Lecture 4: Introduction of Programming Language Perl

(02-10,17-2010)

(Reading: Lecture Notes)

Lecture Outline

1. Overview
2. Constants and literals
3. Variables, arrays, and associative arrays
4. Operators and expressions
5. Control constructs
6. Patterns and pattern matching operators
7. Subroutines
8. Input and output
9. Misc
10. Examples

1. Overview

(1) Salient features

a. Perl is an interpreted language (hence programs are also called Perl scripts)

b. Perl is a type-less language

(a) Perl has been influenced by UNIX shells. Many of its features are from or extensions of UNIX shells.

(b) Variables do not have to be declared before being used

(c) Variables can assume any type of values over their existence

(d) The notion of *associative arrays* can allow arrays with arbitrary subscripts

(e) Strong pattern matching ability

(f) Flexible operations on and conversions between scalar and arrays (called *lists*). This flexibility makes it appealing as a CGI language

(g) Availability of large library functions makes it convenient for various system and application programming

c. Originated from UNIX systems, Perl is now supported by all major platforms (many dialects of UNIX, Windows, VMS, MacOS)

(2) Invoking a script of an interpreted language (in UNIX systems):

a. Invoking the script name directly:

(a) The mode of the script must be *executable*

(b) The first line, with the notion such as:

```
#!/usr/bin/ksh
```

is critical. It specifies which interpreter will be invoked.

b. Call the interpreter to invoke the script directly:

(a) The mode of the script *does not* have to be *executable*. But it must be *readable*.

(3) An introductory example: list the login names and full names of all users in a classical UNIX password file. The following Perl script, named *passwd-list.pl*, will perform the desired actions:

```
#!/usr/local/bin/perl
# Change above line to your absolute path to perl

while (<>) {
    @fields  = split /:/, $_;
    $login_name=@fields[0];
    $full_name=@fields[4];
    print "User ${login_name}'s full name is $full_name\n";
}
```

This Perl script can be invoked by the syntax

perl passwd-list.pl /etc/passwd

2. Constants and literals

(1) Numbers

a. Can be both integer numbers and decimal point numbers. Scientific number expressions such as 39.10e2 are also supported.

b. Normal arithmetic is performed in *double* precision, as in C's double type

c. The standard Perl library (Table 7-6,p.376) include four modules: *integer*, *Math::BigFloat, Math::BigInt*, and *Math::Complex*. The two Big modules allow arbitrary precision floating and integer arithmetics.

(2) Character strings

a. A character string enclosed in a pair of double quotation marks " is char constant. However, *variable interpolation* (i.e. substitution) and *backslash interpolation*

b. A character string enclosed in a pair of single quotation marks ' is char constant and will not subject to interpolation

c. Examples:

$answer = 100;
$x = "$answer-1";
$y = '$answer-1';
$z = ‘finger peng‘;
$p = ‘pwd‘;
print "Value x=$x\n";
print "Value y=$y\n";
print "Value z=$z\n";
print "Value p=$p\n";

3. Variables and their syntax

(1) Main types of variables: *scalar, array* (*list*), *hash* (*associative array*), *subroutine*, and *typeglob*

(2) Variable syntax: each type of variable begins with a special letter. I.e. the beginning letter indicates the nature of the variable.

Type	Begins with character	Example	Is a name for:
Scalar	$	$login_name	An individual value (number or string)
Array	@	@whole_line	A list of values, keyed (indexed) by number
Hash	%	%interest_rates	A group of values, keyed (indexed) by string
Subroutine	&	&login_check	A callable group of Perl code
Typeglob	*	*struck	Everything named *struck*

Perl variable names are case sensitive

(3) Scalar variables

a. Scalar variables are those that can hold a single value, such as a number, a char string, and a file handle

b. Scalar variables always begin with the dollar sign $

(4) Array variables

a. Arrays are ordered list of scalars (hence arrays are also called lists). Array variables begin with the @ letter.

b. Arrays are indexed (or keyed) with integer subscripts normally beginning from 0 (same as in C). Array subscripts are enclosed in pair of brackets [and].

c. If @x is an array, then its elements can be referred by expressions like:

@x[1]

Since array elements are scalars, they can also be be referred by expressions like:

$x[1]

d. Perl arrays have no predefined upbounds. In fact the size of an array can change dynamically at run-time. At any time just calling the array name like

@x

will return the length of the array, i.e. the number of elements in the array.

e. An array variable can be assigned a list of values by using a list expression like:

@x = ("red", "green", "black", "white")

On the other hand, an expression like:

$x = ("red", "green", "black", "white")

will only assign the last element "white" of the list the scalar variable $x

(5) Some special variables

a. Global special variables

Name	Meaning
$_ , $ARG	The default input and pattern searching space
$., $NR $INPUT_LINE_NUMBER	The current input line number of last read file
$?	The status returned by last executed backtick (“) command or the last pipe close
$[	The index of the 1st element of an array, or the first char in a substring. Use is discouraged
$#array_name	The subscript of the last element in array_name This value is always one less than the number of elements in array_name
$/ $,	Input record separator. Default is new line char Input field separator

b. Global special arrays

Name	Meaning
@ARGV	The array containing the command-line arguments

c. Local special arrays

Name	Meaning
@_	The array containing the whole formal parameter

(6) Important array functions

a. The *split* function. A commonly used method of creating an array with initial values is using the *split* function. This function can break a char string into individual words and assign them to the array.

