Origins and Uses of Perl

- Perl was developed in 1987 by Larry Wall. Its design goal was to expand on the text-processing features of awk.
- The language has had many features added since then, including
  - support for communications using sockets
  - support for object-oriented programming
  - powerful text pattern-matching capabilities
- Perl is translated into an intermediate representation, vaguely similar to Java.
Scalars and Their Operations

- There are three categories of variables in Perl:
  - scalar variables (which begin with \$)
  - vector variables (which begin with @)
  - hash variables (which begin with %)
- Scalar variables can store three types of values:
  - numbers (represented internally in double-precision floating-point form)
  - character strings
  - references (addresses)

Numeric Literals

- All numeric values are represented internally as double-precision floating-point values.
- Literal numbers are integers or floating point values.
  - Integers are strings of digits
  - Integers can be written in hexadecimal by preceding them with 0x
  - Floating point values can have decimal points and/or exponents:

\[
\begin{align*}
72 & \quad 7.2 & \quad .72 & \quad 72. & \quad 7E2 & \quad 7e2 \\
.7e2 & \quad 7.e2 & \quad 7.2e-2
\end{align*}
\]
String Literals

• String literals can be delimited by single (') or double (") quotation marks.
  – Strings enclosed in single quotes cannot include escape sequences such as \n. If \n appears in such a string both the backslash and the n are included in the string. However, \' can be used to include a single quote in the string.
• The single quotes can be replaced by another delimited if it is preceded by a "q"
  q$I don't want to go, I can't go, I won't go!$
  q<I don't want to go, I can't go, I won't go!>

String Literals (cont'd)

• Double-quoted string can include special characters (such as \n) and embedded variable names will be replaced by their values.
  "Completion % \t Yards \t Touchdowns"
• A double quote can be embedded by preceding it with a backslash
  "\"Aha !\", he said."
• A different delimited can be used by preceding it with qq.
  qq@"Why, I never!", said she@
• A null string is written as ' ' or "".
Scalar Variables

- The names of scalar variable all begin with $ and then continue with a letter, followed by letters, digits and/or underscores.
- Variable names are case sensitive, so $FRIZZY, $Frizzy and $frizzy are three different names.
- Style dictates that variable names in Perl should not use uppercase.

Variable Names In String Literals

- If variables names are included in double-quoted string literals, the variable names are replaced by their values.
- Example
  
  $age = 47;
  $mystack = "Jack is $age years old ";
  $mystack has the value:
  "Jack is 47 years old"
Declaring Scalar Variables

• In Perl, scalar variables are rarely declared explicitly; it is usually done implicitly when a value is assigned to the variable.
• Scalar variables that have not been assigned a value by the program that the value undef, with a numerical value of 0 and a string value of "."

Implicit Variables

• Perl has several predefined, or implicit, variables, whose names begin with $.
• Many of these names contain only one more special character, such as $ _ $^ $\
Numeric Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>++, --</td>
<td>Nonassociative*</td>
</tr>
<tr>
<td>Unary +, −</td>
<td>Right to left</td>
</tr>
<tr>
<td>** (exponentiation)</td>
<td>Right to left</td>
</tr>
<tr>
<td>*, /, %</td>
<td>Left to right</td>
</tr>
<tr>
<td>Binary +, −</td>
<td>Left to right</td>
</tr>
</tbody>
</table>

* two ++ or -- operators cannot appear in an expression separated by only an operand

In almost all cases, numeric operations are performed in double-precision floating point

String Operators

- Strings are considered single units.
- Concatenation is performed using the . operator:
  
  $first = "Freddy"

  $first . " Freeloader" will produce
  "Freddy Freeloader"

- x is the repetition operator.
  "More! " x 3 will produce
  "More! More! More! "

