Lekcija 4 - Perl heševi i podprogrami

Contents

LearningObject

L4: Perl heševi i podprogrami

Uvod *
Ovo poglavlje predstavlja uvod u oblast Perl heševa i podprograma.

Hashes

Cilj

- Upoznavanja sa Perl heševima

Hashes

Hashes

Heševi

Heševi su napredne forme nizova. Jedno od ograničenja nizova je to što je teško dobiti informacije iz nizova.

Na primer, ako je data lista ljudi sa njihovih godinama.

Heš varijable

Heš varijable je znak (%) nakon čega sledi slovo, pa onda nula ili više slova, digita ili donjih crta (engl. underscores). Dio nakon (%) znaka je isto kao i kod skalara i imena varijabli nizova (engl. array variable names).

Umesto da se referencira celi heš, heš se obično kreira i pristupa mu se referenciranjem njegovih elemenata.

Svaki element heša je posebna skalarna varijabla, kojoj se pristupa preko indeksa stringa koji se zove ključ (engl. key).

Elementi heša %fred se referenciraju sa $fred{$key } gdje je $key bilo koji skalarni izraz..

Treba zapaziti da pristup elementu heša zahteva različitu interpunkciju za razliku od pristupa celom hešu. Kao i kod nizova, kreiraju se novi elementi jednostavnom dodelom novog heš elementa :

$fred{"aaa"} = "bbb"; # creates key "aaa", value "bbb"

$fred{234.5} = 456.7; # creates key "234.5", value 456.7

Ove dve zadnje izjave kreiraju dva elementa u hešu. Narednim pristupima istom element (koristeći isti ključ) vraćaju se predhodno sačuvane vrednosti:

print $fred {"aaa"}; # prints "bbb"

$fred {234.5} += 3; # makes it 459.7

Referenciranje elementa koji ne postoji vraća undef vrednost, isto kao kod nedostajućeg elementa niza ili nedefinisane skalarne varijable.

Heš elegantno rešava problem dozvolajvajući pristup @ages nizu ne preko indeksa, već preko skalarnog ključa. Na primer, da se dobiju godišta različitih ljudi, mogu se koristiti njihova imena kao ključ da se definiše heš .

```
%ages = ('Martin' => 28,
'Sharon' => 35,
'Rikke' => 29,);
print "Rikke is $ages{Rikke} years old\n";
```

Rezultat

```
Rikke is 29 years old
```

Dodatak

Creating Hashes

Hashes are created in one of two following ways. In the first method, you assign a value to a named key on a one-by-one basis:

```
$data{'John Paul'} = 45;
$data{'Lisa'} = 30;
$data{'Kumar'} = 40;
```

In the second case, you use a list, which is converted by taking individual pairs from the list: the first element of the pair is used as the key, and the second, as the value. For example:

```
%data = ('John Paul', 45, 'Lisa', 30, 'Kumar', 40);
```

For clarity, you can use => as an alias for , to indicate the key/value pairs as follows:

```
%data = ('John Paul' => 45, 'Lisa' => 30, 'Kumar' => 40);
```

Here is one more variant of the above form, have a look at it, here all the keys have been preceded by hyphen (-) and no quotation is required around them:

```
%data = (-JohnPaul => 45, -Lisa => 30, -Kumar => 40);
```

But it is important to note that there is a single word ie without spaces keys have been used in this form of hash formation and if you build-up your hash this way then keys will be accessed using hyphen only as shown below.

```
$val = %data{-JohnPaul}
$val = %data{-Lisa}
```

Accessing Hash Elements

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

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

```
%data = ('John Paul' => 45, 'Lisa' => 30, 'Kumar' => 40);

print "$data{'John Paul'}\n";
print "$data{'Lisa'}\n";
print "$data{'Kumar'}\n";
```

This will produce following result:

```
45
30
40
```

Extracting Slices

You can extract slices of a hash just as you can extract slices from an array. You will need to use @ prefix for the variable to store returned value because they will be a list of values:

```
#!/usr/bin/perl

%data = (-JohnPaul => 45, -Lisa => 30, -Kumar => 40);

@array = @data{-JohnPaul, -Lisa};

print "Array : @array\n";
```

This will produce following result:

```
Array : 45 30
```

Extracting Keys and Values

You can get a list of all of the keys from a hash by using keys function which has the following syntax:

```
keys %HASH
```

This function returns an array of all the keys of the named hash. Following is the example:

```
#!/usr/bin/perl

%data = ('John Paul' => 45, 'Lisa' => 30, 'Kumar' => 40);

@names = keys %data;

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

This will produce following result:

```
Lisa
John Paul
Kumar
```

Similarly you can use values function to get a list of all the values. This function has following syntax:

values %HASH

This function returns a normal array consisting of all the values of the named hash. Following is the example:

```
#!/usr/bin/perl
%data = ('John Paul' => 45, 'Lisa' => 30, 'Kumar' => 40);
@ages = values %data;

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

This will produce following result:

```
30
45
40
```

Example

Create an empty hash

```
my %color_of;
```

Insert a key-value pair into a hash

```
$color_of{'apple'} = 'red';
```

Use a variable instead of the key and then you don't need to put the variable in quotes:

```
my $fruit = 'apple';

$color_of{$fruit} = 'red';
$color_of{apple} = 'red';
```

Fetch an element of a hash

```
print $color_of{apple};

print $color_of{orange};

$color_of{orange} = "orange";
$color_of{grape} = "purple";
```

Initialize a hash with values

```
my %color_of = (
"apple" => "red",
```

"orange" => "orange",

"grape" => "purple",

);

=> is called the fat arrow or fat comma, and it is used to indicate pairs of elements. The first name, fat arrow, will be clear once we see the the other, thinner arrow (->) used in Perl. The name fat comma comes from the fact that these arrows are basically the same as commas. So we could have written this too:

my %color_of = (

"apple", "red",

"orange", "orange",

"grape", "purple",

);

my %color_of = (

apple => "red",

orange => "orange",

grape => "purple",

);

Assignment to a hash element

$color_of{apple} = "green";

print $color_of{apple}; # green

The size of a hash

print scalar keys %hash;

Creation of hash

Cilj

- Upoznavanja sa kreiranjem Perl heša

Creation of hash

Creation of hash

Kreiranje heša

Heševi se kreiraju na dva načina.

Prvi način

Dodeljuje se vrednost imenovanom ključu na bazi jedan-na-jedan :

$ages{Martin} = 28;

Drugi način:

Koristiti listu koja se konvertuje uzimanjem pojedinačnih parova iz liste, prvi element para se koristi kao ključ, a drugi kao vrednost. Na primer:

%hash = ('Fred' , 'Flintstone', 'Barney', 'Rubble');

Radi jasnoće koristiti => kao drugi ime za to, za naznačavanje parova ključ/vrednost:

%hash = ('Fred' => 'Flintstone',

'Barney' => 'Rubble');

Dodatak

Creating Hashes

Hashes are created in one of two following ways. In the first method, you assign a value to a named key on a one-by-one basis:

$data{'John Paul'} = 45;

$data{'Lisa'} = 30;

$data{'Kumar'} = 40;

In the second case, you use a list, which is converted by taking individual pairs from the list: the first element of the pair is used as the key, and the second, as the value. For example:

%data = ('John Paul', 45, 'Lisa', 30, 'Kumar', 40);

For clarity, you can use => as an alias for , to indicate the key/value pairs as follows:

%data = ('John Paul' => 45, 'Lisa' => 30, 'Kumar' => 40);

Here is one more variant of the above form, have a look at it, here all the keys have been preceded using by hyphen (-) and no quotation is required around them:

%data = (-JohnPaul => 45, -Lisa => 30, -Kumar => 40);

But it is important to note that there is a single word ie without spaces keys have been used in this form of hash formation and if you build-up your hash this way then keys will be accessed using hyphen only as shown below.

$val = %data{-JohnPaul}

$val = %data{-Lisa}

Accessing Hash Elements

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

#!/usr/bin/perl

%data = ('John Paul' => 45, 'Lisa' => 30, 'Kumar' => 40);

print "$data{'John Paul'}\n";

```
print "$data{'Lisa'}\n";
print "$data{'Kumar'}\n";
```

Display

45

30

40

Extracting individual elements

Cilj

- Upoznavanja sa metodama izdvajanja pojedinih elemenata iz Perl heša

Extracting individual elements

Extracting individual elements

Izdvajanja pojedinih elemenata

Pojedinačni elementi se mogi izdvojiti iz heša specifikacijom elemenata ključa za vrednost koja se traži unutar zagrada.

```
print $hash{Fred};
```

Rezultat

Flintstoner

Izdvajanje delova (engl. extracting slices)

Mogu se izdvojiti delovi heša na isti način na koji se izdvajaju delovi iz niza. Međutim, treba koritiiti @ prefix, jer će vraćena vrednost biti lista vrednosti:

```
#!/uer/bin/perl
%hash = (-Fred => 'Flintstone', -Barney => 'Rubble');
print join("\n",@hash{-Fred,-Barney});
```

Rezultat

Flintstone

Rubble

$hash{-Fred, -Barney} ne vraća ništa..

Izvlačenje ključeva i vrednosti

Može se dobiti lista svih ključeva iz heša koriteći ključeve.

```
#!/usr/bin/perl
%ages = ('Martin' => 28, 'Sharon' => 35, 'Rikke' => 29);
print "The following are in the DB: ", join(', ',values %ages),"\n";
```

Rezultat

The following are in the DB: 29, 28, 35

Ova metoda može biti korisna kod štampanja sadržaja heša:

```
#!/usr/bin/perl
%ages = ('Martin' => 28, 'Sharon' => 35, 'Rikke' => 29);
foreach $key (%ages)
{
print "$key is $ages{$key} years old\n";
}
```

Rezultat

Rikke is 29 years old

29 is years old

Martin is 28 years old

28 is years old

Sharon is 35 years old

35 is years old

Problem ovog pristupa je štot (%ages) vraća listu vrednosti. da bi se rešio ovaj problem, treba koristiti funkciju each koja će vratiti ključ i vrednost par kao što je dato:

Dodatak

The Perl each function is used to iterate over the elements of an array or a hash.

The syntax forms of the each function are as follows:

```
each %hash
each @array
```

If you use it with a hash, it returns a two element list consisting of the key-value of the next pair element of the hash.

In the case of arrays, this function returns a two element list consisting of the next index and the value of the array element associated with it. Please note that using each with arrays is available starting with the Perl 5.12 version only. Because the Perl each function doesn't loop by itself, you can wrap it in awhile loop to access the elements of a hash or an array. The each function returns false when the end of the hash/array is reached.

```
#!/usr/bin/perl
%ages = ('Martin' => 28, 'Sharon' => 35, 'Rikke' => 29);
while (($key, $value) = each %ages)
{
print "$key is $ages{$key} years old\n";
}
```

Display

Rikke is 29 years old

Martin is 28 years old
Sharon is 35 years old

removeelements.pl:

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

# DEFINED HASH
%coins = ( "Quarter" , .25,
"HalfDollar" , .50,
"Penny" , .01,
"Dime" , .10,
"Nickel", .05 );

# print old hash
while (($key, $value) = each(%coins)){
print $key.", ".$value."<br />";
}

# delete the element pairs
delete($coins{Penny});
delete($coins{HalfDollar});

# print the new hash
print "<br />";
while (($key, $value) = each(%coins)){
print $key.", ".$value."<br />";
}
```

Display
Nickel, 0.05
Dime, 0.1
HalfDollar, 0.5
Penny, 0.01
Quarter, 0.25

Checking for existence

Cilj

- Provjera postojanja elemenata u hešu

Checking for existence

Checking for existence

Provjera postojanja

Ako pokušate pristupiti paru ključ/vrijednost (engl. key/value) heša koji ne postoji, normalno se dobija i nedefiniranu vrijednost, a ako su uključena upozorenja (engl. warnings), onda se dobija upozorenje u vrijeme izvršavanja program (engl. at run time).

To se može zaobići korišćenjem funkcije exists, koja vraća true ako je pozvani ključ postoji, neovisno o tome šta može biti njegova vrednost.

```
#!/usr/bin/perl

%ages = ('Martin' => 28, 'Sharon' => 35, 'Rikke' => 29);

if (exists($ages{"alan"}))

{

print "alan if $ages{$name} years old\n";

}

else

{

print "I don't know the age of alan\n";

}
```

Rezultat

I don't know the age of alan

Dodatak

exists EXPR

Given an expression that specifies an element of a hash, returns true if the specified element in the hash has ever been initialized, even if the corresponding value is undefined.

```
print "Exists\n" if exists $hash{$key};

print "Defined\n" if defined $hash{$key};

print "True\n" if $hash{$key};
```

exists may also be called on array elements, but its behavior is much less obvious and is strongly tied to the use of delete on arrays. Be aware that calling exists on array values is deprecated and likely to be removed in a future version of Perl.

```
print "Exists\n" if exists $array[$index];

print "Defined\n" if defined $array[$index];

print "True\n" if $array[$index];
```

A hash or array element can be true only if it's defined and defined only if it exists, but the reverse doesn't necessarily hold true.

Given an expression that specifies the name of a subroutine, returns true if the specified subroutine has ever been declared, even if it is undefined. Mentioning a subroutine name for exists or defined does not count as declaring it.

Note that a subroutine that does not exist may still be callable: its package may have an AUTOLOAD method that makes it spring into existence the first time that it is called;

```
print "Exists\n"  if exists &subroutine;
print "Defined\n" if defined &subroutine;
```

Slika 1 Primer 1

Note that the EXPR can be arbitrarily complicated as long as the final operation is a hash or array key lookup or subroutine name:

```
if (exists $ref->{A}->{B}->{$key}) { }
if (exists $hash{A}{B}{$key})      { }
if (exists $ref->{A}->{B}->[$ix])  { }
if (exists $hash{A}{B}[$ix])       { }
if (exists &{$ref->{A}{B}{$key}})  { }
```

Slika 2 Primer 2

Although the most deeply nested array or hash element will not spring into existence just because its existence was tested, any intervening ones will. Thus $ref->{"A"} and $ref->{"A"}->{"B"} will spring into existence due to the existence test for the $key element above. This happens anywhere the arrow operator is used, including even here:

undef $ref;

if (exists $ref->{"Some key"}) { }

print $ref; # prints HASH(0x80d3d5c)

Use of a subroutine call, rather than a subroutine name, as an argument to exists() is an error.

