Lekcija 3 - Osnove Perl jezika

Contents

LearningObject

L3: Osnove Perl jezika

Uvod *
Ovo poglavlje predstavlja uvod u osnove Perl jezika.

Perl arrays

Cilj

- Upoznavanja sa konceptom Perl nizova

Perl Arrays

PerlArrays

Perl nizovi

Niz je lista od (obično povezanih) skalara koji se drže zajedno.

Niz počinje sa @ znkom.

Niz je varijabla koja čuva uređenu listu skalarnih vrednosti.

@numbers = (1,2,3); # postaviti vrednosti niza @numbers to (1,2,3)

Da se specificira jedan element niza, tjreba koristiti symbol dolara ($) sa imenom varijable (njen skalar), koji sledi indeks elementa u uglastim zagradama (engl. the subscript operator).

Elementi niza se broje od 0. Negativni indeks broji unazad od zadnjeg elementa u listi (na primmer -1 se odnosi na zadnji element liste).

Example

@date = (8, 24, 70);

$date[2] je vrednost trećeg elementa liste, 70.

Initializing an array

Niz se inicijalizira kreiranjem liste vrednosti koje su odvojene zapetom

Example

my @fruits = ("apples", "oranges", "guavas", "passionfruit", "grapes");

my @magic_numbers = (23, 42, 69);

my @random_scalars = ("mumble", 123.45, "willy the wombat", -300);

As you can see, arrays can contain any kind of scalars. They can have just about any number of elements, too, without needing to know how many before you start. Really any number - tens or hundreds of thousands, if your computer has the memory.

Example

```
#!/usr/bin/perl
print "Content-type: text/html \n\n"; #HTTP HEADER
#DEFINE SOME ARRAYS
@days = ("Monday", "Tuesday", "Wednesday");
@months = ("April", "May", "June");
#PRINT MY ARRAYS TO THE BROWSER
print @days;
print "<br />";
print @months;
```

Display

MondayTuesdayWednesday

AprilMayJune

Reading and changing array values

First of all, Perl's arrays, like those in many other languages, are indexed from zero. We can access individual elements of the array like this:

Example

```
print $fruits[0]; # prints "apples"
print $random_scalars[2]; # prints "willy the wombat"
$fruits[0] = "pineapples"; # changes "apples" to "pineapples"
```

Dodatak

An array is a variable that stores an ordered list of scalar values. Array variables are preceded by an "at" (@) sign. To refer to a single element of an array, you will use the dollar sign ($) with the variable name followed by the index of the element in square brackets.

Here is a simple example of using array variables:

```
#!/usr/bin/perl

@ages = (25, 30, 40);
@names = ("John Paul", "Lisa", "Kumar");

print "\$ages[0] = $ages[0]\n";
print "\$ages[1] = $ages[1]\n";
print "\$ages[2] = $ages[2]\n";
print "\$names[0] = $names[0]\n";
print "\$names[1] = $names[1]\n";
print "\$names[2] = $names[2]\n";
```

Here we used escape sign (\) before $ sign just to print it other Perl will understand it as a variable and will print its value. When exected, this will produce following result:

$ages[0] = 25

$ages[1] = 30

$ages[2] = 40

$names[0] = John Paul

$names[1] = Lisa

$names[2] = Kumar

In Perl, List and Array terms are often used as if they're interchangeable. But the list is the data, and the array is the variable.

Array Creation

Array variables are prefixed with the @ sign and are populated using either parentheses or the qw operator. For example:

@array = (1, 2, 'Hello');

@array = qw/This is an array/;

The second line uses the qw// operator, which returns a list of strings, separating the delimited string by white space. In this example, this leads to a four-element array; the first element is 'this' and last (fourth) is 'array'. This means that you can use different lines as follows:

@days = qw/Monday

Tuesday

...

Sunday/;

You can also populate an array by assigning each value individually as follows:

$array[0] = 'Monday';

...

$array[6] = 'Sunday';

Accessing Array Elements

When accessing individual elements from an array, you must prefix the variable with a dollar sign ($) and then append the element index within square brackets after the name of the variable. For example:

#!/usr/bin/perl

@days = qw/Mon Tue Wed Thu Fri Sat Sun/;

print "$days[0]\n";

print "$days[1]\n";

print "$days[2]\n";

print "$days[6]\n";

print "$days[-1]\n";

```
print "$days[-7]\n";
```

This will produce following result:

```
Mon
Tue
Wed
Sun
Sun
Mon
```

Array indices start from zero, so to access first element you need to give 0 as indices. You can also give a negative index, in which case you select the element from the end, rather than the beginning, of the array. This means that

```
print $days[-1]; # outputs Sun
print $days[-7]; # outputs Mon
```

Sequential Number Arrays

Perl offers a shortcut for sequential numbers and letters. Rather than typing out each element when counting to 100 for example, we can do something like as follows:

```
#!/usr/bin/perl

@var_10 = (1..10);
@var_20 = (10..20);
@var_abc = (a..z);

print "@var_10\n"; # Prints number from 1 to 10
print "@var_20\n"; # Prints number from 10 to 20
print "@var_abc\n"; # Prints number from a to z
```

Here double dot (..) is called range operator. This will produce following result:

```
1 2 3 4 5 6 7 8 9 10
10 11 12 13 14 15 16 17 18 19 20
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

Array Size

The size of an array can be determined using scalar context on the array - the returned value will be the number of elements in the array:

```
@array = (1,2,3);
print "Size: ",scalar @array,"\n";
```

The value returned will always be the physical size of the array, not the number of valid elements. You can demonstrate this, and the difference between scalar @array and $#array, using this fragment as follows:

```
#!/uer/bin/perl

@array = (1,2,3);

$array[50] = 4;

$size = @array;

$max_index = $#array;

print "Size: $size\n";

print "Max Index: $max_index\n";
```

This will produce following result:

```
Size: 51

Max Index: 50
```

There are only four elements in the array that contain information, but the array is 51 elements long, with a highest index of 50.

Adding and Removing Elements in Array

Perl provides a number of useful functions to add and remove elements in an array. You may have a question what is a function? So far you have used print function to print various values. Similarly there are various other functions or sometime called sub-routines which can be used for various other functionalities.

1. push @ARRAY, LIST

Pushes the values of the list onto the end of the array.

2. pop @ARRAY

Pops off and returns the last value of the array.

3. shift @ARRAY

Shifts the first value of the array off and returns it, shortening the array by 1 and moving everything down

4. unshift @ARRAY, LIST

Prepends list to the front of the array, and returns the number of elements in the new array.

Example

```
#!/usr/bin/perl

# create a simple array

@coins = ("Quarter","Dime","Nickel");

print "1. \@coins = @coins\n";

# add one element at the end of the array
```

```
push(@coins, "Penny");
print "2. \@coins = @coins\n";

# add one element at the beginning of the array
unshift(@coins, "Dollar");
print "3. \@coins = @coins\n";

# remove one element from the last of the array.
pop(@coins);
print "4. \@coins = @coins\n";

# remove one element from the beginning of the array.
shift(@coins);
print "5. \@coins = @coins\n";
```

This will produce following result:

```
1. @coins = Quarter Dime Nickel
2. @coins = Quarter Dime Nickel Penny
3. @coins = Dollar Quarter Dime Nickel Penny
4. @coins = Dollar Quarter Dime Nickel
5. @coins = Quarter Dime Nickel
```

Array slices

Cilj

- Upoznavanja sa “array slices” konceptom

Array slices

Array slices

Kriške nizova

Ako želimo da radimo sa delovima nizova , koristimo delove ili “kriške” nizova (engl. array slice).

Primer

```
my @fruits = ("apples", "oranges", "guavas", "passionfruit", "grapes");

@fruits[1,2,3]; # oranges, guavas, passionfruit

@fruits[3,1,2]; # passionfruit, oranges, guavas

my @magic_numbers = (23, 42, 69);

@magic_numbers[0..2]; # 23, 42, 69
```

@magic_numbers[1..5] = (46, 19, 88, 12, 23); # Assigns new magic numbers

You'll notice that these array slices have @ signs in front of them. That's because we're still dealing with a list of things, just one that's (typically) smaller than the full array. It is possible to take an array slice of a single element:

@fruits[1]; # array slice of one element

but this usually means that you've made a mistake and Perl will warn you that what you really should be writing is:

$fruits[1];

You just learnt something new back there: the .. ("dot dot") range operator creates a temporary list of numbers between the two you specify. In our case we specified 0 and 2 (then 1 and 5), but it could have been 1 to 100, or 30 to 70, if we'd had an array big enough to use it on. You'll run into this operator again and again.

The previous example will print only values for element 0, 4 and 7. Note that in perl first value of an array is number 0.

That is fine but how do we find the number of elements of an array?

print "Number of elements of an array: $#array1";

Note that $#array1 in our example is number of elements, but because elements from an array in

Perl starts with value 0, the real number of elements of an array is $#array + 1.

qw function

The Perl qw function (quote word) uses embedded whitespace separator to split an expression passed to it into a list of elements and returns that list.

Example

array = ("sweet", "bitter", "sour", "salty", "hot"); # or

@array = qw/sweet bitter sour salty hot/;

There is another method to define an array:

@array2 = qw(Value1 Value2 Value3 Value4);

Perl functions for working with arrays:

pop - remove last element of an array:

push - add an element to the end of array;

shift - removes first element of an array;

unshift - add an element to the beginning of array;

sort - sort an array

Pop Function

Uklanja zadnji element niza

Example

```
#!/usr/bin/perl -w
@array1 = ("Data1", "Data2", "Data3");
print "Array1 values: @array1[0..$#array1]\n";
pop @array1;
print "Array1 after applying pop function: @array1[0..$#array1]\n";
```

Push Function

Dodaje element na kraj niza.

Example

```
#!/usr/bin/perl -w
@array1 = ("Data1", "Data2", "Data3");
print "Array1 values: @array1[0..$#array1]\n";
push @array1, "Data4";
print "Array1 after applying push function: @array1[0..$#array1]\n";
```

Shift Function

Uklanja prvi element niza

Example

```
#!/usr/bin/perl -w
@array1 = ("Data1", "Data2", "Data3");
print "Array1 values: @array1[0..$#array1]\n";
shift @array1;
print "Array1 after applying shift function: @array1[0..$#array1]\n";
```

Array interpolation

Cilj

- Upoznavanja sa interpolacijom nizova

Array interpolation

Array interpolation

Može se pored ostalog i ubaciti string u niz, slično kao kod skalara.

Primer