## String Functions

<table>
<thead>
<tr>
<th>Name</th>
<th>Parameter(s)</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>chomp</td>
<td>A string</td>
<td>Removes trailing newlines and returns # of removed characters</td>
</tr>
<tr>
<td>length</td>
<td>A string</td>
<td>Returns the # of characters in its parameter string</td>
</tr>
<tr>
<td>lc</td>
<td>A string</td>
<td>Returns the parameter with all uppercase convert to lowercase</td>
</tr>
<tr>
<td>uc</td>
<td>A string</td>
<td>Returns the parameter with all lowercase convert to uppercase</td>
</tr>
<tr>
<td>hex</td>
<td>A string</td>
<td>Returns the decimal value of the hex. number in its parameter string</td>
</tr>
<tr>
<td>join</td>
<td>A character and the strings concatenated together with a list of strings</td>
<td>Returns a string constructed by concatenating the strings of the second and subsequent parameters the parameter char. in between.</td>
</tr>
</tbody>
</table>

## Assignment Statements

- Assignment statements in Perl look like those in C/C++/Java:
  
  ```perl
  $salary = 47500;
  ```

- Compound assignment operators also exist that can combine assignments with binary arithmetic operators:
  
  ```perl
  $salary += $value;
  ```
  
  is equivalent to
  
  ```perl
  $salary = $salary + $value;
  ```

- Comments begin with `#` and run until the end of the line:
  
  ```perl
  $salary += $value;  # You just got a raise
  ```
Keyboard Input

• All input and output in Perl is treated as file I/O.
• Files have external names, but are known within the program by their internal names (known as "filehandles"). Every program has three predefined file handles: **STDIN**, **STDOUT** and **STDERR**.
• We get line input by writing:

```perl
$in_data = <STDIN>
```

Getting line input means that we read up to and including the newline, which we can get rid of by rewriting it as:

```perl
chomp($in_data = <STDIN>);
```

Screen Output

• The standard way of producing output is the **print** operator (or **print** function, which requires parentheses), which takes one or more string literals separated by commas.
• There is no newline added to the output, so the programmer has to supply his/her own:

```perl
print "This is pretty easy\n";
```
• C's printf function and its formatting specifiers are also available.
quadeval.pl

# quadeval.pl - A simple Perl program
# Input: Four numbers, representing the values of
#        a, b, c and x
# Output: The value of the expression
#        axx + bx + c
# Get input
print "Please input the value of a ";
$a = <STDIN>;
print "Please input the value of b ";
$b = <STDIN>;
print "Please input the value of c ";
$c = <STDIN>;
print "Please input the value of x ";
$x = <STDIN>;
# Compute and display the result
$result = $a * $x * $x + $b * $x + $c;
print "The value of the expression is: $result 
";

Running Perl programs

- Under Windows, UNIX and Linux, a Perl program is run by typing perl and the program's file name:
  perl quadeval.pl
- If you just want to compile (to see if there are errors), use the –c option. If you want warnings as well (a good idea), use the –w option. Writing
  perl –w quadeval.pl quad.dat
uses quad.dat as standard input.
Control Expressions

- Control expressions are scalar-valued, relational or compound.
- String scalar-valued expressions are true if they are not empty or not "0"; numeric scalar-valued expressions are true if they are non-zero.
- If a string relational operator is given a numeric value, it is coerced into a string. Similarly, if a numeric relational operator is given a string, it is coerced to numbers.
- This coercions are not always intuitive, e.g., 'George' will be coerced into 0, because it cannot be coerced into a numeric value.

Relational Operators

<table>
<thead>
<tr>
<th>Operation</th>
<th>Numeric Operands</th>
<th>String Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is equal to*</td>
<td>==</td>
<td>eq</td>
</tr>
<tr>
<td>Is not equal to*</td>
<td>!=</td>
<td>ne</td>
</tr>
<tr>
<td>Is less than*</td>
<td>&lt;</td>
<td>lt</td>
</tr>
<tr>
<td>Is greater than*</td>
<td>&gt;</td>
<td>gt</td>
</tr>
<tr>
<td>Is less than or equal to*</td>
<td>&lt;=</td>
<td>le</td>
</tr>
<tr>
<td>Is greater than or equal to*</td>
<td>&gt;=</td>
<td>ge</td>
</tr>
<tr>
<td>Compare, returning -1, 0 or +1†</td>
<td>&lt;=&gt;</td>
<td>cmp</td>
</tr>
</tbody>
</table>

