Programming Languages:

Lecture 4

Perl Programming Language

Jinwoo Kim

jwkim@jjay.cuny.edu

Scripting Languages & Perl Programming

- Perl (Practical Extraction and Report Language) is a powerful and adaptable scripting language
  - Become very popular in early 90's as web became a reality
    - **Good for processing web pages containing tags of different types (image tags, url tags etc)**
    - **These tags or substrings can be extracted using Perl commands**
    - **Ideal for managing web applications, such as passwd authentication, database access, network access, and multiplatform**
    - **good for working with html web forms, obtaining user inputs, enabling cookies and tracking clicks and access counters, connecting to mail servers, integrating with html, remote file management via the web, creating dynamic images among many other capabilities capabilities**
  - Perl is ideal for processing text files containing strings

Scripting Languages & Perl Programming

- What is Perl?
  - Strings, literals and variables
    - Perl strings (sequence of characters) can be surrounded by single quotes or double quotes
    - Perl has three types of variables - scalars (strings or numeric's), arrays and hashes
  - Scalars, arrays and hashes
    - A scalar in perl is either a numeric (103, 45.67) or a string
      - $s1 = "hello"; $s2 = "world"; $s3 = $s1.$s2;
    - Array in perl is defined as a list of scalars
      - @array =(10,12,45);
  - Operators
  - Substitutions
  - Control Statements
  - Sorting, Shifting

Larry Wall on Perl

- ``The Perl slogan is, **`There is more than one way to do it.'** People have often taken that to mean that there is more than one way to do it within Perl. But I apply the same thing outside of Perl also. I know a lot of people use Perl for what it's useful for. But I don't expect to take themselves off to a monastery and just write Perl scripts all day.‘’

Larry Wall on Perl Competitors

- Wall says Perl's biggest competitors -- *REXX*, *Tcl*, *Python*, and *Scheme* – are useful for similar things (that Perl is). Although he prefers Perl because of it's efficiency, language structure, and lack of theoretical axes, Wall says he will not engage in language wars. ``I have a firm policy against making enemies," he states.

Resources For Perl

- Books:
  - Learning Perl
    - **By Larry Wall**
    - **Published by O'Reilly**
- Programming Perl
  - By Larry Wall,Tom Christiansen and Jon Orwant
  - Published by O'Reilly
- Web Site
  - http://safari.oreilly.com
    - **Contains both Learning Perl and Programming Perl in ebook form**

Web Sources for Perl Code/Info

- Class homepage
- www.perl.com
- www.perldoc.perl.com
- www.perl.org
- www.perlmonks.org
- www.activestate.com

A simple Perl Script (run this on your machine)

```
#!/usr/bin/perl
print “what is your name? “;
chomp($name = <STDIN>);   # Program waits for user input from keyboard
print “Welcome, $name, are your ready to learn Perl now? “;
chomp($response = <STDIN>);
$response=lc($response);      # response is converted to lowercase
if ($response eq “yes” or $response eq “y”){
            print “Great! Let’s get started learning Perl by example.\n”;
}
else {
            print “O.K. Try again later.\n”;
}
$now = localtime;  # Use a Perl function to get the date and time
print “$name, you ran this script on $now.\n”;
```

The Script File

- A Perl script is created in a text editor
- Normally, there is no special extension required in the file name, unless specified by the application running the script
  - E.g., If running under Apache as a *cgi* program, the file name may be required to have a *.pl* or *.cgi* extension

Running the Script File

- Perl programs are not compiled but interpreted
  - Perl interpreter in your unix system can be found by typing
  - *where perl*
  - *It may show*
    - **/usr/local/bin/perl**
    - **/usr/bin/perl**
  - giving the path of the perl interpreter
  - Perl interpreter is used to run perl programs
    - ***#!/usr/local/bin/perl***
    - ***print "Hello World\n";***

    - ***perl hello.pl*** **(**Assuming above is in a file called hello.pl)

    - ***chmod +x hello.pl***
    - ***./hello.pl***

Free Form

- A Perl is a free-form language

- Statement must be terminated with a semicolon but can be anywhere on the line and span multiple lines

Comments

- Perl comments are preceded by a # sign
- They are ignored by the interpreter
- They can be anywhere on the line and span one line

```
print “Hello, world “; # this is a comment
# And this is a comment
```

Printing Output

- The *print* and *printf* functions are built-in functions used to display output
  - The *print* function arguments consist of a comma-seperated list of strings and/or numbers
  - The *printf* function is similar to the C *printf()* function and is used for formatting output
  - Parentheses are not required around the argument list

```
print “Hello, world\n “;
print “Hello, “, “ world\n”;
print (“Its such a perfect day!\n”); # Parens optional
print “The date and time are: “, localtime(), “\n”;
printf(“Meet %s: Age %5d : Salary \$%10.2f\n”, “John”, 40, 55000);
$string = sprintf(“The name is: %10s\nThe number is: %8.2f\n”, “Ellie”, 33);
print $string       # sprintf allows you to assign the formatted string to a variable
```

Numeric Literals

6	Integer
12.6	Floating Point
1e10	Scientific Notation
6.4E-33	Scientific Notation
4_348_348	Underscores instead of commas for long numbers

String Literals

- “There is more than on way to do it!”
- 'Just don't create a file called -rf.'
- “Beauty?\nWhat's that?\n”
- “”
- “Real programmers can write assembly in any language.”

Quotes from Larry Wall

Data Types/ Variables

- Perl supports three basic data types to hold variables: *scalars*, *arrays*, and *associative arrays* (*hashes*)
- Perl variables don't have to be declared before being used
- Variable names start with a "funny character", followed by a letter and any number of alphanumeric characters, including underscore
  - The funny character ($, @, %) represents the data type and context
  - The characters following the funny symbol are case sensitive

Categories of Variables (Scalar)

- Scalar variable holds a single value, single string, or a number
  - The name of scalar is preceded by a “$” sign
  - $first_name = “Melanie”;
  - $last_name = “Smith”;
  - $salary = 125000.00;
  - print $first_name, $last_name, $salary;

Categories of Variables (Array)

- Array variable: ordered list of scalars (i.e., strings and numbers)
  - The elements of the array are indexed by integers starting at 0
  - The name of the array is preceded by an “@” sign
  - @names = (“Jessica”, “Michelle”, “Linda”);
  - # Prints the array with elements separated by a space
  - print “@names”;
  - print “$names[0] and $names[2]”; # Prints Jessica and Linda
  - print “$names[-1]\n”;
  - $names[3] = “Nicole”; # Assign a new value as the 4th element
  - Some commonly used built-in functions
    - ***pop, push, shift, unshift, splice, sort***

Categories of Variables (Hash)

- Hash variable : Unordered list of key/value pairs, indexed by string (key)
  - The name of the hash is preceded by a “%” symbol
  - %employee = (
  - “Name” => “Jessica Savage”,
  - “Phone” => “(925) 555 -1234,
  - “Position” => “CEO”
  -);
  - print “$employee{“Name”}; #print a value
  - $employedd{“SSN”} = “999-333-1234”; # Assign a key/value
  - Some commonly used built-in functions
    - ***keys, values, each, delete***

Perl Variable Characteristics

- Variables **do not need to be declared**
- Variable type (int, char, ...) is decided at run time
  - $a = 5;   # now an integer
  - $a = “perl”;   # now a string

Operators on Scalar Variables

- Numeric and Logic Operators
  - Typical : +, -, *, /, %, ++, --, +=, -=, *=, /=, ||, &&, ! etc…
  - Not typical: ** for exponentiation
- String Operators
  - Concatenation: “.” - similar to strcat in C
    - **$first_name = “Larry”;**
    - **$last_name = “Wall”;**
    - **$full_name = $first_name . “ “ . $last_name;**

Equality Operators for Strings

- Equality/ Inequality : eq and ne
  - $language = “Perl”; # Assignment
  - if ($language == “Perl”) ... # Wrong!
  - if ($language **eq** “Perl”) ... # Correct

- Use eq / ne rather than == / != for strings

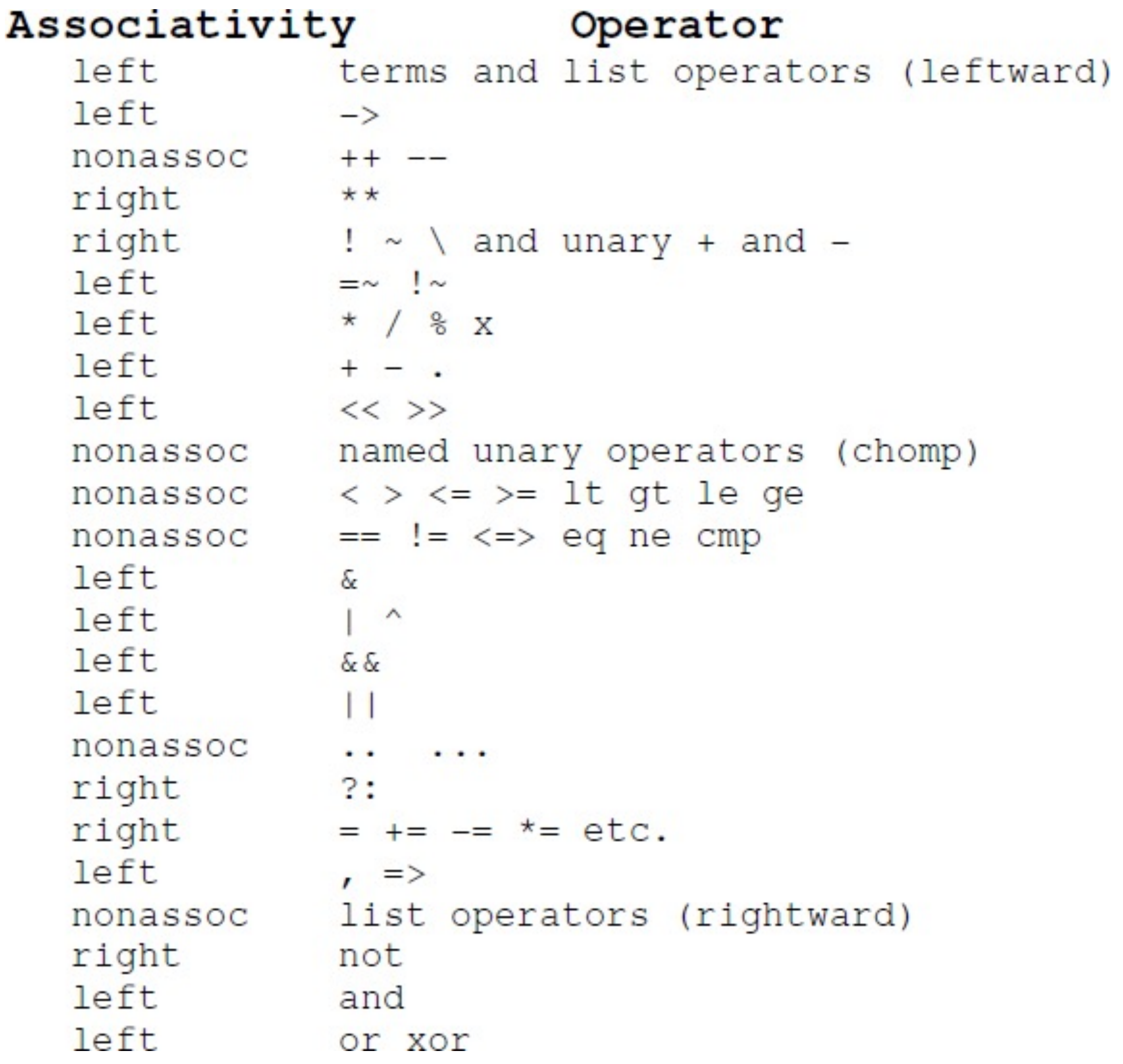
Relational Operators for Strings

- Greater than
  - Numeric : > String : gt
- Greater than or equal to
  - Numeric : >= String : ge
- Less than
  - Numeric : < String : lt
- Less than or equal to
  - Numeric : <= String : le