```
my @fruits = ("apples", "oranges", "guavas", "passionfruit", "grapes");

print "My favourite fruits are @fruits\n"; # whole array
print "Two types of fruit are @fruits[0,2]\n"; # array slice
print "My favourite fruit is $fruits[3]\n"; # single element
```

print "@fruits\n"; # whole array

say "@fruits"; # the same

say @fruits; # not the same

Brojanje unatrag (engl. counting backwards)

Moguće je takođe brojati unatrag, sa kraja na početak:

Primer

$fruits [-1]; # Last fruit in the array, grapes in this case.

$fruits[-3]; # Third last fruit: guavas.

Finding out the size of an array

Cilj

- Upoznavanja sa načinom nalaženja veličine niza

Finding out the size of an array

Finding out the size of an array

To find out the number of elements in an array, there’s a very easy solution. If you evaluate the array in a scalar context (that is, treat the array as if it were a scalar) Perl will give you the only scalar value that makes sense: the number of elements in the array.

my $fruit_count = @fruits; # 5 elements in @fruits

There’s a more explicit way to do it as well --- scalar @fruits and int @fruits will also tell us how many elements there are in the array. Both of these functions force a scalar context, so they’re really using the same mechanism as the $fruit_count example above.We’ll talk more about contexts soon.

say "There are ", scalar(@fruits), " in my list of fruits.";

There’s also an ugly special syntax that gives us the last index used in an array. This looks like

$#array.

my $last_index_used = $#fruits; # index of last element

If you print either $last_index_used or $#fruits you’ll find the value is one less than the size of the array (4 instead of 5). This is because it is the index of the last element and the index starts at zero, so you have to add one to it to find out how many elements are in the array.

my $number_of_fruits = $#fruits + 1;

If the array is empty, $#fruits returns -1.

$#fruits is easily confused with #$fruits (a comment!), and it can often cause off-by-one errors and other bugs.

Thus it is generally best avoided.

Using qw to populate arrays

Cilj

- Upoznavanja sa qw funkcijom za ispunjavanje nizova

Using qw to populate arrays

Using qw// to populate arrays

If you’re working with lists and arrays a lot, you might find that it gets very tiresome to be typing so many quotes and commas. Let’s take our fruit example:

my @fruits = ("apples", "oranges", "guavas", "passionfruit", "grapes");

We had to type the quotes character ten times, along with four commas, and that was only for a short list. If your list is longer, such as all the months in a year, then you’ll need even more punctuation to make it all work. It’s easy to forget a quote, or use the wrong quote, or misplace a comma, and end up with a trivial but bothersome error. Wouldn’t it be nice if there was a better way to create a list?

Well, there is a better way, using the qw// operator. qw//stands for quote words. It takes whitespace separated words and turns them into a list, saving you from having to worry about all that tiresome punctuation. Here’s our fruit example again using qw//:

my @fruits = qw/apples oranges guavas passionfruit grapes/;

As you can see, this is clear, concise, and difficult to get wrong. And it keeps getting better. Your list can stretch over multiple lines, and your delimiter doesn’t need to be a slash. Whatever punctuation character that you place after the qw becomes the delimiter. So if you prefer parentheses overslashes, that’s no problem at all:

my @months = qw(January February March April May June July AugustSeptember October November December);

Example

#!/usr/bin/perl

use strict;

use warnings;

use File::Temp qw(tempfile tempdir);

Example

my @names = ('Kernighan', 'Ritchie', 'Pike');

my @names = qw(Kernighan Ritchie Pike);

Example

@names = qw(Kernighan Ritchie Pike);

@names = qw/Kernighan Ritchie Pike/;

@names = qw'Kernighan Ritchie Pike';

@names = qw{Kernighan Ritchie Pike};

Example

```
#!/usr/bin/perl
use strict;
use warnings;

my $developer = 'Thompson';
my @names = qw(Kernighan Ritchie Pike $developer);
foreach my $name (@names) {
print "name: $name\n";
}
```

Display

```
name: Kernighan
name: Ritchie
name: Pike
name: $developer
```

Printing out the values in an array

Cilj

- Upoznavanja sa načinom štampanja vrijednosti nekog niza

Printing out the values in an array

Printing out the values in an array

Štampanje vrijednosti u nizu

Štampanje svih vrednosti niza se može uraditi na tri načina

Example

```
my @fruits = qw/apples oranges guavas passionfruit grapes/;
print @fruits; # prints "applesorangesguavaspassionfruitgrapes"
print join("|", @fruits); # prints "apples|oranges|guavas|passionfruit|grapes"
print "@fruits"; # prints "apples oranges guavas passionfruit grapes"
```

The first method takes advantage of the fact that print takes a list of arguments to print and prints them out sequentially.

The second uses join() which joins an array or list together by separating each element with the first argument

The third option uses double quote interpolation and a little bit of Perl magic to pick which character(s) to separate the words with (usually a space character).

Formatted output with printf

Cilj

- Upoznavanja sa načinom formatiranja izlaza sa printf funkcijom

Formatted output with printf

Formatted output with printf

The printf operator takes a format string followed by a list of things to print. The format string is a fill-in-the-blanks template showing the desired form of the output:

printf("foo is %d", $decimal);

Example

printf "Hello, %s; your password expires in %d days!\n",

$user, $days_to_die;

The format string holds a number of so-called conversions; each conversion begins with a percent sign (%) and ends with a letter.

There should be the same number of items in the following list as there are conversions; if these don’t match up, it won’t work correctly. In the example above, there are two items and two conversions, so the output might look something like this:

Hello, merlyn; your password expires in 3 days!

To print a number in what’s generally a good way, use %g, which automatically chooses floating-point, integer, or even exponential notation, as needed:

printf "%g %g %g\n", 5/2, 51/17, 51 ** 17; # 2.5 3 1.0683e+29

The %d format means a decimal integer, truncated as needed:

printf "in %d days!\n", 17.85; # in 17 days!

Note that this is truncated, not rounded; you’ll see how to round off a number in a moment.

In Perl, you most often use printf for columnar data, since most formats accept a field width. If the data won’t fit, the field will generally be expanded as needed:

printf "%6d\n", 42; # output like ````42 (the ` symbol stands for a space)

printf "%2d\n", 2e3 + 1.95; # 2001

The %s conversion means a string, so it effectively interpolates the given value as a string, but with a given field width:

printf "%10s\n", "wilma"; # looks like `````wilma

A negative field width is left-justified (in any of these conversions):

printf "%-15s\n", "flintstone"; # looks like flintstone`````

The %f conversion (floating-point) rounds off its output as needed, and even lets you request a certain number of digits after the decimal point:

printf "%12f\n", 6 * 7 + 2/3; # looks like ```42.666667

printf "%12.3f\n", 6 * 7 + 2/3; # looks like ``````42.667

printf "%12.0f\n", 6 * 7 + 2/3; # looks like ``````````43

To print a real percent sign, use %%, which is special in that it uses no element from the list: [143]

printf "Monthly interest rate: %.2f%%\n", 5.25/12; # the value looks like "0.44%"

Arrays and printf

Cilj

- Upoznavanja sa nizovima i funkcijom printf

Arrays and print

Arrays and printf

my @items = qw(wilma dino pebbles);

my $format = "The items are:\n" . ("%10s\n" x @items);

print "the format is >>$format<<\n"; # for debugging

printf $format, @items;

Ovdje se koristi operator x za replikaciju datog stringa određeni broj puta što je dato preko @items (koristi se u skalarnom kontekstu).

Ovdje je to 3, jer postoje tri elementaima što je isto kao da bi napisali;

"The items are:\n%10s\n%10s\n%10s\n".

U rezultatu, svaki element se štampa u novom redu, udesno podešeno u koloni sa deset karaktera

Bolje rešenje:

printf "The items are:\n".("%10s\n" x @items), @items;

Primetiti da se @items jednom koristi u skalarnom kontekstu da se dobije njegova dužina i još jednom u kontekstu liste da se dobije njen sadržaj.

Jasno je da jeovdje kontekst jako važan.

Using the Perl qw() function

The 'quote word' function qw()is used to generate a list of words. If takes a string such as:

tempfile tempdir

Array variables

Cilj

- Upoznavanja sa konceptom varijabli nizova

Array variables

Array variables

Varijable nizova

Niz sadrži listu sklarnih veličina podataka (pojedinačni elementi).Lista sadrži neograničen broj elemenata. U Perlu, nizovi su definisa preko simbola @.

@food = ("apples", "pears", "eels");

@music = ("whistle", "flute");

Nizu se pristupa preko indeksa (engl. indices) koji počinju od 0, uglaste zagrade se koriste za definiciju indeksa. Izraz

$food[2]

vraća eels. Primetiti da se @ menja u $ jer je eels skalar.

Example: Perl array variables

#!/usr/bin/perl

print "Content-type: text/html \n\n"; #HTTP HEADER

#define some arrays

@days = ("Monday", "Tuesday", "Wednesday");

@months = ("April", "May", "June");

#print my arrays to the browser

print @days;

print "
";

print @months;

Display:

MondayTuesdayWednesday

AprilMayJune

Displaying arrays

Since context is important, it shouldn't be too surprising that the following all produce different results:

print @food; # by itself

print "@food"; # embedded in double quotes

print @food.""; # in a scalar context

Context

Cilj

- Upoznavanja sa Context terminom

Context

Context

Kontekst

Kontekst se odnosi na način kako se neki izraz ili varijabla izvršava u Perlu. POstoje dva glavna konteksta:

skalarni kontekst

kontest liste

Skalarne varijable se uvek ocenjuju u skalarnom kontekstu dok se nizovi i heševi ocenjuju u oba konteksta, u sklaranom kada se tretiraju kao skalari i u kontekstu liste kada se tretiraju kao nizovii heševi.

Example

```
my @newarray = @array; # list context
my $howmany = @array; # scalar context
my $howmany2 = scalar(@array); # scalar context again (explicitly)
```

If you look at an array in a scalar context, you'll see how many elements it has; if you look at it in list context, you'll see the contents of the array itself.

Many things in Perl are very specific about which context they require and will force lists into scalar context when required. For example the + (plus or addition) operator expects that its two arguments will be scalars. Hence:

Example

