COMP284 Scripting Languages
Lecture 3: Perl (Part 2)
Handouts

Department of Computer
School of Electrical Engineering, Electronics, an
University of Liverpo

Contents

Control structures: conditional statements

The general format of conditional statements is very similar to that in Java:

```
if (condition) {
    statements
} elsif (condition) {
    sta
} else {
    sta
}
```

- *condition* is an arbitrary expression
- the elsif-clause is optional and there can be more tha
- the else-clause is optional but there can be at most one
- in contrast to Java, the curly brackets must be present even if *statements* consist only of a single statement

Control structures: conditional statements

- Perl also offers two shorter conditional statements:

```
statement if (condition);
```

and

```
state
```

- In ana
  Perl off

```
condition ? if_true_expr :
```

Examples

```
$descr = ($distance < 50) ? "near" : "far";

$size  = ($width < 10) ? "small" :
         ($width < 20) ? "medium" :
                         "large";
```

Blocks

- A sequence of statements in curly brackets is a block
  - ↝ an alternative definition of conditional statements is

```
if (condition) block
elsif (condition) block
else block
```

- In Perl

```
statement if (condition);
statement unless (condition)
```

only a single statement is allowed
but `do block` counts as a single statement,
so we can write

```
do block if (condition);
do block unless (condition);
```

Control structures: switch statement/expression

Starting with Perl 5.10 (released Dec 2007), the language includes a
switch statement and corresponding switch expression

But these are considered experimental and need to be enabled explicitly

Example

```
use feature ...

given ($month) {
  when ([1,3,5,7,8,10,12]) { $days = 31 }
  when ([4,6,9,11])        { $days = 30 }
  when (2)                 { $days = 28 }
  default                  { $days = 0  }
}
```

Note: No explicit break statement is needed

Control structures: while- and until-loops

- Perl offers while-loops and until-loops

```perl
while (condition) {
    statements
}

until (
    st
}
```

- A 'proper' until-loop where the loop is executed at least once
  can be obtained as follows

```perl
do { statements } until (condition)
```

  The same construct also works for `if`,

  In case there is only a single statement it is also possible to write

```perl
statement until (condition);
```

  Again this also works for `if`, `unless` and `while`

Control structures: for-loops

- **for-loops** in Perl take the form

```
for (initialisation; test; increment) {
    statements;
}
```

Again
consis

- Such a f

```
initialisation;
while (test) {
    statements;
    increment;
}
```

Lists and Arrays

- A list is an ordered collection of scalars
- An array (array variable) is a variable that contains a list
- Array variables start with @ followed by a Perl identifier

  `@ identifier`

  An ar

- Perl https://eduassistpro.github.i

  `$ identifier [ index ]`

  to denote the element stored at position
  The first array element has index 0

- Note that

  `$ identifier`
  `@ identifier`

  are two unrelated variables (but this situation should be avoided)

List literals

- A list can be specified by a list literal, a comma-separated list of values enclosed by parentheses

```
(1, 2, 3)
("adam", "ben", "colin", "david")
("adam", 1, "ben", 3)
( )
(1..10, 1
($start
```

- List literals can be assigned to an array:

```
@numbers = (1..10, 15, 20..30);
@names  = ("adam", "ben", "colin", "david");
```

- Examples of more complex assignments, involvi

```
@numbers = (1..10, undef, @numbers, ( ));
@names   = (@names, @numbers);
```

- Note that arrays do not have a pre-defined size/length

Size of an array

- There are three different ways to determine the size of an array

```perl
$arraySize = scalar(@array);
$arraySize = @array;
$arraySize = $#array + 1;
```

- One ca
  in the ra

- But Pe

  The expression $array[-index]
  is equivalent to $array[scalar(@arra

  Example:

  $array[-1] is the same as $array[scalar(@array)-1]
                is the same as $array[$#array]

  that is the last element in @array

Array index out of bound

- Perl, in contrast to Java, allows you to access array indices that are out of bounds

- The value undef will be returned in such a case

```perl
@array = (0, undef, 22, 33);
print '$arra
```

```perl
print '$arra
```
https://eduassistpro.github.i

```
$array[1] = , which IS undef
$array[5] = , which IS undef
```

- The function exists can be used to determin is within bounds and has a value (including undef) associated with it

```perl
print '$array[1]␣exists:␣',exists($array[1]) ? "T":"F","\n";
print '$array[5]␣exists:␣',exists($array[5]) ? "T":"F","\n";
```
```
$array[1] exists: T
$array[5] exists: F
```