```
exists &sub;   # OK
exists &sub(); # Error
```

Slika 3 Primer 3

Sorting/ordering hashes

Cilj

- Upoznavanja sa metodama sortiranja/uređivanja heševa

Sorting/ordering hashes

Sorting/ordering hashes

Sortiranje/ uređivanje heševa

Ne postoje garancije da će redosled u kojem će lista ključeva, vrednosti ili parovi ključ/ vrednost uvek biti ista. Ustvari, bolje je ne oslanjati na redosled između dvije sekvencijalne evaluacije:

```
#!/usr/bin/perl
print(join(', ',keys %hash),"\n");
print(join(', ',keys %hash),"\n");
```

Ako treba garantovati redosled, koristiti funkciju sort, na primer:

```
print(join(', ',sort keys %hash),"\n");
```

Ako se pristupa hešu više puta i ako se želi uvek isti redosled , razmisliti o korišćenju jednog niza za čuvanje uređene sekvence i onda koristiti taj niz (koji ostaje u sortiranom redosledu) i praviti iteracije preko heša . Na primer

```
my @sortorder = sort keys %hash;
foreach my $key (@sortorder)
```

Veličina heša

Dobiti veličinu, odnosno broj elemenata, iz heša koristeći skalarni kontekst u ili ključeve ili vrednosti :

```
#!/usr/bin/perl
%ages = ('Martin' => 28, 'Sharon' => 35, 'Rikke' => 29);
print "Hash size: ",scalar keys %ages,"\n";
```

Display

```
Hash size: 3
```

Dodavanje i uklanjanje elemenata iz heša

Dodavanje novog para ključ/vrednost se može učiniti jednom linijom koda koristeći jednostavni operator dodeljivanja.

Ali za uklanjanje elemenata iz heša treba koristiti funkciju delete.

```
#!/usr/bin/perl
%ages = ('Martin' => 28, 'Sharon' => 35, 'Rikke' => 29);
# Add one more element in the hash
$age{'John'} = 40;
# Remove one element from the hash
delete( $age{'Sharon'} );
```

Primer (ne radi sa brojevima)

```
use strict;
use warnings;

sub main
{
# Declare and initialize an array.
my @items = ("tree", "zebra", "apple", "fox");

# Sort the array.
@items = sort @items;

# print the array contents.
foreach my $item(@items) {
print "$item\n";
}
}
main();
```

Rezultat

```
apple
fox
tree
zebra
```

Primer

```
use strict;
use warnings;

sub main
{
# Declare and initialize an array.
my @items = (0, 5, 10, 20);

# Sort the array.
@items = sort @items;

# print the array contents.
foreach my $item(@items) {
print "$item\n";
}
}
main();
```

Rezultat

0

10

20

5

Dodatak

Perl sorting

1. The syntax forms and a few examples

The Perl sort function sorts a LIST by an alphabetical or numerical order and returns the sorted list value. Keep in mind that the argument list remains unchanged while a new sorted list is returned. In this short free tutorial, I'll give you a few examples about how to use the Perl sort function in your scripts.

The syntax forms of the Perl sort function are as follows:

sort SUBNAME LIST

sort BLOCK LIST

sort LIST

The third syntax form is the simplest of them and is for the standard comparison order. See the following example:

my @array = sort qw(map 23 Perl 101 11 while 1 scalar 102);

print "@array\n";

it prints: 1 101 102 11 23 Perl map scalar while

@array = sort qw(23 101 11 1 102);

print "@array\n";

it prints: 1 101 102 11 23

The problem is that capital letters have a lower ASCII numeric value than the lowercase letters so the words beginning with capital letters will be shown first, as you can see in our example. One alternative is to transform all the list elements into lowercase or uppercase letters and then perform the Perl sort function. You'll see an example below.

As you can see from the second example shown above, even if all the elements of the list are numbers, this simple sort will sort the list in an alphanumerical order.

In practice you need to do some additional performing to make your sort accommodate with your task. And here comes the first and the second syntax forms of the Perl sort function. SUBNAME is the name of a subroutine where you describe how to order the elements of the list. Instead of a subroutine, you can provide a BLOCK as an anonymous in-line subroutine.

This Perl sort function uses two special variables $a and $b and they are any two elements from the list that are compared in pairs by sort to determine how to order the list.

Besides these special variables, the Perl sort function uses two operators: cmp and <=>. So you can sort a list either in an alphanumerical or a numerical order. For this you can use the

cmp (string comparison operator) or <=> (the numerical comparison operator). Please recall how this two operators work:

cmp returns -1, 0 or 1 depending on whether the left argument is stringwise less than, equal to, or greater than the right argument

<=> returns -1, 0 or 1 depending on whether the left argument is numerically less than, equal to, or greater than the right argument

Let's go back to our previous example where we tried to sort a list of numbers. You can do this either by defining a subroutine where you describe how to order the list or by using an in-line subroutine in a block.

In the first case, you can see how this works in the following example:

```
# define a subroutine
sub numSort {
if ($a < $b) { return -1; }
elsif ($a == $b) { return 0;}
elsif ($a > $b) { return 1; }
}

# invoke the Perl sort function with the subroutine
my @array = sort numSort qw(23 101 11 1 102);

print "@array\n";
# it prints: 1 11 23 101 102
```

The previous example is for the first syntax form. You could shorten this code by using the second syntax form of the Perl sort function as in the example below:

```
my @array = sort {$a <=> $b} qw(23 101 11 1 102);
```

Please note that if you explicitly use the cmp and <=> comparisons operators, it matters if $a or $b is on the left or right side of the operator. For instance, if you use $a <=> $b the list will be sorted in an ascending numerical order and if you use $b <=> $a the list will be sorted in a descending numerical order.

If you have a mixed list with either numerical or string elements, you can use the following code to sort it:

```
# Perl sort function with the second syntax form
my @array = sort {$a <=> $b || $a cmp $b}
qw(map 23 Perl 101 11 while 1 scalar 102);

print "@array\n";
# it prints: Perl map scalar while 1 11 23 101 102
```

The numbers will be sorted in a numerical order and the strings in an ASCII order. However, for this example don't use warnings or the –w flag if you want to run it.

If you want to order a list of strings in an alphabetical case-insensitive order, as I mentioned before you can use either the lc or uc function as in the following example:

```
my @array = sort {lc $a cmp lc $b} qw(map Perl while scalar);

print "@array\n";

# it prints: map Perl scalar while
```

In this case the strings will be sorted in an ascending lexicographical order, it doesn’t matter if you have capital letters or not. Please notice that the lc function doesn't modify the values assigned to $a or $b, but returns a lowercase version of the values.

2. How to sort a hash by keys

Considering hashes, you must know that Perl uses internally its own way to store the items. So, generally you can’t keep your hash items in a specific order, except if you use the Tie::IxHash Perl module that preserves the order in which the hash elements were added. But you can access its items in any order you want. The following snippet code shows you how to print the elements of a hash in a specific order of the keys:

```
# define a hash
my %hash = (one => 1, two => 2, three => 3, four =>4);

# using Perl sort function with a foreach loop
foreach my $key (sort keys %hash) {
print "$key: $hash{$key}\n";
}
```

The order of the pair elements of the hash will remain unchanged, but we process the hash elements in the order we need. In the above example, the keys function will return a list with the hash keys, the Perl sort function will sort this list in an alphabetical ascending order (the standard format); the foreach loop will traverse the sorted list and the print function will display the pair elements of the hash.

It produces the following output:

```
four: 4
one: 1
three: 3
two: 2
```

3. How to sort a hash by values

You can access the hash elements in a specific order of their values. See the following code:

```
# define a hash
my %hash = (1 => 'compile', 2 => 'binary',
3 => 'ascii', 4 => 'digit');

# print the hash ordered pairs
```

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

Here we used the Perl sort function with the cmp (string comparison) operator and we get the elements of the hash printed in the values alphabetical descending order. The keys function returns a list with the hash keys.

The elements of this list are assigned to $a and $b for comparisons and the notation $hash{$a} means the corresponding value of the hash key assigned to $a. The Perl sort function will return the sorted list as argument to the foreach loop. The foreach loop will iterate through this list using the $_ special variable. The output is as follows:

```
4: digit
1: compile
2: binary
3: ascii
```

4. How to sort an array of arrays

If you want to sort an array of arrays by the elements values of the sub-arrays you can use the Perl sort function and do something as in the following code:

```
# define an array of arrays
my @AOA = ([25, 49, 33, 200], [145, 32], [11, 121, 78]);

# sort and print the @AOA array
foreach my $item1 (@AOA){
foreach my $item2 (sort {$b <=> $a} @{$item1}){
print "$item2 ";
}
print "\n";
}
```

You know that in Perl a multidimensional array is a usual array that has as elements references to other arrays (or hashes or some other objects). An array of arrays is a bi-dimensional array that has as elements references to other arrays.

In our example, [25, 49, 33, 200] returns a reference to the list (25, 49, 33, 200), so our @AOA array has three scalar elements that are respectively references to the following lists: (25, 49, 33, 200), (145, 32), (11, 121, 78). Using the Perl sort function, the above code sorts the elements of each list in a descending numerical order. To do this we used a nested foreach.

In the outer foreach, the $item is assigned in turn to the elements of the @AOA array, so it contains a reference to a list. In the inner foreach, we used the @{$item1} notation to dereference the $item1 reference. The Perl sort function is invoked for the elements of the sub-arrays (the three list presented above).

Finally, each line will be printed on a separated line, as you can notice from the output of the script:

200 49 33 25

145 32

121 78 11

Keys function

Cilj

- Upoznavanja sa funkcijom keys

Keys function

Keys function

Funkcija keys

Funkcija keys (%hashname) daje listu svih trenutnih kljuceva u hešu % hashname. Drugim recima, to je kao da se neparni elemnti liste (prvi, treci, peti, i tako dalje) vracaju “odmotavajuci” % hashname u kontekstu niza, i ustvari, vraca ih u tom itom redoslijedu. Ako nema elemenata u hešu, funkcija vraca praznu listu.

Na primjer, koristeci heš iz prethodnih primjera, Slika 1:

```
$fred{"aaa"} = "bbb";
$fred{234.5} = 456.7;
@list = keys(%fred); # @list gets ("aaa",234.5) or
```

Slika 1 Primer 1

Kao i u drugim ugradenim funkcijama (engl. built-in function), zagrade su opcija: keys %fred je i keys(%fred).

foreach $key (keys (%fred)) { # once for each key of %fred

print "at $key we have $fred{$key}\n"; # show key and value

}

Ovaj primjer pokazuje da pojedinacni heš elementi mogu biti interpolirati u stringove unutar navodnih znaka. Medutim ne može se interpolirati cijeli heš.

Može se koristiti deo (engl. slice). U skalarnok kontekstu, funkcija keys daje broj elemenata (parovi kljuc-vrednost) u hešu. Na primjer, može se saznati da li hash prazan:

```
if (keys(

%somehash

)) { # if keys() not zero:

...; # array is non empty

}

# ... or ...

while (keys(

%somehash

) < 10) {

...; # keep looping while we have fewer than 10 elements

}
```

Koristeci %somehash u skalarnom kontekstu može otkriti da li je heš prazan ili nije.

```
if (

%somehash

) { # if true, then something's in it

# do something with it

}
```

Dodatak 1

Description

This function returns all the keys of the HASH as a list. The keys are returned in random order but, in fact, share the same order as that used by values and each.

Syntax

Following is the simple syntax for this function:

keys HASH

Return Value

This function returns number of keys in the hash in scalar context and list of keys in list context.

Example

Following is the example showing its basic usage:

#!/usr/bin/perl

%hash = ('One' => 1,

'Two' => 2,

'Three' => 3,

'Four' => 4);

```
@values = values( %hash );
print("Values are  ", join("-", @values), "\n");

@keys = keys( %hash );
print("Keys are ", join("-", @keys), "\n");
```

Slika 2 Primer 2

Display

Values are 4-3-2-1

Keys are Four-Three-Two-One

Dodatak 2

Perl's keys() function is used to iterate (loop) through the the keys of a HASH. The keys are returned as a list (niz).

%contact = ('name' => 'Bob', 'phone' => '111-111-1111');

foreach $key (keys %contact) {

print "$key\n";

```
}
```

In the above example, we start with a simple hashof our contact Bob and his phone number. The keysfunction takes this regular hash as input, and is couched in a foreach loop. Here is the each function standing on it's own:

```
$key (keys %contact)
```

Every time the keys function is called, it grabs a key out of the hash and puts it in the $key string. When couched in the foreach statement it will loop through each element of the hash and pass the keys off to the print statement. The output of the program will be:

```
name

phone
```

There are a couple of things to keep in mind when using the keys function. Because of the way Perl works with hashes, the elements will be returned in random order. Also, the keysfunction does not remove the elements from the hash - the hash is unaffected by the looping process.

So let's say that you want to sort your hash by the keys as you're looping through them. It's a simple matter of setting the keys function inside a sort function like so:

```
%contact = ('name' => 'Bob', 'phone' => '111-111-1111');

foreach $key (sort(keys %contact)) {

print "$key\n";

}
```

Example

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

use strict;

use warnings;

# initialize a hash with a few elements

my %hash = (father => 'John', mother => 'Alice',

son => 'Andrew', daughter => 'Mary');
```

```
# print the hash by using foreach and keys
print "The keys are: ";
foreach my $key (keys % hash) {
print "$key ";
}
print "\n";
```

Display

```
The keys are: son daughter father mother
```

Example

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

use strict;
use warnings;

my %hash = ();
my $count = 1;

while (keys %hash < 10) {
$hash{++$count} = $count * 2 - 1;
}

foreach my $key ( sort {$a <=> $b} keys %hash){
print "$hash{$key} "
}
print "\n";
```

Display

```
1 3 5 7 9 11 13 15 17 19
```

The values function

Cilj

- Upoznavanja sa funkcijom values

The each function

The each function

Funkcija each

Za ispitivanje svakog elementa heša, koristiti funkciju keys, gledajući svaki vraćeni ključ da bi se dobila odgovarajuća vrednost. Mada se ova metoda vrlo često koristi, postoji efikasniji metod,a to je korišćenje funkcije each(%hashname) koja vraća par ključ/vrednost u obliku liste sa dva elementa.

Za svaku evaluaciju ove funkcije za isti heš, sledeći uzastopni par ključ-vrednost se vraća sve dok se ne pristupi svim elementima. Kada nema više parova, svaki povratak vraća praznu listu.

Tako, na primjer, korak kroz %lastname hash iz prethodnog primjera, uraditi nešto poput ovoga :

```
while (($first,$last) = each(%lastname)) {
print "The last name of $first is $last\n";
}
```

Dodela nove vrednosti čitavom hešu resetuje funkciju each na početak .

Dodatak

The each function

Perl's each() function is used to iterate (loop) through the the key / value pairs of a HASH. Each key / value pair is returned as a two-element list.