```
my @a = (1,2,3);
my @b = (4,5,6,7);
print @a + @b;@a + @b;
```

will print 7 (the sum of the two list's lengths) rather than 5 7 9 7 (the individual sums of the list elements). Many things in Perl have different behaviours depending upon whether or not they're in an array or scalar context. This is generally considered a good thing, as it means things can have a "Do What I Mean" (DWIM) behaviour depending upon how they are used.

Arrays are the most common example of this, but we'll see some more as we progress through the course. There's also a third type of context, the null context, where the result of an operation is just thrown away. This usually isn't discussed, because by its very definition we don't care about what result is returned.

Variable Context

Cilj

- Upoznavanja sa konceptom promenljivog konteksta

Variable Context

Variable Context

PERL treats same variable differently based on Context.

For example

my @animals = ("camel", "llama", "owl");

Here @animals is an array, but when it is used in scalar context then it returns number of elements contained in it as following.

if (@animals < 5) { ... } # Here @animals will return 3

Example

my $number = 30;

Here $number is an scalar and contained number in it but when it is called along with a string then it becomes number which is 0, instead of string:

$result = "This is " + "$number";

print "$result";

Here output will be 30.

Context

Cilj

- Upoznavanja sa Context terminom

Context

Context

Kontekst

Kontekst se odnosi na način kako se neki izraz ili varijabla izvršava u Perlu. POstoje dva glavna konteksta:

skalarni kontekst

kontest liste

Skalarne varijable se uvek ocenjuju u skalarnom kontekstu dok se nizovi i heševi ocenjuju u oba konteksta, u sklaranom kada se tretiraju kao skalari i u kontekstu liste kada se tretiraju kao nizovii heševi.

Example

my @newarray = @array; # list context

my $howmany = @array; # scalar context

my $howmany2 = scalar(@array); # scalar context again (explicitly)

If you look at an array in a scalar context, you'll see how many elements it has; if you look at it in list context, you'll see the contents of the array itself.

Many things in Perl are very specific about which context they require and will force lists into scalar context when required. For example the + (plus or addition) operator expects that its two arguments will be scalars. Hence:

Example

my @a = (1,2,3);

my @b = (4,5,6,7);

print @a + @b;@a + @b;

will print 7 (the sum of the two list's lengths) rather than 5 7 9 7 (the individual sums of the list elements). Many things in Perl have different behaviours depending upon whether or not they're in an array or scalar context. This is generally considered a good thing, as it means things can have a "Do What I Mean" (DWIM) behaviour depending upon how they are used.

Arrays are the most common example of this, but we'll see some more as we progress through the course. There's also a third type of context, the null context, where the result of an operation is just thrown away. This usually isn't discussed, because by its very definition we don't care about what result is returned.

Variable Context

Cilj

- Upoznavanja sa konceptom promenljivog konteksta

Variable Context

Variable Context

PERL treats same variable differently based on Context.

For example

my @animals = ("camel", "llama", "owl");

Here @animals is an array, but when it is used in scalar context then it returns number of elements contained in it as following.

if (@animals < 5) { ... } # Here @animals will return 3

Example

my $number = 30;

Here $number is an scalar and contained number in it but when it is called along with a string then it becomes number which is 0, instead of string:

$result = "This is " + "$number";

print "$result";

Here output will be 30.

The difference between a list and an array

Cilj

- Upoznavanja sa razlikama između Perl liste i Perl niza

The difference between a list and an array

The difference between a list and an array

Razlikama između Perl liste i niza

Nisu velike, Lista je niz bez imena,

Example

```
# printing a list of scalars
print ("Hello", " ", $name, "\n");

# printing an array
my @hello = ("Hello", " ", $name, "\n");
print @hello;
```

If you come across something that wants a LIST, you can either give it the elements of list as in the first example above, or you can pass it an array by name. If you come across something that wants an ARRAY, you have to actually give it the name of an array. Examples of functions which insist on wanting an ARRAY are push() and pop(), which can be used for adding and removing elements from the end of an array.

An array

Example

```
my @array2 = ( "1", "2", "3");

for $elem (@array2) {
print $elem."\n";
}
```

Display

1

2

3

List example

```
my @array = [ "1", "2", "3"];

for $elem (@array) {
```

print $elem."\n";

}

Display

ARRAY(0x9c90818)

```
Example
use strict;

my @array = ('foo', 'bar', 'baz');
my $temp;

$temp = @array;
print "$temp\n";
$temp = ('foo', 'bar', 'baz');
print "$temp\n";
($temp) = @array;
print "$temp\n";
($temp) = @array[0..2];
print "$temp\n";
$temp = @array[0..2];
print "$temp\n";
```

```
Display
3
baz
foo
foo
baz
```

Example

```
#!/usr/local/bin/perl
@array = (1, "chicken", 1.23, "\"Having fun?\"", 9.33e+23);
$count = 1;
while ($count <= 5) {

print ("element $count is $array[$count-1]\n");

$count++;
}
```

Display

```
$ program5_1
```

element 1 is 1

element 2 is chicken

element 3 is 1.23

element 4 is "Having fun?"

element 5 is 9.3300000000000005+e23

$

Zadatak 1

Create an array of your friends’ names. (You’re encouraged to use the qw() operator.)

Print out the first element.

Print out the last element.

Print the array within a double-quoted string, ie: print "@friends"; and notice how Perl handles this.

Print out an array slice of the 2nd to 4th items within a double-quoted string (variable interpolation).

Replace every second friend with another friend.

Zadatak 2

Write a print statement to print out your email address. How can you handle the @ when you’re using double quotes?

Operators

Cilj

- Upoznavanja sa Perl operatorima

Operators

Operators

The following figure lists all the Perl operators from highest to lowest precedence and indicates their associativity.

Associativity	Operators
Left	Terms and list operators (leftward)
Left	-> (method call, dereference)
Nonassociative	++ -- (autoincrement, autodecrement)
Right	** (exponentiation)
Right	! ~ \ and unary + and - (logical not, bit-not, reference, unary plus, unary minus)
Left	=~ !~ (matches, doesn't match)
Left	* / % x (multiply, divide, modulus, string replicate)
Left	+ - . (addition, subtraction, string concatenation)
Left	<< >> (left bit-shift, right bit-shift)
Nonassociative	Named unary operators and file-test operators
Nonassociative	< > <= >= lt gt le ge (less than, greater than, less than or equal to, greater than or equal to, and their string equivalents.
Nonassociative	== != <=> eq ne cmp (equal to, not equal to, signed comparison, and their string equivalents)
Left	& (bit-and)
Left	| ^ (bit-or, bit-xor)
Left	&& (logical AND)
Left	|| (logical OR)
Nonassociative	.. ... (range)
Right	?: (ternary conditional)
Right	= += -= *= and so on (assignment operators)
Left	, => (comma, arrow comma)
Nonassociative	List operators (rightward)
Right	not (logical not)
Left	and (logical and)
Left	or xor (logical or, xor)

Slika 1 Perl associativity and Operators, Listed by Precedence

You can make your expressions clear by using parentheses to group any part of an expression. Anything in parentheses will be evaluated as a single unit within a larger expression.

With very few exceptions, Perl operators act upon scalar values only, not upon list values.

Terms that take highest precedence in Perl include variables, quote and quotelike operators, any expression in parentheses, and any function whose arguments are in parentheses.

A list operator is a function that can take a list of values as its argument. List operators take highest precedence when considering what's to the left of them. They have considerably lower precedence when looking at their right side, which is the expected result.

Also parsed as high-precedence terms are the do{} and eval{} constructs, as well as subroutine and method calls, the anonymous array and hash composers ([] and {}), and the anonymous subroutine composer sub{} .

A unary operator is a function that takes a single scalar value as its argument. Unary operators have a lower precedence than list operators because they only expect and take one value.

Sort Operator

Cilj

- Upoznavanja sa sort operatorom

The sort Operator

The sort Operator

Sort operator

Sort operator uzima listu vrijednosti (koje mogu doći iz niza) i sortira ih u interni “character ordering“ .

Sort operator uzima kao ulaz listu, sortira je i daje kao izlaz novu listu.

@rocks = qw/ bedrock slate rubble granite /;

@sorted = sort(@rocks); # gets bedrock, granite, rubble, slate

@back = reverse sort @rocks; # these go from slate to bedrock

@rocks = sort @rocks; # puts sorted result back into @rocks

@numbers = sort 97..102; # gets 100, 101, 102, 97, 98, 99

print $numbers[2];

If you want to sort an array, you must store the result back into that array:

sort @rocks; # WRONG, doesn't modify @rocks

@rocks = sort @rocks; # Now the rock collection is in order

Arrow operator

Cilj

- Upoznavanja sa arrow operatorom

Arrow operator

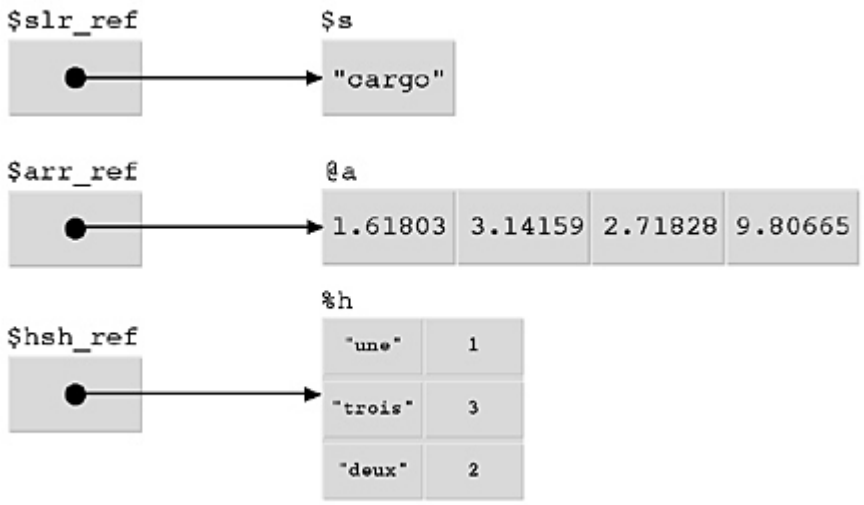
Arrow operator

Strelica operator à

Dereferetni operator (engl. dereference operator or indirection operator)

Dereferetni operator ili indirektni operator, pokazivač na varijable. Pokazuje na adresu varijable i vraća vrednost koja se nalazi na toj adresi, To se zove dereferenciraje pokazivača ili pointera.

Strelica operator je dereferetni operator. Može se koristiti za referenciranje nizova, heševa ili za pozivanje metoda objekata. Slika 1 prikazuje princip referenciranja.



Slika 1 Referenciranje

Example

```
#!/usr/bin/perl
use strict;
use warnings;

my $scalar = "This is a scalar";
my $scalar_ref = \$scalar;

print "Reference: " . $scalar_ref . "\n";
print "Dereferenced: " . $$scalar_ref . "\n";
```

Display

```
Reference: SCALAR <0x420c3c>
Dereferenced: This is a scalar
```

Example

```
#!/usr/bin/perl
use strict;
use warnings;

my $array_ref = ['apple', 'banana', 'orange'];
my @array = @$array_ref;

print "Reference: $array_ref\n";
print "Dereferenced: @array\n";
```

Display

Reference: ARRAY(0x80f6c6c)

Dereferenced: apple banana orange

Arrow operator

