Introduction to Perl

©Sanjiv K. Bhatia

Department of Mathematics & Computer Science
University of Missouri – St. Louis
St. Louis, MO 63121

Contents

List of Tables

1 Introduction

- Practical Extraction and Report Language
- Other (not so serious) expansions for Perl
  - Pathologically Eclectic Rubbish Lister
  - Practically Everything Really Likeable *about shells, awk, sed, grep, and C combined*
- Interpreted language
  - No need to compile programs
  - Functionally similar to scripts in UNIX shells, `awk`, `sed`, and C programs
    - Has built in utilities to duplicate the functionality of `sed`, `tr`, and `awk`; so no need for separate invocation of those utilities
    - Provides for structured data type support (arrays and lists)
    - Built in debugger
  - Particularly suited to manipulate text files (like `sed` and `awk`) while retaining the ability to do mathematics not available in shell
  - Do not have to learn shell commands to write Perl scripts
  - Scores over C in its regular expression capabilities
  - Great for prototyping applications
  - Original language of the web (before Java)
- GNU product
  - Freely available for a number of platforms – from DOS to UNIX
  - Heavy support in Internet forums
  - Installation and configuration is largely automatic

2 Getting started

- Perl is invoked on the shell prompt by typing the command `perl`
- Before we go any further, let us find what version of Perl are we running

```
$ perl -v

This is perl, version 5.004_01

Copyright 1987-1997, Larry Wall

Perl may be copied only under the terms of either the Artistic License or the
GNU General Public License, which may be found in the Perl 5.0 source kit.
$
```

- Perl commands can be executed on the UNIX command line by using the option `-e`
  - First program to say hello to the world

    ```
    $ perl -e ’print "Hello world\n"’
    Hello world
    $
    ```

- Implicit looping through files
  - `sed` and `awk` commands are applied to each line in the file
  - Perl can be forced to implicitly loop through the file one line at a time by using the switch `-n`
  - Example

    ```
    $ head hoax | perl -ne ’print’
    COMPUTERWORLD 1 April
    CREATORS ADMIT UNIX, C HOAX

    In an announcement that has stunned the computer industry, Ken Thompson, Dennis
    Ritchie and Brian Kernighan admitted that the Unix operating system and C
    programming language created by them is an elaborate April Fools prank kept
    alive for over 20 years.  Speaking at the recent UnixWorld Software Development
    Forum, Thompson revealed the following:

    "In 1969, AT&T had just terminated their work with the GE/Honeywell/AT&T Multics
    $ head hoax | perl -ne ’print if /Unix/’
    Ritchie and Brian Kernighan admitted that the Unix operating system and C
    alive for over 20 years.  Speaking at the recent UnixWorld Software Development
    $
    ```

  - Perl can be used as a filter as shown in the above examples
  - When using Perl as a filter, use the `-n` switch to implicitly loop through the entire file
  - Being a filter, Perl can accept input by redirection (from `stdin`), send output by redirection (to `stdout`), or be a part of a pipeline of commands
- Perl uses regular expression metacharacters to specify the patterns to be matched

3 Writing Perl scripts

- Similar to UNIX shell scripts
- List of Perl statements and declarations
- Each statement is terminated with a semicolon (`;`)
- You can declare variables anywhere in the system
  - If you forget to initialize variables, they autoomatically get initialized to `0` or `NULL`, depending on context
- Perl goes over each command only once, unless you provide explicit loop or implicit loop (through `-n` option) [Difference from `sed` and `awk`]

- The first line in every Perl script is

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

  to indicate the location of Perl in your system

  - Check your local system on the location of Perl by using the command

    ```
    % which perl
    ```

- You can put comments in Perl scripts at any point
  - The comments are just like the comments in shell
  - Comments are started with the symbol # and continue to the end of line

3.1 The first Perl script

- Let us write the script to say hello to the world

  ```
  $ cat perl/hello
  #!/usr/bin/perl

  # My first Perl script to say Hello to the World

  print "Hello world\n"                  # The print statement
  ```

- Do not forget to give execute permission on the file

  ```
  $ chmod a+x perl/hello
  ```

- Checking the syntax of the script

  ```
  $ perl -c perl/hello
  perl/hello syntax OK
  ```

