

# COMP284 Scripting Languages

## Lecture 3: Perl (Part 2)

Handouts (8 on 1)

Ullrich Hustadt

Department of Computer Science  
School of Electrical Engineering, Electronics, and Computer Science  
University of Liverpool

Control structures

Conditional statements

## 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

```
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);
```

COMP284 Scripting Languages

Lecture 3

Slide L3 – 4

## Contents

- Control structures
  - Conditional statements
  - Switch statements
  - While- and Until-loops
  - For-loops
- Lists and Arrays
  - Identifiers
  - List literals
  - Contexts
  - List and array functions
  - Foreach-loops
- Hashes
  - Identifiers

COMP284 Scripting Languages

Lecture 3

Slide L3 – 4

Control structures

Switch statements

## 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 "switch";

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

COMP284 Scripting Languages

Lecture 3

Slide L3 – 5

## Control structures: conditional statements

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

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

- condition** is an arbitrary expression
- the **elsif-clauses** is optional and there can be more than one
- 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

COMP284 Scripting Languages

Lecture 3

Slide L3 – 2

Control structures

While- and Until-loops

## Control structures: while- and until-loops

- Perl offers **while-loops** and **until-loops**

```
while (condition) {
    statements
}
```

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

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

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

The same construct also works for **if**, **unless** and **while**

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

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

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

COMP284 Scripting Languages

Lecture 3

Slide L3 – 6

## Control structures: conditional statements

- Perl also offers two shorter conditional statements:

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

and

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

- In analogy to **conditional statements**  
Perl offers **conditional expressions**:

```
condition ? if_true_expr : if_false_expr
```

Examples:

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

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

COMP284 Scripting Languages

Lecture 3

Slide L3 – 3

Control structures

For-loops

## Control structures: for-loops

- for-loops** in Perl take the form

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

Again, the curly brackets are required even if the body of the loop only consists of a single statement

- Such a **for-loop** is equivalent to the following **while-loop**:

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

COMP284 Scripting Languages

Lecture 3

Slide L3 – 7

Lists and Arrays

Identifiers

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 **array variable** denotes the entire list stored in that variable
- Perl uses  
`$identifier[index]`  
to denote the element stored at position `index` in `@identifier`  
The first array element has index 0
- Note that  
`$identifier`  
`@identifier`  
are two unrelated variables (but this situation should be avoided)

COMP284 Scripting LanguagesLecture 3Slide L3 – 8

Lists and Arrays

Contexts

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:  
`$arraySize = @array;`  
~ `@array` stores a list, but returns the number of elements of `@array` in a **scalar context**
- 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**

COMP284 Scripting LanguagesLecture 3Slide L3 – 12

Lists and Arrays

List literals

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, 15, 20..30)`  
`($start..$end)`
- List literals** can be assigned to an **array**:  
`@numbers = (1..10, 15, 20..30);`  
`@names = ("adam", "ben", "colin", "david");`
- Examples of more complex assignments, involving array concatenation:  
`@numbers = (1..10, undef, @numbers, ( ));`  
`@names = (@names, @numbers);`
- Note that arrays do **not** have a pre-defined size/length

COMP284 Scripting LanguagesLecture 3Slide L3 – 9

Lists and Arrays

Contexts

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  
`$line = <IN>; # return one line from IN`  
`@lines = <IN>; # return a list of all lines from IN`
- If an expression returns a **scalar value** in a **list context**, then by default Perl will convert it into a list value with the returned scalar value being 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 taking the last element of the returned list value

COMP284 Scripting LanguagesLecture 3Slide L3 – 13

Lists and Arrays

List literals

Size of an array

- There are three different ways to determine the size of an array  
`$arraySize = scalar(@array);`  
`$arraySize = @array;`  
`$arraySize = $#array + 1;`
- One can access all elements of an array using indices in the range 0 to `$#array`
- But Perl also allows **negative array indices**:  
The expression `$array[-index]` is equivalent to `$array[scalar(@array)-index]`

Example:  
`$array[-1]` is the same as `$array[scalar(@array)-1]`  
is the same as `$array[$#array]`  
that is the last element in `@array`

COMP284 Scripting LanguagesLecture 3Slide L3 – 10

Lists and Arrays

List and array functions

List functions

Function	Semantics
<code>grep(expr, list)</code>	in a list context, returns those elements of <code>list</code> for which <code>expr</code> is true; in a scalar context, returns the number of times the expression was true
<code>join(string, list)</code>	returns a string that contains the elements of <code>list</code> connected through a separator <code>string</code>
<code>reverse(list)</code>	returns a list with elements in reverse order
<code>sort(list)</code>	returns a list with elements sorted in standard string comparison order
<code>split(/regexpr/, string)</code>	returns a list obtained by splitting <code>string</code> into substring using <code>regexpr</code> as separator
<code>(list) x number</code>	returns a list composed of <code>number</code> copies of <code>list</code>

COMP284 Scripting LanguagesLecture 3Slide L3 – 14

Lists and Arrays

List literals

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  
`@array = (0, undef, 22, 33);`  
`print '$array[1]=' . $array[1], ', which ', (defined($array[1]) ? "IS" : "IS NOT") . "undef\n";`  
`print '$array[5]=' . $array[5], ', which ', (defined($array[5]) ? "IS" : "IS NOT") . "undef\n";`  
`$array[1] = , which IS undef`  
`$array[5] = , which IS undef`
- The function `exists` can be used to determine whether an array index is within bounds and has a value (including `undef`) associated with it  
`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`