```
my @array1 = (1, 2, 3, 'four');
my $reference1 = \@array1;
my @array2 = ('one', 'two', 'three', 4);
my $reference2 = \@array2;

my @array = ($reference1, $reference2);

# this refers to the first item of the first array:
print "$array[0]->[0]\n";
```

Display

1

Example

```
my @array = ([1, 2, 3, 'four'], ['one', 'two', 'three', 4]);
print $array[0][0];
```

Display

Unary operator

Cilj

- Upoznavanje karakteristika unarnog operatora

Unary operator

Unarni operator (engl. unary operator)

Unarni operator ! izvršava logičku negaciju ili "NOT." Obična NOT operacija ima niži prioritet od ! operacije.

Unary -performs arithmetic negation if the operand is numeric. If the operand is an identifier, then a string consisting of a minus sign concatenated with the identifier is returned. Otherwise, if the string starts with a plus or minus, a string starting with the opposite sign is returned.

Unary ~performs bitwise negation, that is, one's complement.

For example, on a 32-bit machine, ~0xFF is 0xFFFFFF00 . If the argument to ~ is a string instead of a number, a string of identical length is returned, but with all the bits of the string complemented.

Unary + has no semantic effect whatsoever, even on strings. It is syntactically useful for separating a function name from a parenthesized expression that would otherwise be interpreted as the complete list of function arguments.

Unary \ creates a reference to whatever follows it. Do not confuse this behavior with the behavior of backslash within a string. The \ operator may also be used on a parenthesized list value in a list context, in which case it returns references to each element of the list.

Slika 1 ilustruje unarne operatore.

Operator	Description
Changing the sign of `op1`	
`+op1`	Positive operand
`-op1`	Negative operand
Changing the value of `op1` *before usage*	
`++op1`	Pre-increment operand by one
`--op1`	Pre-decrement operand by one
Changing the value of `op1` *after usage*	
`op1++`	Post-increment operand by one
`op1--`	Post-decrement operand by one

Slika 1 Unarni operatori

Arithmetic operators

Cilj

- Upoznavanja sa Pel arithmetičkim operatoratorima

Arithmetic operators

Arithmetic operators

Aritmetički operator

Binarno ** je eksponencijalni operator. Imati na umu da se veže još čvršće nego unarni minus pa

-2 ** 4 je - (2 ** 4), a ne (-2) ** 4.

Imajte na umu da ** ima pravo asocijativnosti, tako da se $e = 2 ** 3 ** 4; izražava kao 2^{81}, ne $8^{4.}$

Operatori za množenje (*) i delenje (/) se izvršavaju kako se i očekuje, množeći ili deleći dva operanda. Delenje se izvršava u načinu rada sa pomičnom tačkom (engl. floating-point mode), sem ako se ne aktivira rad sa celim brojevima (engl. integer mode) korišćenjem integera.

Modul operator (%) konvertuje operand u cijele brojeve prije izračunavanja ostatka u skladu sa operacijom delenja celog broja. U “floating-point mode”, može se koristiti fmod() funkcija iz POSIX modula.

Slika 1 daje aritmetičke operatore.

Operator	Example	Result	Definition
+	7 + 7	= 14	Addition
-	7 - 7	= 0	Subtraction
*	7 * 7	= 49	Multiplication
/	7 / 7	= 1	Division
**	7 ** 7	= 823543	Exponents
%	7 % 7	= 0	Modulus

Slika 1 Aritmetički operatori

Slika 2 ilustruje Perl aritmetiku, a Slika 3 Perl operatore pridruživanja.

```
#!/usr/bin/perl

print "content-type: text/html \n\n";   #HTTP Header

#PICK A NUMBER
$x = 81;
$add = $x + 9;
$sub = $x - 9;
$mul = $x * 10;
$div = $x / 9;
$exp = $x ** 5;
$mod = $x % 79;
print "$x plus 9 is $add<br />";
print "$x minus 9 is $sub<br />";
print "$x times 10 is $mul<br />";
print "$x divided by 9 is $div<br />";
print "$x to the 5th is $exp<br />";
print "$x modulus 79 is $mod<br />";
```

Slika 2 Perl aritmetika

Operator	Definition	Example
+=	Addition	($x += 10)
-=	Subtraction	($x -= 10)
*=	Multiplication	($x *= 10)
/=	Division	($x /= 10)
%=	Modulus	($x %= 10)
**=	Exponent	($x **= 10)

Slika 3 Perl operatori pridruživanja

Slika 4 daje primer Perl programa.

```
#START WITH A NUMBER
$x = 5;
print '$x plus 10 is '.($x += 10);
print "<br />x is now ".$x;       #ADD 10
print '<br />$x minus 3 is '.($x -= 3);
print "<br />x is now ".$x;       #SUBTRACT 3
print '<br />$x times 10 is '.($x *= 10);
print "<br />x is now ".$x.       #MULTIPLY BY 10
print '<br />$x divided by 10 is '.($x /= 10);
print "<br />x is now ".$x;       #DIVIDE BY 10
print '<br />Modulus of $x mod 10 is '.($x %= 10);
print "<br />x is now ".$x;       #MODULUS
print '<br />$x to the tenth power is '.($x **= 10);
print "<br />x is now ".$x;       #2 to the 10th
```

Slika 4 Primer Perl programa

Dodatak

Perl language supports many operator types but following is a list of important and most frequently used operators:

Arithmetic Operators

Equality Operators

Logical Operators

Assignment Operators

Bitwise Operators

Logical Operators

Quote-like Operators

Miscellaneous Operators

Relational operators

Cilj

- Upoznavanja sa relacionim operatorima

Relational operators

Perl has two classes of relational operators. One class operates on numeric values, and the other operates on string values. String comparisons are based on the ASCII collating sequence. Relational operators are nonassociative, so $a < $b < $ is a syntax error.

Slika 1 Relacioni operatori

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

Slika 2 Primer

Operator	Example	Defined	Result
==,eq	5 == 5 5 eq 5	Test: Is 5 equal to 5?	True
!=,ne	7 != 2 7 ne 2	Test: Is 7 not equal to 2?	True
<,lt	7 < 4 7 lt 4	Test: Is 7 less than 4?	False
>,gt	7 > 4 7 gt 4	Test: Is 7 greater than 4?	True
<=,le	7 <= 11 7 le 11	Test: Is 7 less than or equal to 11?	True
>=,ge	7 >= 11 7 ge 11	Test: Is 7 greater than or equal to 11?	False

Equality operators

Cilj

- Upoznavanja sa operatorima jednakosti

Equality operators

Equality operators

Operatori jednakosti

Operatori jednakosti i nejednakosti (Slika 1) vraćaju 1 z za istinit iskazt i "" za neistinit iskaz (baš kao što rade relacioni operatori).

Operatori <=> i cmp vraćaju -1 ako je levi operand manji od desnog operanda, 0 ako su isti i +1 ako je levi operand veći od desnog

Numeric	String	Meaning
==	eq	Equal to
!=	ne	Not equal to
<=>	cmp	Comparison, with signed result

Slika 1 Operatori jednakosti

Autoincrement and autodecrement

If placed before a variable, the ++ and -- operators increment or decrement the variable before returning the value, and if placed after, they increment or decrement the variable after returning the value.

Assignment operators

Cilj

- Upoznavanja sa operatorima dodeljivanja

Assignment operators

Assignment operators

Operatori dodeljivanja

Perl posjeduje sledeće operatore dodeljivanja za dodelu vrednosti jednoj varijabli.

Operator	Definition	Example
+=	Addition	($x += 10)
-=	Subtraction	($x -= 10)
*=	Multiplication	($x *= 10)
/=	Division	($x /= 10)
%=	Modulus	($x %= 10)
**=	Exponent	($x **= 10)

Slika 1 Operatori dodeljivanja

Each operator requires a variable on the left side and some expression on the right side.

For the simple assignment operator, = , the value of the expression is stored into the designated variable.

For the other operators, Perl evaluates the expression:

$var *OP* = $value

as if it were written:

$var = $var *OP* $value

except that $var is evaluated only once. For example:

$a += 2; # same as $a = $a + 2