* Produces +1 if true, "" is false
† a>b it produces -1, a = b produces 0, a>b produces +1
Operator Precedence

<table>
<thead>
<tr>
<th>Operator</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>+=, -=</td>
<td>Non-associative</td>
</tr>
<tr>
<td>**</td>
<td>Right to left</td>
</tr>
<tr>
<td>unary +, unary -</td>
<td>Right to left</td>
</tr>
<tr>
<td>*, /, %, x</td>
<td>Left to right</td>
</tr>
<tr>
<td>+, -, .</td>
<td>Left to right</td>
</tr>
<tr>
<td>&gt;, &lt;, &gt;=, &lt;=, lt, gt, le, ge</td>
<td>Left to right</td>
</tr>
<tr>
<td>==, !=, eq, ne, cmp</td>
<td>Left to right</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Left to right</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>%=, +=, -=, /=, %=, &amp;=,</td>
<td>Right to left</td>
</tr>
<tr>
<td>&amp;&amp;=,</td>
<td></td>
</tr>
<tr>
<td>not</td>
<td>Right to left</td>
</tr>
<tr>
<td>and</td>
<td>Left to right</td>
</tr>
<tr>
<td>or</td>
<td>Left to right</td>
</tr>
</tbody>
</table>

Assignment Operators as Control Expressions

- Because assignment operators have values (the value being assigned to the left-hand side), they can be used as control expressions.
- The following loop will execute until end of file is encountered:

```plaintext
while ($next = <STDIN>)  {
  ... ...
}
```
Selection and Loop Statements

• Most of the selection and loop structures available in C/C++/Java are also available for Perl.
• These structures require that ALL statements that are part of if-else or loops be enclosed in braces.

if, if-else, if-elsif-else

• if in Perl is similar to C, except that the any statements that are executed must be enclosed in braces:
  if ($c > 10) {
      $b = $a * 2;
  }
• if constructs can include elsif, which is more readable than else if:
  if ($snowrate < 1) {
      print "Light snow\n";
  } elsif ($snowrate < 2) {
      print "Moderate snow\n";
  } else {
      print "Heavy snow\n";
  }
• There is no switch statement in Perl.
**unless**

- **unless** has essentially the same syntax as if, except the statements in the block are executed if the condition is `false`.
- Example
  ```ruby
  unless (sum > 1000) {
    print "We\'re not finished yet!";
  }
  ```
  is the same as:
  ```ruby
  if (sum <= 1000) {
    print "We\'re not finished yet!";
  }
  ```

**while and until**

- **while** and **for** are similar to those in C. There is also **until**, where the loop continue as long as the condition is `false`.
- Examples
  ```ruby
  $sum = 0; $sum = 0;
  $count = 1; $count = 1;
  while ($count <= 100) {
    $sum += $count;
    $count++; $count++;
  }
  until ($count > 100) {
    $sum += $count;
    $count++;
  }
  for ($sum = 0, $count = 1; $count <= 100; $count++) {
    $sum += $count;
  }
  ```
**last** and **next**

- **last** and **next** provides an early exit from a loop.
- **last** breaks the program out of the innermost loop (or another loop if you use labels)
- **next** takes the program to the next iteration of the innermost (or another loop if you use labels).
- Example:

```perl
for ($i = 0; $i > 5; $i++)
    if ($i == 0)
        last;
```

---

**Loops and Labels in Perl**

- Using a label allows the programmer to decide which one of several nested loops will be exited:

```perl
BIGLOOP:
while (...) {
    while (...) {
        while (...) {
            ...
            if (...) { last BIGLOOP} ...
        }
    }
}
```

$_

- $_ is often used in Perl programs in different ways (e.g., default parameter in function calls, default operand of an operator).