```
%contact = ('name' => 'Bob', 'phone' => '111-111-1111');
while (($key, $value) = each %contact) {
print "$key=$value\n";
}
```

In the above example, we start with a simple hashof our contact Bob and his phone number.

The eachfunction takes this regular hash as input, and is couched in a while loop. Here is the each function standing on its own:

```
($key, $value) = each %contact
```

Every time the each function is called, it grabs an element out of the hash and puts them in the $keyand $value strings. When couched in the while statement it will loop through each element of the hash and pass them off to the print statement.

There are a couple of things to keep in mind when using the each function. Because of the way Perl works with hashes, the elements will be returned in random order. Also, the each function does not remove the elements from the hash - the hash is unaffected by the looping process.

The each function

Cilj

- Upoznavanja sa Perl funcijom each

The each function

The each function

Funkcija each

Za ispitivanje svakog elementa heša, koristiti funkciju keys, gledajući svaki vraćeni ključ da bi se dobila odgovarajuća vrednost. Mada se ova metoda vrlo često koristi, postoji efikasniji metod,a to je korišćenje funkcije each(%hashname) koja vraća par ključ/vrednost u obliku liste sa dva elementa.

Za svaku evaluaciju ove funkcije za isti heš, sledeći uzastopni par ključ-vrednost se vraća sve dok se ne pristupi svim elementima. Kada nema više parova, svaki povratak vraća praznu listu.

Tako, na primjer, korak kroz %lastname hash iz prethodnog primjera, uraditi nešto poput ovoga :

```
while (($first,$last) = each(%lastname)) {
print "The last name of $first is $last\n";
}
```

Dodela nove vrednosti čitavom hešu resetuje funkciju each na početak .

Dodatak

The each function

Perl's each() function is used to iterate (loop) through the the key / value pairs of a HASH. Each key / value pair is returned as a two-element list.

```
%contact = ('name' => 'Bob', 'phone' => '111-111-1111');
while (($key, $value) = each %contact) {
print "$key=$value\n";
}
```

In the above example, we start with a simple hashof our contact Bob and his phone number.

The eachfunction takes this regular hash as input, and is couched in a while loop. Here is the each function standing on its own:

```
($key, $value) = each %contact
```

Every time the each function is called, it grabs an element out of the hash and puts them in the $keyand $value strings. When couched in the while statement it will loop through each element of the hash and pass them off to the print statement.

There are a couple of things to keep in mind when using the each function. Because of the way Perl works with hashes, the elements will be returned in random order. Also, the each function does not remove the elements from the hash - the hash is unaffected by the looping process.

The delete function

Cilj

- Upoznavanja sa funkcijom delete

The delete function

The delete function

Funkcija delete

Perl ima funkciju za uklanjanje elemenata iz heša. Operand delete funkcije je heš referenca. Perl uklanja par kluč/vrednost iz heša. Na primer:

%fred = ("aaa","bbb",234.5,34.56); # give %fred two elements

delete $fred{"aaa"};

%fred is now just one key-value pair

1. Napisati program koji čita string i onda ga štampa kao i njegovu mapiranu vrednost u skladu sa Slikom 1:

Ulaz	Izlaz
red	apple
green	leaves
blue	ocean

Slika 1 Primer

2. Napisati program koji čita niz reči, sa jednom reči po jednom redu sve do kraja datoteke (engl. end-of-file), a onda štampa rezime o tome koliko puta se svaka reč pojavljuje (dodatni zadatak. sortirati izlaz u rastućem ASCII redosledu).

Primer

%sampleHash = ('name' => 'Bob', 'phone' => '111-111-1111', 'cell' => '222-222-2222');

print %sampleHash;

print "\n";

$result = delete($sampleHash{'phone'});

print %sampleHash;

print "\n";

print "$result\n";

Primer

```
@myNames = ('jacob', 'alexander', 'ethan', 'andrew');
print "@myNames\n";
$result = delete(@myNames[2]);
print "@myNames\n";
print "$result\n";
```

Primer

```
my @array = (1..10);
foreach (0..$#array) {
delete $array[$_] if $_ % 2;
}

# print the array
foreach (0..$#array) {
print $array[$_], ' ' if exists $array[$_];
}

print "\n\n";
use Data::Dumper;
print Dumper(\@array);
```

Primer

```
my %phoneNumbers = ("Anne", "408566", "Paul", "233375", "Marie", "217302");
delete $phoneNumbers {"Paul"};
delete $phoneNumbers {"John"};

# define a reference to a hash
my $phoneNumbersRef = \%phoneNumbers;
delete $$phoneNumbersRef{"Anne"};
# delete $phoneNumbersRef->{"Anne"};

#print the hash (keys, values)
foreach my $key (keys % phoneNumbers) {
print "Key: $key, Value: $phoneNumbers{$key}\n";
}
```

Dodatak

delete EXPR

Given an expression that specifies an element or slice of a hash, delete deletes the specified elements from that hash so that exists() on that element no longer returns true. Setting a hash element to the undefined value does not remove its key, but deleting it does; see exists.

In list context, returns the value or values deleted, or the last such element in scalar context. The return list's length always matches that of the argument list: deleting non-existent elements returns the undefined value in their corresponding positions.

delete() may also be used on arrays and array slices, but its behavior is less straightforward. Although exists() will return false for deleted entries, deleting array elements never changes indices of existing values; use shift() or splice() for that. However, if all deleted elements fall at the end of an array, the array's size shrinks to the position of the highest element that still tests true for exists(), or to 0 if none do.

WARNING: Calling delete on array values is deprecated and likely to be removed in a future version of Perl.

Deleting from %ENV modifies the environment. Deleting from a hash tied to a DBM file deletes the entry from the DBM file. Deleting from a tied hash or array may not necessarily return anything; it depends on the implementation of the tied package's DELETE method, which may do whatever it pleases.

The delete local EXPR construct localizes the deletion to the current block at run time. Until the block exits, elements locally deleted temporarily no longer exist. See Localized deletion of elements of composite types in perlsub.

```
%hash = (foo => 11, bar => 22, baz => 33);
$scalar = delete $hash{foo}; # $scalar is 11
$scalar = delete @hash{qw(foo bar)}; # $scalar is 22
@array = delete @hash{qw(foo baz)}; # @array is (undef,33)
```

The following (inefficiently) deletes all the values of %HASH and @ARRAY:

```
foreach $key (keys %HASH) {
delete $HASH{$key};
}
foreach $index (0 .. $#ARRAY) {
delete $ARRAY[$index];
}
```

And so do these:

```
delete @HASH{keys %HASH};
delete @ARRAY[0 .. $#ARRAY];
```

But both are slower than assigning the empty list or undefining %HASH or @ARRAY, which is the customary way to empty out an aggregate:

```
%HASH = (); # completely empty %HASH
undef %HASH; # forget %HASH ever existed
@ARRAY = (); # completely empty @ARRAY
undef @ARRAY; # forget @ARRAY ever existed
```

The EXPR can be arbitrarily complicated provided its final operation is an element or slice of an aggregate:

```
delete $ref->[$x][$y]{$key};
delete @{$ref->[$x][$y]}{$key1, $key2, @morekeys};
delete $ref->[$x][$y][$index];
delete @{$ref->[$x][$y]}[$index1, $index2, @moreindices];
```

Subroutines

Cilj

- Upoznavanja sa Perl podprogramima

Subroutines

Subroutines

Podprogrami

Podprogrami dozvoljavaju programerima bolje struktuiranje koda, njegovu bolju organizaciju i ponovno korišćenje (engl. reuse).

Podprogram počinje sa ključnom rečju sub. Dalje je dat podprogram koji sračunava sumu dva broja:

```
#!/usr/bin/perl -w
$var1 = 100;
$var2 = 200;
$result = 0;
$result = my_sum();
print "$result\n";
sub my_sum {
$tmp = $var1 + $var2;
return $tmp;
}
```

Podprogrami se deklarišu koristeći jedan od sledećih formata:

```
sub name
{
block
}
sub name (proto )
{
```

block

}

Prototipovi dozvoljavaju postavljanje ograničenja na argument.

Mogu se kreirati anonimni podprogrami u vreme izvršavanja , koji će biti dostupni za korišćenje preko referenci:

$

subref

= sub {

block

};

Dodatak 1

Calling subroutines

The ampersand is the identifier used to call subroutines. Most of the time, however, subroutines can be used in an expression just like built-in functions.

To call subroutines directly:

```
name (args);  # & is optional with parentheses
nameargs
;             # Parents optional if predeclared/imported
&
name;
              # Passes current @_ to subroutine
```

Slika 1 Primer 1

To call subroutines indirectly (by name or by reference):

```
&$subref (args);      # & is not optional on indirect call
&$subref;             # Passes current @_ to subroutine
```

Slika 2 Primer 2

Passing arguments

Note: Subroutines might have parameters. When passing parameters to subroutines, it will be stored in @_ array. Do not confuse it with $_ which stores elements of an array in a loop.

All arguments to a subroutine are passed as a single, flat list of scalars, and return values are returned the same way. Any arrays or hashes passed in these lists will have their values interpolated into the flattened list.

Any arguments passed to a subroutine come in as the array @_. You may use the explicit return statement to return a value and leave the subroutine at any point. Sometimes we need to transmit parameters to our script files.

@ARGV is an array reserved for parameters transmitted to files (default value of number of arguments is set -1 if no parameters are transmitted.

```
#!/usr/bin/perl -w
if ($#ARGV < 2) {
print "You must have at least 3 parameters.\n";
}
else {
print "Your parameters are: @ARGV[0..$#ARGV]\n";
}
```

Passing references

If you want to pass more than one array or hash into or out of a function and have them maintain their integrity, then you will want to pass references as arguments. The simplest way to do this is to take your named variables and put a backslash in front of them in the argument list:

```
@returnlist = ref_conversion (
@temps1, \@temps2, \@temps3
);
```

This sends references to the three arrays to the subroutine (and saves you the step of creating your own named references to send to the function). The references to the arrays are passed to the subroutine as the three-member @_ array.

The subroutine will have to dereference the arguments so that the data values may be used. Returning references is a simple matter of returning scalars that are references. This way you can return distinct hashes and arrays.

Dodatak 2

Perl subroutine

A Perl subroutine or function is a group of statements that together perform a task. You can divide up your code into separate subroutines. How you divide up your code among different subroutines is up to you, but logically the division usually is so each function performs a specific task. Perl uses the terms subroutine, method and function interchangeably.

Define and Call a Subroutine:

The general form of a subroutine definition in Perl programming language is as follows:

```
sub subroutine_name{
body of the subroutine
}
```

The typical way of calling that Perl subroutine is as follows:

subroutine_name(list of arguments);

In versions of Perl before 5.0 the syntax for calling subroutines was slightly different as shown below. This still works in the newest versions of Perl, but it is not recommended since it bypasses subroutine prototypes.

subroutine_name(list of arguments);

Let's have a look into the following example which defines a simple function and then call it. Because Perl compiles your program before executing it, it doesn't matter where you declare your subroutine.

```
#!/usr/bin/perl

# Function definition
sub Hello{
print "Hello, World!\n";
}

# Function call
Hello();
```

When above program is executed, it produces following result:

```
Hello, World!
```

Passing Arguments to a Subroutine:

You can pass various arguments to a subroutine like you do in any other programming language and they can be accessed inside the function using the special array @_. Thus the first argument to the function is in $_[0], the second is in $_[1], and so on.

You can pass arrays and hashes as arguments like any scalar but passing more than one array or hash normally causes them to lose their separate identities. So we will use references (explained in the next chapter) to pass any array or hash.

Let's try following example which takes a list of numbers and then prints their average:

```
#!/usr/bin/perl

# Function definition
sub Average{
# get total number of arguments passed.
$n = scalar(@_);
$sum = 0;

foreach $item (@_){
$sum += $item;
}
$average = $sum / $n;
```

```
print "Average for the given numbers : $average\n";
}

# Function call
Average(10, 20, 30);
```

Display

```
Average for the given numbers : 20
```

Passing lists to subroutines

Because the @_ variable is an array, it can be used to supply lists to a subroutine. However, because of the way in which Perl accepts and parses lists and arrays, it can be difficult to extract the individual elements from @_. If you have to pass a list along with other scalar arguments then make list as the last argument as shown below:

```
#!/usr/bin/perl
# Function definition
sub PrintList{
my @list = @_;
print "Given list is @list\n";
}
$a = 10;
@b = (1, 2, 3, 4);

# Function call with list parameter
PrintList($a, @b);
```

Display:

```
Given list is 10 1 2 3 4
```

Passing hashes to subroutines

When you supply a hash to a subroutine or operator that accepts a list, then hash is automatically translated into a list of key/value pairs. For example:

```
#!/usr/bin/perl
# Function definition
sub PrintHash{
my (%hash) = @_;

foreach my $key ( keys %hash ){
my $value = $hash{$key};
print "$key : $value\n";
}
}
%hash = ('name' => 'Tom', 'age' => 19);
```

```
# Function call with hash parameter

printHash(%hash);
```

When above program is executed, it produces following result:

```
name : Tom
age : 19
```

Returning Value from a Subroutine:

You can return a value from subroutine like you do in any other programming language. If you are not returning a value from a subroutine then whatever calculation is last performed in a subroutine is automatically also the return value.

You can return arrays and hashes from the subroutine like any scalar but returning more than one array or hash normally causes them to lose their separate identities. So we will use references (explained in the next chapter) to return any array or hash from a function.

Let's try following example which takes a list of numbers and then returns their average:

```
#!/usr/bin/perl

# Function definition
sub Average{
# get total number of arguments passed.
$n = scalar(@_);
$sum = 0;

foreach $item (@_){
$sum += $item;
}
$average = $sum / $n;

return $average;
}

# Function call
$num = Average(10, 20, 30);
print "Average for the given numbers : $num\n";
```

When above program is executed, it produces following result:

```
Average for the given numbers : 20
```

Private Variables in a Subroutine:

By default, all variables in Perl are global variables which means they can be accessed from anywhere in the program. But you can create private variables called lexical variables at any time with the my operator.

The my operator confines a variable to a particular region of code in which it can be used and accessed. Outside that region, this variable can not be used or accessed. This region is called its scope. A lexical scope is usually a block of code with a set of braces around it, such as those defining the body of the subroutine or those marking the code blocks of if, while, for, foreach, and eval statements.

Following is an example showing you how to define a single or multiple private variables using my operator:

