use CGI;
$q = CGI::new();
$mid = $q->param("measid");
$uid = $q->param("uid");
$pwd = $q->param("passwd");
print $q->header();
print $q->head($q->title("Company Evaluation"));
print $q->body(
   $q->h1("$uid: Submit Your Report"),
   $q->hr,
      etc... rest of body elements...
   );
)
```

As shown here, the CGI module provides functions for retrieving environment variables, creating web forms as output, and generating HTML tags. This example is selecting data from a Web form containing text fields called "measid", "uid", and "passwd". It is generating an HTTP-compliant return message with the necessary header and an HTML page for the browser to display. Assuming the "uid" here comes in from the form as "Jones", we get:

```
Content-type: text/html; charset=ISO-8859-1
<head>
  <title>Company Evaluation</title>
</head>
<body>
  <h1>Jones: Submit My Report</h1>
  <hr>
    etc...
```

The CGI module also assists the Web site developer in solving other problems common with Web scripting. Since the HTTP protocol is stateless, one problem is maintaining session state from one invocation of a script

[3] The arrow notation (->) is the Perl syntax for dereferencing a reference (chasing a pointer). In this module, and others following, it is used to access the fields and functions of a Perl object.

to the next. This is normally done with *cookies,* data items a server asks the Web browser to store locally and return on request. However not all browsers allow cookies, and in those that do the user may turn cookies off for security or privacy reasons. To help with this a script using CGI, when called multiple times, will receive default values for its input fields from the previous invocation of the script.

5.3 Web Clients with LWP

While the CGI module supports construction of scripts on the server-side of a Web connection, the modules *LWP (Library for Web access in Perl)* provides support for developing applications on the client side. Most notable among Web clients are the GUI-based browsers, but many other applications acts as clients in HTTP interactions with Web servers. For example, web crawlers and spiders are non-GUI programs (called "bots" or robots) that continuously search the Web for pages meeting various criteria for cataloging.

The LWP modules each support a different aspect of Web client construction and operation. They include:

- *HTML* for parsing and converting HTML files
- *HTTP* for implementing the requests and responses of the HTTP protocol
- *LWP* core module, for network connections and client/server transactions
- *URI* for parsing and handling URLs
- *WWW* implementing robot program standards
- *Font* for handling Adobe fonts
- *File* for parsing directory listings and related information

A Web interaction starts with a client program establishing a network connection to some server. At the low level this is done via sockets with the TCP/IP protocol. Perl does support socket programming directly (see below), and the module *Net* contains functions to allow a program to follow TCP/IP (as well as many others Internet protocols, such as FTP, DNS, and SMTP). On top of sockets and TCP/IP for basic data transfer, the HTTP protocol dictates the structure and content of messages exchanged between client and server. Rather than deal with all these technologies individually, the *LWP::UserAgent* module allows the programmer to manage all client-side activities through a single interface. A simple client script would look like this:

```
use LWP::UserAgent; # imports other modules too
$client = new LWP::UserAgent;
$acmeRep = new URI::URL('www.acme.com/reports/index.html');
$htmlHead =
   new HTTP::Headers(User-Agent=>'RepBot v2.0', Accept=>'text/html');
$outMsg = new HTTP::Request(GET, $acmeRep, $htmlHead);
$inMSg = $client->request($outMsg);
$inMsg->is_success ? {print $inMsg->content;} : {print $inMsg->message;}
```

The network connections and message formatting to HTTP protocol requirements is handled transparently by the LWP functions. Here, the client causes a request like this to be sent to the Web server at www.acme.com:

```
GET /reports/index.html HTTP/1.0
User-Agent: RepBot v2.0
Accept: text/html
```

If the requested file is not there, the response from the server will be an error message. If it is there, the response will contain the HTML file for the "bot" to process in some fashion.

5.4 Database Use

Many Web sites depend on databases to maintain persistent information -- customer data, statistics on site usage, collection of experimental results, medical records. Perl has several ways a CGI script (or any Perl script, Web-based or otherwise) can store and retrieve database records.

Module "DBM" Perl comes with a module called DBM (*DataBase Module*) that contains functions implementing a built-in database structure. DBM treats an external data store internally as hashes, or key/value pairs. This internal database is intended for simple, fast usage; searching requires as few as three lines of script. The method is convenient for fast retrieval, even from very large databases, when there is a unique key for searching (*e.g.* ISBN numbers). It is not as useful for complex data or queries.

