Quick Start Perl

Perl is about saving time. In just 20 minutes, this short introduction to Perl will show you enough to start using Perl straight away. Your new Perl knowledge may save you 20 minutes of work today... so you'll be even. Tomorrow it may save you 20 minutes more.

Read from top to bottom. Leave out nothing, the tutorial is cumulative. Read the example code too, it holds many important insights.

Topics

These links are useful for reference. Read the whole tutorial the first time.

Scalar Variables

Scalar variables have no explicit type. Scalar variables do not have to be declared. Context is very important in Perl. Mathematical operators treat variables as numbers, integers or floating point, depending on the context. String operators treat variables as strings of characters. Conversion between strings and numbers is automatic. Everything in Perl behaves in the most obvious common-sense way. Perl is very intuitive.

Here are some scalar variables and some numeric operators:

```
# End of line comments begin with a #

$a = 17;       # Scalar variables begin with a dollar symbol
               # The Perl assigment operator is =
               # Statements finish with a semicolon ;

$b = 0x11;     # Hexadecimal (17 in decimal)
$c = 021;      # Octal       (17 in decimal)
$d = 0b10001;  # Binary      (17 in decimal)
$f = 3.142;    # Floating point

$a = $a + 1;   # Add 1 to variable $a
$a += 1;       # Add 1 to variable $a
$a++;          # Add 1 to variable $a

$b = $b * 10;  # Multiply variable $b by 10;
$b *= 10;      # Multiply variable $b by 10;
```

```
                # Other arithmetic operators include:
                #  **  Exponentiation
                #  %   Modulo division
                #  ++  Auto increment
                #  --  Auto decrement
                #  <   Numeric less than
                #  >   Numeric greater than
                #  ==  Numeric equality
                #  !=  Numeric inequality
                #  <=  Numeric less than or equal to
                #  >=  Numeric greater than or equal to
                #  <=> Numeric compare: Returns -1 0 1
```

Scalar variables can store strings of characters or words. Single quotes, 'like these', allow spaces to appear inside strings. Double quotes, "like these", allow variables to be automatically substituted or interpolated inside strings.

```
$a = 'Number of DFFs: '; # No interpolation with 'single quotes'
$b = "$a$c\n";           # Interpolation (variable substitution) with "double
quotes"
                         # \n is the newline character
print $b;                # This makes "Number of DFFs: 17\n" appear on the
standard output
print $a, $c, "\n";      # As does this line because print takes
                         #    a comma separated list of arguments to print
print "That's all\n";    # No commas means a list of one element

                         # String operators include:
                         #  lt  String less than
                         #  gt  String greater than
                         #  le  String less than or equal to
                         #  ge  String greater than or equal to
                         #  cmp String compare: Returns -1 0 1
print 'one' lt 'two';    # Prints 1
                         #    ASCII-betically 'o' is less than 't'
print 'buf4' lt 'buf3';  # Prints nothing (that is undef, numerically zero)
                         # Perl's undefined value is undef
                         #    ASCII-betically '4' is not less than '3'
```

Logic and Truth

Perl considers these to be false:

```
  0;      # Integer zero
  0.0;    # Decimal zero
  '0';    # String containing a single zero character
  '';     # Empty string
  undef;  # Undefined
```

Everything else is true. Here are some logical operators:

```
$a = 0; $b = 45;      # More than one statement per line possible
print( $a and $b++ ); # prints 0        *
$a = 22;
print( $a and $b++ ); # prints 45       *
print $b;             # prints 46
                      # *  $b++ only evaluated when $a was true
                      # Some logic operators take shortcuts
                      # Other logical operators include
                      # or  Logical OR
                      # ||  Logical OR
                      # and Logical AND
                      # &&  Logical AND
                      # not Logical NOT
                      # !   Logical NOT
                      # |   Bitwise OR
                      # &   Bitwise AND
                      # ~   Bitwise NOT

print 6 & 5;          # prints 4, 0b0110 & 0b0101 = 0b0100
print 6 | 5;          # prints 7, 0b0110 | 0b0101 = 0b0111
print ! 0;            # prints 1
print ! 5;            # prints nothing (that is undef or false)
print ~5;             # prints 4294967290, same as:
                      # 0b11111111111111111111111111111010
```