(a) Example:

```
$x = "One Two Three";
@y = split / /, $x;
print "$y[0], $y[1], $y[2]\n";
```

(b) Syntax of the *split* function:

```
split /PATTERN/, EXPR, LIMIT
split /PATTERN/, EXPR
split /PATTERN/
split
```

where PATTERN is a regular expression, EXPR an expression that evaluates to string, and LIMIT an arithmetic expression. If LIMIT is non-negative, at most that many fields will be split. The last form of *split* will split the value in the internal variable $_

(c) Semantics: *split* searches for patterns expressed by PATTERN in EXPR.

b. The *shift* function. This function shifts the first (left most) value of a given array off and returns that value.

(a) Syntax

```
shift ARRAY
shift
```

(b) Example: (to be given shortly).

Without an array, the function will either work on @ARGV (if called inside the main function) or @_ (if called inside a subroutine).

c. The *pop* function. This function is similar to the *shift* function. This function pops the last (right most) value of a given array off and returns that value.

(a) Syntax

```
pop ARRAY
pop
```

(b) Example: (to be given shortly).

d. The *push* function. This function pushes a list to the end of an given array (thus extending the length of that array). The function returns the new length as its return value.

(a) Syntax

```
push ARRAY, LIST
push
```

(b) Example: (to be given shortly).

e. The *unshift* function. This function does the opposite of the *shift* function. It prepends a list values to the front (left side) of an array and returns the new length of the array as its return value.

(a) Syntax

unshift ARRAY, LIST
unshift

(b) Example: (to be given shortly).

Without an array, the function will either work on @ARGV (if called inside the main function) or @_ (if called inside a subroutine).

f. The *splice* function.

(a) Syntax

splice ARRAY, OFFSET, LENGTH, LIST
splice ARRAY, OFFSET, LENGTH
splice ARRAY, OFFSET

(b) Semantics. This function removes the elements designated by OFFSET and LENGTH (beginning from OFFSET of length LENGTH) from ARRAY, and replaces them with elements in LIST. The given array can grow or shrink in size depending on the given parameters. It returns the removed elements as return value.

(c) Relations with several other functions:

Direct Method	Splice Equivalent
push(@a,$x, $y)	splice(@a, $#a+1, 0, $x, $y)
pop(@a)	splice(@a, −1)
shift(@a)	splice(@a, 0, 1)
unshift(@a,$x, $y)	splice(@a, 0, 0, $x, $y)
$a[$x] = $y	splice(@a, $x, 1, $y)

(d) Example: (to be given shortly).

g. The *chop* and *chomp* functions.

(a) Syntax. The function applies to both variables and arrays.

chop VARIABLE
chop LIST
chop

Without parameter, it applies to $_

(b) Semantics. This function remove the last char of a string. Applied to an array, it remove the last char of every element of the array.

(c) *chomp* is a safer version of *chop*. It does not do an arbitrary chopping of last char of a given string. It will remove the last char only when the last char is equal to the char in the input record separator variable $/

h. Example 1:

```
#!/usr/local/bin/perl

@a = ('This', 'is', 'a', 'funny', 'var');

$x = @a[0];
$y = @{a}[1];
$z = @a[2];
$v = $a[3];
$w = $a[4];
$s=@a;

print "$x $y $z $v $w\n";
print "The size of the variable @a at this time is $s\n";

$x = "One Two Three Four Five Six Seven Eight Nine Ten";
@y = split / /, $x, @a*2;

print "$y[0]_$y[1]_$y[2]_$y[3]_$y[4]_$y[5]_$y[6]_$y[7]_$y[8]_$y[9]\n";

@y = split /([-,])/, "1-10,20";

print "$y[0]_$y[1]_$y[2]_$y[3]_$y[4]_$y[5]_$y[6]_$y[7]_$y[8]_$y[9]\n";
```

i. Example 2:

```
#!/usr/local/bin/perl

$x = "One Two Three Four Five Six Seven Eight Nine Ten";
@y = split / /, $x, @a*2;

$s = shift @y;
$len=@y;

print "$s|$y[0]|$len\n";

$s = pop @y;
$len=@y;

print "$s|$y[0]|$len\n";

$len = push @y, ("Red", "Green", "Black", "White");
print "$s|$y[$len-1]|$len\n";

$s = chop @y;
$len=@y;
print "\$s=$s, after chopping, the list of @y is now $y[$len-1]\n";
```

j. Example 3:

```
#!/usr/local/bin/perl

sub list_eq {

    my @a = splice(@_, 0, shift);
    my @b = splice(@_, 0, shift);

    my $len_a = @a;
    my $len_b = @b;

    return 0 unless @a == @b;

    print "$len_a=$len_b, the two lists are of the same length\n";
    while (@a) {
        print "Compare $a[@a-1] with $b[@b-1]\n";
        return 0 if pop(@a) ne pop(@b);
    }
    return 1;
}

$x1 = "One Two Three Four Five Six Seven Eight Nine Ten";
@y1 = split / /, $x1;

$x2 = "One Two Three Four Five Six Seven Eight Nine Ten";
@y2 = split / /, $x2;

$len_y1 =@y1;

$diff = list_eq ($len_y1, @y1, scalar(@y2), @y2);

if ($diff == 1) {
   print "The two lists are the same\n";
} else {
   print "The two lists are different\n";
}
```

(6) Hash variables

a. A *hash* is a special array such that each of its element has a *key* that serves as index to that element. The *key* of an hash array element can be any string, not

just limited to integer.

b. If a list is assigned to a hash variable, Perl will take pair of list elements and assign the first of the pair as the key and the second of the pair as the value with that key. Example:

```
#!/usr/bin/perl

%color = ("red", 100, "green", 200, "black", 300, "white", 400);

print "$color{red}, $color{green},$color{black}, $color{white}\n";
```

Notice that hash variables begin with % at the left-hand side. When using its right-hand values, it begins with $

c. A more readable form of initializing a hash is to use the => operator:

```
#!/usr/bin/perl
%color = (
        red =>   100,
        green => 200,
        black => 300,
        white => 400
);
print "$color{red}, $color{green},$color{black}, $color{white}\n";
```

d. Hash is implemented with the hash table concept. When called without key as index, it returns the number of buckets actually used over the number of allocated buckets.

$$\$x = \%color$$

could assign $x the value 3/8, for example.