Comparison Operators

Comparison	Numeric	String
Equal	==	Eq
Not Equal	!=	Ne
Greater than	>	Gt
Less than	<	Lt
Greater or equal	>=	Ge
Less or equal	<=	Le

Operator Precedence and Associativity



Associativity	Operator
left	terms and list operators (leftward)
left	->
nonassoc	++ --
right	**
right	! ~ \ and unary + and -
left	=~ !~
left	* / % x
left	+ - .
left	<< >>
nonassoc	named unary operators (chomp)
nonassoc	< > <= >= lt gt le ge
nonassoc	== != <=> eq ne cmp
left	&
left	| ^
left	&&
left	||
nonassoc	.. ...
right	?:
right	= += -= *= etc.
left	, =>
nonassoc	list operators (rightward)
right	not
left	and
left	or xor

String Functions

- Convert to upper case
  - $name = uc($name);
- Convert only the first char to upper case
  - $name = ucfirst($name);
- Convert to lower case
  - $name = lc($name);
- Convert only the first char to lower case
  - $name = lcfirst($name);

Variables

- Perl has three types of variables - scalars (strings or numeric's), arrays and hashes
  - **$x = 45.67; $var = 'cost'; print "$var is $x";**
  - **@A = ('guna', 'me', 'cmu', 'pgh'); $i = 1; $A[$i] = 'guna';**

- With $#A, we can find
  - **length of an array A**
    - **$len = $#A + 1**
    - **$len = @A;**
  - **resize an array**
    - **@array = (10,12,45);**
    - **$#array = 1;**
    - Will result in an array of size ???

Variable Substitution

- Variables inside **strings** are replaced with their value
  - $stooge = “Larry”
  - print “$stooge is one of the three stooges.\n”;

  Produces the output:
  - Larry is one of the three stooges.

- With **single quotes –** no substitution
  - print '$stooge is one of the three stooges.\n’;

  Produces the output:
  - $stooge is one of the three stooges.\n

Character Escapes

- List of character escapes that are recognized when using double quoted strings
  - \n newline
  - \t tab
  - \r carriage return

- Common Example :
  - print “Hello\n”;
  - # prints Hello and then a return

Numbers and Strings are Interchangeable!

- If a scalar variable looks like a number and Perl needs a number, it will use the variable as a number!
  - $a = 4;   # a number
  - print $a + 18;   # prints 22
  - $b = “50”;   # looks like a string, but ...
  - print $b – 10;   # will print ???

if … else … statements

- Similar to C/C++ - except that *scope braces are REQUIRED!!*
  - if ($os eq “Linux”) {
  - print “Sweet!\n”;
  - }
  - elsif ($os eq “Windows”) { # no e!!!
  - print “Time to move to Linux, buddy!\n”;
  - }
  - else {
  - print “Hmm...!\n”;
  - }

unless ... else Statements

- Unless Statements are the opposite of if ...else statements.
  - unless ($os eq “Linux”) {
  -         print “Time to move to Linux, buddy!\n”;
  - }
  - else {
  -         print “Sweet!\n”;
  - }

- And again remember the braces are required!

while Loop

- While loop: Similar to C/C++ but again the braces are **required!!**

- Example :
  - $i = 0;
  - while ($i <= 1000) {
  -         print “$i\n”;
  -         $i++;
  - }

until Loop

- The until function evaluates an expression repeatedly until a specific condition is met

- Example:
  - $i = 0;
  - until ($i == 1000) {
  -         print “$i\n”;
  -         $i++;
  - }

for Loops

- Like C/C++
  - for ($i = 0; $i <= 1000; $i++) {
  -     print “$i\n”;
  - }

- Another way to create a for loop
  - for $i (0..1000) {
  -     print “$i\n”;
  - }

Control Inside a Loop

- Where you would use **continue** in C, use **next**
- Where you would use **break** in C, use **last**
- What is the output for the following?
  - for ($i = 0; $i < 10; $i++) {
  - if ($i == 1 || $i == 3)
  - { next; }
  - if($i == 5) { last; }
  - print "$i\n";
  - }

Control Structures (Loops and Conditionals)

A While Loop

```
$x = 1;
while ($x < 10){
     print “x is $x\n”;
     $x++;
}
```

Until loop

```
$x = 1;
until ($x >= 10){
     print “x is $x\n”;
     $x++;
}
```

Do-while loop

```
$x = 1;
do{
     print "x is $x\n";
     $x++;
} while ($x < 10);
```

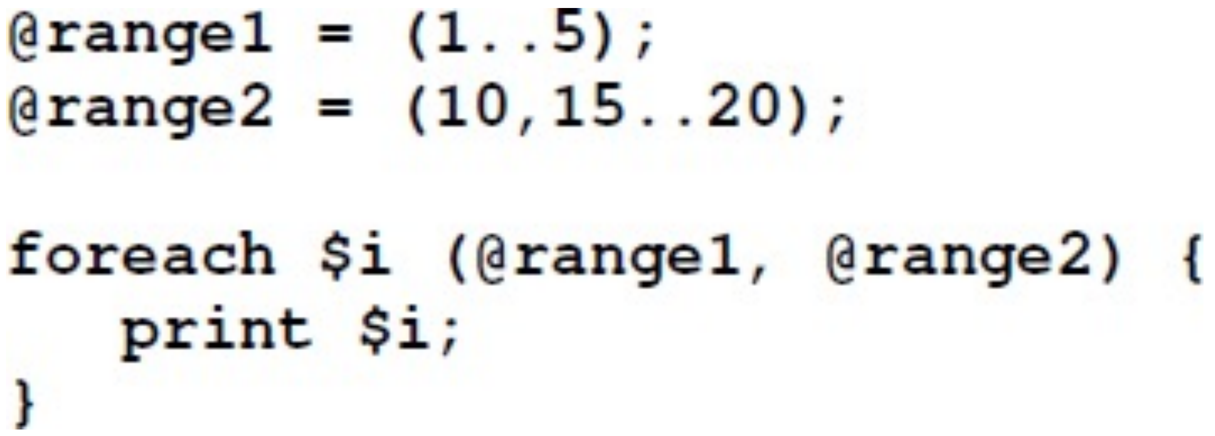
for statement

```
for ($x=1; $x < 10; $x++){
     print “x is $x\n”;
}
```

foreach statement

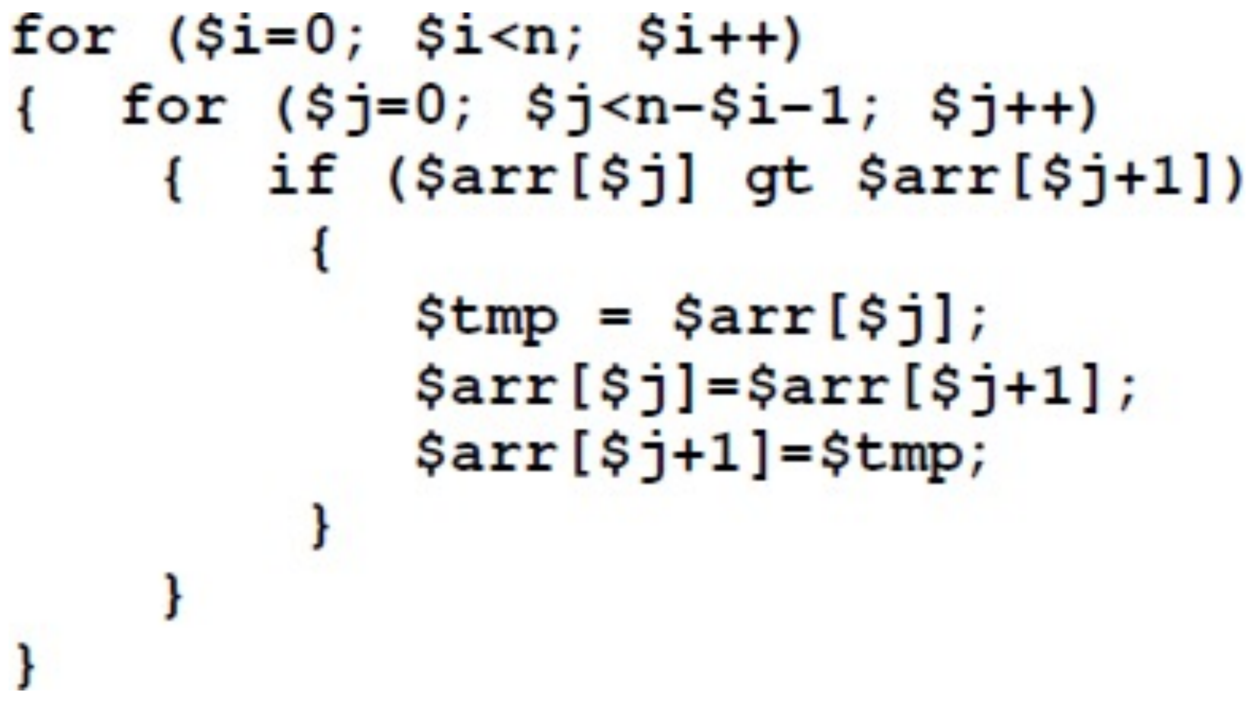
```
foreach $x (1..9) {
     print "x is $x\n";
}
```

What would be the output of the below code?



```
@range1 = (1..5);
@range2 = (10,15..20);

foreach $i (@range1, @range2) {
    print $i;
}
```

bubble sort on an array of strings



```
for ($i=0; $i<n; $i++)
{   for ($j=0; $j<n-$i-1; $j++)
     {   if ($arr[$j] gt $arr[$j+1])
          {
               $tmp = $arr[$j];
               $arr[$j]=$arr[$j+1];
               $arr[$j+1]=$tmp;
          }
     }
}
```

Arrays

- Array variable - denoted by @
  - @names = (“Larry”, “Curly”, “Moe”);
- To access array, use array variable
  - print @names;   # prints :???
  - print “@names”;   # prints :???
- You do not need to loop through the array to print it – Perl does this for you
  - print “The elements of \@names are @names\n”;

Array Elements

- To access one element of the array : use $
- Why? Because every element in the array is scalar
  - print “$names[0]\n”;   # prints : Larry
- What happens if we access $names[3] ?