Arrays and Hashes

An array is a list of scalar variables. The first element has index 0. The @ symbol is used to denote an array variable.

```
@components = ( 'X_LUT4', 'X_AND2', 'X_BUFGMUX', 'X_BUF_PP', 'X_FF' );

# or use qw''. Saves typing commas or quotes, gives the same result
# qw stands for Quoted Words
@components = qw'X_LUT4 X_AND2 X_BUFGMUX X_BUF_PP X_FF';

# or even put the data in columns, gives the same result again
@components = qw'
                     X_LUT4
                     X_AND2
                     X_BUFGMUX
                     X_BUF_PP
                     X_FF
                 ';             # Easier to read this way

push( @components, 'X_MUX2' ); # Push another item onto the top
push( @components, 'X_ONE' );  # And one more

print $components[0];          # Prints element 0, that is, 'X_LUT4'
print $components[5];          # Prints element 5, that is, 'X_MUX2'

print "@components\n";         # Prints everything separated by spaces:
# X_LUT4 X_AND2 X_BUFGMUX X_BUF_PP X_FF X_MUX2 X_ONE

print  @components   ;         # No double quotes, no spaces:
# X_LUT4X_AND2X_BUFGMUXX_BUF_PPX_FFX_MUX2X_ONE
```

In a scalar context an array variable returns its size, the number of elements it contains. The test expression of a while loop is tested for true or false, a scalar question, a scalar context. The shift statement removes items from the bottom of the array, one at a time.

```
while( @components ) {
#     ^^^^^^^^^^^^^                      Array in scalar context
  $next_component = shift( @components );
  print "$next_component\n";
}
# Array variable @components is now empty
```

In this example @components begins with size 7, which is true. After 7 loops each of the 7 elements have been shifted or removed from the bottom of the array. In the while test expression @components would have returned 7, 6, 5, 4, 3, 2, 1 and finally 0. Zero is false, end of while loop.

Hash arrays differ from arrays because the elements are not ordered numerically but associated with unique strings or keys. Hash arrays are associative arrays because they associate values with keys. Maintaining many separate counters is a good hash array application as we will see later. Hashes make a big contribution to Perl's text processing power. The % symbol is used to denote a hash array.

```
# Initialising several hash keys
%components = qw'
                  X_LUT4     0
                  X_AND2     0
                  X_BUFGMUX  0
                  X_BUF_PP   0
                  X_FF       0
                ';
#                 ^^^^^^^^^        keys
#                            ^   values
$components{'X_LUT4'} = 1; # Set key X_LUT4 to the value 1
$components{'X_LUT4'}++;   # Increment value associated with X_LUT4
print $components{'X_FF'}; # Print value associated with X_FF
@keys = keys %components;  # Get a list of hash keys
print "@keys\n";           # Print them - order is indeterminate
%components = ();          # Emptying the components hash
```

Command Line Arguments

There is a special array called @ARGV. Command line arguments are automatically copied into this array variable.

```
# This script is called process_netlist.pl
# Perl scripts often have the file extension .pl

$netlist_filename = $ARGV[0];
$report_filename  = $ARGV[1];
print "    Processing $netlist_filename\n";
print "    Writing report to $report_filename\n";
print "    ARGV contains '@ARGV'\n";

# Use it in this way:

#    C:\perl process_netlist.pl chip_timesim.vhd report.txt
#        Processing chip_timesim.vhd
#        Writing report to report.txt
#        ARGV contains 'chip_timesim.vhd report.txt'
#    C:\
```

Conditions

Perl has many conditional statements. The if statement asks a true/false question. If the answer is true it executes a block of code.

```
if( $ff_count == 1 )
#   ^^^^^^^^^^^^^^ Is this expression true or false?
{
  # Do this action if it is true
  print "There is 1 flip flop\n";
}
else
{
  # Do this action if it is false
  print "There are $ff_count flip flops\n";
}

# More compact layout
if( $ff_count == 1 ) {
  print "There is 1 flip flop\n";
} else {
  print "There are $ff_count flip flops\n";
}
```

It is not necessary to have an else part. The (round brackets) are required around the expression. The {curly brackets} are required around the actions.