4. Operators and expressions

(1) Context

a. The notion of context in Perl determines how Perl evaluates variables.

b. There two major contexts: *scalar* context and *list* context.

c. Scalar context

(a) Used when a single scalar value is being obtained.

(b) Example: @days + 0 would return the number of elements in the list day. The arithmetic operator + and the operand 0 together determine the scalar context.

d. List context

(a) In this context a list value will be obtained.

(b) Example:

```
#!/usr/bin/perl

@whole = split / /, "This is an example string";

($x, $y, @z) = @whole;

print "\$x=$x, \$y=$y, \$z[0]=$z[0], \$z[1]=$z[1], \$z[2]=$z[2]\n";
```

will assign "This" to $x, "is" to $y, "an" to z[0], and etc.

(2) Arithmetic operators (listed on p.464)

```
+  -  *  /  %  **  <<   >>  x  .
+= -= *= /= %= **= <<=  >>= x= .=
```

a. The >> and << are right and left shift operators. Example:

$$1 << 4 \quad \# \text{ returns } 16$$

b. The x operator is the *repeat* operator for strings.

```
$a=123;
$b=3;
print $a x $b;
print "\n";
$a x= $b;
print $a;
```

Both *print* would print 123123123

c. The . and . = are string concatenation operators.

(3) Comparison operators (listed on p.464)

```
<  <=  >  >=  == != <=>
lt le gt  ge  eq ne cmp
```

a. The first line operators are for numbers. The 2nd line are for strings.

b. The operator `<=>` (and cmp) returns -1 if left operand is smaller than right operand, 0 if they are equal, and 1 if left is larger than right.

(4) Bit and unary operators (p.465)

```
&  ^  |  !  ~
```

Name	Meaning
&	Bitwise and
^	Bitwise xor
|	Bitwise or
!	Logic negation
~	Bitwise negation

(5) Increment and decrement operators (p.465)

```
++  --
```

5. Control constructs

(1) Assignment operation and sequence operation (;)

a. The assignment operator is =, same as in C/C++.

b. The sequence operator is semicolon ;, again the same as in C/C++.

(2) Blocks

a. A sequence of statements that defines a scope is called a *block*.

b. The most frequently used block is delimited by a pair of braces: { and }

c. Other block delimiters: the body of a subroutine; the file containing it (with the **required** file); the extent of a string (in case of an **eval** function)

(3) Conditional statements

a. *if* (EXPR) BLOCK;

```
if ( $s eq "This" ) $y++;
```

b. *if* (EXPR) BLOCK *else* BLOCK;

c. *if* (EXPR) BLOCK *elsif* (EXPR) BLOCK ...;

d. *if* (EXPR) BLOCK *elsif* (EXPR) BLOCk ... *else* BLOCK;

e. *unless* can replace *if* in a. Its meaning is opposite of *if*:

```
$y++ unless $s ne "This";
```

(4) Iteration statements

a. **while** statement: it is similar to the **while** loop in C/C++

(a) Syntax

```
while (EXPR) {
   ...
}
```

(b) Example: omitted

b. **for** loop: it is also pretty much like the **for** statement in C/C++

(a) Syntax

```
for (INITIAL;EXPR;ADJUST) {
   ...
}
```

(b) Example: omitted

c. **foreach** loop. This loop has been used more than the **for** loop construct due to its ability to handle lists and hashes naturally.

(a) Syntax

```
foreach VAR (LIST) {
   ...
}
```

(b) Semantics: The variable VAR takes each value from the LIST during each iteration until the LIST is exhausted

(c) Example 1: p.100

```
foreach $count (10,9,8,7,6,5,4,3,2,1,'BOOM') { # Do a countdown
   print $count, "\n"; sleep(1);
}
```

(d) Example 2: p.24

```
foreach $user (@users) {
   if (-f "$home{$user}/.nexrc") {
      print "$user is cool .. they user a pser-aware vi!\n";
   }
}
```

(e) Example 3: p.100. The function *keys* returns the list of keys from the hash array *hash*. That list is sorted and is used by the **foreach** loop.

```
foreach $key (sort keys %hash) { # sorting keys
   print "$key => $hash{$key}\n";
}
```

d. Loop control

(a) *last* LABEL and *next* LABEL

i. The *next* statement skips the rest of the loop and starts next iteration.

ii. The *last* statement skips the rest of the loop and exits the body of the enclosing loop

(b) *redo* LABEL will restart the loop without evaluating the loop condition again.

(5) *goto* function

a. Three forms of goto functions:

```
goto LABEL;
goto EXPR;
goto &NAME;
```

b. The first form has normal semantics: the control switches to the statement labeled with LABEL.

```
#!/usr/bin/perl

$x="This is x";
goto L1;
$y="This is y";
L1: $z="Tha is z";
print "x=$x, y=$y, z=$z\n";
```

c. The second form generalized the second form: the expression must be evaluated to a label name.

```
goto ("FOO", "BAR", "GLARCH")[$i]
```

d. The third form has special meaning that is different from the normal goto statement/function. It substitutes a call to the named subroutine for the currently running subroutine.