COMP284 Scripting LanguagesLecture 3Slide L3 – 11

Lists and Arrays

List and array functions

Array functions: push, pop, shift, unshift

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

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

COMP284 Scripting LanguagesLecture 3Slide L3 – 15

Lists and Arrays

List and array functions

Array operators: push, pop, shift, unshift

Example:

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

COMP284 Scripting LanguagesLecture 3Slide L3 – 16

Lists and Arrays

Foreach-loops

Control structures: foreach-loop

An alternative way to traverse an array is

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

where an element of the array is then accessed using `$array[$index]` in `statements`

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");
```

COMP284 Scripting LanguagesLecture 3Slide L3 – 20

Lists and Arrays

List and array functions

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 position of the highest element that returns true for `exists()`

```
@array = (0, 11, 22, 33);
delete($array[2]);
print 'Array[2] exists: ', exists($array[2]) ? "T":"F", "\n";
print 'Size of @array: ', $#array+1, "\n";
delete($array[3]);
print 'Array[3] exists: ', exists($array[3]) ? "T":"F", "\n";
print 'Size of @array: ', $#array+1, "\n";
```

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

COMP284 Scripting LanguagesLecture 3Slide L3 – 17

Lists and Arrays

Foreach-loops

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 execution of the statements within the loop, the special variable `$_` will be set to consecutive elements of `list`
- Instead of `foreach` we can also use `for`:

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

Example:

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

COMP284 Scripting LanguagesLecture 3Slide L3 – 21

Lists and Arrays

Foreach-loops

Control structures: foreach-loop

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

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

where `$variable`, the `foreach-variable`, stores a different element of the list in each iteration of the loop

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
```

COMP284 Scripting LanguagesLecture 3Slide L3 – 18

Lists and Arrays

Foreach-loops

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; }
}
# Execution of 'last' takes us here
```

- The `next` command stops the execution of the current iteration 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
```

COMP284 Scripting LanguagesLecture 3Slide L3 – 22

Lists and Arrays

Foreach-loops

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 "Before: ", join(" ", @my_list), "\n";
foreach $number (@my_list) {
    $number++;
}
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

COMP284 Scripting LanguagesLecture 3Slide L3 – 19

Hashes

Identifiers

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`
- Remember that Perl can auto-magically convert any `scalar` into a `string`
- `Hash variables` start with a percent sign followed by a `Perl identifier`

```
%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`

COMP284 Scripting LanguagesLecture 3Slide L3 – 23

Hashes

Identifiers

Hashes

- Note that

```
$identifier
%identifier
```

are two unrelated variables (but this situation should be avoided)
- An easy way to print all key-value pairs of a hash `%hash` is the following

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

Note the use of `\%hash` instead of `%hash` (`\%hash` is a reference to `%hash`)  
  
Data::Dumper can produce string representations for arbitrary Perl data structures

COMP284 Scripting LanguagesLecture 3Slide L3 – 24

Hashes

Foreach

The each, keys, and values functions

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

Examples:

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

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

COMP284 Scripting LanguagesLecture 3Slide L3 – 28

Hashes

Basic hash operations

Basic hash operations

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

```
%hash = (key1, value1, key2, value2, ...);
```
- Initialise a hash using a list in big arrow notation

```
%hash = (key1 => value1, key2 => value2, ...);
```
- Associate a single value with a key

```
$hash{key} = value;
```
- Remember that `undef` is a scalar value

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

extends a hash with another key but unknown value

COMP284 Scripting LanguagesLecture 3Slide L3 – 25

Hashes

Foreach

Example: Two-dimensional hash as a 'database'

```
1 use List::Util "sum";
2 $name{'200846369'} = 'Jan_Olsen';
3 $marks{'200846369'}{'COMP201'} = 61;
4 $marks{'200846369'}{'COMP207'} = 57;
5 $marks{'200846369'}{'COMP213'} = 43;
6 $marks{'200846369'}{'COMP219'} = 79;
7
8 $average = sum(values($marks{'200846369'}))/
9             scalar(values($marks{'200846369'}));
10 print("avg: $average\n");
```

Output:  
avg: 60

COMP284 Scripting LanguagesLecture 3Slide L3 – 29

Hashes

Basic hash operations

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 removes a given key and its corresponding value from a hash:

```
delete($hash{key});
```

After executing `delete($hash{key})`, `exists $hash{key}` will be false
- The `undef` function removes the contents and memory allocated to a hash:

```
undef %hash
```

COMP284 Scripting LanguagesLecture 3Slide L3 – 26

Hashes

Foreach

Example: Frequency of words

```
# Create hash the frequency of words in a string
1 $string = "peter_paul_paul_jim_mary_paul";
2
3
4 # Split the string into words and use a hash
5 # to accumulate the word count for each word
6 ++$count{$_} foreach split(/\s+/, $string);
7
8 # Print the frequency of each word found in the
9 # string
10 while ( ($key,$value) = each %count ) {
11     print("$key=>$value\n");
12 }
```

Output:  
jim => 1; peter => 1; mary => 2; paul => 3

COMP284 Scripting LanguagesLecture 3Slide L3 – 30

Hashes

Basic hash operations

Basic hash operations

- It is also possible to assign one hash to another

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

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

Example:

```
%hash1 = ('a' => 1, 'b' => 2);
%hash2 = %hash1;
$hash1{'b'} = 4;
print "\$hash1{'b'}=$_$hash1{'b'}\n";
print "\$hash2{'b'}=$_$hash2{'b'}\n";
```

Output:

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

COMP284 Scripting LanguagesLecture 3Slide L3 – 27

Hashes

Foreach

Revision

Read

- Chapter 3: Lists and Arrays
- Chapter 6: Hashes

of

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

COMP284 Scripting LanguagesLecture 3Slide L3 – 31