- Example

  ```perl
  while (<STDIN>) {
    print;  # default parameter
    chomp;  # default operator
    if ($_ eq "redhead") {
      print "I've really finally found one.\n"
    }
  }
  }
  ```

List Literals

- A list is an ordered sequence of scalar values.
- A list literal is a parenthesized list of scalar values and is the way a list value is specified in a program.
- Since arrays store lists, list literals serve as array literals.
- Lists can store any combination of different types of scalar values:

  ```perl
  (3.14159 * $radius, "circles", 17)
  ```
Arrays

- Arrays are variables that can store lists. All their names begin with @ and occupy a different namespace from scalar variables.
- Arrays can be assigned lists or other arrays:
  ```perl
  @list = ('boy', 'girl', 'dog', 'cat');
  @creatures = @list;
  ```
- If a list is assigned to a scalar variable, the scalar variable is assigned the array's length.

Working with List Literals

- A list literal with all variables can be assigned values in another list literal:
  ```perl
  ($first, $second, $third) = ("George", "Bernard", "Shaw");
  ```
- This is shorthand for:
  ```perl
  $first = "George";
  $second = "Bernard";
  $third = "Shaw";
  ```
- If an array is in that list, it will be assigned any variables not assigned to list members to its left.
Arrays and Subscripts

- Arrays in Perl use integer subscripts (although any numeric expression can be used) beginning at zero.
- Individual elements have a $ in front of their name:
  ```perl
  @list = (2, 4, 6, 8);
  $second = $list[1];
  ```
- Scalars and arrays are in different namespaces, so there is no connection between $a and $a[0].

Array Lengths

- Array lengths are dynamic:
  ```perl
  @list = ("Monday", "Tuesday", "Wednesday",
           "Thursday");
  $list[4] = "Friday";
  ```
- If you write:
  ```perl
  @list = (2, 4, 6);
  $list[27] = 8;
  ```
  The array is now length 28 with 24 vacant positions.
- The last subscript in a list is $#list, so the length of the list is $#list + 1
The **foreach** Statement

- The **foreach** statement in Perl allows the programmer to process every member of an array:
  ```perl
  foreach $value (@list) {
    $value /= 2; # divides every member by 2
  }
  ```

The **foreach** – An Example

- **foreach** assumes that every element of the array from index 0 to index $\#list$ exists and if it is vacant, that it has the "":
  ```perl
  $list[1] = 17;
  $list[3] = 34;
  foreach $value (@list) {
    print "Next: $value \n";
  }
  ```
  prints
  ```perl
  Next:
  Next: 17
  Next:
  Next: 34
  ```
Built-in Array Functions

- There are four functions (and operators) that allow programmers to manipulate the contents of an array:
  - **shift** and **unshift** – which remove from and add items to the beginning of an array
  - **push** and **pop** which insert to and remove items from the end of an array.

**shift and unshift – An Example**

```perl
@list = ("Moe", "Curly", "Larry");
$first = shift @list;
foreach $value (@list) {
    print "Next on the left is $value\n" ;
}
print "Off the list is $first\n"

unshift @list, "Shemp";
foreach $value (@list) {
    print "Next on the left is $value\n" ;
}
print "Off the list is $first\n";
```
**shift and unshift – The output**

Next on the left is Curly  
Next on the left is Larry  
Off the list is Moe

Next on the left is Shemp  
Next on the left is Curly  
Next on the left is Larry  
Off the list is Moe

**push and pop – An Example**

```perl
@list = ("Moe", "Curly", "Larry");
$first = pop @list;
foreach $value (@list) {
    print "Next on the left is $value\n" ;
}
print "Off the list is $first\n\n";

push @list, "Shemp";
foreach $value (@list) {
    print "Next on the left is $value\n" ;
}
print "Off the list is $first\n\n";
```
push and pop – The Output

Next on the left is Moe
Next on the left is Curly
Off the list is Larry

Next on the left is Moe
Next on the left is Curly
Next on the left is Shemp
Off the list is Larry

split

• The split function is used to break strings into parts using a specific character as the basis for the split:
  $stoogestring = "Curly Larry Moe";
  @stooges
       = split (" ", $stoogestring);
  The list is now ("Curly", "Larry", "Moe")
**sort**

- *sort* takes an array parameter and uses a string comparison to sort the elements of the array alphabetically in the list that it returns:
  ```perl
  @list = ("Moe", "Curly", "Larry");
  @newlist = sort @list;
  foreach $value (@newlist) {
      print "$value
  }
  ```