(6) *die* statement: terminate the script.

```
((-e $source_dir) && (-e $target_dir)) ||
    die "At least one of the two given directories does not exist. Give up!\n";
```

The char string is optional. If present it will be output to STDERR

6. Patterns and pattern matching operators

(1) Regular expressions

a. Very similar to those in UNIX systems shells and UNIX utilities such as *vi*, *ed* and *egrep*

b. The simplest regular expressions are those normal character strings. They do not have special meanings and just represent themselves.

c. Meta characters – they are characters with special meanings, i.e. they do not express themselves. Instead they carry special meanings.

```
\ | ( ) [ { ^ $ * + ? .
```

These meta characters have the following special meanings:

Metacharacter	Meaning
\	Escape character; enclosing patterns
|	Alternation (ie selection)
()	Grouping
[]	Any one of a collection of characters enclosed
{}	Quantifiers, specifying matching range
$	End of the line
^	Beginning of the line
*	Quantifier, repeating zero or more times
+	Quantifier, repeating one or more times
?	Quantifier, zero or one time
.	Matching any one char, except \n

d. Rules of regular expression matching by Perl engine: six rules (listed p.60-65)

Rule 1 the engine matches as far left in the string as it can, such that the entire expression matches under Rule 1.

Rule 2 If the whole regular expression consists of a set of alternatives (the | operator), this set of alternatives matches a string if any of the alternatives matches under Rule 3. Alternatives are tried from left to right and the Engine stops as soon as the first match occurs.

Rule 3 an alternative matches if every item in the alternative matches sequentially according to Rules 4 and 5. An item consists of either an *assertion* or a *qualified atom*.

Rule 4 an assertion matches according to the following table:

Assertion	Meaning
^	Matches at the beginning of the string (or line, if /m used)
$	Matches at the end of the string (or line, if /m used)
\b	Matches at word boundary (between \w and \W)
\B	Matches except at word boundary
\A	Matches at beginning of string
\Z	Matches at end of string
\G	Matches where previous m//g left off
(? = ...)	Matches if engine would match ... next
(?!...)	Matches if engine wouldn't match ... next

Rule 5 a qualified atom matches only if the atom itself matches some number of times specified by the qualifier:

Maximal	Minimal	Meaning
$\{n, m\}$	$\{n, m\}?$	Must occur at least n times but no more than m times.
$\{n,\}$	$\{n,\}?$	Must occur at least n times
$\{n\}$	$\{n\}?$	Must match exactly n times
$*$	$*?$	0 or more times (same as {0,})
$+$	$+?$	1 or more times (same as {1,})
?	??	0 or 1 time (same as {0,1})

Rule 6 each atom matches according to its type, as described below:

* A regular expression in parentheses, (...), matches whatever the regular expression matches according to Rule 2. Parentheses serves as a grouping operator. A side effect of parentheses is that the Engine remembers the matched substrings as $1, $2, ... that can be retrieved and used later.
* A "." char matches any character except the newline char \n
* A list of characters in square brackets (called *character class*) matches one of the characters in the list.
  - A caret char at the front of the list matches only characters not in the list.
  - Character ranges are specified using the $a - z$ notation.
  - The backslashed characters \d, \w, \s, \n, \r, \t, \f, plus the \nnn, can also appear in the list.
  - Most metacharacters, except the backslash char \, lost their special meanings in the list.
  - A backslashed letter matches a special char or char class:

Code	Matches
\a	Alarm (beep)
\n	Newline
\r	Carriage return
\t	Tab
\f	Formfeed
\e	Escape
\d	A digit, same as [0-9]
\D	A non-digit
\w	A word character (alphanumeric), same as [a-zA-Z0-9]
\W	A non-word char
\s	A white space character, same as [\t\n\r\f]
\S	A non-white space character

  - Backreferences. \0 matches the null char. Other backslashed single- or two-digit numbers match whatever the corresponding parentheses actually matches.
  - A backslashed two- or three–digit octal number such as \033 matches

the char with the specified value, unless it can be interpreted as back-reference.

e. Examples:

(a) Example 1: swap first two words:

```
s/^([^ ]+) +([^ ]+)/$2 $1/;
```

(b) Example 2: match any pattern: word = word

```
/(\w+)\s*=\s*\1/;
```

(c) Example 3: match lines with at least 80 chars:

```
/.{80,}/;
```

(d) Example 4: pull fields out of a line:

```
if (/Time: (....):(.):(.)/) {
   $hours   = $1;
   $minutes = $2;
   $seconds = $3;
}
```

(e) Example 5: *sed* is UNIX command that performs basic text transformations on input files. It uses notions of regular expressions and pattern matching similar to many other UNIX commands and Perl. Each of the input line in the following sed command has the form:

soc.culture.usa 0000089889 0000087966 y

The following sed command will transform such a line to a line of the form

soc.culture.usa 0000000000 0000000001 y

Notice that in sed the grouping parentheses have to be escaped using back-slash character.

```
sed < active.old > active.new \
   -e 's/^\([^ ]*\) [0-9]* [0-9]* \([^ ]*\)$/\1 000000000 000000001 \2/'
```

(2) Pattern matching operators

a. General forms of pattern matching operators:

m/PATTERN/gimosx
/PATTERN/gimosx

b. Semantics:

(a) The operator searches a string for a pattern match.

- In a scalar context it returns either true (1) or false ("").
- In a list context, it returns a list of matched patterns as ($1,$2,...) if successful, or a null list if matching fails.

(b) If the searched string is not specified (via the =~ or =! operator), the string in $_ is searched.

(c) The delimiter can be any char. If the delimiter is the symbol '/', the initial char *m* can be omitted.