```
sub somefunc {
my $variable; # $variable is invisible outside somefunc()
my ($another, @an_array, %a_hash); # declaring many variables at once
}
```

Let's check following example to distinguish between global and private variables:

```
#!/usr/bin/perl

# Global variable
$string = "Hello, World!";
# Function definition
sub PrintHello{
# Private variable for PrintHello function
my $string;
$string = "Hello, Perl!";
print "Inside the function $string\n";
}

# Function call
PrintHello();
print "Outside the function $string\n";
```

When above program is executed, it produces following result:

```
Inside the function Hello, Perl!
Outside the function Hello, World!
```

Temporary Values via local()

The local is mostly used when the current value of a variable must be visible to called subroutines. A local just gives temporary values to global (meaning package) variables. This is known as dynamic scoping. Lexical scoping is done with my, which works more like C's auto declarations.

If more than one variable or expression is given to local, they must be placed in parentheses. This operator works by saving the current values of those variables in its argument list on a hidden stack and restoring them upon exiting the block, subroutine, or eval.

Let's check following example to distinguish between global and local variables:

```
#!/usr/bin/perl

# Global variable
$string = "Hello, World!";

sub PrintHello{
# Private variable for PrintHello function
local $string;
$string = "Hello, Perl!";
PrintMe();
print "Inside the function PrintHello $string\n";
}
sub PrintMe{
print "Inside the function PrintMe $string\n";
}

# Function call
PrintHello();
print "Outside the function $string\n";
```

When above program is executed, it produces following result:

```
Inside the function PrintMe Hello, Perl!
Inside the function PrintHello Hello, Perl!
Outside the function Hello, World!
```

State Variables via state()

There another type of lexical variables which are similar to private variable but they maintain their state and they do not get reinitialized upon multiple calls of the subroutines. These variables are defined using state operator and available starting from Perl 5.9.4

Let's check following example to demonstrate the use of state variables:

```
#!/usr/bin/perl
use feature 'state';

sub PrintCount{
state $count = 0; # initial value

print "Value of counter is $count\n";
$count++;
}

for (1..5){
PrintCount();
```

```
}
```

When above program is executed, it produces following result:

```
Value of counter is 0
Value of counter is 1
Value of counter is 2
Value of counter is 3
Value of counter is 4
```

Prior to Perl 5.10 you would have to write it like this:

```
#!/usr/bin/perl

{
my $count = 0; # initial value

sub PrintCount {
print "Value of counter is $count\n";
$count++;
}
}

for (1..5){
PrintCount();
}
```

Subroutine Call Context

The context of a subroutine or statement is defined as the type of return value that is expected. This allows you to use a single function that returns different values based on what the user is expecting to receive. For example, the following localtime() returns a string when it is called in scalar context but it returns a list when it is called in list context.

```
my $datestring = localtime( time );
```

In this example, the value of $timestr is now a string made up of the current date and time, for example, Thu Nov 30 15:21:33 2000. Conversely:

```
($sec,$min,$hour,$mday,$mon, $year,$wday,$yday,$isdst) = localtime(time);
```

Now the individual variables contain the corresponding values returned by localtime() subroutine.

Private and local variables

Cilj

- Upoznavanja sa Perl privatnim i lokalnim varijablama

Private and local variables

Private and local variables

Privatne i lokalne variable

Bilo koje varijable koje se koriste u nekoj funkciji, ako nisu deklarisane kao private, onda su globalne varijable.

U potprogramima, programer često želi da koristi varijable koje se neće koristiti bilo gdje drugo u njegovom program te ne treba da zauzimaju prostor u memoriji kada se potprogram ne izvršava. Takođe nekad se ne želi promena varijable u potprogramu koje imaju isto ime kao i globalne varijable.

Funkcija my definiše varijable leksički u domenu se potprogramom. To su privatne varijable koje se nalaze unutar bloka ili se definišu u potprogramu.

Izvan svog domena, one su nevidljive i ne mogu se menjati na bilo koji način. Da se odmah dobije više varijabli u ovakom domenu, treba koristiti listu u zagradama. Mogu se dodeliti variable u my izrazu:

my @list = (44, 55, 66);

my $cd = "orb";

Dinamičke varijable su vidljive drugim podprogramima koji se pozivaju u okviru svoga domena. Dinamičke varijable se definišu kao lokalne i one nisu private varijable. One su globalne varijable sa privremenim vrednostima.

Kada se poziva podprogram, globalna vrednost je sakrivena i koristi se lokalna vrednost. Kada se jednom izađe iz domena, koriste se originalna globalna vrednost..

Najčešće programer želi da koristi my operatora kada se lokalizuju parametri u podprogramu.

Dodatak

Variables

An array variable holds a single list value (zero or more scalar values). Array variable names are similar to

scalar variable names, differing only in the initial character, which is an at sign (@) rather than a dollar

sign ($). For example:

@fred # the array variable @fred

@A_Very_Long_Array_Variable_Name

@A_Very_Long_Array_Variable_Name_that_is_different

Note that the array variable @fred is unrelated to the scalar variable $fred. Perl maintains separate namespaces for different types of things.

The value of an array variable that has not yet been assigned is (), the empty list. An expression can refer to array variables as a whole, or it can examine and modify individual elements

of the array

Assignments

@fred = (1,2,3); # The fred array gets a three-element literal

@barney = @fred; # now that is copied to @barney

If a scalar value is assigned to an array variable, the scalar value becomes the single element of an array:

@huh = 1; # 1 is promoted to the list (1) automatically

Array variable names may appear in a list literal list. When the value of the list is computed, Perl replaces the names with the current values of the array, like so:

@fred = qw(one two);

@barney = (4,5,@fred,6,7); # @barney becomes

(4,5,"one","two",6,7)

@barney = (8,@barney); # puts 8 in front of @barney

@barney = (@barney,"last");# and a "last" at the end

@barney is now (8,4,5,"one","two",6,7,"last")

Note that the inserted array elements are at the same level as the rest of the literal: a list cannot contain another list as an element.

Example

($a,$b,$c) = (1,2,3); # give 1 to $a, 2 to $b, 3 to $c

($a,$b) = ($b,$a); # swap $a and $b

($d,@fred) = ($a,$b,$c); # give $a to $d, and ($b,$c) to @fred

($e,@fred) = @fred; # remove first element of @fred to $e

this makes @fred = ($c) and $e = $b

Dodatak

Dynamic Scoping.

It is a neat concept. Many people don't use it, or understand it.

Basically think of my as creating and anchoring a variable to one block of {}, A.K.A. scope.

my $foo if (true); # $foo lives and dies within the if statement.

So a my variable is what you are used to. Whereas with dynamic scoping $var can be declared anywhere and used anywhere. So with local you basically suspend the use of that global variable, and use a "local value" to work with it. So local creates a temporary scope for a temporary variable.

```
$var = 4;
print $var, "\n";
&hello;
print $var, "\n";

# subroutines
sub hello {
    local $var = 10;
    print $var, "\n";
    &gogo; # calling subroutine gogo
    print $var, "\n";
}
sub gogo {
    $var ++;
}
```

Display

```
4
10
11
4
```

Slika 1 Primer 1

Dodatak

Private Variables via my()

Synopsis:

my $foo; # declare $foo lexically local

my (@wid, %get); # declare list of variables local

my $foo = "flurp"; # declare $foo lexical, and init it

my @oof = @bar; # declare @oof lexical, and init it

my $x : Foo = $y; # similar, with an attribute applied

WARNING: The use of attribute lists on my declarations is still evolving. The current semantics and interface are subject to change. See attributes and Attribute::Handlers.

The my operator declares the listed variables to be lexically confined to the enclosing block, conditional (if/unless/elsif/else), loop (for/foreach/while/until/continue), subroutine, eval, or do/require/use'd file. If more than one value is listed, the list must be placed in parentheses. All listed elements must be legal lvalues. Only alphanumeric identifiers may be lexically scoped--magical built-ins like $/ must currently be localized with local instead.

Unlike dynamic variables created by the local operator, lexical variables declared with my are totally hidden from the outside world, including any called subroutines. This is true if it's the same subroutine called from itself or elsewhere--every call gets its own copy.

This doesn't mean that a my variable declared in a statically enclosing lexical scope would be invisible. Only dynamic scopes are cut off. For example, the bumpx() function below has access to the lexical $x variable because both the my and the sub occurred at the same scope, presumably file scope.

my $x = 10;

sub bumpx { $x++ }

An eval(), however, can see lexical variables of the scope it is being evaluated in, so long as the names aren't hidden by declarations within the eval() itself. See perlref.

The parameter list to my() may be assigned to if desired, which allows you to initialize your variables. (If no initializer is given for a particular variable, it is created with the undefined value.) Commonly this is used to name input parameters to a subroutine.

Example

$arg = "fred"; # "global" variable

$n = cube_root(27);

print "$arg thinks the root is $n\n";

fred thinks the root is 3

sub cube_root {

my $arg = shift; # name doesn't matter

$arg **= 1/3;

return $arg;

}

The my is simply a modifier on something you might assign to. So when you do assign to variables in its argument list, my doesn't change whether those variables are viewed as a scalar or an array. So

my ($foo) = <STDIN>; # WRONG?

my @FOO = <STDIN>;

both supply a list context to the right-hand side, while

my $foo = <STDIN>;

supplies a scalar context. But the following declares only one variable:

my $foo, $bar = 1; # WRONG

That has the same effect as

my $foo;

$bar = 1;

The declared variable is not introduced (is not visible) until after the current statement. Thus,

my $x = $x;

can be used to initialize a new $x with the value of the old $x, and the expression

my $x = 123 and $x == 123

is false unless the old $x happened to have the value 123 .

Lexical scopes of control structures are not bounded precisely by the braces that delimit their controlled blocks; control expressions are part of that scope, too. Thus in the loop

```
while (my $line = <>) {
$line = lc $line;
} continue {
print $line;
}
```

the scope of $line extends from its declaration throughout the rest of the loop construct (including the continue clause), but not beyond it. Similarly, in the conditional

```
if ((my $answer = <STDIN>) =~ /^yes$/i) {
user_agrees();
} elsif ($answer =~ /^no$/i) {
user_disagrees();
} else {
chomp $answer;
die "'$answer' is neither 'yes' nor 'no'";
}
```

the scope of $answer extends from its declaration through the rest of that conditional, including any elsif and else clauses, but not beyond it. See Simple Statements in perlsyn for information on the scope of variables in statements with modifiers.

The foreach loop defaults to scoping its index variable dynamically in the manner of local. However, if the index variable is prefixed with the keyword my, or if there is already a lexical by that name in scope, then a new lexical is created instead. Thus in the loop:

```
for my $i (1, 2, 3) {
some_function();
}
```

the scope of $i extends to the end of the loop, but not beyond it, rendering the value of $i inaccessible within some_function() .

Some users may wish to encourage the use of lexically scoped variables. As an aid to catching implicit uses to package variables, which are always global, if you say:

```
use strict 'vars';
```

then any variable mentioned from there to the end of the enclosing block must either refer to a lexical variable, be pre-declared via our or use vars , or else must be fully qualified with the package name. A compilation error results otherwise. An inner block may countermand this with no strict 'vars' .

A my has both a compile-time and a run-time effect. At compile time, the compiler takes notice of it. The principal usefulness of this is to quiet use strict 'vars' , but it is also essential for generation of closures as detailed in perlref. Actual initialization is delayed until run time, though, so it gets executed at the appropriate time, such as each time through a loop, for example.

Variables declared with my are not part of any package and are therefore never fully qualified with the package name. In particular, you're not allowed to try to make a package variable (or other global) lexical:

my $pack::var; # ERROR! Illegal syntax

In fact, a dynamic variable (also known as package or global variables) are still accessible using the fully qualified :: notation even while a lexical of the same name is also visible:

package main;

local $x = 10;

my $x = 20;

print "$x and $::x\n";

That will print out 20 and 10.

You may declare my variables at the outermost scope of a file to hide any such identifiers from the world outside that file. This is similar in spirit to C's static variables when they are used at the file level. To do this with a subroutine requires the use of a closure (an anonymous function that accesses enclosing lexicals). If you want to create a private subroutine that cannot be called from outside that block, it can declare a lexical variable containing an anonymous sub reference:

```
my $secret_version = '1.001-beta';
my $secret_sub = sub { print $secret_version };
&$secret_sub();
```

Slika 2 Primer 2

As long as the reference is never returned by any function within the module, no outside module can see the subroutine, because its name is not in any package's symbol table. Remember that it's not REALLY called $some_pack::secret_version or anything; it's just $secret_version, unqualified and unqualifiable. This does not work with object methods, however; all object methods have to be in the symbol table of some package to be found..

Dodatak

Private Variables in Functions

One can create your own scalar, array, and hash variables that work the same way. You do this with the my operator, which takes a list of variable names and creates local versions of them (or *instantiations*, if you like bigger words). Here's that add function again, this time using my :

sub add {

my ($sum); # make $sum a local variable

$sum = 0; # initialize the sum

```
foreach $_ (@_) {
$sum += $_; # add each element
}
return $sum; # last expression evaluated: sum of all elements
}
```

Example

```
sub bigger_than_100 {
my (@result); # temporary for holding the return value
foreach $_ (@_) { # step through the arg list
if ($_ > 100) { # is it eligible?
push(@result,$_); # add it
}
}
return @result; # return the final list
}
```

Prototypes

Cilj

- Upoznavanja sa Perl prototipovima

Prototypes

Prototypes

Prototipovi dozvoljavaju kreiranje podprograma koji uzimaju argument sa ogranjčenjem na broj parametara i tipove podataka. Za definisanje funkcije sa prototipovima, koristiti simbole u linijji za deklaraciju kao na primer:

```
sub addem ($$) {
...
}
```

U ovom slučaju, funkcija očekuje dva skalarna argumenta. Slike 1 ilustruje različite tipove simbola prototipova:

Symbol	Meaning
$	Scalar
@	List
%	Hash
&	Anonymous subroutine
*	Typeglob

Slika 1 Simpoli prototipova

Dodatak

A backslash placed before one of these symbols forces the argument to be that exact variable type. For instance, a function that requires a hash variable would be declared like this:

sub hashfunc (\%);

Un backslashed @ or % symbols act exactly alike, and will eat up all remaining arguments, forcing list context. A $ likewise forces scalar context on an argument, so taking an array or hash variable for that parameter would probably yield unwanted results.

A semicolon separates mandatory arguments from optional arguments. For example:

sub newsplit (\@$;$);

requires two arguments: an array variable and a scalar. The third scalar is optional. Placing a semicolon before @ and % is not necessary since lists can be null.

A typeglob prototype symbol (*) will always turn its argument into a reference to a symbol table entry. It is most often used for filehandles.

References and complex data structures

A Perl reference is a fundamental data type that "points" to another piece of data or . A reference knows the location of the information and what type of data is stored there.

A reference is a scalar and can be used anywhere a scalar can be used. Any array element or hash value can contain a reference (a hash key cannot contain a reference), and this is how nested data structures are built in Perl. You can construct lists containing references to other lists, which can contain references to hashes, and so on.

Creating references

Cilj

- Upoznavanja sa načinom kreiranja referenci

Creating References

Creating References

Kreiranje referenci

Referenca se kreira za postojeće variable ili podprograme sa prefiksom koristeći obrnutu kosu crtu (engl. backslash) (\):

```
$a = "fondue";
@alist = ("pitt", "hanks", "cage", "cruise");
%song = ("mother" => "crying", "brother" => "dying");
sub freaky_friday {s/mother/daughter/}
# Create references
$ra = \$a;
$ralist = \@alist;
$rsong = \%song;
$rsub = \&freaky_friday;    # '&' pravilo ya imena podp
```

Slika 1 Primer 1

Reference za skalarne konstante se kreira slično na sledeći način:

$pi = \3.14159;

$myname = \"Charlie";

Sve reference imaju prefix $, čak i kada se odnose na niz ili heš. Sve referene su skalari, tako da se može kopirati referenca na drugi skalar ili čak referenca na drugu referencu.