**qw**

- *qw* can be used to place a sequence of unquoted strings inside quotation marks:
  ```perl
  qw(peaches apples pears kumquats)
  ```
  will produce:
  ```perl
  ("peaches", "apples", "pears", "kumquats")
  ```
die

- **die** takes a variable number of string parameters, concatenates them, sends the result to **STDERR**, and terminates the program.
- The implicit operator $! Stores the number of the most recent error that has occurred:
  
  ```
  die "Error: division by 0 in function fun2 - $! " 
  ```

An Example: **process_names.pl**

```perl
# process_names.pl - A simple program to illustrate
# the use of arrays
# Input: A file, specified on the command line, of
# lines of text, where line is a person's name
# Output: The input names, after all letters are
# converted to uppercase, in alphabetical order
$index = 0;

#>>> Loop to read the names and process them
while ($name = <>) {
  #>>> Convert the name's letters to uppercase and
  #>>> put it in the names array
  $names[$index++] = uc($name);
}
```
Display the sorted list of names
print "\nThe sorted list of names is:\n\n\n";
foreach $name (sort @names) {
    print("$name \n");
}

Hashes

- An **associative array** is an array in which each data item is paired with a **key** (a value that uniquely identifies the data item that you seek). Associative arrays in Perl are called **hashes**.
- Hashes are different from arrays because:
  - Array use numeric subscripts; hashes uses string values as keys.
  - Arrays are ordered by subscript; hashes are not really ordered.
- Hash names begin with $.
Initializing Hashes

- Hashes can be initialized using a list literal:
  ```perl
  %kids_age = ("John" => 38,
                 "Genny" => 36, "Jake" => 22,
                 "Darcie" => 21);
  ```
- If you assign an array to a hash, the even subscripts are the hashes, and the odd subscripts are the values:
  ```perl
  @anarray = (3, 5, 6, 99);
  %ahash = @anarray
  ```
  is equivalent to
  ```perl
  %ahash = ("3" => 5, "6" => 99);
  ```

Referencing a Hash Element

- An individual value element of a hash can be referenced by using the hash name along with the appropriate key. Braces are used instead of brackets:
  ```perl
  $genny_age = $kids_age{"Genny"};
  ```
- New values are added by assigning the new element’s value together with its key:
  ```perl
  $kids_age{"Aidan"} = 7;
  ```
Removing Hash Elements

- A single member of a hash can be removed by writing:
  ```perl
delete $kids_age{"Genny"};
```
- An entire hash can be set to empty by:
  - assigning it an empty list:
    ```perl
    %kids_age = ();
    ```
  - or by using the undef ("undefined") operator:
    ```perl
    undef %kids_age;
    ```

Working With Hashes

- The exists operator can be used to determine if an element with a specific key value is in the hash:
  ```perl
  if (exists $kids_age{"Freddie"}) …
  ```
- A hash's keys and values can be extracted into an array by writing:
  ```perl
  foreach $child (keys %kids_age) {
    print("The age of $child is $kids_age{$child} \n";
  }
  @ages = values %kids_age;
  print "All of the ages are @ages \n";
  ```
Environmental Variables

- Perl has a predefined hash called `%ENV` that stores operating systems environmental variables, which store information about the system on which Perl is running.
- The environmental variables can be accessed by any Perl program, where the keys are the variable names. They can be accessed:

```perl
foreach $key (sort keys %ENV) {
    print "$key = $ENV{$key} \n";
}
```

References

- A reference is a scalar variable that references another variable or a literal (i.e., it's an address).
- Perl references are related to pointers in C/C++, but they are safer to use.
- A reference can be obtained by the backslash operator before the variable name:

  ```perl
  $age = 42;
  $ref_age = \$age;
  @stooges = ("Curly", "Larry", "Moe");
  $ref_stooges = \@stooges;
  ```
References to Literals

• A reference to a list literal can be created by putting the literal value in brackets:

```perl
$ref_salaries
    = [42500, 29800, 50000, 35250];
```

• A reference to a hash literal is created by putting the literal value in braces:

```perl
$ref_ages = { 'Curly' => 41,
    'Larry' => 38,
    'Moe' => 43};
```

Dereferencing

• A reference can be used to specify two values: its own (which is an address) and the value stored at that address. The latter is called dereferencing.