```
#!/usr/bin/perl

print "content-type: text/html \n\n";   #HTTP HEADER

#START WITH A NUMBER
$x = 5;
print '$x plus 10 is '.($x += 10);
print "<br />x is now ".$x;       #ADD 10
print '<br />$x minus 3 is '.($x -= 3);
print "<br />x is now ".$x;       #SUBTRACT 3
print '<br />$x times 10 is '.($x *= 10);
print "<br />x is now ".$x.       #MULTIPLY BY 10
print '<br />$x divided by 10 is '.($x /= 10);
print "<br />x is now ".$x;       #DIVIDE BY 10
print '<br />Modulus of $x mod 10 is '.($x %= 10);
print "<br />x is now ".$x;       #MODULUS
print '<br />$x to the tenth power is '.($x **= 10);
print "<br />x is now ".$x;       #2 to the 10th
```

Slika 2 Perl dodeljivanje

Pattern match operators

Cilj

- Upoznavanja sa operatorima podudaranja šablona

Pattern match operators

Pattern match operators

Binary =~ binds a scalar expression to a pattern match, substitution, or translation. These operations search or modify the string $_ by default.

Binary !~ is just like =~ except the return value is negated in the logical sense. The following expressions are functionally equivalent:

$string !~ /

pattern

/

not $string =~ /

pattern

/

Dodatak

Special pattern matching character operators:

\ Quote the next metacharacter

^ Match the beginning of the line

. Match any character (except newline)

$ Match the end of the line (or before newline at the end)

| Alternation

() Grouping

[] Character class

The simplest and very common pattern matching character operators is the .

This simply allows for any single character to match where a . is placed in a regular expression.

For example /b.t/ can match to bat, bit, but or anything like bbt, bct ....

Square brackets ([..]) allow for any one of the letters listed inside the brackets to be matched at the specified position.

For example /b[aiu]t/ can only match to bat, bit or but.

You can specify a range inside[..]. For example (regex.pl):

[012345679] # any single digit

[0-9] # also any single digit

[a-z] # any single lower case letter

[a-zA-Z] # any single letter

[0-9\-] # 0-9 plus minus character

Some Simple Examples

fa*t matches to ft, fat, faat, faaat *etc*

(.*) can be used a *wild card* match for any number (zero or more) of any characters.

Thus f.*k matches to fk, fak, fork, flunk, *etc.*

fa+t matches to fat, faat, faaat *etc*

.+ can be used to match to one or more of any character *i.e.* at least something must be there.

Thus f.+k matches to fak, fork, flunk, *etc.* but not fk.

? matches to zero or one character.

Thus ba?t matches to bt or bat.

b.?t matches to bt, bat, bbt, *etc.* but not bunt or higher than four-letter words.

ba{3}t} only matches to baaat.

ba{1,4} matches to bat, baat, baaat and baaaat

Because patterns are processed as double quoted strings, the following also work:

\t tab (HT, TAB)

\n newline (LF, NL)

\r return (CR)

\f form feed (FF)

\a alarm (bell) (BEL)

\e escape (think troff) (ESC)

\033 octal char (think of a PDP-11)

\x1B hex char

\c[control char

\l lowercase next char (think vi)

\u uppercase next char (think vi)

\L lowercase till \E (think vi)

\U uppercase till \E (think vi)

\E end case modification (think vi)

\Q quote regular expression metacharacters till \E

File test operators

Cilj

- Upoznavanja sa operatorima za testiranje datoteka

File test operators

File test operators

Operatori za testiranje datoteka

Ova tip operatora je unarni operator koji testira ime datoteke ili ručku datoteke (engl. filehandle), Slika 1.

Operator	Meaning
-r	File is readable by effective uid/gid.
-w	File is writable by effective uid/gid.
-x	File is executable by effective uid/gid.
-o	File is owned by effective uid.
-R	File is readable by real uid/gid.
-W	File is writable by real uid/gid.
-X	File is executable by real uid/gid.
-O	File is owned by real uid.
-e	File exists.
-z	File has zero size.
-s	File has non-zero size (returns size).
-f	File is a plain file.
-d	File is a directory.
-l	File is a symbolic link.
-S	File is a socket.
-b	File is a block special file.
-c	File is a character special file.
-t	Filehandle is opened to a tty.
-u	File has setuid bit set.
-g	File has setgid bit set.
-k	File has sticky bit set.
-T	File is a text file.
-B	File is a binary file (opposite of -T).
-M	Age of file (at startup) in days since modification.
-A	Age of file (at startup) in days since last access.
-C	Age of file (at startup) in days since inode change.

Slika 1 Operatori za testiranje datoteka

Logical operators

Cilj

- Upoznavanja sa Perl logičkim operatorima

Logical operators

Logical operators

Logički operatori

Perl posjeduje && (logical AND) i || (logical OR) operator. Oni ocenjuju sa leva na de
izjava.

Example	Name	Result
$a && $b	And	$a if $a is false, $b otherwise
$a || $b	Or	$a if $a is true, $b otherwise

Slika 1 Logički operatori

For example, an oft-appearing idiom in Perl programs is:

open(FILE, "somefile") || die "Cannot open somefile: $!\n";

In this case, Perl first evaluates the open function. If the value is true (because somefile was successfully opened), the execution of the die function is unnecessary and is skipped.

Perl also provides lower-precedence and and or operators that are more readable.

Bitwise operators

Perl has bitwise AND, OR, and XOR (exclusive OR) operators:

& , | , and ^ .

Slika 2 Bitwise operators i

These operators work differently on numeric values than they do on strings. If either operand is a number, then both operands are converted to integers, and the bitwise operation is performed between the two integers.

If both operands are strings, these operators do bitwise operations between corresponding bits from the two strings.

Ternary operator (?)

Example

my $size =

($width < 10) ? "small" :

($width < 20) ? "medium" :

($width < 50) ? "large" :

"extra-large"; # default

Dodatak

Perl Bitwise Operators Example

There are following Bitwise operators supported by Perl language, assume if $a = 60; and $b = 13:

```
&
 Binary AND Operator  copies a bit to the result if it exists in both operands.
($a & $b) will give 12 which is 0000 1100

|
Binary OR Operator  copies a bit if it exists in either operand.
($a | $b) will give 61 which is 0011 1101

^
Binary XOR Operator  copies the bit if it is set in one operand  but not both.
($a ^ $b) will give 49 which is 0011  0001

~
Binary Ones Complement  Operator is unary  and has the effect of 'flipping' bits.
(~$a ) will give -61 which is 1100  0011  in 2's complement  form due to a signed binary numb

<<
 Binary Left Shift Operator.  The left operands  value is moved left by the number  of bits spec
operand.
$a << 2 will give 240 which is 1111  0000

>>
Binary Right Shift Operator.  The left operands  value is moved right by the number  of bits sp
operand.
$a >> 2 will give 15 which is 0000  1111
```

Slika 3 Primer 1

Example

Try following example to understand all the bitwise operators available in Perl,.

```
#!/usr/local/bin/perl
use integer;

$a = 60;
$b = 13;

print "Value of \$a = $a and value of \$b = $b\n";

$c = $a & $b;
print "Value of \$a & \$b = $c\n";

$c = $a | $b;
print "Value of \$a | \$b = $c\n";

$c = $a ^ $b;
print "Value of \$a ^ \$b = $c\n";

$c = ~$a;
print "Value of ~\$a = $c\n";

$c = $a << 2;
print "Value of \$a << 2 = $c\n";

$c = $a >> 2;
print "Value of \$a >> 2 = $c\n";
```

Slika 4 Primer 2

When the above code is executed, it produces following result:

```
Value of $a = 60 and value of $b = 13
Value of $a & $b = 12
Value of $a | $b = 61
Value of $a ^ $b = 49
Value of ~$a = 18446744073709551555
Value of $a << 2 = 240
Value of $a >> 2 = 15
```

Slika 5 Primer 3

Miscellaneous operators

Cilj

- Upoznavanja sa raznim vrstama Perl operatora

Miscellaneous operators

Miscellaneous operators

Range operator

The range operator is really two different operators depending on the context. In a list context, it returns a list of values counting (by ones) from the left value to the right value.

In a scalar context, returns a Boolean value. It is false as long as its left operand is false. Once the left operand is true, the range operator stays true until the right operand is true, after which the range operator becomes false again. The right operand is not evaluated while the operator is in the false state, and the left operand is not evaluated while the operator is in the true state.

The alternate version of this operator does not test the right operand immediately when the operator becomes true; it waits until the next evaluation.

Conditional operator

Ternary ?: is the conditional operator. It works much like an if-then-else statement, but it can safely be embedded within other operations and functions.

test_expr

?

if_true_expr

:

if_false_expr

If the test_expr is true, only the if_true_expr is evaluated. Otherwise, only the if_false_expr is evaluated. Either way, the value of the evaluated expression becomes the value of the entire expression.

Comma operator

In a list context, " ," is the list argument separator and inserts both its arguments into the list. In scalar context, " , " evaluates its left argument, throws that value away, then evaluates its right argument and returns that value.

The =>operator is mostly just a synonym for the comma operator. It's useful for documenting arguments that come in pairs. It also forces any identifier to the left of it to be interpreted as a string.

String operator

The concatenation operator "." is used to add strings together:

print 'abc' . 'def'; # prints abcdef

print $a . $b; # concatenates the string values of $a and $b

Binary x is the string repetition operator. In scalar context, it returns a concatenated string consisting of the left operand repeated the number of times specified by the right operand.

print '-' x 80; # prints row of dashes

print "\t" x ($tab/8), ' ' x ($tab%8); # tabs over

In list context, if the left operand is a list in parentheses, the x works as a list replicator rather than a string replicator. This is useful for initializing all the elements of an array of indeterminate length to the same value:

@ones = (1) x 80; # a list of 80 1s

@ones = (5) x @ones; # set all elements to 5

The each operator

Starting with Perl 5.12, you can use the each operator on arrays. Every time that you call each on an array, it returns two values for the next element in the array—the index of the value and the value itself:

use 5.012;

my @rocks = qw/ bedrock slate rubble granite /;

while(my($index, $value) = each @rocks) {

say "$index: $value";

}

If you wanted to do this withouteach, you have to iterate through all of the indices of the array and use the index to get the value:

@rocks = qw/ bedrock slate rubble granite /;

foreach $index (0 .. $#rocks) {

print "$index: $rocks[$index]\n";

}

Dodatak

Miscellaneous operators supported by Perl language

Assume variable a holds 10 and variable b holds 20 then:

.

Binary operator dot (.) concatenates two strings. If $a="abc", $b="def" then $a.$b will give "abcdef"

x

The repetition operator x returns a string consisting of the left operand repeated the number of times specified by the right operand.

('-' x 3) will give ---.

..

The range operator .. returns a list of values counting (up by ones) from the left value to the right value (2..5) will give (2, 3, 4, 5)

++

Auto Increment operator increases integer value by one

$a++ will give 11

--

Auto Decrement operator decreases integer value by one

$a-- will give 9

->

The arrow operator is mostly used in dereferencing a method or variable from an object or a class name $obj->$a is an example to access variable $a from object $obj.

Example

Try following example to understand all the miscellaneous operators available in Perl. Copy and paste following Perl program in test.pl file and execute this program.

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

$a = "abc";
$b = "def";

print "Value of \$a = $a and value of \$b = $b\n";

$c = $a . $b;
print "Value of \$a . \$b = $c\n";

$c = "-" x 3;
print "Value of \"-\" x 3 = $c\n";

@c = (2..5);
print "Value of (2..5) = @c\n";

$a = 10;
$b = 15;
print "Value of \$a = $a and value of \$b = $b\n";

$a++;
$c = $a ;
print "Value of \$a after \$a++ = $c\n";

$b--;
$c = $b ;
print "Value of \$b after \$b-- = $c\n";
```

When the above code is executed, it produces following result:

```
Value of $a = abc and value of $b = def
Value of $a . $b = abcdef
Value of "-" x 3 = ---
Value of (2..5) = 2 3 4 5
Value of $a = 10 and value of $b = 15
Value of $a after $a++ = 11
Value of $b after $b-- = 14
```

Regular expressions

Cilj

- Upoznavanja sa konceptom regularnih izraza

Regular expressions

Regular expressions

A regular expression (abbreviated regex or regexp) is a sequence of characters that forms a search pattern, mainly for use in pattern matching with strings, or string matching, i.e. "find and replace"-like operations.

The concept arose in the 1950s, when the American mathematician Stephen Kleene formalized the description of a regular language, and came into common use with the UNIX text processing utilities ed, an editor, and grep (global regular expression print), a filter.

A regular expression processor processes a regular expression statement expressed in terms of a grammar in a given formal language, and with that examines the target text string, parsing it to identify substrings that are members of its language, the regular expressions.

Regular expressions are used several ways in Perl. They're used in conditionals to determine whether a string matches a particular pattern. They're also used to find patterns in strings and replace the match with something else.

The ordinary pattern match operator looks like /*pattern*/. It matches against the $_ variable by default. If the pattern is found in the string, the operator returns true ("1"); if there is no match, a false value ("") is returned.