Array Elements (Continued)

- The index of the last element in the array
  - print $#names;   # prints 2 in the previous example

- Find the number of elements in the array:
  - $array_si ze = @names;
    - $array_size now has 3 in the above example because there are 3 elements in the array

Array Elements (Continued)

```
print "Enter the number of the element you wish to view :";
chomp ($x=<STDIN>);
@names=("Muriel","Gavin","Susanne","Sarah","Anna","Paul","Simon");
print "The first two elements are @names[0,1]\n";
print "The first three elements are @names[0..2]\n";
print "You requested element $x who is $names[$x-1]\n"; # starts at 0
print "The elements before and after are : @names[$x-2,$x]\n";
print "The first, second, third and fifth elements are @names[0..2,4]\n";
print "a) The last element is $names[$#names]\n";      # one way
print "b) The last element is @names[-1]\n";           # different way
```

Array Elements (Continued)

- The two variables $myvar and @myvar are not, in any way, related
  - $myvar="scalar variable";
  - @myvar=("one“, "element“, "of“, "an“, "array“, "called“, "myvar");
  - print $myvar; # refers to the contents of a scalar variable
    # called myvar
  - print $myvar[1]; # refers to the second element of the array
    # myvar
  - print @myvar; # refers to all the elements of array myvar

Changing & Adding Array Elements

- Access an array element and assign it as a new value
  - @browser = ("NS", "IE", "Opera");
  - $browser[2]="Mosaic";
  - # This changes the value of the element with the third list element

- We can add a new element in the last position by just assigning the next position a value
  - @browser = ("NS", "IE", "Opera");
  - $browser[3]="Mosaic";

Splice Function

- Using the splice function, you can delete or replace elements within the array
  - @browser = ("NS", "IE", "Opera");
  - splice(@browser, 1, 1);
    - **argument of splice function: name of the array you want to splice, list number of the element where you wish to start the splice (starts counting at zero), number of elements you wish to splice.**

- If you want to delete more than one element, change that third number to the number of elements you wish to delete
  - @browser = ("NS", "IE", "Opera");
  - splice(@browser, 0, 2);

Splice Function (Continued)

- You can also use splice to replace elements
  - @browser = ("NS", "IE", "Opera");
  - splice(@browser, 1, 2, "NeoPlanet", "Mosaic");
    - **You just need to list your replacement elements after your other three arguments within the splice function**

- If you put the third parameter as 0, you simply add items
  - @new_browser = (“X0”, “X1”, “X2”, X3”, “X4”);
  - splice(@browser, 1, 0, @new_browser[1..3]);

Splice Function (Continued)

- Splice returns the elements removed from the array
  - @browser = ("NS", "IE", "Opera");
  - @dwarfs = qw(Doc Grumpy Happy Sleepy Sneezy);
  - @who = splice(@dwarfs, 3, 2, "NeoPlanet", "Mosaic");
  - print “@who\n”;
    - **$who2 = splice(@dwarfs, 3, 2, "NeoPlanet", "Mosaic");**
    - **print “Swho2\n”;**

Splice Function (Continued)

- Both the offset and the length can be negative numbers (count from end of array)
  - @browser = ("NS", "IE", "Opera“, “Safari”);
  - @new = splice(@ browser, 1, -1);
  - print “@new\n”;
  - @browser = ("NS", "IE", "Opera“, “Safari”);
  - @new = splice(@ browser, -3, 2);
  - print “@new\n”;

Splice Function (Continued)

- Can you see the difference between following codes?

```
@fruits = (“apples”, “bananas”, “tomatoes”, “pineapples”);

$fruits[1] = “”;
```

```
splice (@fruits, 1, 1);
```

Unshift/Shift

- To add an element to the left side, you would use the unshift function
  - @browser = ("NS", "IE", "Opera");
  - unshift(@browser, "Mosaic");

- To delete an element from the left side, you would use the shift function
  - @browser = ("NS", "IE", "Opera");
  - shift(@browser);

Unshift/Shift (Continued)

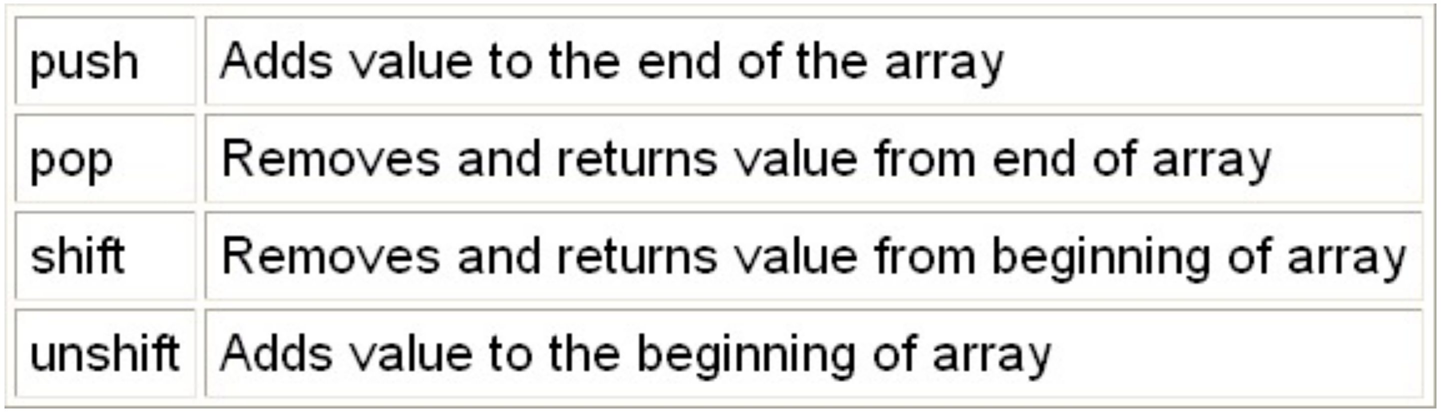
- You can keep the value you deleted from the array by assigning the shift function to a variable
  - @browser = ("NS", "IE", "Opera");
  - $old_first_element= shift(@browser);

Push/Pop

- These two functions are just like unshift and shift, except they add or delete from the right side of an array (the last position)

- So, if you want to add an element to the end of an array, you would use the push function and to delete from the right side, you would use the pop function
  - @browser = ("NS", "IE", "Opera");
  - push(@browser, "Mosaic");
  - push (@browser, “Safari“, @browser[1..2]);
  - $last_element = pop(@browser);

A table of array hacking functions



push	Adds value to the end of the array
pop	Removes and returns value from end of array
shift	Removes and returns value from beginning of array
unshift	Adds value to the beginning of array

Chop & Chomp

- If you want to take the last character of each element in an array and "chop it off", or delete it, you can use the chop function
  - @browser = ("NS4", "IE5", "Opera3");
  - chop(@browser);

- If you want to remove newline characters from the end of each array element, you can use the chomp function
  - @browser = ("NS4\n", "IE5\n", "Opera3\n");
  - chomp(@browser);

- The chomp function is much safer than the chop function, as it will not remove the last character if it is not \n

Chop & Chomp (Continued)

- Guess what happened with following codes…
  - print "Please tell me your name: ";
  - $name=<STDIN>;
  - print "Thanks for making me happy, $name !\n";

- How can we correct this???

Sort

- You can sort in ascending or descending order with numbers or strings

- Numbers will go by the size of the number, strings will go in alphabetical order

```
@browser = ("NS", "IE", "Opera");
@sortedBrowser = sort (ascend @browser);
print “@sortedBrowser\n”;
sub ascend {
 $a cmp $b;
}
```

Built-in Sorting of Arrays

- Two ways to sort:
- Default : sorts in a standard string comparisons order
  - sort LIST
- Usersub: create your own subroutine that returns an integer less than, equal to or greater than 0
  - sort USERSUB LIST
  - The **<=>** and **cmp** operators make creating sorting subroutines very easy
    - **If you are comparing numbers, your comparison operator should contain non-alphas, if you are comparing strings the operator should contains alphas only**

Sorting Numerically

- The sort function compares two variables, $a and $b
- The result is
  - **1** if $a is greater than $b
  - **-1** if $b is greater than $a
  - **0** if $a and $b are equal

Sorting Example

```
– #!/usr/local/bin/perl -w
– @unsortedArray = (3, 10, 76, 23, 1, 54);
– @sortedArray = sort numeric @unsortedArray;

– print “@unsortedArray\n”;          # prints 3 10 76 23 1 54
– print “@sortedArray\n”;            # prints 1 3 10 23 54 76

– sub numeric {
–           $a <=> $b
– }
```

Sorting Example (Continued)

```
%countries=('976', 'Mongolia', '52', 'Mexico', '212', 'Morocco',
'64', 'New Zealand', '33', 'France');

foreach (sort { $a <=> $b } keys %countries){
        print "$_ $countries{$_}\n";
}

foreach (sort values %countries){
        print "$_ \n";
}

foreach (sort { $countries{$a} cmp $countries{$b} } keys %countries){
        print "$_ $countries{$_}\n";
}
```

Sorting Example (Continued)

```
# You can sort several lists at the same time:

%countries=('976', 'Mongolia', '52', 'Mexico', '212', 'Morocco',
'64', 'New Zealand', '33', 'France');
@nations=qw(China Hungary Japan Canada Fiji);

# This sorts @nations and the values from %countries into a new array
@sorted= sort values %countries, @nations;


foreach (@sorted) {
        print "$_\n";
}
```

foreach

- foreach – automatic iteration over an array

- Example:
  - foreach $element (@array) {
  -         print “$element\n”;
  - }
  - This goes through each member ('iterates', another good technical word to use) of @array, and assigns each member in turn to the variable $element

- This is similar to :
  - for ($i = 0; $i <= $#array; $i++) {
  -         print “$array[$i]\n”;
  - }

$_ : Default Input and Searching Variable

- Previous example can be much shorter with $_
- Example:
  - foreach (@array) {
  -           print “$_”;
  - }
  - If you don't specify a variable to put each member into, $_ is used instead as it is the default for this operation

- This is similar to :
  - foreach (@array) {
  -           print ;
  - }
  - As we haven't supplied any arguments to print , $_ is printed as default

Sorting with foreach

- The sort function sorts the array and returns the items in sorted order

- Example :
  - @array = (“Larry”, “Curly”, “Moe”);
  - foreach $element (sort @array) {
  -         print “$element ”;
  - }

- Prints the elements in sorted order:
  - Curly Larry Moe

reverse /join

- You can reverse the order of the array elements with the ***reverse*** function
  - @browser = ("NS", "IE", "Opera");
  - reverse(@browser);