(d) The six characters g,i,m,o,s, and x are called modifiers. They supply supplemental information about how patterns are matched:

Modifier	Meaning
g	Match globally
i	Case-insensitive pattern matching
m	Treat string as multiple lines
o	Compile pattern only once
s	Treat string as single line
x	Use extended regular expressions

c. The binding operators =~ and !~

(a) The binary operator =~ *binds* a scalar expression to a pattern match, substitution, or translation operation. For example, the expression:

if ($string =~ /pat/)

will perform the pattern matching using the pattern pat against the value of the variable $string.

(b) The binary operator !~ is like =~, but it returns a value that is logical negation of =~.

(c) By default the match, substitution, and translation operations operate on the variable $_. The binding operator changes this default behavior.

(d) Substitutions and translations return the number of successful substitutions and translations. The values returned for match operations depend on the context.

d. File test operators. They are unary operators that take one parameter (either a file name or file handle), tests the given file for truth of the specified property. Some of them:

Operator	Meaning
-f	File is a plain file
-d	File is a directory
-l	File is a symbolic link
-S	File is a socket
-e	File exists
-z	File has zero size
-s	File has non-zero size
-r	File is readable by effective uid/gid
-w	File is writable by effective uid/gid
-x	File is executable by effective uid/gid
-o	File is owned by effective uid
-M	Age of file in days since last modification
-A	Age of file in days since last access
-C	Age of file in days since last inode change
-T	File is a text file
-B	File is a binary file

(a) Example 1: read from input until a char string that is the name of a plain file is read.

```
while (<>) {
    chomp;
    next unless -f $_;
    ...
}
```

(b) Example 2:

```
if ( -d "$a/$a_s_file_name" ) {
    ...
}
```

7. Subroutines

(1) Declaring and defining subroutines. Subroutines can be declared (without giving body of the subroutines) alone or declared and defined together.

a. Declaring subroutines

```
sub Name;           # A "forward" declaration
sub Name (PROTO);   # A "forward" declaration with prototype
```

b. Declaring and defining subroutines

```
sub Name BLOCK;           # A declaration and a definition
sub Name (PROTO) BLOCK;   # A declaration and a definition with  prototype
```

c. Defining an anonymous subroutine:

```
$subref = sub BLOCK;
```

d. Importing subroutines defined in another package:

```
use PACKAGE qw(NAME1 NAME2 NAME3 ...);
```

(2) Calling subroutines

a. To call a subroutine directly:

```
NAME(LIST);       # Symbol & optional with parentheses
NAME LIST;        # Parentheses optional if predeclared or imported
&NAME;            # Passes current @_ to subroutine
```

b. To call a subroutine indirectly:

```
&$subref(LIST);      # Symbol &  not optional on indirect call
&$subref;       # Parentheses optional if predeclared or imported
&NAME;          # Passes current @_ to subroutine
```

(3) Formal parameters and their passing

a. Perl's parameter passing is special: values all formal parameters are passed as a single flat list of scalar values. Same for return values. There are no prespecified type and number of parameters for any subroutine. A subroutine can be called with any number of parameters of any type. Values of all these parameters are flattened and passed as a list of scalar values.

(a) The subroutine implementation must know what parameters are to be given in order to extract and use them.

(b) A subroutine can return any type of values and different type values according to different parameters passed. All of them are returned as a list of scalar values.

b. If no parameter is provided when calling a subroutine, the value in the special global variable @_ can be used by the subroutine to obtain parameter values.

c. Example: the *list_eq* subroutine defined as Example 3 previously.

(4) Subroutines declared with prototypes

a. Subroutine prototype has been provided since Perl 5.003. The prototype associated with a function declaration/definition places certain constraints on the type and number of parameters.

b. Example: (p.119)

Declared as	Example calls
sub mylink ($$)	mylink $old, $new
sub myvec ($$$)	mylink $var, $offset, 1
sub myindex ($$;$)	myindex &getstring, "substr"
sub mysyswrite ($$$;$)	myindex $buf, 0, length($buf) - $off, $off
sub myreverse (@)	myreverse $a,$b,$c
sub myjoin ($@)	myjoin ":", $a,$b,$c
sub mypop (\@)	mypop @array
sub mysplice (\@$$@)	mysplice @array,@array,0,@pushme
sub mykeys (\%)	mykeys %{$hashref}
sub myopen (*;$)	myopen HANDLE, $name
sub mygrep (&@)	mygrep { /foo/ } $a,$b,$c
sub mytime ()	mytime

(5) Scoping rules

a. Perl variables by default are all *global*. That means that a normal Perl variable assigned a value anywhere is visible anywhere else.

b. In subroutines a variable can be declared as *local*. Such variables are visible within the body of the subroutine and are also visible by subsequent recursive calls of the function. However a *local* variable's value is lost once the subroutine terminates.

c. In subroutines a variable can be declared as *my*. Such variables are visible within the body of the subroutine, but are not visible by subsequent calls of the same function. The value of a *local* variable is lost as soon as the using subroutine terminates

d. Example. In the following example, watch for the value changes of variable $t, $l, and $my_var

```
#!/usr/local/bin/perl

$t=1;
$x=sub_test($t);

print "x=$x, y=$y\n";

print "www=$www\n";
print "l=$l\n";

sub sub_test {

        local($l);
        my $my_var;

        $l=$_[0];
        $my_var++;

        print "In sub_test, l=$l\n";
        $l=$l+13;
        $y=10;
        $www=100;
        print "\$my_var=$my_var\n";
        if ($t == 1) {
           $t++;
           sub_test($t);
        }
        return $l;
}
```

8. Input/Output and format

(1) Input operations

a. Input from executing shell commands: the backtick notions (eg. ‘date‘)

b. Line input operator: <FILEHANDLE>

(a) Input from stdin: the STDIN file handle. Example: the following are equivalent

```
while (defined($_=<STDIN>)) {print $_;} # The long way. defined is function
                                        #   testing if an expr has value
while (defined(<STDIN>)) {print;}        # The short way
```

```
for (;<STDIN>;) {print;}                # while loop in disguise
print $_ while defined($_=<STDIN>);     # Long statement modifier
print $_ while <STDIN>;                 # Short statement modifier
```

Get a single line from STDIN:

```
$x=<STDIN>;    # Do not forget the colon char!
```

(b) Reading a single char from stdin: *getc* function:

```
getc FILEHANDLE
getc
```

If FILEHANDLE is omitted, it reads from STDIN.