```
$aref = \@names;
$bref = $aref;  # both refer to @names
$cref = \$aref; # $cref is a reference to $aref
```

Slika 2 Primer 2

Zbog toga što su nizovi i heševi skupovi skalara, mogu se kreirati reference na pojedine elemente dodavanjem prefiksa na njihova imena sa obrnutim kosim crtama \.

```
$star = \$alist[2];          # refers to third element of @alis
$action = \$song{mother};  # refers to the 'mother' value of %
```

Slika 3 Primer 3

Dodatak

Making References

References can be created in several ways.

1.

By using the backslash operator on a variable, subroutine, or value. This typically creates another reference to a variable, because there's already a reference to the variable in the symbol table. But the symbol table reference might go away, and you'll still have the reference that the backslash returned. Here are some examples:

```
$scalarref = \$foo;
$arrayref  = \@ARGV;
$hashref   = \%ENV;
$coderef   = \&handler;
$globref   = \*foo;
```

Slika 4 Primer 4

It isn't possible to create a true reference to an IO handle (filehandle or dirhandle) using the backslash operator. The most you can get is a reference to a typeglob, which is actually a complete symbol table entry. But see the explanation of the *foo{THING} syntax below. However, you can still use type globs and globrefs as though they were IO handles.

2.

A reference to an anonymous array can be created using square brackets:

$arrayref = [1, 2, ['a', 'b', 'c']];

Here we've created a reference to an anonymous array of three elements whose final element is itself a reference to another anonymous array of three elements. (The multidimensional syntax described later can be used to access this. For example, after the above, $arrayref->[2][1] would have the value "b".)

Taking a reference to an enumerated list is not the same as using square brackets--instead it's the same as creating a list of references!

```
@list = (\$a, \@b, \%c);
@list = \($a, @b, %c);     # same thing!
```

As a special case, \(@foo) returns a list of references to the contents of @foo , not
Likewise for %foo , except that the key references are to copies (since the keys are
fledged scalars).

Slika 5 Primer 5

3.

A reference to an anonymous hash can be created using curly brackets:

$hashref = {

'Adam' => 'Eve',

'Clyde' => 'Bonnie',

};

4.

A reference to an anonymous subroutine can be created by using sub without a subname:

$coderef = sub { print "Boink!\n" };

Note the semicolon. Except for the code inside not being immediately executed, a sub {} is not so much a declaration as it is an operator, like do{} or eval{}. (However, no matter how many times you execute that particular line (unless you're in an eval("...")), $coderef will still have a reference to the same anonymous subroutine.)

Anonymous subroutines act as closures with respect to my() variables, that is, variables lexically visible within the current scope. Closure is a notion out of the Lisp world that says if you define an anonymous function in a particular lexical context, it pretends to run in that context even when it's called outside the context.

In human terms, it's a funny way of passing arguments to a subroutine when you define it as well as when you call it. It's useful for setting up little bits of code to run later, such as callbacks. You can even do object-oriented stuff with it, though Perl already provides a different mechanism to do that--see perlobj.

You might also think of closure as a way to write a subroutine template without using eval(). Here's a small example of how closures work:

```
sub newprint {
   my $x = shift;
   return sub { my $y = shift; print "$x, $y!\n";
}
$h = newprint("Howdy");
$g = newprint("Greetings");
# Time passes...
&$h("world");
&$g("earthlings");
```

Slika 6 Primer 6

Display

Howdy, world!

Greetings, earthlings!

Note particularly that $x continues to refer to the value passed into newprint() despite "my $x" having gone out of scope by the time the anonymous subroutine runs. That's what a closure is all about.

This applies only to lexical variables, by the way. Dynamic variables continue to work as they have always worked. Closure is not something that most Perl programmers need trouble themselves about to begin with.

5.

References are often returned by special subroutines called constructors. Perl objects are just references to a special type of object that happens to know which package it's associated with. Constructors are just special subroutines that know how to create that association. They do so by starting with an ordinary reference, and it remains an ordinary reference even while it's also being an object. Constructors are often named new() . You can call them indirectly:

```
$objref = new Doggie( Tail => 'short', Ears => 'long' );
```

But that can produce ambiguous syntax in certain cases, so it's often better to use the direct method invocation approach:

```
$objref = Doggie->new(Tail => 'short', Ears => 'long');

use Term::Cap;

$terminal = Term::Cap->Tgetent( { OSPEED => 9600 });

use Tk;

$main = MainWindow->new();

$menubar = $main->Frame(-relief => "raised", -borderwidth => 2)
```

6.

References of the appropriate type can spring into existence if you dereference them in a context that assumes they exist. Because we haven't talked about dereferencing yet, we can't show you any examples yet.

7.

A reference can be created by using a special syntax, lovingly known as the *foo{THING} syntax. *foo{THING} returns a reference to the THING slot in *foo (which is the symbol table entry which holds everything known as foo).

```
$scalarref = *foo{SCALAR};

$arrayref = *ARGV{ARRAY};

$hashref = *ENV{HASH};

$coderef = *handler{CODE};

$ioref = *STDIN{IO};

$globref = *foo{GLOB};

$formatref = *foo{FORMAT};

$globname = *foo{NAME}; # "foo"

$pkgname = *foo{PACKAGE}; # "main"
```

Most of these are self-explanatory, but *foo{IO} deserves special attention. It returns the IO handle, used for file handles (open), sockets (socket and socketpair), and directory handles (opendir). For compatibility with previous versions of Perl, *foo{FILEHANDLE} is a synonym

for *foo{IO} , though it is deprecated as of 5.8.0. If deprecation warnings are in effect, it will warn of its use.

*foo{THING} returns undef if that particular THING hasn't been used yet, except in the case of scalars. *foo{SCALAR} returns a reference to an anonymous scalar if $foo hasn't been used yet. This might change in a future release.

*foo{NAME} and *foo{PACKAGE} are the exception, in that they return strings, rather than references. These return the package and name of the typeglob itself, rather than one that has been assigned to it. So, after *foo=*Foo::bar , *foo will become "*Foo::bar" when used as a string, but *foo{PACKAGE} and *foo{NAME} will continue to produce "main" and "foo", respectively.

*foo{IO} is an alternative to the *HANDLE mechanism given in Typeglobs and Filehandles in perldata for passing filehandles into or out of subroutines, or storing into larger data structures. Its disadvantage is that it won't create a new filehandle for you. Its advantage is that you have less risk of clobbering more than you want to with a typeglob assignment. (It still conflates file and directory handles, though.) However, if you assign the incoming value to a scalar instead of a typeglob as we do in the examples below, there's no risk of that happening.

splutter(*STDOUT); # pass the whole glob

splutter(*STDOUT{IO}); # pass both file and dir handles

sub splutter {

my $fh = shift;

print $fh "her um well a hmmm\n";

}

$rec = get_rec(*STDIN); # pass the whole glob

$rec = get_rec(*STDIN{IO}); # pass both file and dir handles

sub get_rec {

my $fh = shift;

return scalar <$fh>;

}

Perl file handling

Cilj

- Upoznavanja sa identifikatorom Perl datoteka

Perl file handling

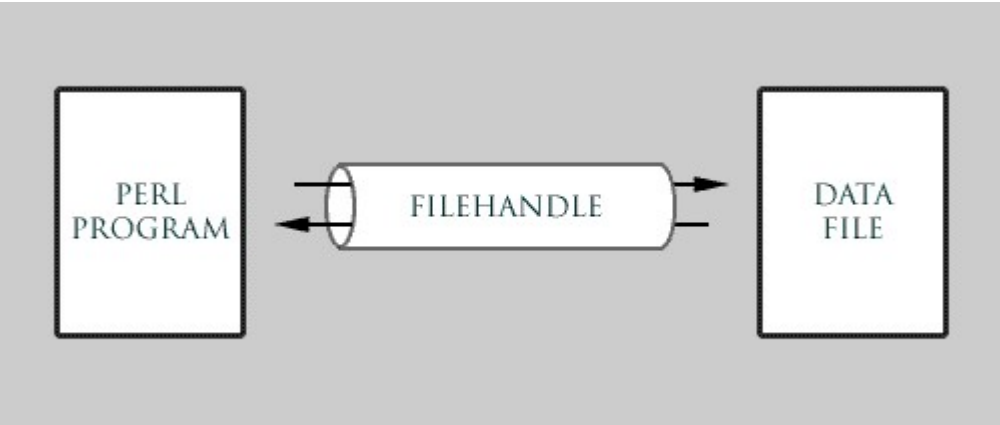
Perl file handling

Identifikator datoteke

Šta je identifikator datoteke?

U Perl, kada se program "poveže" sa vanjskom datotekom podataka, Perl označava tu vezu (ne na samu datoteku!) sa labelom koja se zove Identifikator datoteke (engl. filehandle).

Programer bira ime identifikatora i svaki ulaz ili izlaz u program se prenosi preko veze označene kao filehandle, Slika 1. Tako iygleda kao da čitanje i pisanje ide direktno u filehandle, a ne u samu datoteku.

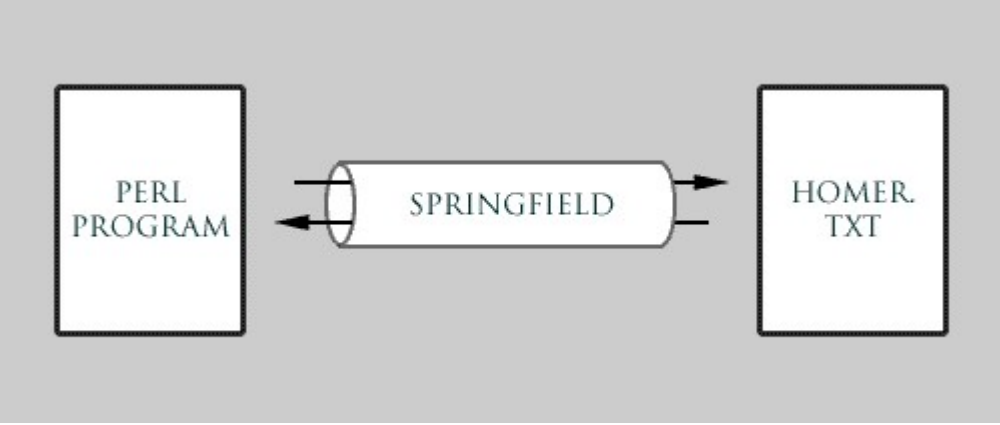


Slika 1 Identifikator datoteke

Filehandle omogućava takođe komunikaciju jednog programa sa drugim programima. Filehandle je ime I/O veze između Perl procesa i operativnog sistema. Filehandle imena su kao imena labela, ali koriste svoj prostor imena (engl. namespace). Kao i imena labela, veze koriste velika slova u svom imenu.

Svaki Perl program ima tri filehandle veza koje se automatski otvaraju: STDIN, STDOUT i STDERR. Po pravilu, print i write funkcije upisuju u STDOUT. Ostale filehandles se kreiraju koristeći funkciju open:

open (DATA, "numbers.txt");



Slika 2 Otvaranje i zatvaranje datoteka

DATA je novi filehandle koji se dodaje na vanjsku datoteku, koja je sada otvorena za čitanje. Može se otvoriti filehandle za čitanje, pisanje i dodavanje (engl. appending) vanjskim datotekama i uređajima.

Za otvaranje datoteke za pisanje, koristi se prefiks na ime datoteke u obliku “veći-nego” (>) znaka

open(OUT, ">outfile");

Za otvaranje datoteke za dodavanje, prefiks na ime datoteke je dvostruki “veći-nego” znak:

open(LOGFILE, ">>error_log");

Funkcija open vraća true ako je datoteka uspešno otvorena, a false ako nije uspešno otvorena.

Otvaranje datoteke može biti neuspešno iz više razloga: datoteka ne postoji, datoteka je zaštićena od pisanja, ili korisnik nema dozvolu za datoteku ili direktorij. Međutim, filehandle koji nije uspešno otvoren još uvijek se može čitati od (dajući direktan EOF) ili pisanje, bez vidljivih učinaka

Treba se uvijek odmah proveriti rezultat otvoranja i izvestiti o grešci, ako operacija ne uspije.

Funkcija **warn** daje izveštaj o grešci , ako nešto pođe po zlu, a funckija die može prekinuti program i reći što je pošlo po zlu.

Example

open(LOGFILE, "/usr/httpd/error_log") || warn "Could not open /usr/httpd/error_log.\n";

open(DATA, ">/tmp/data") || die "Could not create /tmp/data\n.";

Dodatak

Once the file is opened, you can access the data using the diamond operator, < *filehandle* >. This is the line-input operator. When used on a filehandle in a scalar context, it will return a line from a filehandle as a string. Each time it is called it will return the next line from the filehandle, until it reaches the end-of-file. The operator keeps track of which line it is on in the file, unless the filehandle is closed and reopened, resetting the operator to the top-of-file.

For example, to print any line containing the word "secret.html" from the LOGFILE filehandle:

while (<LOGFILE>) {

print "$_\n" if /secret\.html/;

}

In a list context, the line-input operator returns a list in which each line is an element. The empty <>operator reads from the ARGV filehandle, which reads the array of filenames from the Perl command line. If this filehandle is empty, the operator resorts to standard input.

A number of functions send output to a filehandle. The filehandle must already be opened for writing, of course. In the previous example, print wrote to the STDOUT filehandle, even though it wasn't specified. Without a filehandle, print defaults to the currently selected output filehandle, which will be STDOUT until you open and select another one in your program. See the selectfunction (filehandle version) for more information.

If your program involves more than a couple of open filehandles, you should be safe and specify the filehandles for all of your IO functions:

print LOGFILE "======= Generated report $date =======";

To close a filehandle, use the close function. Filehandles are also closed when the program exits.

Primer

SYNOPSIS

use FileHandle;