- You can create a flat file database from your array with the join function (this is more useful if you are reading and writing from files)

- The function creates a variable for each element, joined by your delimiter
  - $result = join $glue, @pieces;
  - @browser = ("NS", "IE", "Opera");
  - $brower_sting = join(":", @browser);

Arrays to Strings – join (Example)

- Array to space separated string
  - @array = (“Larry”, “Curly”, “Moe”);
  - $string = join(“ “, @array);
  - # string = “Larry Curly Moe”

- Array of characters to string
  - @stooge = (“c”, “u”, “r”, “l”, “y”);
  - $string = join(“”, @stooge);
  - # string = “curly”

split

- It allows you to create an array of elements by splitting a string every time a certain delimiter (a character of your choice) shows up within the string
  - @fields = split /separator/, $string;
  - $browser_list="NS:IE:Opera";
  - @browser= split(/:/, $browser_list);
- Notice in the split function that you place your delimiter between two forward slashes
- You then place the string you want to split as the second argument

Strings to Arrays : split (Example)

- Split a string into words and put into an array
  - @array = split(/ /, “Larry Curly Moe”);
  - # creates the same array as we saw previously

- Split into characters
  - @stooge = split(//, “curly”);
  - # array @stooge has 5 elements: c, u, r, l, y

- Split on any character
  - @array = split(/:/, “10:20:30:40”);
  - # array has 4 elements : 10, 20, 30, 40

Strings to Arrays : split (Continued)

- split on Multiple White Space
  - @array = split(/\s+/, “this   is   a \t   test”;
  - # array has 4 elements : this, is, a, test

- Default for split is to break up $_ on white space
  - @fileds = split;
  - # like split /\s+/, $_;

- More on ‘\s+’ later

Perl Associative Arrays (Hashes)

- Why use hashes ?
  - When you want to look something up by a keyword
    - **Suppose we wanted to create a program which returns the name of the country when given a country code**
    - **We'd input ES, and the program would come back with Spain**

- Typical application of Hashes in Perl
  - Given name, family name
  - Host name, IP address # %ip_address
    - **http://www.stonehenge.com, 123.45.67.89**
  - IP address, host name # %host_name
    - **%host_name = *reverse* %ip_address;**
  - Username, number of disk blocks they are using
  - Driver's license number, name
  - Plate number, registered address

Perl Associative Arrays (Hashes)

- Unlike a regular array, however, you get to use your own text strings to access elements in the array

- When a regular array is created, its variables stay in the same order you created them in. With a hash, perl reorders elements for quick access

Perl Associative Arrays (Hashes)

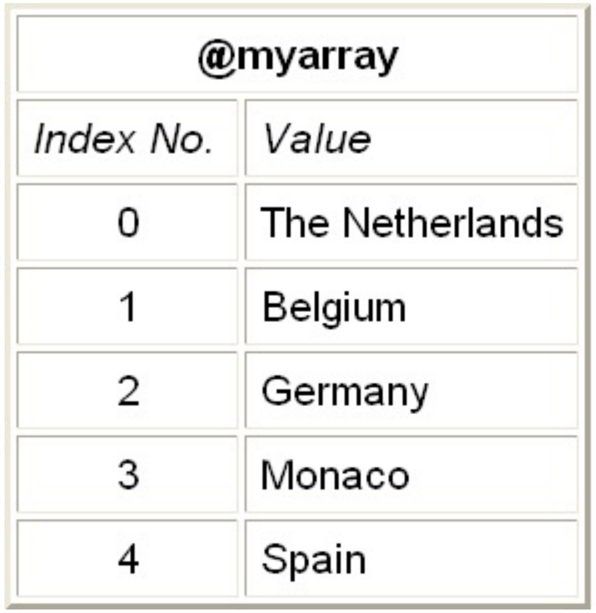
- Associative arrays are created with a set of key/value pairs
  - A key is a text string of your choice that will help you remember the value later
  - The value, then, is the value of the variable you want to store

- Each key of a hash must be unique
  - If you do define a certain key twice, the second value overwrites the first
  - The values of a hash can be duplicates, but never the keys

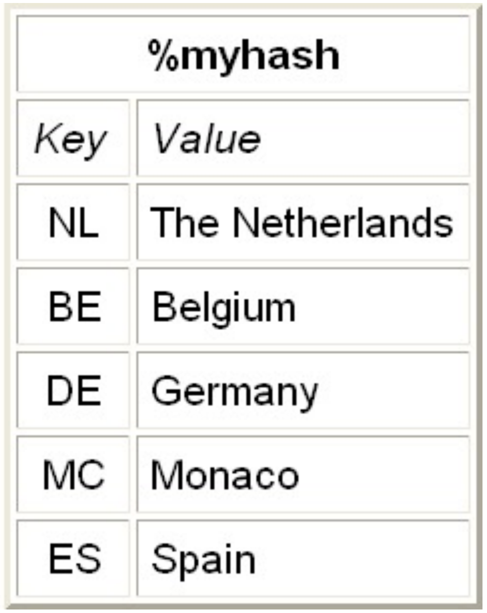
Example (Hashes)

```
my %family_name = (“fred” => “flintstone, “dino” => undef,
                                “barney” = “rubble”, “betty” => “rubble”);
my @k = keys %family_name;
my $k_count = @k;
my @v = values %family_name;

$foo = “bar;
print "$family_name{ $foo . “ney”}";  # prints ???

while ( ($key, $value) = each %family_name) {
            print "$key => $value\n";
}
```

Arrays & Associative Arrays (Hashes)



@myarray	
Index No.	*Value*
0	The Netherlands
1	Belgium
2	Germany
3	Monaco
4	Spain



%myhash	
Key	*Value*
NL	The Netherlands
BE	Belgium
DE	Germany
MC	Monaco
ES	Spain

- If we want 'Belgium' from @myarray and also from %myhash
  - print "$myarray[1]";
  - print "$myhash{'BE'}";

Define an Associative Array

- The percent (%) sign at the beginning in front of the array name
  - %array_name = ('key1', 'value1', 'key2', 'value2');

- This indicates that what follows is an associative array, so the interpreter knows to use the keys and values, rather than assign index numbers to each string
  - %our_friends = ('best', 'Don', 'good', 'Robert', 'worst', 'Joe');

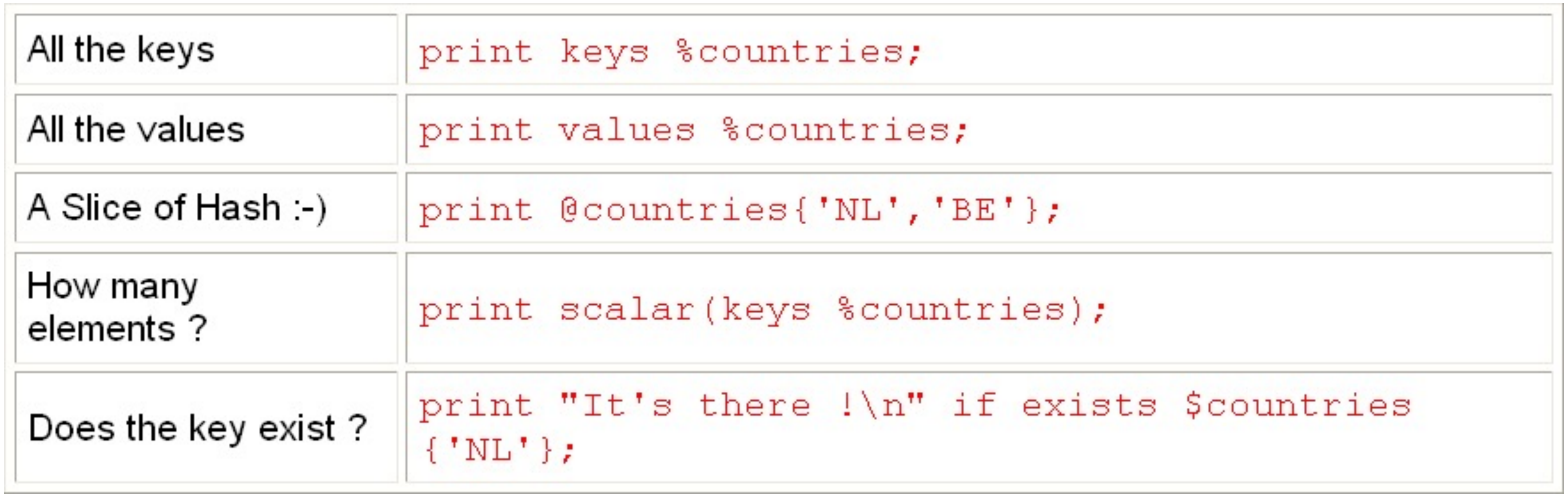
Access Your Elements in Hash

- To access an element, you use your key string in place of a number
  - You define a plain variable with the array name followed by its key
    - **So if we wanted to get the name of our 'good' friend, we would use**
    - **$our_friends{'good'}**

- Notice the key string 'good', which will give us back our good friend Robert
  - $good_friend = $our_friends{'good'};
  - print "I have a good friend named $good_friend.\n";

Access Your Elements in Hash (Continued)

%countries=('NL', 'The Netherlands', 'BE', 'Belgium', 'DE', 'Germany',
'MC', 'Monaco', 'ES', 'Spain');



All the keys	`print keys %countries;`
All the values	`print values %countries;`
A Slice of Hash :-)	`print @countries{'NL','BE'};`
How many elements ?	`print scalar(keys %countries);`
Does the key exist ?	`print "It's there !\n" if exists $countries {'NL'};`

Adding to the Hash

- Like a regular array, you can add a value to an associative array by simply defining a new value in your script
  - %our_friends = ('best', 'Don', 'good', 'Robert', 'worst', 'Joe');
  - $our_friends{'cool'} = "Karen";

- This adds the key/value pair of 'cool' and 'Karen' to the %our_friends array

Deleting from the Hash

- You can also delete a key/value pair from an associative array using the delete function, which is a little different than the regular array
  - %our_friends = ('best', 'Don', 'good', 'Robert', 'worst', 'Joe');
  - delete ($our_friends{'worst'});

- Associative arrays are very handy when you are trying to get input from forms on a web page
  - Most scripts that do this read the form values in as key/value pairs to an associative array, thus making it easy to use the values you need

Scoping rule with my & local

- Scope (visibility) of a variable
  - Lexical (static) scoping: my
  - Dynamic scoping: local
- A ***my*** variable has a block of code as its scope
  - A block is often declared with braces { }
- A ***local*** variable can affect what happens outside of the block of code in which it is used
  - Compiler can’t tell its behavior
  - Suspend the use of global variable, and use a local value
  - Understand concept with run-time stack during function calls

Example (my & local variables)

```
#!/usr/bin/perl
$x = 1;
sub foo {
      print "$x\n";
}
sub bar {
      $x = 2;    # my, local $x ???
      foo();
}
&foo;
&bar;
&foo;
```