The substitution operator looks like s/*pattern*/*replace*/. This operator searches $_ by default. If it finds the specified *pattern*, it is replaced with the string in*replace*. If *pattern* is not matched, nothing happens. You may specify a variable other than $_ with the =~ binding operator (or the negated !~ binding operator, which returns true if the pattern is not matched).

For example:

$text =~ /sampo/;

/

Dodatak

Regular expressions in Perl.

Slika 1 prikazuje metakaraktere. Slika 2 prikazuje ponavljanja, Slika 3 oosebnu notaciju sa \ karakterom, a

Slika 4 daje primere.

char	meaning
^	beginning of string
$	end of string
.	any character except newline
*	match 0 or more times
+	match 1 or more times
?	match 0 or 1 times; *or*: shortest match
|	alternative
( )	grouping; "storing"
[ ]	set of characters
{ }	repetition modifier
\	quote or special

Slika 1 Metakarakteri

*a**	zero or more *a*'s
a+	one or more *a*'s
a?	zero or one *a*'s (i.e., optional *a*)
a{m}	exactly *m* *a*'s
a{m,}	at least *m* *a*'s
a{m,n}	at least *m* but at most *n* *a*'s
repetition?	same as *repetition* but the *shortest* match is taken

Slika 2 Ponavljanja

Single characters

\t	tab
\n	newline
\r	return (CR)
\x*hh*	character with hex. code *hh*

"Zero-width assertions"

\b	"word" boundary
\B	not a "word" boundary

Matching

\w	matches any *single* character classified as a "word" character (alphanumeric or "_")
\W	matches any non-"word" character
\s	matches any whitespace character (space, tab, newline)
\S	matches any non-whitespace character
\d	matches any digit character, equiv. to [0-9]
\D	matches any non-digit character

Sliak 3 Posebna notacija sa \ karakterom

expression	matches...
abc	abc (that exact character sequence, but anywhere in the string)
^abc	abc at the *beginning* of the string
abc$	abc at the *end* of the string
a|b	either of a and b
^abc|abc$	the string abc at the beginning or at the end of the string
ab{2,4}c	an a followed by two, three or four b's followed by a c
ab{2,}c	an a followed by at least two b's followed by a c
ab*c	an a followed by any number (zero or more) of b's followed by a c
ab+c	an a followed by one or more b's followed by a c
ab?c	an a followed by an optional b followed by a c; that is, either abc or ac
a.c	an a followed by any single character (not newline) followed by a c
a\.c	a.c exactly
[abc]	any one of a, b and c
[Aa]bc	either of Abc and abc
[abc]+	any (nonempty) string of a's, b's and c's (such as a, abba, acbabcacaa)
[^abc]+	any (nonempty) string which does *not* contain any of a, b and c (such as defg)
\d\d	any two decimal digits, such as 42; same as \d{2}
\w+	a "word": a nonempty sequence of alphanumeric characters and low lines (underscores), such as foo and 12bar8 and foo_1
100\s*mk	the strings 100 and mk optionally separated by any amount of white space (spaces, tabs, newlines)
abc\b	abc when followed by a word boundary (e.g. in abc! but not in abcd)
perl\B	perl when *not* followed by a word boundary (e.g. in perlert but not in perl stuff)

Slike 4 Primeri

POSIX	Description	ASCII	Unicode	Shorthand	Jav
[:alnum:]	Alphanumeric characters	[a-zA-Z0-9]	[\p{L&}\p{Nd}]		\p{Aln
[:alpha:]	Alphabetic characters	[a-zA-Z]	\p{L&}		\p{Alp
[:ascii:]	ASCII characters	[\x00-\x7F]	\p{InBasicLatin}		\p{ASC
[:blank:]	Space and tab	[\t]	[\p{Zs}\t]	\h	\p{Bla
[:cntrl:]	Control characters	[\x00-\x1F\x7F]	\p{Cc}		\p{Cnt
[:digit:]	Digits	[0-9]	\p{Nd}	\d	\p{Dig
[:graph:]	Visible characters (i.e. anything except spaces, control characters, etc.)	[\x21-\x7E]	[^\p{Z}\p{C}]		\p{Gra
[:lower:]	Lowercase letters	[a-z]	\p{Ll}		\p{Low
[:print:]	Visible characters and spaces (i.e. anything except control characters, etc.)	[\x20-\x7E]	\P{C}		\p{Pri
[:punct:]	Punctuation and symbols.	[!"#$%&'()*+, \-./:;<=>?@ [\\\]^_`{|}~]	[\p{P}\p{S}]		\p{Pun
[:space:]	All whitespace characters, including line breaks	[\t\r\n\v\f]	[\p{Z}\t\r\n\v\f]	\s	\p{Spa
[:upper:]	Uppercase letters	[A-Z]	\p{Lu}		\p{Upp
[:word:]	Word characters (letters, numbers and underscores)	[A-Za-z0-9_]	[\p{L}\p{N}\p{Pc}]	\w	
[:xdigit:]	Hexadecimal digits	[A-Fa-f0-9]	[A-Fa-f0-9]		\p{XDi

Slika 5 daje POSIX izraze sa zagradom (engl. POSIX bracket expression).

Slika 5 POSIX izrazi sa zagradom

Dodatak

A regular expression is a string of characters that define the pattern or patterns you are viewing. The syntax of regular expressions in Perl is very similar to what you will find within other regular expression.supporting programs, such as **sed**, **grep**, and **awk**.

The basic method for applying a regular expression is to use the pattern binding operators =~ and !~. The first operator is a test and assignment operator.

There are three regular expression operators within Perl

Match Regular Expression - m//

Substitute Regular Expression - s///

Transliterate Regular Expression - tr///

The forward slashes in each case act as delimiters for the regular expression (regex) that you are specifying. If you are comfortable with any other delimiter then you can use in place of forward slash.

The Match Operator

The match operator, m//, is used to match a string or statement to a regular expression. For example, to match the character sequence "foo" against the scalar $bar, you might use a statement like this:

if ($bar =~ /foo/)

The m// actually works in the same fashion as the q// operator series.you can use any combination of naturally matching characters to act as delimiters for the expression. For example, m{}, m(), and m>< are all valid.

You can omit the m from m// if the delimiters are forward slashes, but for all other delimiters you must use the m prefix.

Note that the entire match expression.that is the expression on the left of =~ or !~ and the match operator, returns true (in a scalar context) if the expression matches. Therefore the statement:

$true = ($foo =~ m/foo/);

Will set $true to 1 if $foo matches the regex, or 0 if the match fails.

In a list context, the match returns the contents of any grouped expressions. For example, when extracting the hours, minutes, and seconds from a time string, we can use:

my ($hours, $minutes, $seconds) = ($time =~ m/(\d+):(\d+):(\d+)/);

Match Operator Modifiers

The match operator supports its own set of modifiers. The /g modifier allows for global matching. The /i modifier will make the match case insensitive. Here is the complete list of modifiers

Modifier Description

i Makes the match case insensitive

m Specifies that if the string has newline or carriage

return characters, the ^ and $ operators will now

match against a newline boundary, instead of a

string boundary

o Evaluates the expression only once

s Allows use of . to match a newline character

x Allows you to use white space in the expression for clarity

g Globally finds all matches

cg Allows the search to continue even after a global match fails

Matching Only Once

There is also a simpler version of the match operator - the ?PATTERN? operator. This is basically identical to the m// operator except that it only matches once within the string you are searching between each call to reset.

For example, you can use this to get the first and last elements within a list:

```
#!/usr/bin/perl

@list = qw/food foosball subeo footnote terfoot canic footbrdige/;

foreach (@list)
{
$first = $1 if ?(foo.*)?;
$last = $1 if /(foo.*)/;
}
print "First: $first, Last: $last\n";
```

This will produce following result:

```
First: food, Last: footbrdige
```

The Substitution Operator

The substitution operator, s///, is really just an extension of the match operator that allows you to replace the text matched with some new text. The basic form of the operator is:

```
s/PATTERN/REPLACEMENT/;
```

The PATTERN is the regular expression for the text that we are looking for. The REPLACEMENT is a specification for the text or regular expression that we want to use to replace the found text with.

For example, we can replace all occurrences of .dog. with .cat. using

```
$string =~ s/dog/cat/;
```

Another example:

```
#/user/bin/perl

$string = 'The cat sat on the mat';
$string =~ s/cat/dog/;

print "Final Result is $string\n";
```

This will produce following result:

```
The dog sat on the mat
```

Substitution Operator Modifiers

Here is the list of all modifiers used with substitution operator

Modifier Description

i Makes the match case insensitive

m Specifies that if the string has newline or carriage

return characters, the ^ and $ operators will now

match against a newline boundary, instead of a

string boundary

o Evaluates the expression only once

s Allows use of . to match a newline character

x Allows you to use white space in the expression

for clarity

g Replaces all occurrences of the found expression

with the replacement text

e Evaluates the replacement as if it were a Perl statement,

and uses its return value as the replacement text

Translation

Translation is similar, but not identical, to the principles of substitution, but unlike substitution, translation (or transliteration) does not use regular expressions for its search on replacement values. The translation operators are:

tr/SEARCHLIST/REPLACEMENTLIST/cds

y/SEARCHLIST/REPLACEMENTLIST/cds

The translation replaces all occurrences of the characters in SEARCHLIST with the corresponding characters in REPLACEMENTLIST. For example, using the "The cat sat on the mat." string we have been using in this chapter:

```
#/user/bin/perl

$string = 'The cat sat on the mat';
$string =~ tr/a/o/;

print "$string\n";
```

This will produce following result:

```
The cot sot on the mot.
```

Standard Perl ranges can also be used, allowing you to specify ranges of characters either by letter or numerical value. To change the case of the string, you might use following syntax in place of the **uc** function.

```
$string =~ tr/a-z/A-Z/;
```

Translation Operator Modifiers

Following is the list of operators related to translation

Modifier Description

c Complement SEARCHLIST.

d Delete found but unreplaced characters.

s Squash duplicate replaced characters

The /d modifier deletes the characters matching SEARCHLIST that do not have a corresponding entry in REPLACEMENTLIST. For example:

```
#!/usr/bin/perl

$string = 'the cat sat on the mat.';
$string =~ tr/a-z/b/d;

print "$string\n";
```

This will produce following result:

```
b b b.
```

The last modifier, /s, removes the duplicate sequences of characters that were replaced, so:

```
#!/usr/bin/perl

$string = 'food';
$string = 'food';
$string =~ tr/a-z/a-z/s;

print $string;
```

This will produce following result:

```
fod
```

Perl control structures

Cilj

- Upoznavanja sa Perl kontrolnim strukturama

Perl control structures

Perl control structures

The if and unless statements execute blocks of code depending on whether a condition is met.