• All dereferencing in Perl is explicit. This can be done by placing a extra $ in front of the variable's name:

```perl
$$ref_stooges[3] = "Maxine";
```

This can also be done by using the -> operator:

```perl
$ref_stooges -> [3] = "Maxine";
```
Functions

- A function definition includes the function header and a block of code that specifies its actions.
- Neither the parameters nor the type of the result is specified.
- The header contains the reserved word `sub` and the function’s name.
- A function declaration tells the compiler that a function with a given name will be defined elsewhere.
- Function’s that return values can be used in an expression. Functions that do not return anything can be standalone statements.

Local Variables

- Variables that appear only in a function and that are implicitly declared are `global`. You can force variables to be local by using the word `my` in front of the variable:
  - `my $count = 0;`
  - `my ($count, $sum) = (0, 0);`
- When local and global variables conflict in name, we use the local variable.
- We can also declared local variables with the reserved word `local`. 
Parameters Passing

- Parameters can be passed by one of two ways:
  - Passing by value – a copy of the parameters’ values are given to the function. (one-way communication)
  - Passing by reference – a copy of the values’ addresses are given to the function so it can use the actual values in their actual memory location. (two-way communication)
- If changes do not need to be shared with the main program, passing by value is preferable.

Passing Parameters in Perl

- All Perl parameters are passed through a special variable called @_. The actual parameters are copied into @_.
- If an array is a parameter, it is also copied into @_ (and should be at the end of the parameter list). Hashes are flattened into arrays.
- @_ is passed by value.
Passing Parameters in Perl - Examples

```perl
sub plus10 {
    $_[0] += 10
}

plus10 ($a) # $a is increased by 10

// Evaluates a quadratic equation
sub fun_eval {
    my ($a, $b, $c) = @_;  
    return $a * $x * $x + $b * $x + $c
}
```

References As Parameters

- References to variables can be used as actual parameters, which provides a pass by reference mechanism.

```perl
sub squeeze {
    my $ref_LIST = $-[0];
    my $value, @new;
    foreach $value (@$ref_list) {
        if ($value > 0) {
            push(@new, $values);
        }
    }
}

squeeze(@mylist);
```
The `sort` Function, Revisited

- Sort normally compares members of an array as if they were string (coercing numeric values if necessary).
- This can be changed by giving it an explicit block of code that specifies what comparison to use:

  ```perl
  # sort numbers in ascending order
  @new_list = sort { $a <=> $b } @list;
  # sort numbers in descending order
  @new_list = sort { $b <=> $a } @list;
  # sort strings in descending order
  @new_list = sort { $b cmp $a } @list;
  ```

An Example – `tst_median.pl`

```perl
# program to test a function that computes the
#   median of a given array
#
# median - a function
# Parameter:
#   A reference to an array of numbers
# Return value:
#   The median of the array, where median is the
#   middle element of the sorted array, if the
#   length is odd; if the length is even, the median
#   is the average of the two middle elements of the
#   sorted array.
```
An Example – \texttt{tst\_median.pl}

\begin{verbatim}
sub median{
my $ref\_list = $_[0];

#>>> Compute the length of the passed array
my $len = $#$ref\_list + 1;

#>>> Sort the parameter array
@list = sort {
a <=> b
} @$ref\_list;

#>>> Compute the median
if ($len %2 == 1) { # length is odd
  return $list[$len / 2];
} else { # length is odd
  return ($list[$len / 2] + $list[$len / 2 - 1]) / 2;
} #>>> End of function median
}

#>>> Begin main program
#>>> Create two test arrays, one with odd length and one with
#>>> even length
@list1 = (11, 26, 5, 29, 41, 6, 8, 0, 9);
@list2 = (43, 77, 11, 29, 8, 51, 9, 18);