Example (my & local variables)

```
$var = 5;
print $var, "\n";
&fun1;
print $var, "\n";
sub fun1 {
      $var = 10;  #my, local $var ???
      print $var, "\n";
      &fun2;
      print $var, "\n";
}
sub fun2 {
      $var++;
}
@foo = (1,2, 3, 4, 5);
foreach my $var (@foo) { print $var, "\n";}
```

Length

- The length function simply gives you back the number of characters in a string variable
  - $ice="cold";
  - $length_ice = length ($ice);

- The length function always works on strings and it creates *scalar* context for its parameters

- What will be the output? Why?

```
$my_string = “abc”;
@my_array = (1,2,3,4,5);
print (length $my_string);
print (length @my_array);
```

Substring (substr)

- The substring function is a way to get a portion of a string value, rather than using the entire value
  - $portion = substr($string_variable, start number, length);
    - **$string_variable will be the variable from which you wish to create the substring**
    - **The start number is the character within the string from which you want to start your substring**
      - **Remember, though- the first number in a string here is zero rather than 1, so be careful when you make the count**
    - **The length above is the amount of characters you wish to take out of the string**

  -
    ```
    $ice="cold";
    $age = substr($ice, 1, 3);
    print "It sure is $ice out here today.";
    print "I wonder if I am $age enough to play in the snow?";
    ```

Example

```
%countries=('NL', 'The Netherlands', 'BE', 'Belgium', 'DE', 'Germany',
'MC', 'Monaco', 'ES', 'Spain');
print "Enter the country code:";
chomp ($find=<STDIN>);
$find=~tr/a-z/A-Z/;     # make sure everything is in uppercase
print "$countries{$find} has the code $find\n";
foreach (reverse sort keys %countries) {
        print "The key $_ contains $countries{$_}\n";
 }
```

Subroutines

- User-defined functions in Perl
  – Let us recycle one chunk of code many times in one program
  – Name of subroutine is another Perl identifier
    – **Letters, digits, and underscores, but it can't start with a digit**
  – To invoke subroutine, place & in front of its name

```
$n = &max(10, 15);

sub max {

        my ($m, $n) = @_;      # new private variable for this block

        if ($m > $n) { $m } else { $n }

}
```

Persistent, Private Variable

- Perl will keep private variable values between calls with "state"
  - The first time we call a subroutine, Perl declares and initialize state variable(s)
  - Perl ignores the statement on subsequent calls
  - Between calls, Perl retains the value of state variable

```
sub marine {
        state $n = 0;       # private, persistent variable $n
        $n += 1;
        print "Hello, sailor number $n!\n";
}
```

Perl Start

- ActivePerl - install at home
  - www.perl.org
  - www.perl.com Has rpm's for Linux
  - www.activestate.com binaries for Windows
- Perl resources - from homepage

Hello World

- Program:
  - #!/usr/local/bin/perl –w
  - print “Hello World!\n”;
- Save this as “hello.pl”
- Give it executable permissions
  - chmod ug+x hello.pl
- Run it by typing:
  - ./hello.pl
  - or
  - perl hello.pl

Hello World (Continued)

- “.pl” extension is optional but is commonly used
  - E.g. .pro or .prlg for Prolog

- The first line “#!/usr/local/bin/perl” tells UNIX where to find Perl

- “-w” switches on warnings : not required but a really good idea

A String Example Program

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

$var1 = “oscar”;
$var2 = “henry”;
$var3 = “diana”;

print ucfirst($var1); # Prints ‘Oscar'
print uc($var2); # Prints ‘HENRY'
print lcfirst(uc($var3)); # Prints ‘dIANA'
```

Pattern Matching

- Perl has many built in capabilities for pattern matching
  - Pattern matching is one of Perl's most powerful and probably least understood features

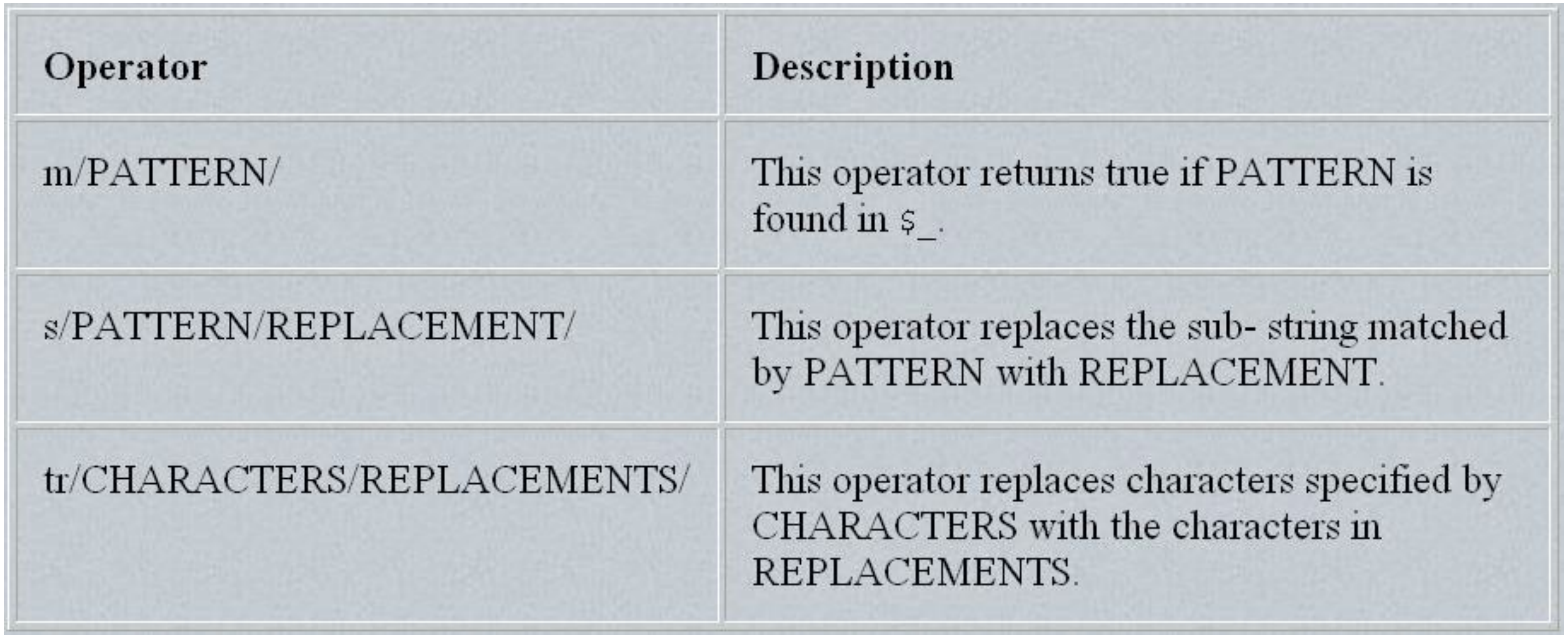
- When do we use pattern matching?
  - Pattern matching allows programs to scan data for patterns and extract data
    - **For example, to look for a specific name in a phone list or all of the names that start with the letter *a*.**
  - Can be used to reformat documents
  - Search & Replace

Regular Expressions

- You can use a *regular expression* (*regex*) to find patterns in strings
  - A regular expression is a pattern to be matched

- Three main use for *regex*
  - Matching
    - **use normal characters to match single characters**
    - **uses the m/ / operator, which evaluates to a true or false value**
  - Substitution
    - **substitutes one expression for another; it uses the s/ / / operator**
  - Translation
    - **translates one set of characters to another and uses the tr/// operator.**

Perl's Regular Expression Operators



Operator	Description
m/PATTERN/	This operator returns true if PATTERN is found in $_.
s/PATTERN/REPLACEMENT/	This operator replaces the sub- string matched by PATTERN with REPLACEMENT.
tr/CHARACTERS/REPLACEMENTS/	This operator replaces characters specified by CHARACTERS with the characters in REPLACEMENTS.

All three regular expression operators work with $_ as the string to search. You can use the binding operators (=~ and !=) to search a variable other than $_.

Regular Expressions (Continued)

- Using a regex to match against a string returns either true or false
  - $name=~m/John/
  - $name=~/John/ # Simpler way
  - # If "John" is inside $name, then True
  - The regular expression itself is between / / slashes, and the =~ operator assigns the target for the search

- Sometimes you just want to see if there is a match

- Other times you might want to do something with the matched string
  - Replace it or store it in a variable

The Matching Operator (m//)

- The matching operator (m//) is used to find patterns in strings
  - One of its more common uses is to look for a specific string inside a data file
    - **For instance, you might look for all customers whose last name is "Johnson" or you might need a list of all names starting with the letter *s***

- The matching operator only searches the $_ variable
  - makes the match statement shorter because you don't need to specify where to search
  - Example
    - **$needToFind = "bbb";**
    - **$_ = "AAA bbb AAA";**
    - **print "Found bbb\n" if m/$needToFind/;**

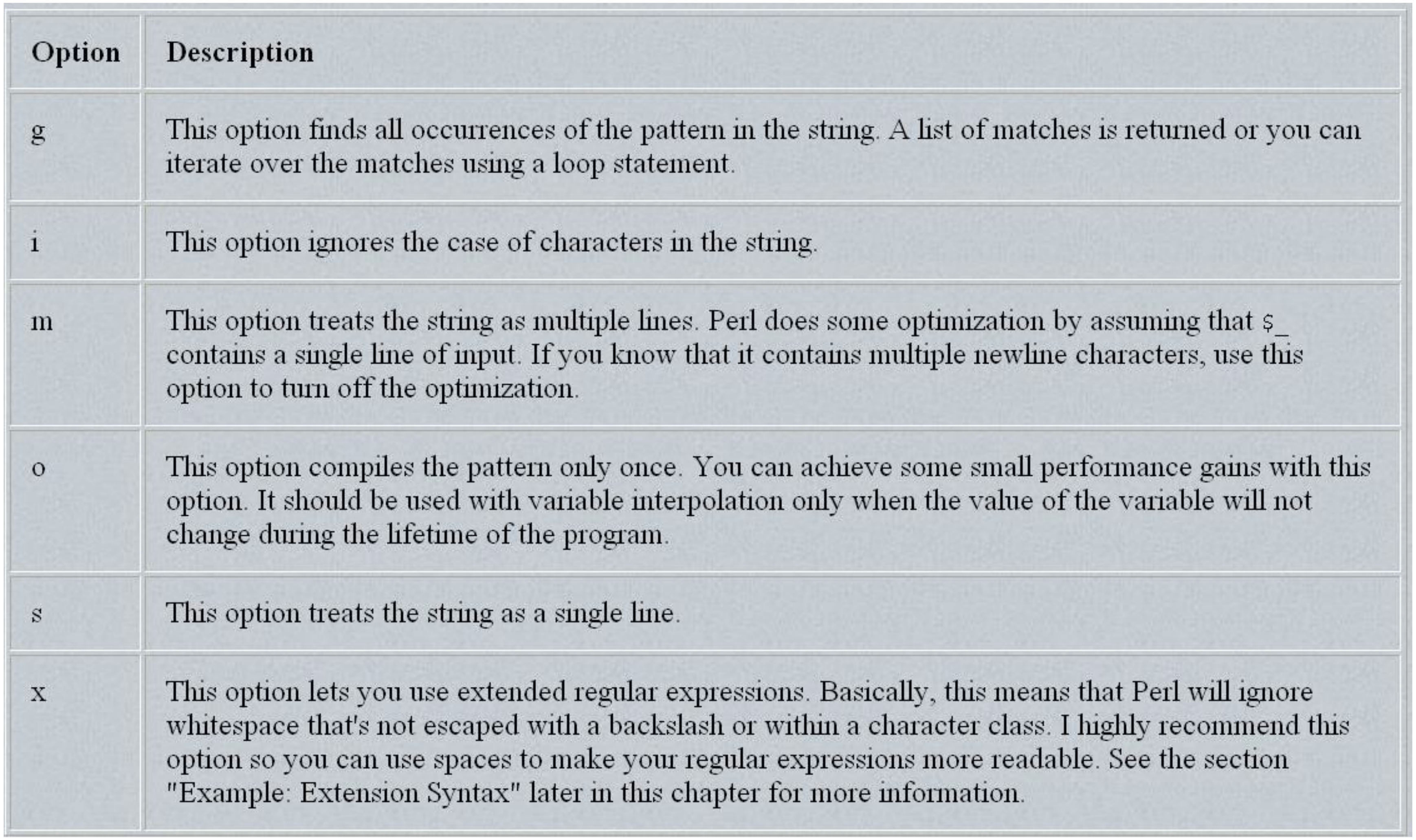
The Matching Operator (m//) (Continued)