These statements take the following forms:

```
if ( expression ) {
block
} else {
```

```
block
}
```

Example: If

```
#!/usr/bin/perl
# HTTP HEADER
print "content-type: text/html \n\n";

# SOME VARIABLES
$x = 7;
$y = 7;

# TESTING...ONE, TWO...TESTING
if ($x == 7) {
print '$x is equal to 7!';
print "<br />";
}
if (($x == 7) || ($y == 7)) {
print '$x or $y is equal to 7!';
print "<br />";
}
if (($x == 7) ($y == 7)) {
print '$x and $y are equal to 7!';
print "<br />";
}
```

Display

```
$x is equal to 7!
$x or $y is equal to 7!
$x and $y are equal to 7!
```

```
unless ( expression ) {
block
} else {
block
}
```

Example: Unless

```
$a = 35;
$b = 22;
unless($a > $b) {
print "min($a, $b) = $a\n";
```

```
} else {
print "min($a, $b) = $b\n";
}
# it prints: min(35, 22) = 22
if ( expression1) {
block
}
elsif ( expression2) {
block
}
...
elsif ( lastexpression) {
block
}
else {
block
}
```

Example: Elsif

```
#!/usr/bin/perl
# HTTP HEADER
print "content-type: text/html \n\n";

# SOME VARIABLES
$x = 5;

# PLAY THE GUESSING GAME
if ($x == 6) {
print "X must be 6.";
}
elsif ($x == 4) {
print "X must be 4.";
}
elsif ($x == 5) {
print "X must be 5!";
}
```

Example: Elsif

```
my ($a, $b);
print "Enter 2 numbers, one per line\n";
```

```
chomp($a = <STDIN>);
chomp($b = <STDIN>);
if ( $a < $b ) {
print "$a is less than $b\n";
} elsif ( $b < $a ) {
print "$b is less than $a\n";
} else {
print "The numbers are equal.\n";
}
```

Example: next

```
for ( my $i = 0; $i < 30; $i++) {
if ( $i % 3 != 0 ) { next; }
print "$i\n";
}
```

Example: infinite loop

```
for (; ; ) {
# block of statements
}
```

Modifiers

Cilj

- Upoznavanja sa konceptom Perl modifikatora

Modifiers

Modifiers

Any simple statement may be followed by a single modifier that gives the statement a conditional or looping mechanism. This syntax provides a simpler and often more elegant method than using the corresponding compound statements. These modifiers are:

statement
if
EXPR
;
statement
unless
EXPR
;

statement

while

EXPR

;

statement

until

EXPR

;

For example:

$i = $num if ($num < 50); # $i will be less than 50

$j = $cnt unless ($cnt < 100); # $j will equal 100 or greater

$lines++ while <FILE>;

print "$_\n" until /The end/;

The conditional is evaluated first with the while and until modifiers except when applied to a do {} statement, in which case the block executes once before the conditional is evaluated.

For example:

do {

$line = <STDIN>;

...

} until $line eq ".\n"

Example

$firstVar = 20;

$secondVar = 20;

$firstVar++ if ($secondVar == 10);

print("firstVar = $firstVar\n");

print("secondVar = $secondVar\n");;

Display

firstVar = 20

secondVar = 20

Example

$firstVar = 20;

$secondVar = 20;

$firstVar++ unless ($secondVar == 10);

print("firstVar = $firstVar\n");

print("secondVar = $secondVar\n");

Display

firstVar = 21

secondVar = 20

Example

$firstVar = 10;

$firstVar++ until ($firstVar > 2);

print("firstVar = $firstVar\n");

Display

firstVar = 10

Loops

Cilj

- Upoznavanja sa tipovima petlji

Loops

Loops

While loops

The while statement repeatedly executes a block as long as its conditional expression is true. For example:

while (<INFILE>) {

print OUTFILE, "$_\n";

}

This loop reads each line from the file opened with the filehandle INFILE and prints them to the OUTFILE filehandle. The loop will cease when it encounters an end-of-file.

If the word while is replaced by the word until , the sense of the test is reversed. The conditional is still tested before the first iteration, though.

The while statement has an optional extra block on the end called a continue block. This block is executed before every successive iteration of the loop, even if the main while block is exited early by the loop control command next . However, the continue block is not executed if the main block is exited by a last statement. The continue block is always executed before the conditional is evaluated again.

Example: While loop

#!/usr/bin/perl

print "content-type: text/html \n\n";

SET A VARIABLE

```
$count = 0;
# RUN A WHILE LOOP
while ($count <= 7) {
# PRINT THE VARIABLE AND AN HTML LINE BREAK
print "$count<br />";
# INCREMENT THE VARIABLE EACH TIME
$count ++;
}
print "Finished Counting!";
```

Display

```
0
1
2
3
4
5
6
7
Finished Counting!
```

For loops

The for loop has three semicolon-separated expressions within its parentheses. These three expressions function respectively as the initialization, the condition, and the re-initialization expressions of the loop. The for loop can be defined in terms of the corresponding while loop:

```
for ($i = 1; $i < 10; $i++) {
...
}
```

is the same as:

```
$i = 1;
while ($i < 10) {
...
}
continue {
$i++;
}
```

Example: For loop

forloop.pl

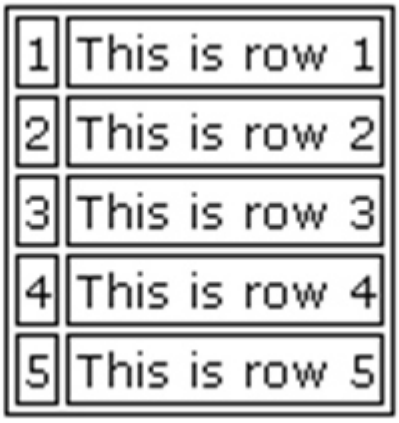
```
#!/usr/bin/perl
```

```
print "content-type: text/html \n\n";

# SET UP THE HTML TABLE
print "<table border='1'>";

# START THE LOOP, $i is the most common counter name for a loop!
for ($i = 1; $i < 5; $i++) {
# PRINT A NEW ROW EACH TIME THROUGH W/ INCREMENT
print "<tr><td>$i</td><td>This is row $i</td></tr>";
}
# FINISH THE TABLE
print "</table>";
```

Display



Slika 1 Primer

Example: For loop using C style

```
#!/usr/bin/perl -w
# for loop example 1
for ($i = 1; $i < 100; $i++) {
print "$i\n";
}
```

Example: For loops using ranges:

```
#!/usr/bin/perl -w
# for loop example 2
$var1 = 1;
$var2 = 100;
$i = 1;
for ($var1..$var2) {
```

```
print "$i\n";
$i+=1;
}
```

Foreach loops

The foreach loop iterates over a list value and sets the control variable (*var*) to be each element of the list in turn:

```
foreach
var
(
list
) {
...
}
```

Example: foreach

```
#!/usr/bin/perl
print "content-type: text/html \n\n"; #The header

# SET UP THE HTML TABLE
print "<table border='1'>";

# CREATE AN ARRAY
andnames = qw(Steve Bill Connor Bradley);

# SET A COUNT VARIABLE
$count = 1;

# BEGIN THE LOOP
foreach $names(andnames) {
print "<tr><td>$count</td><td>$names</td></tr>";
$count++;
}
print "</table>";
```

Like the while statement, the foreach statement can also take a continue block.

Loop control

Cilj

- Upoznavanja sa načinima upravljanja petljama

Loop control

Loop control

Upravljanje petljom

Može se staviti labela na vrhu petlje da joj se da ime. Labela petlje identifikuje petlju za komande upravljanja petljnom (engl. loop-control commands) kao što su next , last i redo.

LINE: while (<SCRIPT>) {

print;

next LINE if /^#/; # discard comments

}

Sintaksa za za komande upravljanja petljnom je data kao:

last

label

next

label

redo

label

Ako nema labele, komande upravljanja petljnom se odnose na najdublju unutrašnju ugnježdenu petlju.

Komanda **last** je slična **break** izjavi u programskom jeziku C (koja se koristi u petlji); ova komanda Ova

Komanda **next** je kao **continue** izjava u programskom jeziku C; ona preskače ostatak tekuće iteracije i počinje novu iteraciju petlje. Ako postoji blok **continue** u petlji, on se obavezno izvršava upravo prije nego se uslov petlje ponovo proveri.

Komanda **redo** restartuje blok petlje bez ponovne evaluacije uslova petlje.

Goto

Perl podražava **goto** komandu.

Format **goto*label*** traži izjavu sa imenom ***label*** i nastavlja izvršavanje tamo. Ne može se koristiti da uđe u unutar strukture koja zahteva inicijalizaciju, kao što su podprogram ili **for each** loop.

Dodatak

Loops

while loop

Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.

until loop

Repeats a statement or group of statements until a given condition becomes true. It tests the condition before executing the loop body.

for loop

Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.

foreach loop

The foreach loop iterates over a normal list value and sets the variable VAR to be each element of the list in turn.

do...while loop

Like a while statement, except that it tests the condition at the end of the loop body

nested loops

You can use one or more loop inside any another while, for or do..while loop.

Loop Control Statements:

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

C supports the following control statements. Click the following links to check their detail.

next statement

Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.

last statement

Terminates the loop statement and transfers execution to the statement immediately following the loop.

continue statement

A continue BLOCK, it is always executed just before the conditional is about to be evaluated again.

redo statement

The redo command restarts the loop block without evaluating the conditional again. The

continue block, if any, is not executed.

goto statement

Perl supports a goto command with three forms.

The Infinite Loop

A loop becomes infinite loop if a condition never becomes false. The for loop is traditionally used for this purpose. Since none of the three expressions that form the for loop are required, you can make an endless loop by leaving the conditional expression empty.

