BVB BibliotheksVerbund Bayern

The SFX Administrator‘s Perl 101

Dr. Mathias Kratzer, Bavarian Library Network Center

Introduction

What is Perl?

What can Perl do for me?

Where to find more information?

What is Perl?

- according to its UNIX man page it‘s an acronym for
  - **P**ractical **E**xtraction and **R**eport **L**anguage
  - **P**athologically **E**clectic **R**ubbish **L**ister ;-)
- author: Larry Wall (*1954), American **linguist** and programmer
- year of publication: 1987
- licence: dual open source (GPL + AL)
- still ongoing development: currently about 30 official maintainers plus several hundred volunteers from all over the world

Is Perl easy or difficult?

- Perl is **easy to use**, ...
- ... but (admittedly) **not that easy to learn**.
- reason for both: **high code density** due to operators and built-in functions with „standard behaviour“
- **advantage:** shorter programs which are easier to maintain and are less likely to contain bugs
- **disadvantage:** can tend to look cryptic at first glance
- Any effort is worthwhile: You have to learn Perl **only once** … but can use it **again and again** ☺

What can Perl do for me?

- „Swiss Army Chainsaw of Programming Languages“
- **Mission statement** by Larry Wall:

  „Common things should be easy, advanced things should at least be possible.“
- … and as if that wasn‘t sufficient: **TMTOWTDI**
- Perl is as good for quick hacks (tiny scripts in order to automate frequently recurring tasks) as for professional software development (cf. SFX)
- **ideal** for „text crunching“ problems (rule of thumb #1: 90% is about text and 10% is about anything else)

Perl on the Web

- http://www.perl.org: official homepage of Perl
- http://perldoc.perl.org: official documentation on Perl
- http://www.perlmonks.org: free but fast & high-quality support from the Monastery (also for non-members)
- http://www.perl.com: O'Reilly's Perl news site
- http://www.cpan.org: homepage of the Comprehensive Perl Archive Network (CPAN), a completely free pool of Perl modules for almost every imaginable purpose
- http://www.activestate.com/activeperl: free and easy to install Perl interpreter for Windows and Mac OS X

Literature on Perl

- Learning Perl

  aka „Llama book“ by Randal L. Schwartz, Tom Phoenix, Brian D. Foy
- Learning Perl Objects, References, and Modules

  aka „Alpaka book“ by Randal L. Schwartz, Tom Phoenix
- Programming Perl

  aka „Camel book“ by Larry Wall, Tom Christiansen, Jon Orwant
- Perl Cookbook by Tom Christiansen, Nathan Torkington
- Perl Best Practices by Damian Conway
- Perl Pocket Reference by Johan Vromans

Rule of thumb #2: 80/20

- 80% of your Perl problems are solved by 20% of the Perl documentation ...
- ... and the remaining 80% of the documentation address the remaining 20% of your problems.

Introduction

What is Perl?

What can Perl do for me?

Where to find more information?

First Steps

„Hello World“ program

Basic rules and recommendations

Literals and operators

Hello World!

- For your first steps with Perl a simple text editor (e.g. „Notepad“ on Windows systems) is sufficient.
- For maintaining or even developing bigger programs your editor should at least support syntax highlighting (e.g. GNU Emacs).
- Please **don‘t use a text processor** (like MS-Word)!
- Let‘s get started:

```
#!/usr/bin/perl
print "Hello, World!\n";
```

Perl basics

- Programs begin with a so-called „she-bang“-line like `#!/usr/bin/perl` or a `package`-declaration (in case your program is a module).
- **Recommended:** `use strict;` and `use warnings;`
- Any sequence of any kind of whitespace is interpreted like a single space character which allows for generous **indentation!**
- Any text between a `#`-character and the end of the current line is ignored which allows for **comments!**
- „Elementary“ expressions must be terminated by a **semicolon** (`;`).