- Using the matching operator to find a string inside a file is very easy because the defaults are designed to facilitate this activity

- Example

```
– $target = "M";
– open(INPUT, "<findstr.dat");
– while (<INPUT>)  {
–             if (/$target/) {
–                 print "Found $target on line $.";
–              }
– }
– close(INPUT);
```

The Matching Operator (m//) (Continued)



Option	Description
g	This option finds all occurrences of the pattern in the string. A list of matches is returned or you can iterate over the matches using a loop statement.
i	This option ignores the case of characters in the string.
m	This option treats the string as multiple lines. Perl does some optimization by assuming that $_ contains a single line of input. If you know that it contains multiple newline characters, use this option to turn off the optimization.
o	This option compiles the pattern only once. You can achieve some small performance gains with this option. It should be used with variable interpolation only when the value of the variable will not change during the lifetime of the program.
s	This option treats the string as a single line.
x	This option lets you use extended regular expressions. Basically, this means that Perl will ignore whitespace that's not escaped with a backslash or within a character class. I highly recommend this option so you can use spaces to make your regular expressions more readable. See the section "Example: Extension Syntax" later in this chapter for more information.

Example

```
if (/barney.*fred/is) {          # both /i and /s
        print "That string mentioneds Fred after Barney!\n";
}
```

=~ *operator*

- The search, modify, and translation operations work on the $_ variable by default

- What if the string to be searched is in some other variable?
  - That's where the binding operators come into play
  - Perl lets you bind the regular expression operators to a variable other than $_

- There are two forms of the binding operator: the regular =~ and its complement !~

=~ operator (Continued)

- Perl uses the =~ operator to compare a string to a regular expression
- The string to compare is on the left, regex on the right
- This returns either 1 or 0 (true or false)
  - $mycar =<STDIN>;
  - if ($mycar =~ /abc/) {
  -        print ”$mycar contains ‘abc’!”;
  - }
  - else {
  -        print ”$mycar does not contains ‘abc’!”;
  - }

Example

```
$scalar = "The root has many leaves and root";

$match = $scalar =~ m/root/;

$substitution = $scalar =~ s/root/tree/g;

$translate = $scalar =~ tr/h/H/;

print("\$match = $match\n");                      # $match = ???

print("String has not root.\n") if $scalar !~ m/root/;

print("\$substitution = $substitution\n");        # $substitution = ???

print("\$translate = $translate\n");              # $translate = ???

print("\$scalar = $scalar\n");                     # $scalar = ???
```

[ ] – Character Class

- A group of characters in square brackets will match any characters in the bracket

- A caret (^) negates the match
  - /[ab][cd]/ This will match a or b followed by c or d
  - /[aeiou]/ This will match all vowels
  - /[^aeiou]/ This will match anything but vowels
  - /[0123456789]/ All numerals
    - **compare this with m/0123456789/ # exact digit sequence**
  - /[0-9]/ All numerals (Use the – to specify a range)

Symbolic Character Class

```
.     # matches all characters except for the newline
\d    # same as [0-9]
\D   # same as [^0-9]
\s    # same as [\t \n]
\S    # same as [^\t \n]
\w    # same as [a- zA-Z0-9_]
\W   # same as [^a-zA-Z0-9_]
```

Symbolic Character Class (Example)

```
$_ = "AAABBBCCC";
$charList = "ADE";
print "matched" if m/[$charList]/;  #will display matched


$_ = "AAABBBCCC";
print "matched" if m/[^ABC]/;  #will display nothing
(This match returns true only if a character besides A, B, or C   is
in the searched string)


$_ = "AAABBBCCC";
print "matched" if m/[^A]/;   # string "matched" will be displayed
(because there is a character beside A)
```

Word Boundaries

- */b option* is used for exact word boundaries
- m/foo\b/;   # will match foo but not foobar
- m/\bwiz/;   # match wizard but not geewiz

```
$mystr = “abcfeed”;
$ print "matched" if mystr =~ m/foo|far|fee/;
$ print "matched" if mystr =~ m/\b(foo|far|fee)\b/;
```

Matching at specific points

- Can you see the difference below???

- if (/n/i) {
  - True if the word contains an ‘N’ or ‘n’ anywhere in it

- if (/^n/i) {
  - True if the string starts with an ‘N’ or 'n‘

- if (/n$/i) {
  - True if the string ends with an ‘N’ or 'n‘

- if (/[^n]/i) {
  - True if the word does not contain an ‘N’ or 'n‘ anywhere in it

Example

```
@names=qw(Karlson Carleon Karla Carla Karin Carina);
foreach (@names) {      # sets each element of @names to $_ in turn
        if (/[\-a-eKCZ]arl[^sa]/)  {
                print "Match ! $_\n";
         } else {
                 print "Sorry. $_\n";
          }
}
```

Negating the regex

- If you want to negate the entire regex, change =~ to !~
- if ($_ !~/[KC]arl/) {
- if (!/[KC]arl/) {   # another way

Grouping

- Grouping allows you to look for a certain (or arbitrary) amount of something
  - You can look for at least *n* times
  - At least *n* but not more than *m* (a range)
  - 1 or more of something
  - 0 or 1, 0 or more (optional)

Grouping Quantifiers

- A quantifier with a pattern determines how many times pattern must appear

- Quantifiers:

*	0 or more times
+	1 or more times
?	0 or 1 time
{n}	Exactly n times
{n,}	At least n times
{n,m}	At least n , but no more than m times

```
/-?\d+\.?\d*/                  # what is this doing?
/ -?   \d+   \.?   \d* /x      # a little better
```

Grouping Quantifiers (Continued)

- Parentheses may be used for grouping
  - /fred+/ matches strings like fredddddd but not like fredfredfred
  - /(fred)*/   # matches strings like “hello, world”
- Parentheses also gives us a way to reuse part of the string directly in the match
  - Back references (\1, \2, \3, etc)

Example

```
$_ = “abba”;
if (/(.)\1/) {     #  matcjes ‘bb’
      print “It matches same character next to itself!\n”;
}
$_ = “yabba dabba doo”;
if (/y(….) d\1/)                  # matches ???
if (/y(.)(.)\2\1/)                # matches ???

$_ = “aa11bb”;
if (/(.)\111/)             # matches ???
if (/(.)\g{1}11/)           # same
```

Example

$_ = "AA AB AC AD AE";

m/^(\w+\W+){5}$/;

# match statement will be true only if five words are present in the $_variable

m/^\w+/; # will match "QQQ" and "AAAAA" but not "" or " BBB ".

# match a starting word whose length is unknown

m/\s*\w+/; # will match "QQQ" and "AAAAA" and " BBB ".

# allow leading white

$_ = "AAA BBB CCC";*** ***m/(\w+)/;*** ***print("$1\n");

# looked for the first word in a string and stored it into the first buffer, $1 (will display AAA)

Example

```
$_ =  "AAA BBB CCC";
@matches = m/(\w+)/g;
print("@matches\n");
```

want to find all the words in the string, you need to use the /g (The program will display **AAA BBB CCC)**

```
m/(.)\1/;
```

need to find repeated characters in a string like the AA in "ABC AA ABC“

```
m/^\s*(\w+)/;
```

need to find the first word in a string

Saving Parts of a Match

- Parentheses allow you to save pieces of your match
- If the match is successful, the item in parentheses get stored in $1, $2, $3, etc according to their order
  - $myvar =~ /System Configuration: (.*)/;
    - **everything from (to) will be put into $1 if the match is successful**
    - **. matches any non-newline character**
    - *** means 0 or more of the previous character**
    - **So, this saves everything after “System Configuration: “ into $1**

Example

```
$_ = ‘My email address is John@Yahoo.com.’;   # any change with “” ???

/(john)\@(yahoo.com)/i;         # Paren forces matches to $1, $2, etc

print “Found it! $1 at $2\n”;     # What will be printed???



$_='My email address is <john@yahoo.com> !.';

/<([^>]+)/i;

print "Found it ! $1\n";               # What will be printed???
```

Alteration

- The | character is used to specify alternative expressions
  - m/abc/ will match "abc" but not "cab" or "bca"
- Basically an OR function for regex’s
  - m/\w|\w\w/ will match a single word character or two consecutive word characters
  - $class =~ /mat374|mat375/;
- This works well with parentheses
  - $class =~ /(mat374|mat375)/;
  - print “The class is $1”;

Searching & Replacing

• /PATTERN/	Match Operator
• m/PATTERN/	Match Operator
• s/PATTERN/REPLACE/	Search & Replace
• tr/searchlist/replacelist/	Character Search & Replace
• split(/PATTERN/, $line)	split on regex

The Substitution Operator (s///)