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

for( ; ; )
```

```
{
printf "This loop will run forever.\n";
}
```

You can terminate above infinite loop by pressing Ctrl + C keys.

When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but as a programmer more commonly use the for(;;) construct to signify an infinite loop.

Dodatak

Primeri

```
for($i = 5; $i >= -5; $i--) {
print "$i "
}
print "\n";
# expected: 5 4 3 2 1 0 -1 -2 -3 -4 -5

print "$_ " for(-5..5);
print "\n";
# expected: -5 -4 -3 -2 -1 0 1 2 3 4 5

$i = 0;
for (;;) {
$i++;
last if $i > 10;
}
print "i = $i\n"; # expected i = 11

# initialize a hash (associative array)
%ages = (John => 22, Harry => 25, Anne => 51);
# put the names in an array
@names = keys(%ages);
for($count = 0; @names; $count++) {
print shift @names;
print " ";
}

print "\nThe hash has $count elements\n";
```

Display

Slika 1 Rezultat

Special Variables

Cilj

- Upoznavanja sa posebnim Perl varijablama

Special Variables

Special Variables

Some variables have a predefined and special meaning in Perl. They are the variables that use punctuation characters after the usual variable indicator ($, @, or %), such as $_. The explicit, long-form names shown are the variables' equivalents when you use the English module by including "use English;" at the top of your program.

Global Special Variables

The most commonly used special variable is $_, which contains the default input and pattern-searching string. For example, in the following lines:

foreach ('hickory','dickory','doc') {

print;

}

The first time the loop is executed, "hickory" is printed. The second time around, "dickory" is printed, and the third time, "doc" is printed. That's because in each iteration of the loop, the current string is placed in $_, and is used by default by print . Here are the places where Perl will assume $_ even if you don't specify it:

Various unary functions, including functions like ord and int , as well as the all file tests (-f , -d) except for -t , which defaults to STDIN .

Various list functions like print and unlink.

The pattern-matching operations m// , s/// , and tr/// when used without an =~ operator.

The default iterator variable in a foreach loop if no other variable is supplied.

The implicit iterator variable in the grep and map functions.

The default place to put an input record when a line-input operation's result is tested by itself as the sole criterion of a while test (i.e., < filehandle >). Note that outside of a whil test, this will not happen.

Dodatak

Special Vars Quick Reference

`$_`

The default parameter for a lot of functions.

`$.`

Holds the current record or line number of the file handle that was last read. It is read-only and will be reset to 0 when the file handle is closed.

`$/`

Holds the input record separator. The record separator is usually the newline character. However, if `$/` is set to an empty string, two or more newlines in the input file will be treated as one.

`$,`

The output separator for the `print()` function. Nor-mally, this variable is an empty string. However, setting `$,` to a newline might be useful if you need to print each element in the parameter list on a separate line.

`$\`

Added as an invisible last element to the parameters passed to the `print()` function. Normally, an empty string, but if you want to add a newline or some other suffix to everything that is printed, you can assign the suffix to `$`.

`$#`

The default format for printed numbers. Normally, it's set to `%.20g`, but you can use the format specifiers covered in the section "Example: Printing Revisited" in Chapter 9to specify your own default format.

`$%`

Holds the current page number for the default file handle. If you use `select()` to change the default file handle, `$%` will change to reflect the page number of the newly selected file handle.

`$=`

Holds the current page length for the default file handle. Changing the default file handle will change `$=` to reflect the page length of the new file handle.

`$-`

Holds the number of lines left to print for the default file handle. Changing the default file handle will change `$-` to reflect the number of lines left to print for the new file handle.

`$~`

Holds the name of the default line format for the default file handle. Normally, it is equal to the file handle's name.

`$^`

Holds the name of the default heading format for the default file handle. Normally, it is equal to the file handle's name with `_TOP` appended to it.

`$|`

If nonzero, will flush the output buffer after every `write()` or `print()` function. Normally, it is set to 0.

`$$`

This UNIX-based variable holds the process number of the process running the Perl interpreter.

`$?`

Holds the status of the last pipe close, back-quote string, or `system()` function.

`$``

Holds the string that preceded whatever was matched by the last successful pattern match.

`$'`

Holds the string that followed whatever was matched by the last successful pattern match.

`$0`

Holds the name of the file containing the Perl script being executed.

`$<number>`

This group of variables (`$1`, `$2`, `$3`, and so on) holds the regular expression pattern memory. Each set of parentheses in a pattern stores the string that match the components surrounded by the parentheses into one of the `$<number>` variables.

`$[`

Holds the base array index. Normally, it's set to 0. Most Perl authors recommend against changing it without a very good reason.

`$]`

Holds a string that identifies which version of Perl you are using. When used in a numeric context, it will be equal to the version number plus the patch level divided by 1000.

`$"`

This is the separator used between list elements when an array variable is interpolated into a double-quoted string. Normally, its value is a space character.

`$;`

Holds the subscript separator for multidimensional array emulation. Its use is beyond the scope of this book.

`$!`

When used in a numeric context, holds the current value of `errno`. If used in a string context, will hold the error string associated with `errno`.

`$@`

Holds the syntax error message, if any, from the last `eval()` function call.

`$<`

This UNIX-based variable holds the read `uid` of the current process.

`$>`

This UNIX-based variable holds the effective `uid` of the current process.

`$)`

This UNIX-based variable holds the read `gid` of the current process. If the process belongs to multiple groups, then `$)` will hold a string consisting of the group names separated by spaces.

`$:`

Holds a string that consists of the characters that can be used to end a word when word-wrapping is performed by the `^` report formatting character. Normally, the string consists of the space, newline, and dash characters.

`$^D`

Holds the current value of the debugging flags. For more information.

`$^F`

Holds the value of the maximum system file description. Normally, it's set to 2. The use of this variable is beyond the scope of this book.

`$^I`

Holds the file extension used to create a backup file for the in-place editing specified by the `-i` command line option. For example, it could be equal to ".bak."

`$^L`

Holds the string used to eject a page for report printing.

`$^P`

This variable is an internal flag that the debugger clears so it will not debug itself.

`$^T`

Holds the time, in seconds, at which the script begins running.

`$^W`

Holds the current value of the `-w` command line option.

`$^X`

Holds the full pathname of the Perl interpreter being used to run the current script.

`$ARGV`

Holds the name of the current file being read when using the diamond operator `(<>)`.

`@ARGV`

This array variable holds a list of the command line arguments. You can use `$#ARGV` to determine the number of arguments minus one.

`@F`

This array variable holds the list returned from autosplit mode. Autosplit mode is associated with the `-a` command line option.

`@Inc`

This array variable holds a list of directories where Perl can look for scripts to execute. The list is mainly used by the `require` statement.

`%Inc`

This hash variable has entries for each filename included by `do` or `require` statements. The key of the hash entries are the filenames, and the values are the paths where the files were found.

`%ENV`

This hash variable contains entries for your current environment variables. Changing or adding an entry affects only the current process or a child process, never the parent process. See the section "Example: Using the `%ENV` Variable" later in this chapter.

`%SIG`

This hash variable contains entries for signal handlers. For more information about signal handlers

`_`

This file handle (the underscore) can be used when testing files. If used, the information about the last file tested will be used to evaluate the new test.

`DATA`

This file handle refers to any data following `__END__`.

`STDERR`

This file handle is used to send output to the standard error file. Normally, this is connected to the display, but it can be redirected if needed.

`STDIN`

This file handle is used to read input from the standard input file. Normally, this is connected to the keyboard, but it can be changed.

`STDOUT`

This file handle is used to send output to the standard output file. Normally, this is the display, but it can be changed.

L3: Osnove Perl jezika

Zaključak*
Nakon studiranja sadržaja ovog poglavlja, studenti će steći osnovna znanja iz oblasti programiranja u Perl jeziku.

LearningObject

Uvod u Perl jezik

Perl - nizovi i implicitne promenljive

- Uz pomoć foreach petlje, proći kroz brojeve od 1 do 10 i ispisati njihove kvadrate. Koristiti range operator za kreiranje liste brojeva.
- Kreirati niz @numbers koji sadrži brojeve od 1 do 10. Koristiti while petlju i shift funkciju, i ispisati kvadrat svakog broja u nizu.
- Napisati skriptu (reverse_word.pl) koja od korisnika sa STDIN uzima rečenicu i ispisuje je tako da su sve reči napisane obrnuto. Primer:

Ulaz: Hello world!

Izlaz: olleH !dlroW

Možete koristiti funkcije split i join.

Naslov mejla treba da bude:

IT2008-IPT-SCRP-STRL-DZ-DZ03-Ime-Prezime-BrojIndeksa.docx

Perl programski jezik

Perl - nizovi i implicitne promenljive

Perl - array variables

```
#!/usr/bin/perl
# DEFINE AN ARRAY
@coins = ("Quarter","Dime","Nickel");

# PRINT THE ARRAY
print "@coins";
print @coins;
```

perl - array indexing

```
#!/usr/bin/perl
# DEFINE AN ARRAY
@coins = ("Quarter","Dime","Nickel");

# PRINT THE WHOLE ARRAY
print "@coins";
# PRINT EACH SCALAR ELEMENT
print $coins[0]; #Prints the first element
print "\n";
```

```
print $coins[1]; #Prints the 2nd element
print "\n";
print $coins[2]; #Prints the 3rd element
```

Perl - sequential number arrays

```
#!/usr/bin/perl
# Intervals (ranges)
@10 = (1 .. 10);
@100 = (1 .. 100);
@1000 = (100 .. 1000);
@abc = (a .. z);

# PRINT 'EM
print "@10\n";
print "@100\n";
print "@1000\n";
print "@abc\n
```

perl - finding the length of an array

```
#!/usr/bin/perl
@nums = (1 .. 20);
@alpha = ("a" .. "z");

# SCALAR FUNCTION
print scalar(@nums)." \n ";
print scalar(@alpha)." \n ";

# FORCING INTO A SCALAR CONTEXT
$nums = @nums;
$alpha = @alpha;
print "$nums\n";
print "$alpha\n";
print "There are $nums numerical elements\n";
print "There are $alpha letters in the alphabet!";
```

Perl - adding and removing elements

```
#!/usr/bin/perl
# AN ARRAY
@coins = ("Quarter","Dime","Nickel");

# ADD ELEMENTS
push(@coins, "Penny"); # Inserts an element at the end of the array
```

```
print "@coins";
print "\n";
unshift(@coins, "Dollar"); # Inserts an element at the beginning of the array
print "@coins";

# REMOVE ELEMENTS
pop(@coins);
print "\n";
print "@coins";
shift(@coins);
print "\n";
# BACK TO HOW IT WAS
print "@coins";
```

perl - slicing array elements

```
#!/usr/bin/perl
@coins = qw(Quarter Dime Nickel Penny); # Word array syntax
@slicecoins = @coins[0,2];
print "@slicecoins\n";
```

perl - replacing array elements

```
#!/usr/bin/perl
@nums = (1..20);
splice(@nums, 5,5,21..25);
print "@nums \n";
```

perl - sorting arrays

```
#!/usr/bin/perl
# TWO ARRAYS
@foods = qw(pizza steak chicken burgers);
@Foods = qw(Pizza Steak chicken burgers);

# SORTING
@foods = sort(@foods);
@Foods = sort(@Foods);

# PRINT THE NEW ARRAYS
print "@foods\n";
print "@Foods\n";
```

$_ - the default variable

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

```
use strict;
use warnings;

print "Enter a string and I'll reverse it.\nEnter 'quit' or 'q' to quit.\n";
while(<STDIN>) {
if($_ =~ /q/) {
last;
}
$_ = reverse;
print "$_\n"
}
```