```
dbmopen (%db, $database, 0400) || die "Can't open DB"; #open read only
```

```
for ($k=$max; $k< $max+20; $k++){ print "$k $db{$k}" } #get and print data
dbmclose (%db);                                         #close database
```

In this example, we know the index values are numbers and are unique. The database looks to the Perl script like a hash, so the associative array variable *%db* is used to get the data values from it.

One useful feature Perl provides is DBM filters. These are small data transformation routines, written by a programmer to manage situations where the format of the data fields in a database are not quite compatible with the form needed by a script. Rather than put small data manipulation code chunks scattered throughout the script, or write and call extra functions, DBM filters can be attached directly to the fields of a database; data transformation then takes place automatically as field values are moved into and out of the database. This feature makes the code using the database easier to read, and less error prone due to less code replication.

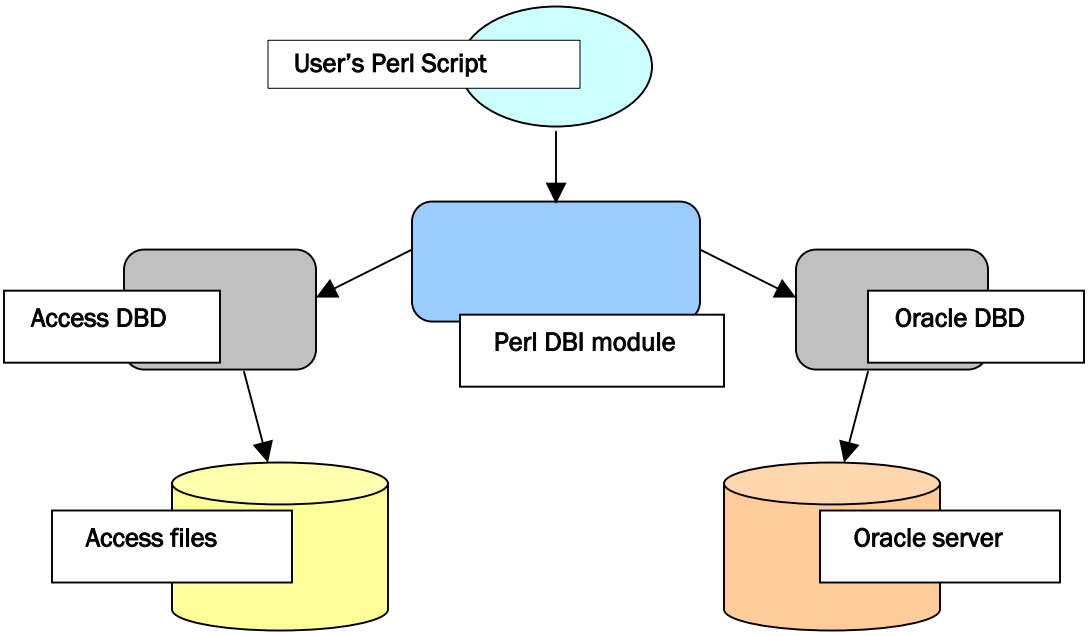

Figure 1: DBI provides an abstract interface for specific database systems

Module "DBI" For more advanced applications, a relational database is often more desirable than the simple structure of DBM. For this Perl provides the DBI module, or *Data Base Interface*. DBI is designed to hide the details of specific database systems, allowing the programmer to create scripts that are general. The interface allows expressing SQL queries, executing them against a specific database, and retrieving the results. The DBI module does not contain code specific to any database vendor's product, but it does have references to numerous vendor-specific modules called DBD's (*DataBase Drivers*). A DBD module will contain the detailed code needed to communicate with a specific brand of database system.

Figure 1 illustrates the relationship among the executing script, the DBI, the DBD, and the physical database. When a Perl script invokes a DBI function to execute a query, the query is routed to the appropriate DBD module according to how the DBI database handle was opened (as an Oracle DB, as an Access DB, etc.). The DBD module communicates with the actual files or tables of the physical database system, and produces query results. These results are communicated back to the DBI module, which relays them back to the user's Perl script. This layer of indirection gives Perl scripts a generality that makes migration from one physical DB system to another relative painless.

Module "ODBC" In addition to the DBI module, programmers wishing to write Perl scripts that interface to external databases, such as Access or Oracle, can obtain an ODBC compatibility package as a free download from several 3rd party distributors. This module contains functions that implement the ODBC standard

database interface. Perl scripts written to use this standard can then work with any relational database under them, as long as that database has its own ODBC interface. Almost all major relational database vendors provide an ODBC implementation for their products. ODBC provides essentially the same advantages as DBI, but the ODBC standard was designed outside Perl and is available in many other programming languages as well.

5.5 Processes and IPC

While the Web is the major Internet application, it is certainly not the only one. Perl includes facilities to support general network applications, and many Perl programs are written to use the Internet in many different ways. While not designed specifically for concurrent or multiprocess computations, Perl does support various forms of processes, interprocess communication (IPC), and concurrency.[4] Processes are relatively heavyweight computations that each have their own resources and address spaces. Multiple processes may execute on the same machine, or on different machines across the Internet. Data exchange among processes is usually done via files, pipes (channels), or lower level sockets.

Simple Unix-style process communications can be established in Perl using file I/O operations. Processes are given filehandles, and communication paths are established with the *open* statement using a pipe symbol "|" on command the process will execute. To read from an executing program, for example, the pipe goes at the end:

```
$pid = open(DATAGEN, "ls -lrt |") || die "Couldn't fork: $!\n";
while (<DATAGEN>) {
    print ;
}
close(DATAGEN) || die "Couldn't close: $!\n";
```

This program creates a process that executes the Unix "ls" command with arguments "-lrt" to generate a listing of the current directory. The pipe symbol tells Perl that the *open* is specifying a process command to run instead of a simple file name. To write to a process, the pipe goes at the beginning of the command:

```
$pid = open(DATASINK, "| myProg args") || die "Couldn't fork: $!\n";
print DATASINK "some data for you\n";
close(DATASINK) || die "Couldn't close: $!\n";
```

A script can use *pipe* and *fork* to create two related processes that communicate, with better control than can be had from *open, system*, and *backtick*s:

```
pipe(INFROM, OUTTO);  # opens connected filehandles
if (fork) { # both processes share all open filehandles
    # run parent code for writing
    close(INFROM);
    # now the writer code...
} else {
    # run child code for reading
    close(OUTTO);
    # now the reader code...
}
```

For more complicated situations, such as reading and writing to the same executing process, the previous methods are not sufficient. There is a special forking form of *open* that can be used. However, using the *IPC::Open2* module is a better approach:

```
use IPC::Open2;
open2(*READFROM, *WRITETO, "myProg arg1 arg2");
print WRITETO "here's your input\n";
$output = <READFROM>;
   # etc...
close(WRITETO);
```

[4] Perl having been born of Unix, this section is heavy on Unix process concepts such as pipes and forking.