```
$fh = FileHandle->new;
if ($fh->open("< file")) {
print <$fh>;
$fh->close;
}
$fh = FileHandle->new("> FOO");
if (defined $fh) {
print $fh "bar\n";
$fh->close;
}
$fh = FileHandle->new("file", "r");
if (defined $fh) {
print <$fh>;
undef $fh; # automatically closes the file
}
$fh = FileHandle->new("file", O_WRONLY|O_APPEND);
if (defined $fh) {
print $fh "corge\n";
undef $fh; # automatically closes the file
}
$pos = $fh->getpos;
$fh->setpos($pos);
$fh->setvbuf($buffer_var, _IOLBF, 1024);
($readfh, $writefh) = FileHandle::pipe;
autoflush STDOUT 1;
```

FileHandle::open accepts one parameter or two. With one parameter, it is just a front end for the built-in open function. With two parameters, the first parameter is a filename that may include whitespace or other special characters, and the second parameter is the open mode, optionally followed by a file permission value.

If FileHandle::open receives a Perl mode string (">", "+<", etc.) or a POSIX fopen() mode string ("w", "r+", etc.), it uses the basic Perl open operator.

If FileHandle::open is given a numeric mode, it passes that mode and the optional permissions value to the Perl sysopen operator. For convenience, FileHandle::import tries to import the O_XXX constants from the Fcntl module. If dynamic loading is not available, this may fail, but the rest of FileHandle will still work.

FileHandle::fdopen is like open except that its first parameter is not a filename but rather a file handle name, a FileHandle object, or a file descriptor number.

If the C functions fgetpos() and fsetpos() are available, then FileHandle::getpos returns an opaque value that represents the current position of the FileHandle, and FileHandle::setpos uses that value to return to a previously visited position.

If the C function setvbuf() is available, then FileHandle::setvbuf sets the buffering policy for the FileHandle. The calling sequence for the Perl function is the same as its C counterpart, including the macros _IOFBF, _IOLBF , and _IONBF, except that the buffer parameter specifies a scalar variable to use as a buffer.

WARNING:

A variable used as a buffer by FileHandle::setvbuf must not be modified in any way until the FileHandle is closed or until FileHandle::setvbuf is called again, or memory corruption may result!

See perlfunc for complete descriptions of each of the following supported FileHandle methods, which are just front ends for the corresponding built-in functions:

close

fileno

getc

gets

eof

clearerr

seek

tell

Example

UNIX cat command

```
#!/usr/local/bin/perl
#
# Program to open the password file, read it in,
# print it, and close it again.

$file = '/etc/passwd';     # Name the file
open(INFO, $file);         # Open the file
@lines = <INFO>;           # Read it into an array
close(INFO);               # Close the file
print @lines;              # Print the array
```

Slika 3 primer

Example

```
open(INFO, $file); # Open for input
open(INFO, ">$file"); # Open for output
open(INFO, ">>$file"); # Open for appending
open(INFO, "<$file"); # Also open for input
```

Perl assigning handles

Cilj

- Upoznavanja sa načinom na kojiPerl dodeljuje identifikatore

Perl assigning handles

Perl assigning handles

Perl dodeljuje identifikatore

Filehandle ili identifikatior je ništa više od nadimka za datoteku koja se koristi u PERL skriptu i programima. Handle je privremeno ime koje se dodeljuje datoteci . Sledeći primer pokazuje kako se koristi filehandle u PERL kodu.

Primer

```
#!/usr/bin/perl
print "content-type: text/html \n\n"; #The header
$FilePath = "home/html/myhtml.html"
sysopen(HANDLE, $FilePath, O_RDWR);
printf HANDLE "Welcome to Tizag!";
close (HANDLE);
```

perl –kreiranje datoteke

Primer

```
#!/usr/bin/perl
use Fcntl;
#The Module
print "content-type: text/html \n\n"; #The header
sysopen (HTML, 'myhtml.html', O_RDWR|O_EXCL|O_CREAT, 0755);
printf HTML "<html>\n";
printf HTML "<head>\n";
printf HTML "<title>My Home Page</title>";
printf HTML "</head>\n";
printf HTML "<body>\n";
printf HTML "<p align='center'>Here we have an HTML
page with a paragraph.</p>";
printf HTML "</body>\n";
printf HTML "</html>\n";
close (HTML);
```

myhtml.html

```
<html>
<head>
<title>My Home Page</title></head>
<body>
<p align='center'>Here we have an HTML page with a paragraph.</p>
</body>
</html>
```

Perl – čitanje datoteke

Primer

```
#!/usr/bin/perl
print "content-type: text/html \n\n"; #The header
$HTML = "myhtml.html";
open (HTML) or die "Can't open the file!";
print <HTML>;
close (HTML);
```

myhtml.pl

Here we have an HTML page with a paragraph.

Example

```
#!/usr/bin/perl

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

$FilePath = "home/html/myhtml.html"
sysopen(HANDLE, $FilePath, O_RDWR);
printf HANDLE "Welcome to Beograd!";

close (HANDLE);
```

Perl input array

Cilj

- Upoznavanja sa karakteristkama Perl ulaznog niza

Perl - input array

Perl input array

Perl ulazni niz

Perl može da štampa podatke drugih datoteka korišćenjem nizova (engl. array). Kada se pozove HTML identifkator datoteke (engl. filehandle) koristeći input operator, Perl automatskii čuva svaku liniju datoteke u globalnom nizu.

Onda može da procesira svaki element niza. Na taj način je moguće integrisati dinamične bitove HTML koda sa već postojećim HTML datotekama.

dynamic.pl:

```
#!/usr/bin/perl
print "content-type: text/html \n\n"; #The header
$HTML = "myhtml.html";
open (HTML) or die "Can't open the file!";
@fileinput = <HTML>;
print $fileinput[0];
print $fileinput[1];
print $fileinput[2];
print $fileinput[3];
print "<table border='1' align='center'><tr>
<td>Dynamic</td><td>Table</td></tr>";
print "<tr><td>Temporarily Inserted</td>
<td>Using PERL!</td></tr></table>";
print $fileinput[4];
print $fileinput[5];
close (HTML);
```

Rezultat, Slika 1.

Dynamic	Table
Temporarily Inserted	Using PERL!

Slika 1 Rezultat

Da bi se permanetno promenila datoteka, treba zameniti funkciju print sa funkcijom printf.

Perl – čitanje iz datoteke

Moguće je čitanje linija iz datoteka i njihovo ubacivanje pomoću <> input operatora. Postavljenjem identifikatora unutar input operatora, skript će učitati liniju u datoteku.

Primer

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

```
print "content-type: text/html \n\n"; #The header
$HTML = "myhtml.html";
open (HTML) or die "Can't open the file!";
print <HTML>;
close (HTML);
```

Dodatak

This page shows different ways to accept user input using perl.

Accepting user input from the command line

There are a number of ways to accept command line input. Firstly you can use the @ARGV array:

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

print $ARGV[0] . "\n";
```

Note that $ARGV[0] actually is the first argument and not the program name.

Or if you need a bit more flexibility, use the Getopt::Std module:

```
#!/usr/bin/perl
use strict;
use warnings;
use Getopt::Std;

my %args;
# -p is just a flag
# -t has a value
getopts('pt:', \%args);

if ($args{p}) {
print "p was passed in.\n";
}
if ($args{t}) {
print "t has a value of: $args{t}\n";
}
```

If you called the above program as follows:

```
./test.pl -p -t hello
```

Display

p was passed in.

t has a value of: hello

Accepting user input from a terminal

You can read directly from standard in with the angle operators <>.

```
#!/usr/bin/perl
use strict;
use warnings;
while (<STDIN>) {
last if ($_ =~ /^\s*$/); # Exit if it was just spaces (or just an enter)
print "You typed: $_";
}
```

You could use the angle operators without the word 'STDIN':

```
while (<>) {..
```

Accepting user input graphically

The code below will display a perl Tk widget, with text field and two buttons. Enter some text in the text field then press the 'Click here' button. You will see the text you typed appear in your terminal. When you are finished click the 'Done' button.

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

my $mw = MainWindow->new;
my $text;

$mw->title("Text Entry");

$mw->Label(
-text => "Enter some text:"
)->pack();

$mw->Entry(
-textvariable => \$text
)->pack();

$mw->Button(
-text => "Click here",
-command =>
sub {
print "You typed: $text\n";
```

```
}
)->pack();

$mw->Button(
-text => "Done",
-command => sub {exit}
)->pack();

MainLoop;

exit 0;
```

Example

<STDIN> returns the next line of input in a scalar context. However, in a list context, it returns all remaining lines up to end of file. Each line is returned as a separate element of the list. For example:

```
@a = <STDIN>; # read standard input in a list context
```

If the person running the program types three lines, then presses CTRL-D (to indicate "end of file"), the array ends up with three elements. Each element will be a string that ends in a newline, corresponding to the three newline-terminated lines entered.

Some systems use CTRL-Z to indicate end of file, while others use it to suspend a running process

Accepting user input from a web form

Use the CGI module, create a CGI object (in the variable $q in our case), and call the CGI method param() to retrieve the form values. The example below also prints out the form:

```
#!/usr/bin/perl
use strict;
use warnings;
use CGI;
use CGI::Carp qw(fatalsToBrowser);

my $q = new CGI;

print $q->header();

print $q->start_html(
-style => qq(
body {font-family: verdana, arial, sans-serif;}
)
);

if ($q->param()) {
```

```
print "Your name is " . $q->param('username') . "<BR>";
print $q->br;
}

# Output the form
print $q->start_form();
print "Name: ";
print $q->textfield(-name => "username");
print $q->br;
print $q->submit(-value => "Click here");
print $q->end_form();
print $q->end_html();

exit 0;
```

Perl assigning handles

Cilj

- Upoznavanja sa načinom na kojiPerl dodeljuje identifikatore

Perl assigning handles

Perl assigning handles

Perl dodeljuje identifikatore

Filehandle ili identifikatior je ništa više od nadimka za datoteku koja se koristi u PERL skriptu i programima. Handle je privremeno ime koje se dodeljuje datoteci . Sledeći primer pokazuje kako se koristi filehandle u PERL kodu.

Primer

```
#!/usr/bin/perl
print "content-type: text/html \n\n"; #The header
$FilePath = "home/html/myhtml.html"
sysopen(HANDLE, $FilePath, O_RDWR);
printf HANDLE "Welcome to Tizag!";
close (HANDLE);
```

perl –kreiranje datoteke

Primer

```
#!/usr/bin/perl
use Fcntl;
```

```
#The Module
print "content-type: text/html \n\n"; #The header
sysopen (HTML, 'myhtml.html', O_RDWR|O_EXCL|O_CREAT, 0755);
printf HTML "<html>\n";
printf HTML "<head>\n";
printf HTML "<title>My Home Page</title>";
printf HTML "</head>\n";
printf HTML "<body>\n";
printf HTML "<p align='center'>Here we have an HTML
page with a paragraph.</p>";
printf HTML "</body>\n";
printf HTML "</html>\n";
close (HTML);
```

myhtml.html

```
<html>
<head>
<title>My Home Page</title></head>
<body>
<p align='center'>Here we have an HTML page with a paragraph.</p>
</body>
</html>
```

Perl – čitanje datoteke

Primer

```
#!/usr/bin/perl
print "content-type: text/html \n\n"; #The header
$HTML = "myhtml.html";
open (HTML) or die "Can't open the file!";
print <HTML>;
close (HTML);
```

myhtml.pl

Here we have an HTML page with a paragraph.

Example

```
#!/usr/bin/perl

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

$FilePath = "home/html/myhtml.html"
```

```
sysopen(HANDLE, $FilePath, O_RDWR);
printf HANDLE "Welcome to Beograd!";

close (HANDLE);
```

Perl input array

Cilj

- Upoznavanja sa karakteristkama Perl ulaznog niza

Perl - input array

Perl input array

Perl ulazni niz

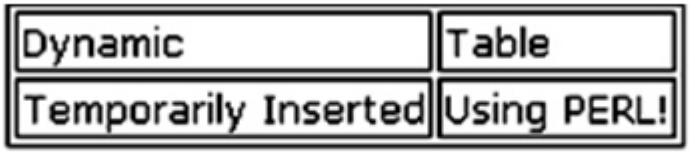
Perl može da štampa podatke drugih datoteka korišćenjem nizova (engl. array). Kada se pozove HTML identifkator datoteke (engl. filehandle) koristeći input operator, Perl automatskii čuva svaku liniju datoteke u globalnom nizu.

Onda može da procesira svaki element niza. Na taj način je moguće integrisati dinamične bitove HTML koda sa već postojećim HTML datotekama.

dynamic.pl:

```
#!/usr/bin/perl
print "content-type: text/html \n\n"; #The header
$HTML = "myhtml.html";
open (HTML) or die "Can't open the file!";
@fileinput = <HTML>;
print $fileinput[0];
print $fileinput[1];
print $fileinput[2];
print $fileinput[3];
print "<table border='1' align='center'><tr>
<td>Dynamic</td><td>Table</td></tr>";
print "<tr><td>Temporarily Inserted</td>
<td>Using PERL!</td></tr></table>";
print $fileinput[4];
print $fileinput[5];
close (HTML);
```

Rezultat, Slika 1.



Slika 1 Rezultat

Da bi se permanetno promenila datoteka, treba zameniti funkciju print sa funkcijom printf.

Perl – čitanje iz datoteke

Moguće je čitanje linija iz datoteka i njihovo ubacivanje pomoću <> input operatora. Postavljenjem identifikatora unutar input operatora, skript će učitati liniju u datoteku.

Primer

```
#!/usr/bin/perl

print "content-type: text/html \n\n"; #The header
$HTML = "myhtml.html";
open (HTML) or die "Can't open the file!";
print <HTML>;
close (HTML);
```

Dodatak

This page shows different ways to accept user input using perl.

Accepting user input from the command line

There are a number of ways to accept command line input. Firstly you can use the @ARGV array:

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

print $ARGV[0] . "\n";
```

Note that $ARGV[0] actually is the first argument and not the program name.