- Executing the script

  ```
  $ perl/hello
  Hello world
  ```

4 Identifiers

- Identifier is the name of a variable or constant
- Each identifier starts with a prefix followed by a letter followed by any number of letters, digits, or underscore character (_)
  - The part of identifier following the prefix is called a *word*
  - Words should be normally quoted
  - If a word has no special meaning to Perl, it is treated as if surrounded by single quotes
    - The following two statements are equivalent

      ```
      print "Hello world\n";
      print "Hello", world, "\n";
      ```
- Identifiers are case sensitive
- The prefix defines the variable type and can be either of the following

Prefix	Identifier type	Example

`$`	Scalar	`$answer = 42;`
		`$city = "St. Louis";`

`@`	Array	`@list = ( "McGwire", 70, "homers", $city);`
		`@nums[0,1,2] = ( 9, 6, $answer );`

`%`	Hash	`%list = ( "a", "apple", "b", "bat");`
		`print $list{'a'};`

`&`	Subroutine	`&list ( @items );`
`*`	Wildcard	List of all items with this name

  - Since the prefix determines the variable type, the same name can be used for scalar, array, or hash without conflict
    - Different types of variables are said to have their own namespace
    - Changing the value of one kind of variable does not affect the value contained in another kind of variable with the same name
    - The following two variables i(`$a` for scalar and `@a` for array are both valid and can coexist in the same code

      ```
      $a = 42;
      @a = ( 1, 2, 3 );
      ```
  - If the identifier name starts with a letter, it can consist of any number of letters; otherwise, it can be only one character long (such as `$_`)
  - An uninitialized variable gets a value of `0` or `NULL` depending on context
  - Hash was known as *associative array* in pre-Perl 5.
  - The & character for subroutine name is now optional

- Scalars
  - A scalar variable can hold only one value which could be a number or a string
  - Each scalar variable is preceded by the character `$`
  - By default, all numbers are stored as doubles
  - Items to be stored in scalars include `42` and `"Hello there"`
    - In case of literals, the rules of quoting in shell are obeyed for using single quotes or double quote
- Array
  - A data structure with more or less permanently allocated storage
    - Differs from `list` that is a set of values on the run-time stack
  - Array name starts with `@`
  - Items can be accessed by a subscript that follows the array name and is enclosed in square brackets
    - Subscripts must be integers and indexing starts with 0
  - Perl arrays can contain elements of different types
    - It is legal to mix strings and numbers in arrays
  - Arrays are initialized and accessed as follows

    ```
    @students = ("John", "James", "Mary", "Joe", "Shawn", "Pat");
    $student = @students;
    $student = @students[4];
    ```

    - The first assignment assigns `6` to `$student`
    - The second assignment assigns `Shawn` to `student`
  - The array can be empty or take up all the available memory
  - You can use the *list constructor* to fill in consecutive values

    ```
    @letters = ( ’a’..’z’ );
    @nums = ( 0.5..5.5 );
    ```

- Subroutine
  - Subroutine names may be optionally prefixed with an ampersand
  - The parentheses around the subroutine arguments can also be omitted

    ```
    sub hi
    {
        $name = shift; "hi, $name\n"
    }

    print &hi ( "John" );      # Old style syntax
    print hi ( "John" );       # Helps in recognizing sub name
    print hi "John";           # Parens can be omitted
    ```

4.1 System variables

- Perl has a set of predefined variables
  - The scalar variable `$_` is used to hold the current line
  - Example

    ```
    $ date | perl -ne ’print "Today is $_"’
    Today is Thu Jan  7 20:58:55 CST 1999
    $
    ```

    We did not have to include the newline character for printing as that is included in the output of `date`
  - The UNIX `cat` command can be written as

    ```
    while ( <> )
    {
        print $_;
    }
    ```

4.2 User-defined variables

- User-defined variables in Perl are automatically declared and compiled
  - No need for separate declaration of variables before use (like in C)
- The scope of the variable is over the entire script (global) and the variable can be modified at any point in the script
- Using the same name for different kind of variables

```
$a = 1;                            # Scalar a
@a = ( 1, 2, 3 );                  # Array a
%a = (’a’ => 97, ’b’ => 98);  # Hash a
$a[3] = 4;                         # $a is still 1; @a is (1,2,3,4)
$a{’c’} = 99;                      # $a, @a unchanged;
                                   # %a has three key-value pairs
```