(2) File opening: the *open* function

```
open FILEHANDLE, EXPR
open FILEHANDLE
```

a. Semantics: the function opens the file whose filename is given by EXPR, and associate it with the given FILEHANDLE. If EXPR not present, the scalar variable of the same name as the FILEHANDLE must contain the filename.

b. The three file handles STDIN, STDOUT, STDERR for stdin, stdout, stderr are open by default and there is no need to open them.

c. The EXPR can begin with char/strings $>, >>, <$, which indicates that the file is opened for output, for appending, and for input. A + sign in front of $>$ or $<$ indicates both read write.

d. If EXPR begins with a letter |, the filename is interpreted as a command to which the output to be piped.

e. If EXPR ends with a letter |, the filename is interpreted as a command which pipes input to the script.

f. Examples:

(a) Open a file without EXPR

```
$ARTICLE = "/usr/local/news/spool/articles/comp/lang/perl/112233";
open ARTICLE or die "Can't find article $ARTICLE: $!\n";  # $! contains errno
while (<ARTICLE>) { ... }
```

(b) Open a file for appending

```
open(CLASS_ADMIN_FILE_F,">>$class_account_file");
```

(c) Open a file for output piping:

```
if (!($opt_nis || $opt_nisplus)) {
   open(ADDUSER, "| /usr/local/sbin/my_adduser") unless ($test_on);
}
```

(3) Formats

a. Perl formats are intended to provide simple and easy to use formatted output and reports

b. Output format declarations

```
format NAME =
FORMLIST
.
```

Notice the format ends with a dot sign '.' in the last line. Each line of the FORMLIST can be one of the three types:

(a) A *comment*, indicated by the '#' at the first column of the line;

(b) A *picture* line, providing the format for one output line;

(c) An argument line supplying values to plug into the previous picture line.

c. Picture lines are printed as they are except for certain fields that substitute values into the line.

(a) The letter @ is a place holder for substitution.

(b) The caret letter ^ is another special place holder letter. So is the string @*

d. Examples.

(a) Example 1

```
#
# Format for the admin file for the class (one such file for
# each Class,Section,and Semester:
# login_name uid  full_name vms_user_name initial_pw exp_date exp_time
#    9        6      24           15             9          9       10
format CLASS_ADMIN_FILE_F =
@<<<<<<<<@<<<<<@<<<<<<<<<<<<<<<<<<<<<<<@<<<<<<<<<<@<<<<<<<<@<<<<<<<<
$login_name,$uid,$full_name,$vms_user_name,$password,$expiration_date
.
```

(b) Example 2

```
#
# Format for the account admin file. This file contains the
# account information for all users in the file. It should
# pre-exist before used.
#
#LogName  VMSUNam Class#  Sect# Semester   ExpDate  CreaDate InitPW
#    9       8       8      6      12        9        9       9        9
#@<<<<<<<<@<<<<<<<@<<<<<<<@<<<<<@<<<<<<<<<<<@<<<<<<<<@<<<<<<<<@<<<<<<<<@<<<<<<<<#$login_name,$full_name,$vms_user_name,$password,$exp_date
#@>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
#$full_name
#
format SYSTEM_ADMIN_FILE_F =
@<<<<<<<<@<<<<<@<<<<<<<@<<<<<<<@<<<<<@<<<<<<<<<<<@<<<<<<<<@<<<<<<<<@<<<<<<<<
$login_name,$uid,$vms_user_name,$class_no,$section_no,$semester_without_space,$create_date,$expiration_date,$password
.
```

e. There is a *TOP-OF-FORM* format for each defined format. This format has a name that is same as the format name with _TOP appended to it. This format specifies how to print the top of each page for multi-page output.

f. Format variables

(a) The current format name is stored in the global special variable $~ – changing the value of this variable allows a script to use different output format at different locations.

(b) The current top-of-form format name is stored in the global special variable $^ – changing the value of this variable allows a script to use different top-of-form format at different locations.

(4) The *write* function

a. The output format only declares the form of output. Actual output is initiated by calling the write function.

b. Syntax:

```
write FILEHANDLE
write
```

If FILEHANDLE is not present, the format that is the value of the variable $~ is used.

c. Examples:

(a)
```
if ($opt_h || $opt_help) {
    $~ = "USAGE_OUTPUT_F";
    write;
    die "\n";
}
```

(b)
```
write CLASS_ADMIN_FILE_F;
```

9. Misc

(1) Handling command line options

a. The Getopt module

b. Example:

```
use Getopt::Long;
$Getopt::Long::ignorecase = 0;

##
## Section 1: handling command line options and usage
&GetOptions(split(' ', "d h r t"));

if ($opt_r) {
   $remove_objects_in_target_that_not_in_source = 1;
} else {
   $remove_objects_in_target_that_not_in_source = 0;
}
```

See Example 3 in next section for a more complex handling of command-line options.