Or if you need a bit more flexibility, use the Getopt::Std module:

```
#!/usr/bin/perl
use strict;
use warnings;
use Getopt::Std;

my %args;
```

```
# -p is just a flag
# -t has a value
getopts('pt:', \%args);

if ($args{p}) {
print "p was passed in.\n";
}
if ($args{t}) {
print "t has a value of: $args{t}\n";
}
```

If you called the above program as follows:

```
./test.pl -p -t hello
```

Display

```
p was passed in.
t has a value of: hello
```

Accepting user input from a terminal

You can read directly from standard in with the angle operators <>.

```
#!/usr/bin/perl
use strict;
use warnings;
while (<STDIN>) {
last if ($_ =~ /^\s*$/); # Exit if it was just spaces (or just an enter)
print "You typed: $_";
}
```

You could use the angle operators without the word 'STDIN':

```
while (<>) {..
```

Accepting user input graphically

The code below will display a perl Tk widget, with text field and two buttons. Enter some text in the text field then press the 'Click here' button. You will see the text you typed appear in your terminal. When you are finished click the 'Done' button.

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

my $mw = MainWindow->new;
```

```
my $text;

$mw->title("Text Entry");

$mw->Label(
-text => "Enter some text:"
)->pack();

$mw->Entry(
-textvariable => \$text
)->pack();

$mw->Button(
-text => "Click here",
-command =>
sub {
print "You typed: $text\n";
}
)->pack();

$mw->Button(
-text => "Done",
-command => sub {exit}
)->pack();

MainLoop;

exit 0;
```

Example

<STDIN> returns the next line of input in a scalar context. However, in a list context, it returns all remaining lines up to end of file. Each line is returned as a separate element of the list. For example:

```
@a = <STDIN>; # read standard input in a list context
```

If the person running the program types three lines, then presses CTRL-D (to indicate "end of file"), the array ends up with three elements. Each element will be a string that ends in a newline, corresponding to the three newline-terminated lines entered.

Some systems use CTRL-Z to indicate end of file, while others use it to suspend a running process

Accepting user input from a web form

Use the CGI module, create a CGI object (in the variable $q in our case), and call the CGI method param() to retrieve the form values. The example below also prints out the form:

```
#!/usr/bin/perl
use strict;
use warnings;
use CGI;
use CGI::Carp qw(fatalsToBrowser);

my $q = new CGI;

print $q->header();

print $q->start_html(
-style => qq(
body {font-family: verdana, arial, sans-serif;}
)
);

if ($q->param()) {
print "Your name is " . $q->param('username') . "<BR>";
print $q->br;
}

# Output the form
print $q->start_form();
print "Name: ";
print $q->textfield(-name => "username");
print $q->br;
print $q->submit(-value => "Click here");
print $q->end_form();
print $q->end_html();

exit 0;
```

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

Perl function reference

Cilj

- Upoznavanja sa refrencom za Perl funkcije

Perl function reference

Perl function reference

Ovdje se objašnjava pojam “Perl ugrađena funkcija” (engl. Perl's built-in function). Svaki opis daje sintaksu funkcije sa tipom i redosledom argumenata.

Traženi argument su prikazani u “italic” stilu i odvojeni su zapetom. Ako argument treba da bude određeni tip varijable, identifikator te variable će biti korišćen (na primer, znak procenta za heš, % *hash*).

Opcioni argument se stavljaju u zagrade. Postoji nekoliko načina za korišćenje ugrađenih funkcija.

Bilo koji argument, koji zahteva skalarnu vrednost, može biti napravljen od bilo kojeg izraza koji vraća jedan. Na primer:

$root = sqrt (shift @numbers);

Funkcija **shift** uklanja prvi element @numbers i vraća gad a se koristi sa fukcijom sqrt .

Mnoge funkcije uzimaju listu skalara kao argument. Bilo koja varijabla nizaili drugi izraz koji vraća lista se može koristiti za sve ili dio argumenata.

chmod (split /,/ FILELIST>); # an expression returns a list

chmod 0755, @executables; # array used for part of arguments

U prvoj liniji, **split** izraz čita string iz identifikatora datoteke (engl. filehandle) i ubacuje ga u listu.

Lista daje korektne argument za **chmod** .

Druga linija koristi niz koji se sastoji od liste imena datoteka koje se daje chmod funkciji.

Zagrade nisu potrebne oko argumeneta funkcija . Međutim, bez zagrada funkcije se mogu gledati kao operatori u izrazu.

U Perlu, može se preneti podprogramu samo jedan tip argumenta i to skalar. Za prenošenje bilo kojeg drugog tipa argumenta treba ga konvertovati u scalar. To se radi prenošenjem reference.

Referenca na bilo što je skalar.

Sledeća slika ilustruje metode refrenciranja i dereferenciranja variable.

U slučaju list ii heševa, referencira se ili dereferencira čitava lista ili heš kao Celina, ne pojedinačni elementi.

Variable	Instantiating the scalar	Instantiating a reference to it	Referencing it	Dereferencing it	Acc
$scalar	$scalar = "steve";	$sref = \"steve";	$sref = \$scalar	$$sref or ${$sref}	N/A
@list	@list = ("steve", "fred");	$sref = ["steve", "fred"];	$sref = \@list	@{$sref}	${$ Sref
%hash	%hash = ("name" => "steve", "job" => "Troubleshooter");	$shash = {"name" => "steve", "job" => "Troubleshooter"};	$sref = \%hash	%{$sref}	${$ Sref
FILE			$sref = *FILE	{$sref} or scalar <$sref>	

Slika 1 Primer

Ovi principi su demonstrirani u sledećim izvornim kodovima .

```
sub doscalar
{
my($scalar) = "This is the scalar";
my($ref) = \$scalar;
print "${$ref}\n"; # Prints "This is the scalar".
}

sub dolist
{
my(@list) = ("Element 0", "Element 1", "Element 2");
my($ref) = \@list;
print "@{$ref}\n"; # Prints "Element 0 Element 1 Element 2".
print "${$ref}[1]\n"; # Prints "Element 1".
}

sub dohash
{
my(%hash) = ("president"=>"Clinton",
"vice president" => "Gore",
"intern" => "Lewinsky");
my($ref) = \%hash;

# NOTE: Can't put %{ref} inside doublequotes!!! Doesn't work!!!
# Prints "internLewinskyvice presidentGorepresidentClinton".
# NOTE: Hash elements might print in any order!
print %{$ref}; print "\n";

# NOTE: OK to put ${$ref}{} in doublequotes.
# NOTE: Prints "Gore".
print "${$ref}{'vice president'}\n";
}
```

Perl functions by category

Cilj

- Upoznavanja sa kategorijama Perl funkcija

Perl functions by category

Perl functions by category

Kategorije Perl funkcija

Ovdje su date Perl funkcije i ključne reči, organizovane po kategorijama. Neke funkcije se pojavljuju u nekoliko kategorija.

Scalar manipulation

chomp , chop , chr , crypt , hex , index , lc , lcfirst , length , oct , ord , pack , q// , qq// , reverse , rindex ,

sprintf , substr , tr/// , uc , ucfirst , y///

Regular expressions and pattern matching

m// , pos , qr// , quotemeta , s/// , split , study

Numeric functions

abs , atan2 , cos , exp , hex , int , log , oct , rand , sin , sqrt , srand

Array processing

pop , push , shift , splice , unshift

List processing

grep , join , map , qw// , reverse , sort , unpack

Hash processing

delete , each , exists , keys , values

Input and output

binmode , close , closedir , dbmclose , dbmopen , die , eof , fileno , flock , format , getc , print , printf ,

read , readdir , rewinddir , seek , seekdir , select , syscall , sysread , sysseek , syswrite , tell , telldir , truncate ,

warn , write

Fixed-length data and records

pack , read , syscall , sysread , syswrite , unpack

Filehandles, files, and directories

chdir , chmod , chown , chroot , fcntl , glob , ioctl , link , lstat , mkdir , open, opendir , readlink , rename ,

rmdir , stat , symlink , sysopen , umask , unlink, utime

Flow of program control

caller , continue , die , do ,dump , eval ,exit , goto , last ,next , redo ,return , sub ,wantarray

Scoping

caller , import , local , my , package , use

Miscellaneous

define , dump , eval , formline , local , my , prototype , reset , scalar ,undef , wantarray

Processes and process groups

alarm , exec , fork , getpgrp , getppid , getpriority , kill , pipe , qx// ,setpgrp , setpriority , sleep , system ,

times , wait , waitpid

Library modules

do , import ,no , package ,require , use

Classes and objects

bless ,dbmclose , dbmopen , package , ref , tie , tied , untie , use

Low-level socket access

accept , bind , connect , getpeername , getsockname , getsockopt , listen , recv , send , setsockopt , shutdown , socket , socketpair

System V interprocess communication

msgctl , msgget , msgrcv , msgsnd , semctl , semget , semop , shmctl , shmget , shmread , shmwrite

Fetching user and group information

endgrent , endhostent , endnetent , endpwent , getgrent , getgrgid , getgrnam , getlogin , getpwent ,

getpwnam , getpwuid , setgrent , setpwent

Fetching network information

endprotoent , endservent , gethostbyaddr , gethostbyname , gethostent , getnetbyaddr , getnetbyname , getnetet , getprotobyname , getprotobynumber , getprotoent , getservbyname , getservbyport , sethostent , setnetent , setprotoent , setservent

Time

gmtime , localtime , time , times

Debugging

Cilj

- Upoznavanja sa načinima ispravljanje grešaka

Debugging

Debugging

Ispravljanje grešaka

Kada se kod programiranja Perl skipta javljaju greške, postoji nekoliko stvari koje programer može da učini:

Pokrenuti skript sa ***–w*** opcijom (engl. switch) koja štampa upozorenja o mogućim problemima s kodom.

Koristiti Perl debugger.

Koristiti drugi debugger ili program za optimizaciju (engl. profiler) kao što je Devel::DProf modul.

Perl debugger obezbeđuje interaktivno Perl okruženje. Tu se korisei DProf modul i ***dprofpp*** program koji je dio modula. Zajedno obezbeđuju profil za Perl script.

Dodatak

SYNOPSIS

perl -d:DProf test.pl

Devel::DProf is DEPRECATED and will be removed from a future version of Perl. We strongly recommend that you install and use Devel::NYTProf instead, as it offers significantly improved profiling and reporting.

DESCRIPTION

The Devel::DProf package is a Perl code profiler. This will collect information on the execution time of a Perl script and of the subs in that script. This information can be used to determine which subroutines are using the most time and which subroutines are being called most often. This information can also be used to create an execution graph of the script, showing subroutine relationships.

To profile a Perl script run the perl interpreter with the -d debugging switch. The profiler uses the debugging hooks. So to profile script test.pl the following command should be used:

perl -d:DProf test.pl

When the script terminates (or when the output buffer is filled) the profiler will dump the profile information to a file called tmon.out. A tool like dprofpp can be used to interpret the information which is in that profile. The following command will print the top 15 subroutines which used the most time:

dprofpp

To print an execution graph of the subroutines in the script use the following command:

dprofpp -T

Consult dprofpp for other options.

Dodatak

Debugging a Perl program

To debug a perl program, invoke the perl debugger using “perl -d” as shown below.

perl -d ./perl_debugger.pl

To understand the perl debugger commands in detail, let us create the following sample perl program (perl_debugger.pl).

```
$ cat perl_debugger.pl
#!/usr/bin/perl -w

# Script to list out the filenames (in the pwd) that contains specific pattern.
#Enabling slurp mode
$/=undef;

# Function : get_pattern
# Description : to get the pattern to be matched in files.
sub get_pattern
{
my $pattern;
print "Enter search string: ";
chomp ($pattern = <> );
return $pattern;
}

# Function : find_files
# Description : to get list of filenames that contains the input pattern.
sub find_files
{
my $pattern = shift;
my (@files,@list,$file);

# using glob, obtaining the filenames,
@files = <./*>;

# taking out the filenames that contains pattern.
@list = grep {
$file = $_;
open $FH,"$file";
@lines = <$FH>;
$count = grep { /$pattern/ } @lines;
$file if($count);
} @files;
return @list;
}
# to obtain the pattern from STDIN
$pattern = get_pattern();

# to find-out the list of filenames which has the input pattern.
@list = find_files($pattern);
```

```
print join "\n",@list;
```

1. Enter Perl Debugger

```
# perl -d ./perl_debugger.pl
```

it prompts,

```
DB<1>
```

2. View specific lines or subroutine statements using (l)

```
DB<1> l 10
10: my $pattern;
```

```
DB<2> l get_pattern
11 {
12: my $pattern;
13: print “Enter search string: “;
14: chomp ($pattern = );
15: return $pattern;
16 }
```

3. Set the breakpoint on get_pattern function using (b)

```
DB<3> b find_files
```

4. Set the breakpoint on specific line using (b)

```
DB<4> b 44
```

5. View the breakpoints using (L)

```
DB<5> L
./perl_debugger.pl:
22: my $pattern = shift;
break if (1)
44: print join “\n”,@list;
break if (1)
```

6. Step by step execution using (s and n)