- The substitution operator (s///) is used to change strings

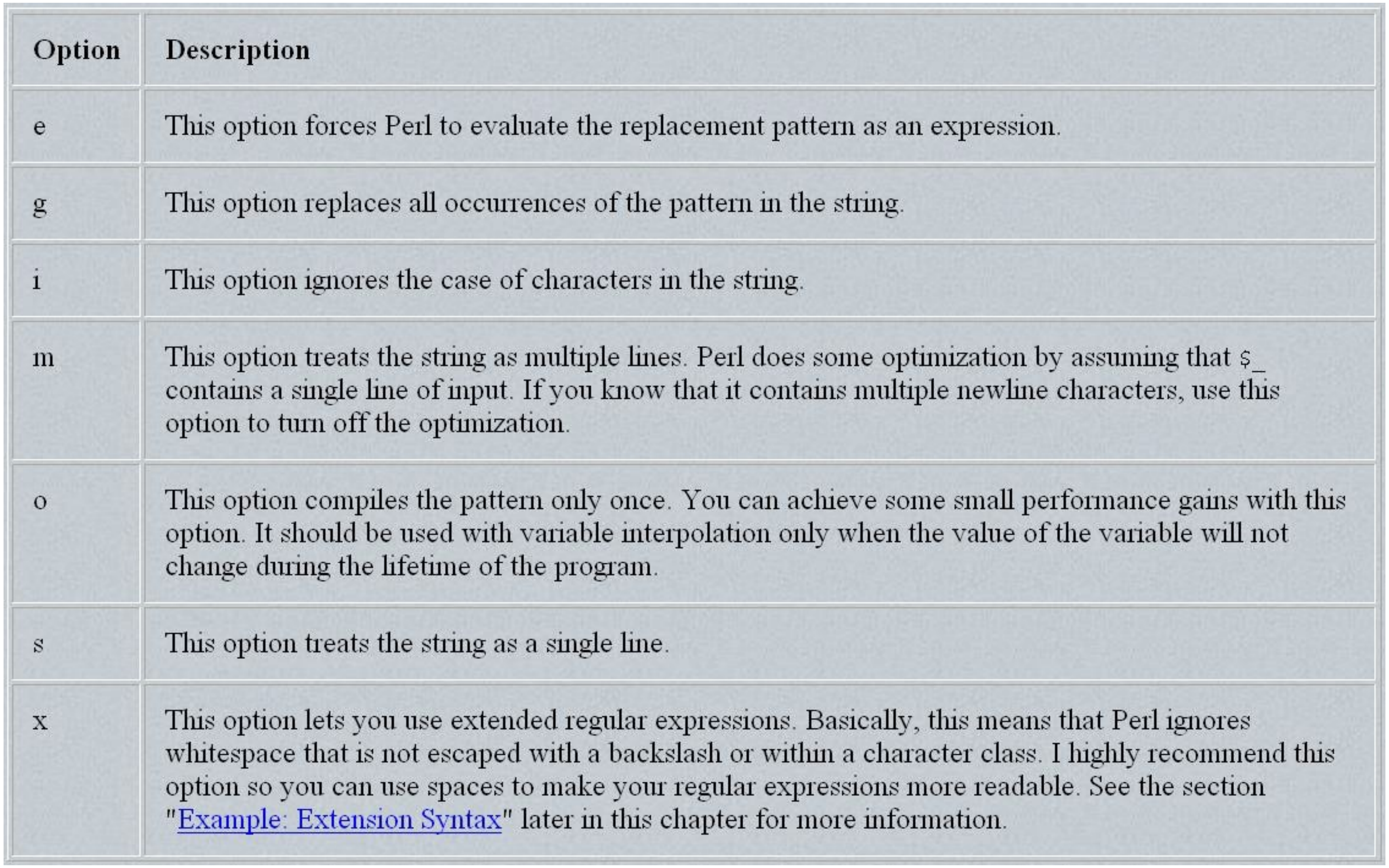
- Examples
  - $needToReplace = "bbb";
  - $replacementText = “1234567890”;
  - $_ = "AAA bbb AAA";
  - $result = s/$needToReplace/$replacementText/;
  - print $_;
  - print $result ; # the number of substitutions made

The Substitution Operator (s///) (Continued)

- The substitution operator is used to remove substrings

- Examples
  - $needToReplace = "bbb";
  - $replacementText = “1234567890”;
  - $_ = "AAA bbb AAA";
  - s/bbb//;
  - print $_;

Options for the Substitution Operator



Option	Description
e	This option forces Perl to evaluate the replacement pattern as an expression.
g	This option replaces all occurrences of the pattern in the string.
i	This option ignores the case of characters in the string.
m	This option treats the string as multiple lines. Perl does some optimization by assuming that $_ contains a single line of input. If you know that it contains multiple newline characters, use this option to turn off the optimization.
o	This option compiles the pattern only once. You can achieve some small performance gains with this option. It should be used with variable interpolation only when the value of the variable will not change during the lifetime of the program.
s	This option treats the string as a single line.
x	This option lets you use extended regular expressions. Basically, this means that Perl ignores whitespace that is not escaped with a backslash or within a character class. I highly recommend this option so you can use spaces to make your regular expressions more readable. See the section "Example: Extension Syntax" later in this chapter for more information.

Example

```
$_ = “home, sweet home!”;
s/home/cave/g;
print “$_\n”;         # print what???

$_ = “    Input     data\t may have              extra whitespace.   ”;
s/\s+/ /g;            # collapse white spaces
s/^\s+//;             # replace leading white spaces with nothing
????                  # replace trailing white spaces with nothing
```

The Translation Operator (tr///)

- The translation operator (tr///) is used to change individual characters in the $_ variable
- It requires two operands, like this: **tr/a/z/;**
  - This statement translates all occurrences of a into z
- If you specify more than one character in the match character list, you can translate multiple characters at a time, like this: **tr/ab/z/;**
  - This statement translates all a and all b characters into the z character

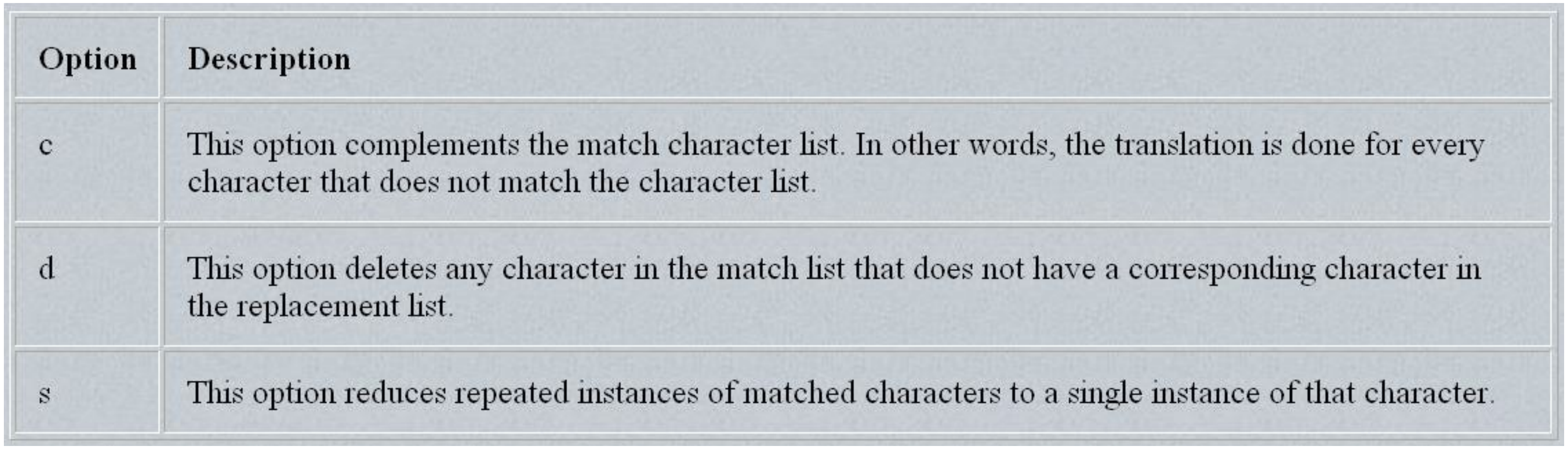
The Translation Operator (tr///) (Continued)

- If the replacement list of characters is shorter than the target list of characters, the last character in the replacement list is repeated as often as needed

- However, if more than one replacement character is given for a matched character, only the first is used, like this: **tr/WWW/ABC/;**
  - This statement results in all W characters being converted to an A character
  - The rest of the replacement list is ignored

- Unlike the matching and substitution operators, the translation operator doesn't perform variable interpolation

Options for the Translation Operator



Option	Description
c	This option complements the match character list. In other words, the translation is done for every character that does not match the character list.
d	This option deletes any character in the match list that does not have a corresponding character in the replacement list.
s	This option reduces repeated instances of matched characters to a single instance of that character.

Options for the Translation Operator (Continued)

- Normally, if the match list is longer than the replacement list, the last character in the replacement list is used as the replacement for the extra characters. However, when the d option is used, the matched characters are simply deleted.

- If the replacement list is empty, then no translation is done. The operator will still return the number of characters that matched, though. This is useful when you need to know how often a given letter appears in a string. This feature also can compress repeated characters using the s option.

Example

```
$x = “ ’quoted words’ ”;
$x =~ s/^’(.*)’$/$1/;
print “$x”;            # print what???

$y = "I'm fine. Thank you.";
$count = ($y =~ tr/./././);   # $count = ($y =~ s/././); ???
print $y, "\n";
print $count, "\n";
```

Examples of Regex

1. /[abc]|[123]/ --- matches a string that contains a character from [abc] or [123]
2. /\dguna/ --- matches a string that begins with a digit and followed by guna
3. /gu+n?a/ -- matches guna, gun, guuna, etc
4. $_ = "i like   programming"; s/ +/_/; -- replaces consecutive spaces with _ character.
5. /g{3,8}/ -- matches 3 to 8 g's
6. /g{2,}/ -- matches with strings with at least 2 g's
7. /g{4}/ -- matches with strings with exactly 4 g's

Basic I/O: Reading Input

- To get input from a user, use <STDIN> filehandle
- When assigned as a scalar, one line is read
  - $a = <STDIN>;
    - **$a will contain one line**
- When assigned as an array, multiple lines are read
  - @a = <STDIN>;
    - **@a will contain an element for each line entered**

@ARGV

- The @ARGV array holds any arguments typed on the command line
- $0 is a special variable which holds the name of your script
- If you invoke a program like this:
  - % myprog.pl file1.txt file2.txt
- The following variables get set automatically:
  - $ARGV[0] = “file1.txt”
  - $ARGV[1] = “file2.txt”
  - $0 = “myprog.pl”

Diamond Operator

- Special kind of line-input operator
  - Input comes from the user's choice of input

- Unlike <STDIN> , the diamond operator gets its data from the file or files specified on the command line that invoked the Perl program
  - # In yourprog.pl
  - while (<>) {

```
        print $_;
 }
# Now you invoke your program
yourprog.pl file1 file2 file3
```

  - **It will just print each line of file until EOF is reached**

```
yourprog.pl file1 -  file2
```

Diamond Operator (Continued)

- You can even set this array within your program and have the diamond operator work on that new list rather than the command-line arguments
  - @ARGV = ("aaa“, "bbb“, "ccc");

```
while (<>) {       # process files aaa, bbb, and ccc
      print $_;
}
```