Number Literals

- examples for **floating point literals**:
  `3.141`, `-0.005`, `-6.539e24`, `2E-4`
- examples for **decimal integer literals**:
  `0`, `2006`, `-80`, `87654321`, `87_654_321`
- examples for **other integer literals**:
  - binary: `0b11111111`,
  - octal: `0377`,
  - hexadecimal: `0xFF`, `0xff`

Arithmetic Operators

Examples for arithmetic operators in Perl:

- negation: `-6.0221415e23`
- addition: `2 + 3`
- subtraction: `5.1 - 2.4`
- multiplication: `3 * 12`
- division: `10.2 / 0.3`
- modulo operation: `25 % 8` (yields `1`)
- exponentiation: `2 ** 13` (yields `8192`)

String Literals

- Strings may contain arbitrary characters.
- Perl distinguishes strings in
  - **single quotes** such as `'Fred'` from strings in
  - **double quotes** such as `"Barney"`.
- The string of minimal length is calles the **empty string** and contains no characters at all. It is recommended to denote it by `q{}` instead of `''` or `""`.
- A string of maximal length would consume all available RAM, i.e. Perl does not limit the length of your strings!

single quoted vs. double quoted

- In a single quoted string each character except for the backslash (`\`) and the single quote (`'`) denotes itself.
- In a double quoted string Perl will recognize the use of **backslash escapes** (`\n`, `\t`, `\"`, `\\`, ...) to denote output control characters oder any other characters in form of their octal (`\0..`) or hexadecimal (`\x..`) byte codes, e.g.:

  `\xC4` or `\xC3\x84` for Ä in ISO 8859-1 or UTF-8, resp.

- **Rule of thumb #3:** Never use a double quoted string when its features exceed your needs!

String Operators

Examples for string operators in Perl:

- **Concatenation**:

```
'Hello' . 'World'      yields 'HelloWorld'
'Hello World' . "\n"   yields "Hello World\n"
```

- **Repetition**:

```
    '-' x 15      yields '---------------'
      3 x 7       yields '3333333'
```

Number or String?

- Perl converts numbers into strings and vice versa on demand, i.e. **depending on operator context**.
- In order to convert a number into a string Perl virtually adds the missing quotes.
- The conversion of a string into a number yields the **longest numerical prefix** of the string (any leading whitespace is ignored) or `0` if this prefix is empty.
- Examples:

```
"12three45\n" + '    3'   yields 15
'Z39.' . (5 * 10)         yields 'Z39.50'
'true' / 2                yields 0
```

First Steps

„Hello World“ program

Basic rules and recommendations

Literals and operators

Native Data Structures

Scalars and Lists

$calar Variables

@rrays

%ashes

Scalars and Lists

- Natural languages (or more precisely: their grammars) have the concept of singular and plural.
- Likewise, Perl has the concept of **scalars** and **lists**.
- Perl distinguishes four types of scalars:
  - numbers,
  - strings,
  - references (memory addresses) and
  - the special value **`undef`**.
- Lists are totally ordered sets (aka „tuples“) of scalars of **arbitrary** type.

More on Lists

- The **elements of a list**
  - **are always scalars, never lists;**
  - are **indexed** by subsequent integers **starting at** `0` (which induces the total order).
- Perl automagically increases the amount of memory allocated for a list on demand, i.e. as more and more elements are added to the list.
- The list of minimal length is called the **empty list** and contains no elements at all. It is denoted by `( )`.
- A list of maximal length would consume all available RAM, i.e. Perl does not limit the length of your lists!

List Literals

- **Option 1:** comma separated, delimited by parentheses

```
(1,2,3)
( 16-15, 4*0.5, 67%8, )
(23..42, (10,10,71), "gr\xFCn")
( 'cyan', 'magenta', 'yellow' )
```

- **Option 2:** by means of the „quoted words“-operator

```
qw/cyan magenta yellow/
qw</usr/lib /etc/lib /home/lib>
```

$calar Variables

- A scalar variable is a storage container für scalar values of **arbitrary** type (number, string, reference, `undef`).
- A **Perl identifier** consists of a letter or the underscore (_) followed by any combination of letters, digits, and underscores.
- The name of a scalar variable needs to be a Perl identifier preceded by the dollar sign (`$`).
- Examples: `$isbn`, `$ctx_obj`, `$_`, `$_DATABASE`

Good Names for Variables

... are **meaningful** and **unmistakable**:

- Better use `$number_of_rows` than `$nr`!
- Does `$stopid` mean `$sto_pid` or `$stop_id` – or is there just a stupid typo?
- Please, use neither `$OOOOOOOO` nor `$OOOOOOOO`!
- There is at least one indisputable advantage in relying on `$underscores_for_longer_names` instead of using `$TheCamelCaseAlternative` : ...
- ... `$THIS_WILL_NOT_WORK_IN_CAMEL_CASE` ;-)

Assignment Operators

- The operator **=** assigns the value of the expression on its right to the variable on its left.
- Perl offers several combined assignment operators to abbreviate frequently used forms of expressions:
  ***=**, **/=**, **+=**, **-=**, **.=**, etc.

```
$time  = 3;                 # $time contains 3
$time  = $time + 3;         # incremented by 3
$time += 3;                 # … and once again
$time .= ':00 AM';          # $time conains '9:00 AM'
```

Comparison Operators

Perl distinguishes comparison operators

- for numbers ...