The while loop repeatedly executes a block of statements while a conditional expression is true.

```
# Counting to one hundred
while( $count < 100 ) {
  $count++;              # Perl assumes $count == 0 the first time
  print "$count\n";
}
```

Variables do not have to be declared or initialised. Perl will create the variable when it is first used. If the variable is first used in a numeric context then its undefined initial value will be interpreted as zero. If the variable is first used in a string context then its undefined initial value will be interpreted as an empty string. Perl's default behaviour makes good sense. Counting begins at zero. Writing begins with an blank page.

Another loop statement is foreach. It is used for looping through a list of scalars, numeric or string.

```
foreach $course ( 'VHDL', 'SystemVerilog', 'SystemC', 'Perl', 'Tcl/Tk', 'PSL' )
{
  print "There is a $course Doulos training course\n";
}
# $course is the loop variable.
# It takes the string value 'VHDL' for the first loop
# and 'PSL' for the last loop.

# Get a list from an array variable
foreach $component ( @components ) {
  print "Component is $component\n";
}
```

Files

Text files are created with the open and print statements. Perl uses file handles to refer to each open file. A file handle is just a string, conventionally written in uppercase letters without quotes. Use the print statement to write text into a file.

```
open( FILE1, '>file1.txt' );
```

```
#             ^                               > means open in write mode
print FILE1 "The first line to file1.txt\n";
print FILE1 "The final line to file1.txt\n";
close( FILE1 );                               # Don't have to explicitly close a
file

print STDOUT "This goes to the standard output\n";
print        "So does this\n";
#     ^^^^^^    STDOUT is a file handle that always
#               refers to the standard output.
#               It is the default so doesn't have to be stated.
```

Text files are read using the open statement and the input record operator. Standard input, the input typed into the keyboard in a command line application, can be read from the STDIN file handle.

```
open( FILE2, 'file2.txt' );  # Open in read mode - the default mode
$first_line = <FILE2>;       # Reads the first line from file2.txt into
$first_line.
                             # Includes the newline character, \n.
while( $line = <FILE2> ) {
  print $line;               # Read and print remaining lines from file2.txt.
}                            # When every line has been read <FILE2> returns
undef.

$standard_input = <STDIN>;   # Read a line from the standard input.
                             # Can be the keyboard if run from the command line.

chomp( $standard_input );    # Remove the trailing newline character
```

Here is a short program. It reads every line from a file named in the first command line argument. The lines are written to a report file named in the second command line argument. Numbers are printed at the beginning of each line.

```
$netlist_filename = $ARGV[0];
$report_filename  = $ARGV[1];
open( FILE_IN, $netlist_filename );
open( FILE_OUT, ">$report_filename" );
while( $line = <FILE_IN> ) {
  $line_count++;
  print FILE_OUT "$line_count: $line";
}
# perl filter_netlist.pl chip_timesim.vhd report.txt
```

This Perl script does the same using standard input and standard output.

```
while( $line = <STDIN> ) {
  $line_count++;
  print "$line_count: $line";
}
# perl filter_netlist.pl < chip_timesim.vhd > report.txt
```

Pattern Matching

Perl's matching operator uses regular expressions to search a string for a match. Regular expressions are patterns used to find a match for a string of characters contained in a longer string. Regular expressions are built from character classes. The simplest character class is a single character. The letter A matches a capital letter A once. Character classes only match one character but the character can be any from a list of qualifying characters.

```
$string = "Novice to Expert in a 3 day Perl course.\n";
print $string;
if( $string =~ m/Expert/ ) {
  # A successful match returns 1 so this statement is executed
  print "This string contains the substring 'Expert'\n";
}
# m stands for match
# Forward slashes are used to /delimit/ regular expressions.
# =~ tells the m operator which string to search.
# The m is optional when // are used.
```

Regular Expressions

Individual letters are very limited character classes. Ranges of characters are matched using character class shortcuts. Any alphanumeric character matches \w, including underscores. Conveniently, most languages recognise identifiers containing letters, digits and underscores.