Printing Output

- print() – takes a list of strings and sends each one to STDOUT
- printf() - takes a format control string that defines how to print the remaining arguments
  - Good for formatting numbers – allows you to specify how many decimal places to show
- Example
  - $a = 1244.3892104;
  - printf("%10.2f %d %15s", $a, $a, "Gumby");

Filehandles

- A filehandle is a variable name used to access a file
- Filehandle should use UPPERCASE letters for their name
- A filehandle is not prefixed by any special character

Using Filehandles

- There are usually three steps to using a filehandle:
- 1) Open the filehandle
- 2) Read/Write from/to the file
- 3) Close the filehandle

Creating a Filehandles

- The open () command creates a filehandle
- open () takes two arguments, the filehandle name and an expression
- The expression is a mode followed by a file name
  - There are three modes:
  - read   <
  - write   >
  - append   >>

Reading from a File

- open(FH, “<filename”)

- Example
  - open(DICT, “</usr/dict/words”);
    - #open /usr/dict/words for reading
  - while ($line = <DICT>) {   # Loop through each line
  -       chomp($line);
  -       print “$line\n”;   # print out the words…
  - }
  - close(DICT);   # close the filehandle

Writing to a File

- open(FH, “>filename”)

- Example
  - open(LOGFILE, “>log.txt”);
    - **#open log.txt for writing**
  - $user = “bart”;
  - print LOGFILE “$user\n”;   # write an entry to the log file…
  - close(LOGFILE);   # close the filehandle

- Opening a file for writing will overwrite the file if it already exists

Appending to a File

- open(FH, “>>filename”)
  - Opening a file for appending will create the file if it does not exists
  - Otherwise, lines are added to the end of the file as they are written

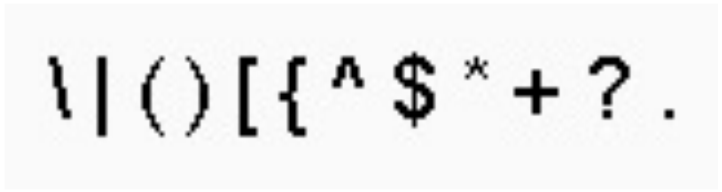
- Example
  - open(LOGFILE, “>>log.txt”); # open log.txt for writing
  - $date = `date`; # get today’s date from system
  - chomp($date);
  - $user = “bart”;
  - print LOGFILE “$date - $user\n “; # adds an entry to the log file…
  - close(LOGFILE); # close the filehandle

backticks

- Backticks (`command`) can be used to execute system commands
- The result can be assigned as a scalar or list
- Example
  - @dirlist = `ls –l`
    - **# get a listing of current directory, ls = `dir` for windows**
  - $num = $#dirlist + 1;
    - **# get the number of lines returned**
  - print “There are $num files in this directory\n”;

Summary: Metacharacters

- For all their power and expressivity, patterns in Perl recognize the same 12 traditional metacharacters



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

- Some simple metacharacters stand by themselves
  - They don't directly affect anything around them
  - . ^ $

Summary: Metacharacters (Continued)

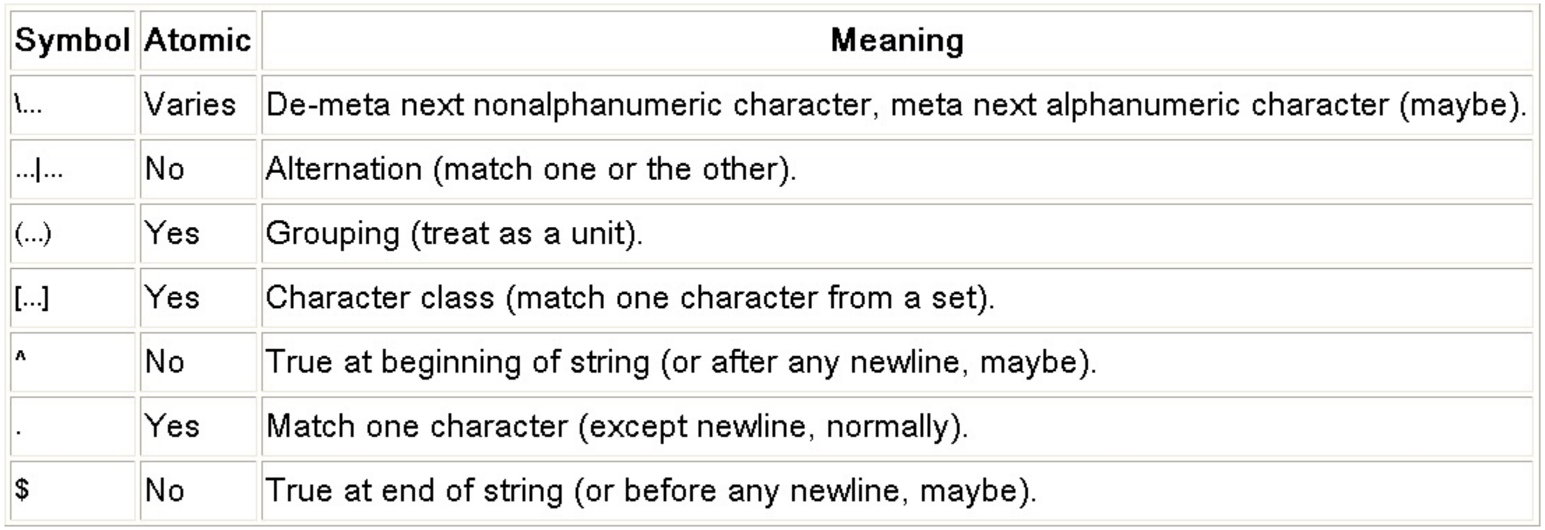
- Some work like postfix operators
  - They govern what immediately precedes them
  - * + ?
- One metacharacter acts like an infix operator
  - It stands between the operands it governs
  - |
- Some work like circumfix operators
  - (...) [...]
  - They govern something contained inside them

Summary: Metacharacters (Continued)

- Backslash disables the others
  - When a backslash precedes a nonalphanumeric character in a Perl pattern, it always makes that next character a literal
  - \. matches a real dot, \$ a real dollar sign, \\ a real backslash, and so on

Summary: Metacharacters (Continued)

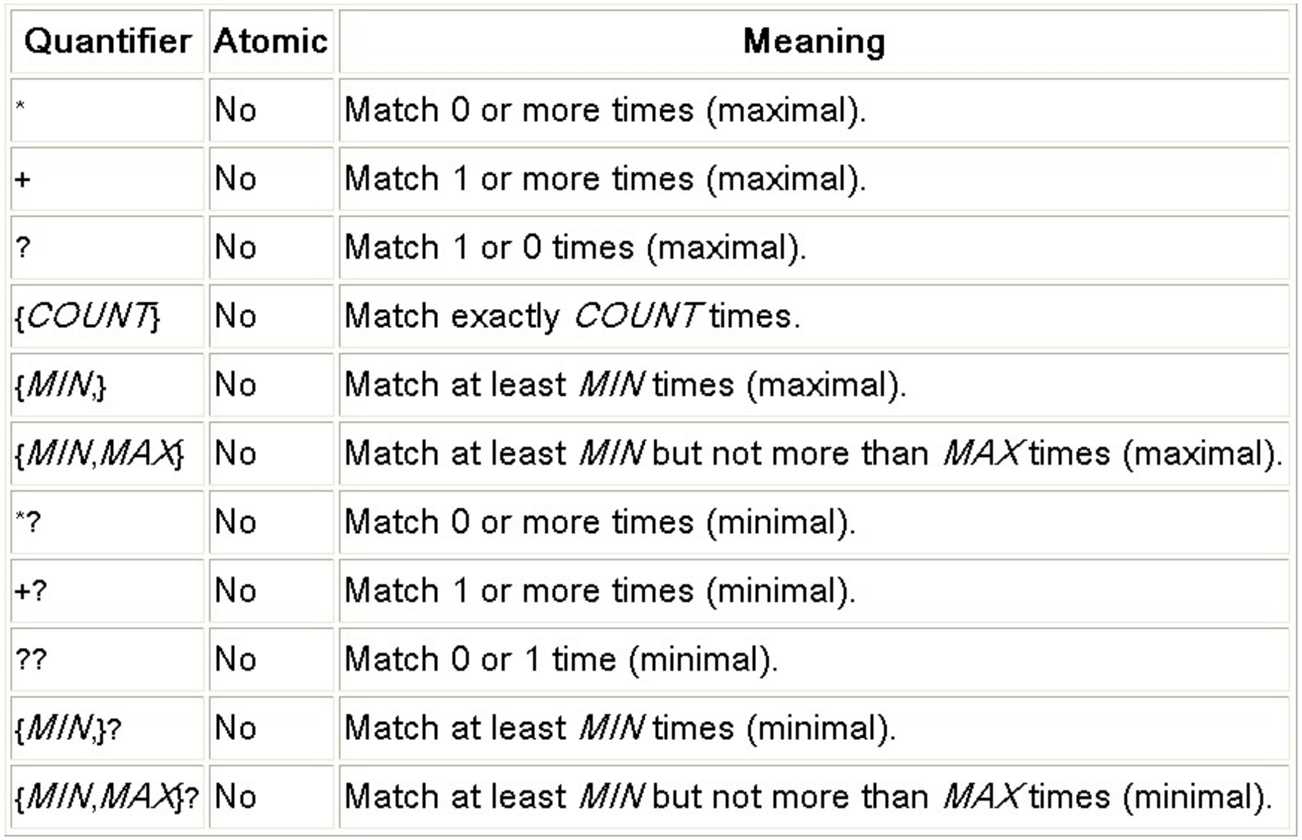
- General Regex Metacharacters



Symbol	Atomic	Meaning
\...	Varies	De-meta next nonalphanumeric character, meta next alphanumeric character (maybe).
...|...	No	Alternation (match one or the other).
(...)	Yes	Grouping (treat as a unit).
[...]	Yes	Character class (match one character from a set).
^	No	True at beginning of string (or after any newline, maybe).
.	Yes	Match one character (except newline, normally).
$	No	True at end of string (or before any newline, maybe).

Summary: Metacharacters (Continued)

- Regex Quantifiers



Quantifier	Atomic	Meaning
*	No	Match 0 or more times (maximal).
+	No	Match 1 or more times (maximal).
?	No	Match 1 or 0 times (maximal).
{*COUNT*}	No	Match exactly *COUNT* times.
{*MIN*,}	No	Match at least *MIN* times (maximal).
{*MIN*,*MAX*}	No	Match at least *MIN* but not more than *MAX* times (maximal).
*?	No	Match 0 or more times (minimal).
+?	No	Match 1 or more times (minimal).
??	No	Match 0 or 1 time (minimal).
{*MIN*,}?	No	Match at least *MIN* times (minimal).
{*MIN*,*MAX*}?	No	Match at least *MIN* but not more than *MAX* times (minimal).