Add WeChat edu_assist_pr

Scalar context versus list context

- Scalar context

  when an expression is used as an argument of an operation that requires a scalar value, the expression will be evaluated in a scalar context

  Example:

  ```
  $array
  ```

  ⤳ @array   @array
     in a s

- List context

  when an expression is used as an argument of an operation that requires a list value, the expression will be evaluated in a list context

  Example:

  ```
  @sorted = sort 5;
  ```

  ⤳ A single scalar value is treated as a list with one element in a
     list context

Scalar context versus list context

Expressions behave differently in different contexts following these rules:

- Some operators and functions automatically return different values in different contexts

```perl
$line  = <IN>;       # return one line from IN
@lines = <I
```

- If an exp
  Perl will convert it into a list value with the returned sca
  the one and only element

- If an expression returns a list value in a scalar context, then by default
  Perl will convert it into a scalar value by take the last element of the
  returned list value

List functions

Function	Semantics
`grep(expr, list)`	in a list context, returns those elements of *list* for which *expr* is true; in a scalar context, returns the number of
`join(`	ents
`reverse(list)`	returns a lis
`sort(list)`	returns a list wi standard s
`split(/regexpr/, string)`	returns a list obtained by splitting *string* into substring using *regexpr* as separator
`(list) x number`	returns a list composed of *number* copies of *list*

Array functions: push, pop, shift, unshift

Perl has no stack or queue data structures,
but has stack and queue functions for arrays:

Function	Semantics
push(@array1, value) push(	appends an element or an entire list to the
pop(@array1)	extracts the last element from an array and returns it
shift(@array1)	shift extracts and returns it
unshift(@array1, value) unshift(@array1, list)	insert an element or an entire list at the start of an array variable; returns the number of elements in the resulting array

Array operators: push, pop, shift, unshift

Example:

```perl
1 @planets = ("earth");
2 unshift(@planets,"mercury","venus");
3 push(@planets,"mars","jupiter","saturn");
4 print "A
5 $last = po
6 print
7 $firs  = s
8 print "Array\@3: ", join(" ",@planets),"\n";
9 print "    \@4: ", $first, " ", $last, "\n";
```

Output:

```
Array@1: mercury venus earth mars jupiter saturn
Array@2: mercury venus earth mars jupiter
Array@3: venus earth mars jupiter
    @4: mercury saturn
```

Array operators: delete

- It is possible to delete array elements

- delete($*array*[*index*])
  - removes the value stored at *index* in @*array* and returns it
  - only if *index* equals $#*array* will the array's size shrink to the
    posi   ts()

```
@array = (0, 1, 2, 3
delete($a
print '$array[2]␣exists:␣',exists($array[2])?"T":"F", "\n";
print 'Size␣of␣$array:␣',$#array+1
delete($array[3])
print '$array[3]␣exists:␣',exists($array[3])?"T":"F",
print 'Size␣of␣$array:␣',$#array+1
```

```
$array[2] exists: F
Size of $array: 4
$array[3] exists: F
Size of $array: 2
```

Control structures: foreach-loop

Perl provides the `foreach`-construct to 'loop' through the elements of a list

```
foreach $variable (list) {
    statements
}
```

where `$`

list in each

Example:

```
@my_list = (1..5,20,11..18);
foreach $number (@my_list) {
    $max = $number if (!defined($max) || $number > $max);
}
print("Maximum number in ",join(',',@my_list)," is $max\n");
```

Output:

```
Maximum number in 1,2,3,4,5,20,11,12,13,14,15,16,17,18 is 20
```

Control structures: foreach-loop

Changing the value of the foreach-variable changes the element of the list that it currently stores

Example

```
@my_list = (1..5,20,11..18);
print "Befor
foreach $num
  $numbe
}
print "After:␣␣".join(",␣",@my_list)."\n";
```

Output:

```
Before: 1, 2, 3, 4, 5, 20, 11, 12, 13, 14, 15, 16, 17, 18
After:  2, 3, 4, 5, 6, 21, 12, 13, 14, 15, 16, 17, 18, 19
```

Note: If no variable is specified, then the special variable $_ will be used to store the array elements

Control structures: foreach-loop

An alternative way to traverse an array is

```
foreach $index (0..$#array) {
  statements
}
```

where an el $index] in
statem

Example

```
@my_list = (1..5,20,11..18);
foreach $index (0..$#my_list) {
  $max = $my_list[$index] if ($my_list[$index] > $max);
}
print("Maximum number in ",join(',',@my_list)," is $max\n");
```

Control structures: foreach-loop

- In analogy to while- and until-loops, there are the following variants of foreach-loops:

```
do { statements } foreach list
statement foreach list;
```