4.3 Quoting variables

- Quoting rules are extremely important
- Strings are normally delimited by a matching pair of single or double quotes
  - With single quotes, all characters are treated as literals
  - With double quotes, *almost* all characters are treated as literals with the exceptions being
    - Characters for variable substitution

    - Special escape sequences

- Characters such as `$`, `@`, and `%` need to be escaped with the \ or enclosed within single quotes

- Literals

  - Literals or constants in Perl can be represented as integers in decimal, octal, or hexadecimal format

    - Numers can be positive or negative, using the numeric representation from C (octal with preceding 0 and hex with preceding 0x)
    - Examples

Decimal literal	`42`
Octal literal	`052`
Hexadeciaml literal	`0x2A`

  - Floating point literals can be represented in floating point or fixed point notation

    - Examples

Fixed point	`42.0`
Scientific notation	`0.42E2`

  - Strings enclosed in *double quotes* may contain string literals such as `"\n"`, `"\t"`, or `"\033"` (`ESC`)

    - Also known as *escape sequences*
    - String literals are alphanumeric characters preceded by \ and may be represented in decimal, octal, hexadecimal, or control characters
    - Examples in Table 1

  - Special literals are available to represent things like current file name

    - Special literals are used as separate words and are not interpreted if quoted
    - There are two underscore characters on either side of the special literals
    - Examples

`__LINE__`	Current line number
`__FILE__`	Current filename
`__END__`	Logical end of the script; anything following is ignored

5 Checking Perl syntax

- You can check the syntax of any Perl script by using the option `-c`

- This allows the syntax to be checked without actually executing the script

- Examples

```
$ perl -ce 'print if /Unix'
Search pattern not terminated at -e line 1.
$ perl -ce 'print if /Unix/'
-e syntax OK
```

Table 1: Escape sequences in Perl

`\\`	Backslash
`\033`	Octal character for escape
`\0x1B`	Hexadecimal character for escape
`\a`	Alarm
`\b`	Backspace
`\c[`	Control character
`\e`	Escape
`\f`	Form feed
`\n`	Newline
`\r`	Carriage return
`\t`	Tab
`\l`	Convert next character to lower case
`\u`	Convert next character to upper case
`\L`	Convert following characters to lower case until a `\E` is reached
`\U`	Convert following characters to upper case until a `\E` is reached
`\E`	Stop lower/upper case conversions started with `\L` or `\U`

6 I/O streams in Perl

- The three streams `stdin`, `stdout` and `stderr` are inherited from the shell
- The inherited streams are not accessed directly but through filehandles
- Perl calls the three streams `stdin`, `stdout`, and `stderr` as `STDIN`, `STDOUT`, and `STDERR`, respectively
- By default, `print` and `printf` functions send their output to `STDOUT`
- With file handle, the `print` command is written as

  ```
  print STDOUT "Hello world\n";
  print STDOUT Hello, " ", world, "\n";
  ```

  - There is no comma between `STDOUT` and the next literal
- The `print` function, if successful, returns a 1; if unsuccessful, it returns a 0
- if the strings are not quoted, `STDOUT` must be specified and is not followed by a comma

6.1 Basic I/O

- You can read a file from the standard input into a scalar variable `$i` using the following syntax

  ```
  $i = <STDIN>;
  ```

  - If `STDIN` is omitted from within the angular brackets, the default stream for input is `STDIN`

    ```
    $i = <>;
    ```

- The input line thus read contains the newline character due to the user having to press the enter key
- The newline character can be discarded from the input line by using the function `chop`

  ```
  chop ( $i );
  ```

- Since we have already seen the `print` statement above, we can write a small script to read and process the inputs

  ```
  #!/usr/bin/perl

  # This is a script to illustrate reading from keyboard and
  # discarding the newline character

  print STDOUT "Enter a number to be squared: ";
  $n = <STDIN>;
  chop ( $n );
  print STDOUT "The square of ", $n, " is ", $n * $n, "\n";
  ```

  - Upon execution, the above script gives the following result:

    ```
    Enter a number to be squared: 5
    The square of 5 is 25
    ```

- The formatted output is performed by using `printf` function which is very similar to the function in `awk` and C by the same name
  - The quoted control string may contain format specifiers just like C
  - Each format specifier is preceded by the `%` character
  - Format specifiers are described in Table 2
  - Modifiers for format specifiers control the placement and are listed as