(2) References

a. Perl references are similar to the reference concept in C (or pointers). But Perl references are more flexible and more powerful.

b. There are two types of references: *hard references* and *symbolic references*. They are similar to the symbolic links and hard links in UNIX.

c. Hard references

(a) Hard references are declared using the backslash char in front of a variable.

(b) Declaration examples:

```
$scalarref = \$foo;
$constref  = \186_282.42;
$arrayref  = \@ARGV;
$hashref   = \%ENV;
$coderef   = \&handler;
$globref   = \$STDOUT;
```

(3) Packages and modules

a. Packages are units that define their own name space. For example, a variable $x used in a specific package *pk* is only visible inside that package. Packages provide

basic building blocks for modular programming. When a Perl script runs, it has its own default package.

b. A module is a reusable package that is defined in a Perl library file whose name is the same as the name of the package (with .pm on the end). To use objects in another module, explicit operations are needed. Modules are included in a script by using the *use* function or the *required* function.

(4) Special data structures

a. With the help of references, many special data structures like *arrays of arrays, hashes of hashes, hashes of arrays*, and *arrays of hashes* can be defined and used.

b. Other complex data structures can be defined.

(5) TCP/IP API

a. Perl provides a complete suite of TCP/IP socket API functions that can be used for networking programming.

b. Most of these functions have syntax and semantics similar to those in C in UNIX.

10. Examples

(1) Example 1: The grader program: p.10

```
#!/usr/bin/perl

open(GRADES, "grades") or die "Can't open grades: $!\n";
while ($line = <GRADES>) {
    ($student, $grade) = split(/ /, $line);
    $grades{$student} .= $grade . " ";
}

foreach $student (sort keys %grades) {
    $scores = 0;
    $total = 0;
    @grades = split(/ /, $grades{$student});
    foreach $grade (@grades) {
        $total += $grade;
        $scores++;
    }
    $average = $total / $scores;
    print "$student: $grades{$student}\tAverage: $average\n";
}
```

Notes: only one space char is allowed between the student's name and the score on each line.