Quantifiers allow character classes to be repeated. The most common quantifiers are question mark, ?, asterisk, *, and plus, +. Zero or one repetition is ?. Zero or more repetitions is *. One or more repetitions is +.

```
use English;
$string = "Novice to Expert in a 3 day Perl course.\n";
if( $string =~ /\w+/ ) {
  # \w+ matches one or more alphanumeric characters in a row
  print "Matched: $MATCH\n";  # Matched: Novice
}
```

Readable English names for some Perl special variables are provided by the English Perl module. $MATCH gets a copy of the substring successfully matched. Without using the English Perl module $MATCH would have to be called $&.

```
use English;
$string = "Novice to Expert in a 3 day Perl course.\n";
if( $string =~ /Perl\s+\w+/ ) {
  #             ^^^^          matches Perl
  #                 ^^^       matches one or more white space characters
  #                                    (including space, tab and newline)
  #                    ^^^    matches one or more alphanumeric characters
  print "Matched: $MATCH\n";  # Matched: Perl course
}
#  \w?    Zero or one letter, digit or underscore
#  \w     One letter, digit or underscore
#  \w*    Zero or more letters, digits or underscores
#  \w+    One or more letters, digits or underscores
#  \W     One character but not a letter, digit or underscore

#  \s     White space character, space, tab or newline
#  \S     One character but not a space, tab or newline
```

Groups

Sub expressions can be grouped together and stored in back reference buffers. Round brackets are used to (group) sub expressions. The back reference buffers have the names $1, $2, $3 etc. The substring that matches the first bracketed group is copied into $1. The second into $2 etc.

There is a Xilinx Vital timing model in a file called chip_timesim.vhd. Here is one component instantiation from it:

```
  c4_n001449 : X_LUT4
    generic map(
      INIT => X"0001"
    )
    port map (
      ADR0 => c4_count(4),
      ADR1 => c4_count(18),
      ADR2 => c4_count(3),
      ADR3 => c4_count(5),
      O => CHOICE121_pack_1
    );
```

The component name, X_LUT4, could be matched using a regular expression containing a group. This short script opens the file, finds every component instantiation reporting the component name.

```
open( VHDL_FILE, 'chip_timesim.vhd' );
while( $line = <VHDL_FILE> ) {
  if( $line =~ /\w+\s*:\s*(X_\w+)/ ) {
#               ^^^                     Instance label
#                  ^^^                  Zero or more white space characters
#                     ^                 :
#                      ^^^              Zero or more white space characters
#                          ^^^^^        Group containing a word beginning with
X_
#                                                                (copied into
$1)
    print "Found instantiation of $1\n";
  }
}
```

Netlist Filtering

The following script takes the filename of a Xilinx netlist from the command line. It finds and counts every component instantiation. Finally, it prints a list of all the component names found and the number of appearances.

```
# Pulling it all together
# Everything in this script is described above

$netlist_filename = $ARGV[0];
open( VHDL_FILE, $netlist_filename );

while( $line = <VHDL_FILE> ) {
  if( $line =~ /\w+\s*:\s*(X_\w+)/ ) {
    $component_hash{$1}++;
  }
}

@name_array = keys %component_hash;
foreach $component_name ( @name_array ) {
  print "$component_name: $component_hash{$component_name}\n";
}
```

Extracting information from text files is easy given a little Perl knowledge. The following output was generated by the above script:

```
X_FF: 56
X_AND2: 29
X_ONE: 25
X_INV_PP: 23
X_BUF_PP: 395
X_ZERO: 4
X_TOC: 1
X_XOR2: 53
X_BUFGMUX: 1
X_OR2: 8
X_ROC: 1
X_MUX2: 96
X_LUT4: 123
X_TRI_PP: 20
```

The formatting can be improved. There are many more features and tricks in Perl. This is a useful subset that can help you to do some things straight away. To learn more about Perl quickly book your place on the Doulos Essential Perl course . You will learn how to write Perl scripts to solve many design flow related problems. Read the syllabus to find out more.