#>>> Call median on both arrays and display the results
$med = median(@list1);
print "The median of the first array is $med \n";
$med = median(@list2);
print "The median of the second array is $med \n";
\end{verbatim}
Pattern Matching

- Perl has powerful pattern-matching features, including a pattern-matching operator `m`, which we won't be using explicitly.
- If we delimit our pattern between slashes, we can match it implicitly against the implicit variable `$_`:
  ```perl
  if (/rabbit/) {
      print "rabbit appears in \$_\n";
  }
  ```
- The binding operator `=~` matches a variable of our choosing against the pattern:
  ```perl
  if ($str =~ /^rabbit/) {
      print "$str begins with rabbit\n"
  }
  ```

More About `split`

- `split` can divide a string based on more than single-character separators; it can separate string based on any pattern based on regular expressions.
- Example
  ```perl
  @words = split /[ .,]s*/, $str
  ```
- `$str` will be divided into strings separated by a blank, period or comma followed by any whitespace.
word_table.pl

# word_table.pl

# Input: A file of text in which all words are
# separated by whitespace or punctuation,
# possibly followed by whitespace, where
# the punctuation can be a comma, a
# semicolon, a question mark , an
# exclamation point, a period or a colon.
# The input file is specified on the
# command line.

# Output: A list of all unique words in the input
# file, in alphabetical order.

### Main loop to get and process lines of input
### text

while (<>) {
    # Split the lines into words
    @line_words = split /\[,.\;:\!\?]\s*/;

    # Loop to count the words (either increment or
    # initialize tho ])
    foreach $word (@line_words) {
        if (exists $freq{$word}) {
            $freq{$word} ++;
        } else {
            $freq{$word} = 1;
        }
    }
}
#>>> Display the words and their frequencies
print "\nWord \t Frequency \n\n";
foreach $word (sort keys %freq) {
    print "$word \t $freq($word) \n";
}

Remembering Matches

• Parts of a string can be divided up among implicit variables:
  
  "4 July 1776" =~ /\d+ \w+ /d+/;
  print "$2 \$1, $3\n";

  will display
  July 4, 1776

• It can be very useful to match a pattern and save the portions of the string before the match, matching the pattern and after the match. These are $` (before), $& (matching) and $' (after).
Remembering Matches – An Example

```
$str = "This is a rabbit test\n";
$str =~ /rabbit/;
print
    "Before \"$\`, during \"\$\`\" and after",
    \"\"$\`\"\n";
```

prints
Before "This is a ", during "rabbit" and after "
test"

```
Substitutions

• Sometimes we may wish to alter a string that we are
  saving; we can do this using the substitute operator:
  \textit{s/Pattern/NewString/}

• Example
  \texttt{$str = "It ain't going to rain no more"; $str =~ s/ain't/is not/;}
```
Substitutions With Modifiers

- **g** modifier tells the substitute operator to make the changes on all pattern matches, not just the first one:
  
  ```perl
  $str = "Rob, Robbie and Robette are siblings\n";
  $str =~ s/Rob/Bob/g;
  ```
  changes all three names.

- **i** modifier tells the substitute operator to ignore case of letters:
  
  ```perl
  $str = "Is it Rose, rose or ROSE"
  $str =~ s/Rose/rose/ig;
  ```
  changes all three to lower case.

The Transliterate Operator

- **tr** is the translate operator, which will replace a character (or a class of characters) with another character (or class of characters):
  
  ```perl
  # replace semi-colons with colons
  $str =~ tr/;:/;=
  # translate upper to lower case
  $str =~ tr/A-Z/a-z/;
  Remove all commas and periods
  $str =~ tr/\,\./\,\./;
  ```
File Input and Output

- Files are references by using filehandles, whose names do NOT begin with special characters and are usually written in uppercase for greater readability.
- The connection between external name and filehandle are established using the open function:

  ```c
  open (FileHandle, "FileUseSpecifier FileName");
  ```