  `==, !=, <, <=, >, >=, <=>`

- ... from those for strings:

  `eq, ne, lt, le, gt, ge, cmp`

Two **non-trivial** examples:

`'Fred' == "Barney"` is „true“.

`(6+7)*8 ge 6+(7*8)` is „false“.

What is true and what is false?

- **Perl doesn‘t come with any Boolean literals** (such as `true` and `false` in C/C++ or Java).
- Instead of this a Boolean context makes Perl interpret **any scalar value** as a Boolean value according to the following definitions:
  - The special value `undef` is „false“.
  - The number `0` and the string `'0'` are „false“.
  - The empty string `q{}` is „false“.
  - Any other scalar values are „true“.

Context-sensitive undef

- `undef` is the value of undefined scalar variables, i.e. variables which currently do not store a „real“ value.
- If used in **numerical context** `undef` evals to `0`, ...

  Example: `$sum = $sum + 108;`
- ... if used in **string context** `undef` evals to `q{}`, ...

  Example: `$text = $text . "line 1\n";`
- ... but `undef` **is neither a number nor a string!**
- In particular, `undef` is different from `'undef'`!
- In order to test on `undef`-inedness Perl programmers use the unary comparison operator `defined`.

@rrays

- An **array variable** (or for short: **array**) is a storage container for **lists** which allows for element access by numerical index (starting at **0** – always remember!).
- The name of an array needs to be a Perl identifier preceded by the at sign (@).

```
@num = (1,2,3,4);                # array definition
$one = $num[0];                  # $one contains 1
$sum = $one + $num[$one+2]; # $sum contains 5
$num[4] *= $sum;                 # $num[4] contains 0
```

Array Caveats

- The elements of lists are always scalars. Thus, the fifth element of a list stored in `@array` has to be accessed by `$array[4]` – and **not** by `@array[4]`!
- An array always contains at least the empty list `()` – and, thus, is **never** `undef`-ined in Perl!
- A list assignment `@array2 = @array1` **additionally allocates** the amount of memory already allocated for `@array1`!

The Concept of Key-Value Pairs ...

Here you see some quite common key-value pairs:

- hostname – IP address
- IP address – institution
- user name – password
- patron ID – charges account
- ISSN – journal title
- license number – vehicle owner
- word – number of its occurrences in a given document
- URL – number of hyperlinks referring to that URL
- OpenURL parameter – piece of context information

… is the Basic Idea of %ashes

- A **hash variable** (or for short: **hash**) is a storage container for **lists** which allows for access to every second element – the **value** – by specifiying its predecessor – serving as the **key**.
- Values may be arbitrary scalars while keys need to be **pairwise different** strings!
- The name of a hash needs to be a Perl identifier preceded by the percent sign (**%**).

Hash Examples

```
# The natural way of defining a hash:
%age = ( 'Peggy', 40, 'Al', 42, 'Kelly', 17 );
$var1 = $age{'Peggy'};      # $var1 contains 40
$name = 'elly';
$var2 = $age{'K'.$name};    # $var2 contains 17
$var3 = $age{'Bud'};        # $var3 is undef
$age{'Bud'} = "15 years";   # a new hash entry!


# The more common way of defining a hash:
%Matrix = ('Neo'      => 'Keanu Reeves',
           'Trinity'  => 'Carrie-Anne Moss',
           'Morpheus' => 'Lawrence Fishburne',);
```

Hash Caveats

- The elements of lists are always scalars. Thus, the value assigned to key `'k'` in the hash `%value` has to be accessed by `$value{'k'}` – and **not** by `%value{'k'}`!
- A hash always contains at least the empty list `()` – and, thus, is **never** `undef`-ined in Perl!
- The list assignment `%hash2 = %hash1` forces Perl to „unwind“ (i.e. explicitely build a temporary defining list for) `%hash1` and, thus, can be quite **expensive** both in terms of runtime and consumption of memory!

Interpolation of Variable Names

Within **double quoted strings** Perl interpolates variable names as follows:

- Names of **scalar** variables are replaced by the current value of the variable (in string context).
- Names of **non-scalar** variables (i.e. arrays and hashes) are replaced by a concatenation of the currently stored list elements (in string context). The current value of the global special variable **$"** serves as glue.