```
close(READFROM);
```

Here the program "myProg" is executed as a process, using the supplied arguments, and the filehandles READFROM and WRITETO are connected to its standard input and output respectively so the Perl script can exchange data with it.

Module "Socket" Even finer-grained control over processes can be obtained if the programmer is willing to program lower into the operating system. Sockets are the underlying technical mechanism for network programming on the Internet. Perl gives access to this level with built-in functions that operate on sockets. These include

- *socket* to assign a filehandle
- *bind* to associate a socket with a port and address
- *listen* to wait for activity on the server-side connection
- *accept* to allow incoming connection on server-side
- *connect* to establish communications with a server
- *recv* to get data off a socket
- *send* to put data onto a socket
- *close* to end it all

Sockets are given filehandles on creation, so Perl functions that operate on filehandles can be used on sockets as well. Socket functions tend to need hard-coded values related to machine specifics and IP address, which limits portability of scripts. The Perl module *Socket* should be used in conjunction with the Perl socket functions to pre-load machine-specific constants and data into Perl variables.

Module "Thread" Lighter weight and finer grained than processes, many languages support concurrent computations that share common resources and address spaces. Known as *multithreading,* programming such applications is a specialized endeavor. As Larry Wall says in the text *Programming Perl*, "Perl is a rich language, and multithreading can make a muddle of even the simplest of language." Nonetheless, recent releases of Perl do support a simple thread model for concurrency within a script. Though obviously with different syntax, the behavior is similar to Java threads, with thread creation, synchronization, queuing for execution, resource locking, and signals. Because of its specialized and error-prone nature, we only mention the threads capability here; for more detailed information see references on the module *Thread* found at CPAN.

6 On Beyond Perl

Perl was created when object-oriented (OO) programming concepts were young and less well understood than today. Early versions of Perl, in fact, did not even contain objects or references. As understanding of OO concepts has advanced, Perl has evolved to contain OO features. They work fairly well, but since they were not part of the original design rationale of Perl, many consider them less elegant and cohesive than the original set of language features.

Two more recently developed programming languages – *Python* and *Ruby* -- claim Perl in their heritage, but have been designed specifically as OO programming tools. The designers of each wanted to retain the extreme convenience and flexibility of Perl's text handling and string matching, but to incorporate as well other capabilities that go beyond Perl. The results are two notations that are still largely thought of as "scripting" languages, but with more highly integrated object semantics. Following are brief discussions of each with respect to Perl.

6.1 Python

Guido van Rossum began work on the design of Python in late 1989. One of his goals for the new language was to cater to infrequent users and not just experienced ones. Infrequent users of a notation can find rich syntax (such as that of Perl) to be more burdensome than helpful. This means that Python is a *compact* language. A programmer can easily keep the entire feature set in mind without frequent references to a manual. C is famously compact in much the same way, but Perl most certainly is not. The Perl design principle of ``more than one way to do it'' shows up in Perl's wealth of features, notational shortcuts, and style idioms. Van Rossum also wanted a language designed from the beginning to be object-oriented and to have clear module semantics. In Python everything is an object or class including the base data types.

Python has unusual syntax for a modern programming language. Rather than being a free-form notation, in Python whitespace is important for defining the block structure of a script. Indentation levels serve the same purpose in Python that pairs of "{ }" do in Perl, C, and others. Here, the body of the loop, and the bodies of the conditional clauses, are defined by vertical alignment :