(2) Example 2: Mirroring a directory

a. Requirements

b. The code and demonstration

```
#!/usr/bin/perl
#
#  This is a pseudo 'mirroring' script. It can be run at scheduled
#      intervals to keep the target directory updated according to
#      the source directory specified.
#
#  Input: two directors, source and target. It is assumed that
#      the target directory is a backup copy of the given source
#      directory.
#
```

```
#  Output: This Perl script compares the given source and target
#      directories, recursively, and performs the following actions
#      at each corresponding level of recursion:
#
#      1. If a file exists in a subdirectory of the source directory's
#         hierarchy and that same file does not exists in the corresponding
#         subdirectory of the target directory's hierarchy, the file
#         will be copied to the corresponding subdirectory;
#      2. If a file exists in both the source directory's hierarchy
#         and target directory's hierarchy, the two copies of the file
#         will be compared. If the copy on the source is newer, this
#         newer copy will be copied to the corresponding target subdirectory.
#      3. If a directory exists in the source hierarchy but does not
#         exist inside the target hierarchy, the directory together with
#         its contents will be copied to the target hierarchy.
#
#   Notes: 1. Both the source and the target directories must
#              pre-exist before calling this script;
#    2. The two directories must be different directories;
#             Mirroring to the same directory does not make sense anyway;
#          3. Without the -d option the source and target directories must
#             have the same same at the lowest level. Namely, source can be
#             /home/john/project,  while target can be /home1/alice/project.
#             Calling this script with parameters such as
#                        /home/john/project /home/john/project1 are not
#             allowed. If you want it to be allowed, use -d option.
#
#   Created:  01/12/2002
#   Modified: 01/31/2002
#   Modified: 02/03/2002
#
#   Further work:
#
#      1. The -r option has not been implemented. With the -r option,
#   files or directories that exist inside the  target directory
#   but do not exist inside the source directory will be removed.
#   But this option can be destructive and should be used carefully.
#

use Getopt::Long;

$Getopt::Long::ignorecase = 0;
```

```
##
## Section 1: handling command line options and usage
&GetOptions(split(' ', "d h r t"));

if ($opt_r) {
   $remove_objects_in_target_that_not_in_source = 1;
} else {
   $remove_objects_in_target_that_not_in_source = 0;
}
$parameter_num = @ARGV;

if (($parameter_num == 1) || $opt_h) {
   $~ = "USAGE_OUTPUT_F";
   write;
   die "\n";
} elsif ($parameter_num == 2) {
   $source_dir = $ARGV[0];
   $target_dir = $ARGV[1];
} else {
   print "Please enter the name of the source directory: ";
   $source_dir = <STDIN>;
   chop $source_dir;
   print "Please enter the name of the target directory: ";
   $target_dir = <STDIN>;
   chop $target_dir;
}

##
## Section 2: body of main function

## Make sure both source and target directories exist
((-e $source_dir) && (-e $target_dir)) ||
    die "At least one of the two given directories does not exist. Give up!\n";

## Make sure both source and target directories are not the same directory
($source_dir ne $target_dir) ||
    die "The source and target directory are the same directories. Give up!\n";

## Make sure both source and target directories are the same same
## at the lowest level. Namely, source can be /home/john/project,
## while target can be /home1/alice/project.
##
```

```
if (!$opt_d) {
    $lowest_s_dir_name=$source_dir;
    $lowest_t_dir_name=$target_dir;
    if ($source_dir =~ /\//) {
       $x = reverse ($source_dir);
       @y = split /\//, $x;
       $z = @y[0];
       $lowest_s_dir_name=reverse("$z");
    }

    if ($target_dir =~ /\//) {
       $x = reverse ($target_dir);
       @y = split /\//, $x;
       $z = @y[0];
       $lowest_t_dir_name=reverse("$z");
    }

    print "$lowest_s_dir_name=$lowest_t_dir_name=\n";
    if ($lowest_s_dir_name ne $lowest_t_dir_name) {

       print "The lowest dir names of the given source and target dir names\n";
       print "    must be the same. You can override this using -d option.\n";
       die   "    Give up!\n";
    }
}

print "\@ARGV[0] = $ARGV[0]\n";
print "source_dir=$source_dir, target_dir=$target_dir\n";

@source_file_list = `ls $source_dir`;
print "source_file_list=@source_file_list\n";

## Global statistics collecting variables
$total_files_updated = $total_files_copied = 0;
$total_directories_copied = $total_same_files = 0;

recursively_compare_and_update ($source_dir, $target_dir);

print "Operation statistics:\n";
print "  Total number of files updated:    $total_files_updated\n";
print "  Total number of files copied:     $total_files_copied\n";
print "  Total number of dir copied:       $total_directories_copied\n";
```

```
print "  Total number of unchanged files:  $total_same_files\n";

##
## Section 3: The main subroutine that does all
##
sub recursively_compare_and_update {

    my $a = shift;
    my $b = shift;

    my $a_s_file_name, $a_target_file_name;
    my @as_file_list, @bs_file_list;

    my $s_d_flag, $t_d_flag;

    my @c;
    my @a_stat, @b_stat;
    my $source_file_exists_in_target;

    @as_file_list = `ls $a`;
    @bs_file_list = `ls $b`;

    while ( $a_s_file_name = shift @as_file_list ) {

        chop $a_s_file_name;

        $s_d_flag = 0; # default value for file type: not directory
        $source_file_exists_in_target = 0;

        if ( -d "$a/$a_s_file_name" ) {
                print "SOURCE file $a_s_file_name is a DIRECTORY\n";
                $s_d_flag = 1;
        }
        elsif ( -f "$a/$a_s_file_name" ) {
                print  "$a/$a_s_file_name is a file\n";
        }

        $bs_file_list_str = `ls $b`;
        if ($bs_file_list_str =~ /$a_s_file_name/) {

           # Have to use original @bs_file_list for each source file under $a
           @bs_file_list = `ls $b`;
           @c = @bs_file_list;
```

```
   # Check each file in the corresponding target directory
   while ( $a_target_file_name = shift @c ) {

      chop $a_target_file_name;
      $t_d_flag = 0; # default value for file type: not directory
      if ( -d "$b/$a_target_file_name" ) {
         print "The target file $a_target_file_name is a directory\n";
         $t_d_flag = 1;
      }
      elsif ( -f "$b/$a_target_file_name" ) {
         print  "$The target file $b/$a_target_file_name is a file\n";
      }

      if ($a_s_file_name eq $a_target_file_name) {
         print "FILE $a_s_file_name exists in both directory $a";
         print " and target dir $b\n";
         $source_file_exists_in_target = 1;

         if (($s_d_flag == 1) && ($t_d_flag == 1)) {
            print " They are both directories\n";
            recursively_compare_and_update
               ("$a/$a_s_file_name", "$b/$a_target_file_name");
         }
         last; # A directory or file exists in both source and
               #  target directory hierarchies. Stop the current
               #  loop on target hierarchy
      }
   }
}

if ($s_d_flag == 0) {  # handling files

   @s_stat = stat("$a/$a_s_file_name");
   print "s_stat=@s_stat\n";
   print "FILE $a/$a_s_file_name is $s_stat[7] bytes\n";
   if ($source_file_exists_in_target) {
         # The stat function returns
         # ($dev,$ino,$mode,$nlink,$uid,$gid,$rdev,$size,$atime,
         #  $mtime,$ctime,$blksize,$blocks)
      @t_stat = stat("$b/$a_target_file_name");
      print "t_stat=@t_stat\n";
      print "FILE $b/$a_target_file_name is $t_stat[7] bytes\n";
```

```
            }
            # Source file newer than target file or does not exist on target
            if (($s_stat[9] > $t_stat[9]) || (!$source_file_exists_in_target)) {
               if (!$opt_t) {
                  `/bin/cp -p "$a/$a_s_file_name" "$b/"`;
                  if ($source_file_exists_in_target)  {
                     $total_files_updated++;
                     print "UPDATE ";
                  } else {
                     $total_files_copied++;
                     print "COPY ";
                  }
                  print " FILE $a/$a_s_file_name to $b\n";
               }
            } else {
                 $total_same_files++;
                 print "File $a/$a_s_file_name and\n";
                 print "     $b/$a_target_file_name are the same,\n";
                 print "      and no updating needed\n";
            }

         } elsif ($source_file_exists_in_target == 0) {
               # Source file is a dir, but it does not exist on the target
               if (!$opt_t) {
                  print "COPY directory $a/$a_s_file_name to $b\n";
                  print "s_d_flag=$s_d_flag, t_d_flag=$t_d_flag\n";
                  $total_directories_copied++;
                  `/bin/cp -p -R "$a/$a_s_file_name" "$b/"`;
               }
         }

    }

}

#
format USAGE_OUTPUT_F =

usage: mirror_a_dir [-d] [-h] [-r] [-t] [source_directory_name target_directory_name]

       -d: The source and the target directories are different at the lowest
           level. Namely, the source can be /home/john/project and target
           can be /home_1/john/project1. The default is that the two directory
```

```
            names at the lowest level must be the same.
        -h: Print this help page.
        -r: Remove files/directories that exist on the target but do not
            exist on the source (not implemented yet).
        -t: Test mode. No real updating or copying will be done.
```

.

(3) The batch_adduser.pl script for adding UNIX user accounts