In the ex
$_ wil

- Instea

```
do { statements } for list
statement for list;
```

Example:

```
print "Hello␣$_!\n" foreach ("Peter","Paul",
                             "Mary");
```

Control structures: last and next

- The last command can be used in while-, until-, and foreach-loops and discontinues the execution of a loop

```
while ($value = shift(@data)) {
   $written = print(FILE $value);
   if (!$written) { last; }
}
# Executi
```

- The ne
  of a loop and moves the execution to the next iteration

```
foreach $x (-2..2) {
   if ($x == 0) { next; }
   printf("10 / %2d = %3d\n",$x,(10/$x));
}
10 /  -2 =   -5
10 /  -1 =  -10
10 /   1 =   10
10 /   2 =    5
```

Hashes

- A hash is a data structure similar to an array but it associates scalars with a string instead of a number

- Alternatively, a hash can be seen as a partial function mapping strings to scalars

- Reme

- Hash v

  ```
  %identifier
  ```

  A hash variable denotes the entirety of the hash

- Perl uses

  ```
  $identifier{key}
  ```

  where *key* is a string, to refer to the value associated with *key*

Hashes

- Note that

```
$identifier
%identifier
```

are two unrelated variables (but this situation should be avoided)

- An easy way to ... the following

```
use Data::Dumper;
$Data::Dumper::Terse = 1;
print Dumper \%hash;
```

Note the use of \%*hash* instead of %*ha*
(\%*hash* is a reference to %*hash*)

```
Data::Dumper
```
can produce string representations for
arbitrary Perl data structures

Basic hash operations

- Initialise a hash using a list of key-value pairs

```perl
%hash = (key1, value1, key2, value2, ...);
```

- Initialise a hash using a list in big arrow notation

```perl
%hash                                                  ..);
```

- Assoc

```perl
$hash{key} = value;
```

- Remember that undef is a scalar value

```perl
$hash{key} = undef;
```

extends a hash with another key but unknown value

Basic hash operations

- One can use the `exists` or `defined` function to check whether a key exists in a hash:

  `if (exists $hash{key}) {...}`

  Note that if `$hash{key}` eq `undef`, then `exists $hash{key}` is true

- The `delete` function ... from a hash:

  `delete($hash{key});`

  After executing `delete($hash{key})`, ... ll be false

- The `undef` function removes the contents and memory allocated to a `hash`:

  `undef %hash`

Basic hash operations

- It is also possible to assign one hash to another

```perl
%hash1 = %hash2;
```

In contrast to C or Java this operation creates a copy of %hash2
that is then assigned to %hash1

Example:

```perl
%hash1 = ...
%hash2 = %hash1;
$hash1{'b'} = 4;
print "\$hash1{'b'}␣=␣$hash1{'b'}\n";
print "\$hash2{'b'}␣=␣$hash2{'b'}\n";
```

Output:

```
$hash1{'b'} = 4
$hash2{'b'} = 2
```

The each, keys, and values functions

each %hash	returns a 2-element list consisting of the key and value for the next element of %hash, so that one can iterate over it
values %hash	returns a list consisting of all the values of %hash,
keys %hash	returns a list consisting of all the keys of %hash

Examples:

```perl
while ( ($key, $value) = each %hash) {
    statements
}
```

```perl
foreach $key (sort keys %hash) {
    $value = $hash{$key};
}
```

Example: Two-dimensional hash as a 'database'

```perl
1  use List::Util "sum";
2  $name{'200846369'} = 'Jan Olsen';
3  $marks{'200846369'}{'COMP210'} = 61;
4  $marks{'200846369'}{'COMP207'} = 57;
5  $mark
6  $mark
7
8  $average = sum(values($marks{'200846369'}))/
9             scalar(values($marks{'
10 print("avg: $average\n");
```

Output:

```
avg: 60
```

Example: Frequency of words

```
1  # Establish the frequency of words in a string
2  $string = "peter paul mary paul jim mary paul";
3
4  # Split the string into words and use a hash
5  # to accum
6  ++$c
7
8  # Print the frequency of each word found in the
9  # string
10 while (($key,$value) = each %count) {
11    print("$key =>  $value;  ");
12 }
```

Output:

```
jim => 1; peter => 1; mary => 2; paul => 3
```

Revision

Read

- Chapter 3: Lists and Arrays

- Cha

of

R. L. Schwartz, brian d foy, T. Phoenix:
Learning Perl.
O'Reilly, 2011.
Harold Cohen Library: 518.579.86.S39 or e-book