```
while x < y:
    buf = fp.read(blocksize)
    if not buf: break
    conn.send(buf)
x = 3
if x == 4:
   result = x + 2
   print x
else:
   print 'Try again.'
```

Though this initially appears to be a cumbersome throwback, in many ways this makes Python easy to use. Python has a very clean and structured layout, and it's very easy to follow what's going on. Perl can frequently look noisy, and new programmers particularly can have difficulty trying to understand the behavior they see from their Perl scripts. Python programmers report that after a small training period, they can produce working code about as fast as they can type, and that they begin to think of Python as *executable pseudocode.*

Python gives programmers good support for modern programming practices like design of data structures and object-oriented programming. The compactness causes programmers to write readable, maintainable scripts by eliminating much of the cryptic notations in Perl. In Perl's original application domains, Python comes close but rarely beats Perl for programming flexibility and speed. On the other hand, Python is proving quite usable well beyond Perl's best application domains.

6.2 Ruby

Ruby is another language that is advertised as being a natural successor to Perl. It was developed in Japan in 1993 by Yukihiro Matsumoto, and began attracting a user community in the United States by the year 2000. In Japan, Ruby has overtaken Python in popularity. Like Python, Ruby is open sourced and so is easy to extend, correct or modify by others.

Matsumoto was looking to design an object oriented scripting language that did not have the messy "Swiss Army chainsaw" aspects of Perl, but he considered Python to be not object oriented enough. Ruby has retained many of the text manipulation and string matching features of Perl that Python leaves out, but they are elevated to the class level (e.g., regular expressions are classes). Even operators on the base types are methods of the base classes. In addition to classes, Ruby allows metaclass reasoning, allowing scripts to understand and exploit the dynamic structure of objects at runtime. Ruby is best thought of as having modern object semantics, as Python does, but also retaining more of the Perl features and syntax than Python does.

7 For More Information

If you are looking to learn programming this text teaches it, using Perl:

- *Elements of Programming with Perl,* by A. L. Johnson (Manning Publications, October1999)

These books give more details on the Perl language for readers who understand programming:

- *Perl: The Programmers Companion*, by N. Chapman (John Wiley and Sons, September 1997)
- *Programming Perl,* by L. Wall, T. Cristiansen, and J. Orwant (O'Reilly and Associates, July 2000, 3rd ed.)
- *Perl in a Nutshell*, by E. Siever, S. Spainhour, and N. Patwardhan (O'reilly and Associates, June 2002)
- *Learning Perl: Making Easy Things Easy and Hard Things Possible,* by R. Schwartz and T. Phoenix (O'Reilly and Associates, July 2001, 3rd ed.)
- *Perl Cookbook: Solutions and Examples for Perl Programmers*, by T. Christiansen and N. Torkington (O'Reilly and Associates, August 1998)
- *Perl Pocket Reference*, 4th Edition, by J. Vromans (O'Reilly and Associates, July 2002)

These texts give detailed, "under the covers" explanations of the advanced features in Perl:

- *Advanced Perl Programming,* by S. Srinivasan (O'Reilly and Associates, August 1997)
- *Object Oriented Perl,* by D. Conway (Manning Publications, August 1999)
- *Mastering Regular Expressions,* by J. E. F. Friedl (O'Reilly and Associates, July 2002, 2nd ed.)

These references show how to apply Perl and Perl modules in specific application domains:

- *Writing Apache Modules with Perl and C,* by L. Stein and D. MacEachern (O'Reilly and Associates, March 1999)
- *Perl and LWP,* by S. M. Burke (O'Reilly and Associates, June 2002)
- *Perl and XML*, by E. T. Ray and J. McIntosh (O'Reilly and Associates, April 2002)
- *CGI/Perl Cookbook*, by M. Wright (Wiley Computer Publishing, October 1997).
- *CGI Programming with Perl,* by S. Guelich, S. Gundavaram, and G. Birznieks (O'Reilly and Associates, July 2000, 2nd ed.)

These references give more information on languages with some similarity to Perl:

- *Programming Ruby: The Pragmatic Programmer's Guide*, by D. Thomas and A. Hunt, (Addison Wesley Longman, October 2000), http://www.rubycentral.com/book/
- *The Quick Python Book*, by D. Harms and K. McDonald (Manning Publications, October 1999)
- *Learning Python*, by M. Lutz and D. Ascher (O'Reilly and Associates, April 1999)
- *Python Essential Reference*, by D. Beazley (New Riders, 2001, 2nd ed.)

These web sites contain extensive information about the Perl language definition and standard, tutorials on programming in Perl, example programs and useful scripts, libraries, and upcoming conferences:

- http://www.perl.com/ main perl commercial distribution site
- http://cpan.org/ perl archives
- http://use.perl.org/ news clearinghouse
- http://history.perl.org/PerlTimeline.html specific development timeline
- http://www.perl.org/ Perl mongers
- http://dev.perl.org/perl6/ latest in development of Perl 6
- http://www.activestate.com/ Perl implementations for Windows platforms
- http://history.perl.org/ perl through the ages

Finally, there are the Perl newsgroups. Use your favorite news reader to access these groups. There you will find discussions about Perl, and a place to ask and answer questions.

- comp.lang.perl.misc The Perl language in general.
- comp.lang.perl.announce Announcements about Perl. (moderated)