```
DB<5> s
main::(./perl_debugger.pl:39): $pattern = get_pattern();
```

```
DB<5> s
main::get_pattern(./perl_debugger.pl:12):
12: my $pattern;
```

Option s and n does step by step execution of each statements. Option s steps into the subroutine. Option n executes the subroutine in a single step (stepping over it).

The s option does stepping into the subroutine but while n option which would execute the subroutine (stepping over it).

7. Continue till next breakpoint (or line number, or subroutine) using (c)

DB<5> c

Enter search string: perl

main::find_files(./perl_debugger.pl:22):

22: my $pattern = shift;

8. Continue down to the specific line number using (c)

DB<5> c 36

main::find_files(./perl_debugger.pl:36):

36: return @list;

9. Print the value in the specific variable using (p)

DB<6> p $pattern

perl

DB<7> c

main::(./perl_debugger.pl:44): print join “\n”,@list;

DB<7> c

./perl_debugger.pl

Debugged program terminated. Use q to quit or R to restart, use o inhibit_exit to avoid stopping after program termination, h q, h R or h o to get additional info.

After the last continue operation, the output gets printed on the stdout as “./perl_debugger.pl” since it matches the pattern “perl”.

10. Get debug commands from the file (source)

Perl debugger can get the debug command from the file and execute it. For example, create the file called “debug_cmds” with the perl debug commands as,

c

p $pattern

q

Note that R is used to restart the operation(no need quit and start debugger again).

DB<7> R

DB<7> source debug_cmds

>> c

Enter search string: perl

./perl_debugger.pl

Debugged program terminated. Use q to quit or R to restart, use o inhibit_exit to avoid stopping after program termination, h q, h R or h o to get additional info.

>> p $pattern

perl

>> q

Note: If you are relatively new to perl, refer to our previous article: 20 perl programming tips for beginners.

Summary of Perl debugger commands

Following options can be used once you enter the Perl debugger.

h or h h – for help page

c – to continue down from current execution till the breakpoint otherwise till the subroutine name or line number,

p – to show the values of variables,

b – to place the breakpoints,

L – to see the breakpoints set,

d – to delete the breakpoints,

s – to step into the next line execution.

n – to step over the next line execution, so if next line is subroutine call, it would execute subroutine but not descend into it for inspection.

source file – to take the debug commands from the file.

l subname – to see the execution statements available in a subroutine.

q – to quit from the debugger mode.

Perl debugger

Cilj

- Upoznavanja sa Perl debuggerom

Perl debugger

Perl debugger

Definicija

Debugger je računarski program koji omogućava izvršavanje programa liniju po liniju, ispitivanje vrednosti varijabli ili praćenje vrednosti koje se unose u funkcije te objašnjava zašto se program izvršava drugačije od očekivanog.

Za izvršavanje skripta pod Perl izvornim debugger programom, treba pokrenuti Perl sa prekidačem *–d*.:

perl -d myprogram

Program se izvršava u interativnom Perl okruženje koje traži debugger komande koje ispituju izvorni kod, postavljaju tačke prekida (engl. breakpoints), menjaju vrednosti varijabl itd. Ako program koristi neke prekidače ili argument, moraju se uključiti u komandu :

perl -d myprogram myinput

U Perlu debugger nije poseban program kao kod tipičnih prevodilačkih okruženja (engl. compiled environment). Umesto toga, *-d* zastavica kaže prevodiocu da ubaci izvornu informaciju u drvo za rasčlanjivanje (engl. parse tree) koje se zatim predaje interpreteru.

Programski kod se mora prvo korektno prevesti da bi debugger mogao da radi na njemu, debugger se neće pokrenuti dok se ne otklone sve greške u prevođenju..

Kada se kod korektno prevede i kada se pokrene debugger, program se zaustavlja prije prve izvršne izjave i čeka na unošenje prve debugger komande. Kada se debugger zaustavi i pokaže liniju koda, uvijek pokaže liniju koju će se izvršiti, umjesto liniju koju je upravo završila.

Dodatak

Using the debugger

If you have any compile time executable statements (code within a BEGIN block or a use statement), they are not stopped by the debugger, although require s are. The debugger prompt is something like this:

DB<8>

or even this:

DB<<17>>

where the number in angle brackets is the command number. A *csh* -like history mechanism lets you access previous commands by number. For example, !17 repeats command number 17. The number of angle brackets indicates the depth of the debugger. You get more than one set of brackets, for example, if you're already at a breakpoint and then you print out the result of a function call that itself also has a breakpoint.

If you want to enter a multiline command, such as a subroutine definition with several statements, you can use a backslash to escape the newline that would normally end the debugger command:

DB<1> sub foo { \

cont: print "fooline\n"; \

cont: }

DB<2> foo

fooline

You can maintain limited control over the Perl debugger from within your Perl script. You might do this, for example, to set an automatic breakpoint at a certain subroutine whenever a particular program is run under the debugger.

Setting $DB::single to 1 causes execution to stop at the next statement, as though you'd used the debugger's s command.

Setting $DB::single to2 is equivalent to typing the n command, and the $DB::trace variable can be set to 1 to simulate the t command.

Once you are in the debugger, you can terminate the session by entering q or CTRL-D at the prompt. You can also restart the debugger with R.

Namespaces and packages

Cilj

- Upoznavanja sa Perl prostorima imena i pakovanjima

Namespaces and packages

Namespaces and packages

Prostor imena i pakovanja

Prostor imena čuva imena ili identifikatore , uključujući varijable, podprograme, identifikatore datoteka (engl. filehandles) i formate.

Svaki prostor imena ima svoju tabelu simbola. što je ustvari heš sa ključem za svaki identifikator

Podrazumevani (engl. default) proctor imena za programe je **main**, ali korinsik može definisati druge prostore imena i varijabli i koristiti ih u svojim programima.

Varijable u drugim prostorima imena mogu čak imati i ista imena, ali su one potpuno druačije jedne od drugih.

U Perlu, prostor imena je pakovanje (engl. package) . Po konvenciji, imena pakovanja počinju velikim slovom.

Svako pakovanje počinje deklaracijom pakovanja .

Izjava pakovanja prebacuje tekući kontekst imenovanja na određeni proctor imena (symbol table)

Poziv pakovanja koristi jedan argument , ime pakovanja. Unutra domena (engl. scope) deklaracije pakovanja, dozvoljeni su svi regularni identifikatori u okviru tog pakovanja (osim my varijabli).

Primer

PACKAGE_NAME::VARIABLE_NAME

$i = 1; print "$i\n"; # Prints "1"

package foo;

$i = 2; print "$i\n"; # Prints "2"

package main;

print "$i\n"; # Prints "1"

```
print "$foo::i\n"; # Prints "2"
```

Rezultat

1

2

1

2

Dodatak

From inside one package, you can refer to variables from another package by "qualifying" them with the package name. To do this, place the name of the package followed by two colons (::) before the identifier's name, i.e., $Package::varname .

If the package name is null, the main package is assumed. For example, $var and $::var are the same as $main::var .

Packages may be nested inside other packages. However, the package name must still be fully qualified. For example, if the package Province is declared inside the package Nation , a variable in the Province package is called as $Nation::Province::var . You cannot use a "relative" package name such as $Province::var within the Nation package for the same thing.

The default main namespace contains all other packages within it.

Dodatak

What are Packages?

A **package** is a collection of which lives in its own namespace.

A **namespace** is a named collection of unique variable names (also called a symbol table).

Namespaces prevent variable name collisions between packages

Packages enable the construction of modules which, when used, won't clobber variables and functions outside of the modulus’s own namespace.

Modules

Cilj

- Upoznavanja sa Perl modulima

Modules

Modules

Perl moduli

Perl modul je pakovanje za višekratnu upotrebu definisano u biblioteci čije ime je isto kao i ime pakovanja

(s .pm kao ekstenzijom na kraju).

Perl modul poznaje modul koji se zove "Foo.pm" i koji može sadržavati sledeće izjave:

#!/usr/bin/perl

package Foo;

sub bar {

print "Hello $_[0]\n"

}

sub blat {

print "World $_[0]\n"

}

1;

Karakteristke

Funkcije require i use pune (engl. load) modul.

Obe koriste pretraživanje u @INC za nalaženje modula (mogu se menjati!)

Obe pozivaju funkciju eval za procesiranje koda

1; na kraju koda izaziva izvršavanje eval za evaluaciju TRUE

Perl locira module pretraživanjem @INC niza koji sadrži listu direktorija biblioteke.

Perlovo korišćenje @INC je približno uporedivo kada UNIX ljuske koristi varijablu PATH okruženja za lociranje izvršnih programa.

@INC je definisan kada se Perl kreira a može se dopuniti sa *-I* opcijom komandne linije ili uz uporebu lib unutar programa.

Primer

MyModule.pm

```
package MyModule;
use strict;
use Exporter;
use vars qw($VERSION @ISA @EXPORT @EXPORT_OK %EXPORT_TAGS);

$VERSION     = 1.00;
@ISA   = qw(Exporter);
@EXPORT    = ();
@EXPORT_OK   = qw(func1 func2);
%EXPORT_TAGS = ( DEFAULT => [qw(&func1)],
            Both    => [qw(&func1 &func2)]);

sub func1  {return reverse @_  }
sub func2  {return map{ uc }@_ }
```

Slika 1 Primer

Dodatak

The Require Function

A module can be loaded by calling the require function

```
#!/usr/bin/perl
require Foo;

Foo::bar( "a" );
Foo::blat( "b" );
```

Notice above that the subroutine names must be fully qualified (because they are isolated in their own package)

It would be nice to enable the functions bar and blat to be imported into our own namespace so we wouldn't have to use the Foo:: qualifier.

The Use Function

A module can be loaded by calling the use function

```
#!/usr/bin/perl
use Foo;

bar( "a" );
blat( "b" );
```

Dodatak

How to install CPAN modules

Here are some recommended approaches to installing modules from CPAN, as with much of Perl there are several alternatives.

Some basics

Most Perl modules are written in Perl, some use XS (they are written in C) so require a C compiler (it's easy to get this setup - don't panic), see your OS of choice below to find out how to get the right compiler. Modules may have dependencies on other modules (almost always on CPAN) and cannot be installed without them (or without a specific version of them). It is worth throughly reading the documentation for the options below. Many modules on CPAN now require a recent version of Perl (version 5.8 or above).

Quick start

Install cpanm to make installing other modules easier (you'll thank us later). You need to type these commands into a Terminal emulator (Mac OS X, Win32, X Windows/Linux)

cpan App::cpanminus

Now install any module you can find.

cpanm Module::Name

Tools

To help you install and manage your modules:

local::lib enables you to install modules into a specified directory, without requiring root or administrator access. See the bootstrapping technique for how to get started. You can create a directory per user/project/company and deploy to other servers, by copying the directory (as long as you are on the same operating system).

cpanm from App::cpanminus is a script to get, unpack, build and install modules from CPAN. It's dependency free (can bootstrap itself) and requires zero configuration (install instructions). It automates the entire build process for the majority of modules on CPAN and works well with local::lib and perlbrew. Many experienced Perl developers use this as their tool of choice. Related tools: cpan-outdated, pm-uninstall, cpan-listchanges.

perlbrew from App::perlbrew is useful if your system perl is too old to support modern CPAN modules, or if it's troublesome in other capacities (RedHat/CentOS are included in this list). perlbrew makes the process of installing a Perl in any directory much easier, so that you can work completely independently of any system Perl without needing root or administrator privileges.

You can use multiple versions of Perl (maybe as you upgrade) across different projects. The separation from your system Perl makes server maintenance much easier and you more confident about how your project is setup. Currently (March 2011) Win32/64 is not supported.

cpan from CPAN has been distributed with Perl since 1997 (5.004). It has many more options than cpanm, it is also much more verbose.

cpanp from CPANPLUS has been distributed with Perl since 2007 (5.009). This offers even more options than cpanm or cpan.

Perl on Windows (Win32 and Win64)

Strawberry Perl is an open source binary distribution of Perl for the Windows operating system. It includes a compiler and pre-installed modules that offer the ability to install XS CPAN modules directly from CPAN. cpanm can be installed by running cpan App::cpanminus.

ActiveState provide a binary distribution of Perl (for many platforms), as well as their own perl package manager (ppm). Some modules are not available as ppm's or have reported errors on the ppm build system, this does not mean they do not work. You to use the cpan script to build modules from CPAN against ActiveState Perl.

Perl on Mac OSX

OSX comes with Perl pre-installed, in order to build and install your own modules you will need to install the "Command Line Tools for X" or "X" package - details on our ports page. Once you have done this you can use all of the tools mentioned above.

Perl on other Unix like OSs

Install 'make' through your package manager. You can then use all of the tools mentioned above.

Other tools

CPAN::Mini can provide you with a minimal mirror of CPAN (just the latest version of all modules). This makes working offline easy.

CPAN::Mini::Inject allows you to add your own modules to your local CPAN::Mini mirror of CPAN. So you can install and deploy your own modules through the same tools you use for CPAN modules.

Which modules should I use?

Task::Kensho lists suggested best practice modules for a wide range of tasks. http://search.cpan.org/ will let you search CPAN. You could also get involved with the community, ask on a mailing list or find your nearest Perl Mongers group.

L4: Perl heševi i podprogrami

Zaključak*
Nakon studiranja sadržaja ovog poglavlja, studenti će steći osnovna znanja iz oblasti Perl heševa i podprograma.

LearningObject

Perl programski jezik

Perl - hashes

Defining subroutines

sub hello_world {

print "Hello world!\n";

}

Slika 1 Poziv podprograma

Parameter handling

sub print_params {

print join(', ', @_) . "\n";

}

print_params(1,2,3,4); # Ispisuje "1, 2, 3, 4\n"

Svi nizovi se "ravnaju" u jedan niz

print_params(1, (2, 3), (4,)); # Takodje ispisuje "1, 2, 3, 4\n"

Arguments as references

Ukoliko ne želimo da sve argumente spljoštimo u isti niz, array i hash vrednosti možemo slati po referenci:

my @a1 = (1,2,3);

my @a2 = (4,5,6);

print_params(\@a1, \@a2);

Number of parameters

Ukoliko se @_ interpretira u skalarnom kontekstu (konvertuje u skalar na primer), vrednost je

duzina param niza.

sub number_of_args {

print scalar @_ . "\n";

}

Uvod u Perl jezik

Perl - hashes and subroutines

Napisati program koji traži od korisnika ime, i vraća prezime. Koristiti imena poznatih ljudi ili ljudi koje poznajete.

Napisati program koji čita tekstualni fajl i obrće redosled linija (prva je poslednja, poslednja je prva).

Napisati program koji uzima listu brojeva i ispisuje one koji su iznad proseka u listi (napisati poseban sub koji računa prosek).

Napisati sub (nazvan zip) koji prima dva niza (po referenci) iste dužina, i vraća jedan niz u kome je svaki i-ti element i-ti element prvog niza, a svaki i+1 element i-ti element drugog niza. Primer:

my @a1 = (1,2,3,4);

my @a2 = ('a', 'b', 'c', 'd');

zip(\@a1, \@a2); # Vraća (1, 'a', 2, 'b', 3, 'c', 4, 'd')

Obraditi slučaj nizova različite dužine, ili ispisivanjem greške ili odsecanjem dužeg niza.

Naslov mejla treba da bude:

IT2008-IPT-SCRP-STRL-DZ-DZ04-Ime-Prezime-BrojIndeksa.docx