File Use Specifiers

<table>
<thead>
<tr>
<th>Character(s)</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;</code></td>
<td>Input (the default)</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>Output, starting at the beginning of the file</td>
</tr>
<tr>
<td><code>&gt;&gt;</code></td>
<td>Output, starting at the end of the existing data on the file</td>
</tr>
<tr>
<td><code>+&gt;</code></td>
<td>Input from and output to the file</td>
</tr>
</tbody>
</table>
Using a File – An Example

```perl
# Open the file for input. If it won't open, 
# terminate and print the error message
open (INDAT, "<temperatures")
    or die "Error - unable to open temperatures "$!";
...
# Print a line of output
print OUTDAT "The result is: $result \n";
# Read one line of input
$next_line = <INDAT>;
```

Reading Multiple Lines of Input

- The read function can be used to read multiple lines of input into a scalar variable. The syntax is:
  
  ```perl
  read(filehandle, buffer, length
  [, offset]);
  ```

- This will read `length` bytes into scalar variable `buffer`.

- If `offset` is used, it will begin `offset` bytes after the file pointer.

- Example
  
  ```perl
  $chars = read (ANIMALS, $buf, 255);
  ```
Using **split**

- The lines of text in the buffer can be separated using `split`:
  ```perl
  @lines = split /
/, $buf;
  ```

Using **seek**

- Some applications seek to re-write data that has just been read. This requires the use of the `->` file use specification and then moving back to the beginning of the data just read using `seek`.
- Syntax
  ```perl
  seek (filehandle, offset, base);
  ```
  The base is either 0 (beginning of file), 1 (current position), 2 (end of file)
- Example
  ```perl
  seek(UPDAT, 240, 0);
  ```
An Example – wages.pl

# wages.pl - An example program to illustrate some of
# the features of Perl
# Input: A file of lines of employee data, where
# each line has name, age, department
# code, salary
# Output: 1. The names of all employees whose
# names end with "son"
# 2. Percentage of employees under 40
# years old
# 3. Average salary of employees under
# 40 years old
# 4. An alphabetical list of employees
# who are under 40 years old and who
# have salaries more than $40,000

#>>> Open the data file and display a head for
# employees whose names end in 'son'
open(EMPLOYEES, "employees.txt") || die "Can't open employees $!";
print "Names that end in 'son'\n\n";

#>>> Loop to read and process the employee data
while (<EMPLOYEES>) {
    #>>> Increment the number of employees and chop
    # off the newline
    $total_employees++;
    chomp;

    #>>> Split the input line into its four parts
    ($name, $age, $dept, $salary) = split(/:/);
#>>> If the name ends in 'son', print the name
if ($name =~ /son$/) {
    print "$name \n";
}

#>>> If the employee is under 40, count him or her
# and add his or her salary to the sum of such
# salaries
if ($age < 40) {
    $under_40++;
    $salary_sum += $salary;

    #>>> If the salary is over $40,000, add the
    # person and his or her salary to the hash of
    # such people
    if ($salary > 40000) {
        $sublist{$name} = $salary;
    }
}

#>>> If there was at least one employee, continue
if ($total_employees > 0) {

    #>>> If there was at least one under 40, continue
    if ($under_40 > 0) {
        #>>> Compute and display the % of employees
        # under 40 and their average salaries
        $percent = 100 * $under_40 / $total_employees;
        print \nPercent of employees under 40 is: ", " $percent \n";
        $avg = $salary_sum / $under_40;
        print "Average salary of employees under 40 ", "is: $avg \n";
    }
}
#>>> If there was at least one under 40 who
#    earned salary > 40,000, continue
if (keys(%sublist)) {
    #>>> Sort and display the names of the
    #    employees under 40 with salaries > 40,000
    print "Sorted list of employees under 40",
        " with salaries > \$40,000 \n";
    @sorted_names = sort (keys(%sublist));
    print "\nName \t\t Salary \n";
    foreach $name (@sorted_names) {
        print "$name \t\t\$sublist{$name} \n";
    }
} else {
    print "There were no employees under 40 ",
        " who earned over \$40,000 \n"
} #>> of if (keys(%sublist))
}

else {
    print "There were no employees under 40 \n"
} #>> of if ($under_40 > 0)
} else {
    print "There were no employees
"
} #>> of if ($total_employees > 0)