-	Left justified output
#	Integers in octal format displayed with leading 0 Integers in hexadecimal displayed with leading 0x
+	Force + or - sign for integers
0	Pad the displayed value with 0 instead of space

Table 2: Format specifiers for `print` statement

`%d`	Decimal number
`%c`	Character
`%e`	Scientific notation for floating point
`%f`	Fixed point
`%g`	Floating point or fixed point (whichever takes less space)
`%ld`	Long decimal
`%lu`	Long unsigned decimal
`%lx`	Long hexadecimal
`%o`	Octal
`%lo`	Long octal
`%s`	String
`%u`	Unsigned decimal
`%x`	Hexadecimal

- Example of `printf`

```
#!/usr/bin/perl

print STDOUT "Please type your name: ";
$n = <STDIN>;

printf (  STDOUT "Hello %s", $n );
```

- Example 2

```
#!/usr/bin/perl

print STDOUT "Please enter the balance: ";
$b = <STDIN>;
chop ( $b );

print STDOUT "Please enter interest rate: ";
$i = <STDIN>;
chop ( $i );

printf ( STDOUT "The interest amount for the month is \$%05.2f\n",
  $b * $i / ( 12 * 100 ) );
```

7 Numeric operators

- Arithmetic and logical operations on numbers are performed in Perl by numeric scalar operators
- The standard arithmetic operators from C are retained with the same semantics and are listed as

  - Addition operator +
  - Subtraction operator -
  - Multiplication operator *
  - Division operator /
  - Modulus operator %

- An additional operator for exponentiation is provided as **
  - 2 ** 10 evaluates to 1024

- Numeric comparisons can be performed as per the following operators

<	Less than
<=	Less than or equal to
==	Equal to
!=	Not equal to
>=	Greater than or equal to
>	Greater than
<=>	Comparison

  - The operator <=> requires some explanation
    * $x <=> $y compares $x with $y and returns the following

1	if $x > $y
0	if $x == $y
-1	if $x < $y

- Assignment operators
  - The assignment operators are defined by the following

=	$answer = 42;	Simple assignment
+=	$x += 32;	Add 32 to $x
-=	$y -= 2;	Subtract 2 from $y
*=	$a *= 3;	Multiply $a by 3
/=	$b /= 2;	Divide $b by 2
**=	$c **= 3;	Raise $c to the third power
%=	$d %= 4;	Divide $d by 4 and assign the remained to $d

- Auto increment/decrement
  - Each of the operators work in pre- or post-mode as follows

$x++	Post increment	$x = $x + 1
$x--	Post decrement	$x = $x - 1
++$x	Pre increment	$x = $x + 1
--$x	Pre decrement	$x = $x - 1

  - Unlike in C, the auto increment/decrement operators are not limited to integers but can work with floating point data as well

- Example: A script to get two numbers from keyboard and add them

```
#!/usr/bin/perl

print STDOUT "Enter the first number: ";
$n1 = <STDIN>;
print STDOUT "Enter the second number: ";
$n2 = <STDIN>;

chop ( $n1 );
chop ( $n2 );

print STDOUT $n1, " + ", $n2, " = ", $n1 + $n2, "\n";
```

8 String operators

- Perl provides a rich set of scalar operators for strings
- Concatenation of strings
  - Operator `.` is used to concatenate strings
  - `"Hello" . "world"` gives ”Helloworld”
  - `The answer is " . ( 7 * 6 )` gives `"The answer is 42"`
- String comparisons can be performed as per the following operators

`lt`	Less than
`le`	Less than or equal to
`eq`	Equal to
`ne`	Not equal to
`ge`	Greater than or equal to
`gt`	Greater than

  - Make sure that you do not use the numeric comparison operators for string comparisons

- Substring operator
  - Works just like `awk`
  - `substr ( "I am God", 4, 2 )` gives `" G"`
  - The indexing starts with 0 for the first character in the string
- Repetition operator `x`

  - Allows the repetition of the preceding string as per the specified number
  - Example is

    ```
    print "Hello " x 5;
    ```

- Auto increment/decrement
  - The autoincrement operator works on strings as well
  - Consider the following illustration

    ```
    $x = "Helo";
    $x++;
    print "$x\n";
    ```

- Arrays and control structures
- Pattern matching with regular expressions
- File handling