Interpolation Examples & Caveats

```
my $food = 'dino burger';
print "Fred ate a $food.\n";           # yum yum!
print "|\$food| eq |$food|\n";         # debugging
print "Barney ate two $foods.\n";      # oops???
print "Barney ate two ${food}s.\n";    # ok
print 'Two ' . $food . "s? Wow!\n";    # TMTOWTDI

my @drinks = qw/juice soda coke beer/;
print "I'll take a $drinks[3]. And you?\n";
$" = ', ';
print "@drinks ... no wine? Too bad!\n";

my $email = "kratzer@bsb-muenchen.de"; # oh dear!
```

Why My?

- The scope of a variable declared with the `my` modifier ist the innermost surrounding block or the code file.
- Within its scope a `my`-variable hides any global variable of the same name

  ⇒ **independence of imported foreign code**
- A `my`-variable can only be used – in particular, misused – by code within its scope

  ⇒ **security, easier localisation of errors**
- Among other things the recommended `use strict;` pragma enforces usage of the `my` modifier whenever you introduce (i.e. declare) a new „local“ variable.

Native Data Structures

Scalars and Lists

$calar Variables

@rrays

%ashes

Control Flow

Conditional Statements

Loops

Special Variables

Subroutines

Conditional Statements (I)

- Code which is to be executed under certain conditions only belongs into an **`if`-clause**.
- An **`if`**-clause **may be** followed by one or more **`elsif`-clauses** and/or one **`else`-clause**:

```
if ( c1 ) {
    # execute if c1 is true
}
elsif ( c2 ) {
    # execute if c1 is false but c2 is true
}
else {
    # execute in all other cases
}
```

Boolean Operators

In order to express more complex conditions Perl offers **two sets** of Boolean operators.

- Boolean operators of low precedence:

```
not,    and,    or,    xor
```

- Boolean operators of high precedence:

```
!,    &&,    ||
```

Conditional Statements (II)

- Instead of an **`if`**-clause with logically negated condition you might prefer/find an **`unless`-clause**:

```
unless ( condition ) {
    # execute only if condition is false
}
```

- The ternary **`?:`-operator** (a compact if-else) comes in handy for the conditional creation of values:

```
$what = ($counter > 1) ? 'lines' : 'line';
print „Parsed $counter $what of input.\n";
```

while-Loop

Code which is to be executed repeatedly **while** a certain condition remains true belongs into a **`while`-loop** ...

- ... that might never be entered at all:

```
while ( condition ) {
    # code to be executed repeatedly (perhaps never)
}
```

- ... that has to be entered at least once:

```
do {
    # code to be executed repeatedly (at least once)
} while condition;
```

foreach-Loop

- Code which is to be executed **exactly once for each element of a list** belongs into a **`foreach`-loop**:

```
foreach my control_variable ( list ) {
    # code to be executed once per element of the list
}
```

- At the beginning of each iteration the „current“ element of *`list`* is automagically assigned to the *`control_variable`*.
- Of course, *`list`* may also be an array.

$_ and @_

- Perl provides a whole lot of **special variables** which store certain useful static or dynamic values during the runtime of your program, e.g. `$"`.
- The most frequently used and, thus, most important special variable is `$_` , the **standard argument**.
- For instance, in a `foreach`-loop without explicit control variable `$_` takes on that role.
- Likewise, whenever a subroutine is called Perl pushes the current arguments into the **standard parameter array** `@_`.

  Access to the elements: `$_[0]`, `$_[1]`, `$_[2]`, ...

Subroutines

- In addition to Perl's many built-in functions (such as `print`) you can also write your own functions which are called **subroutines**.
- The name of a subroutine needs to be a Perl identifier preceded by an ampersand sign (`&`).
- The definition of a subroutine starts with the keyword `sub` followed by the name of the subroutine (**omitting the** `&`) and its **code body** enclosed in curly braces.
- In principle, subroutines may be defined at any place in the program. Their names are always global.

Arguments and Return Values

- You **can** pass a list of arguments to a subroutine when you call it – **but** you **don‘t necessarily** have to do so.
- A subroutine **can** explicitely `return` one or more values – **but** it **doesn‘t necessarily** have to do so.

- However, it is **good programming style** to
  - keep your subroutines as flexible as possible with regard to the number of arguments;
  - always `return` explicitely.

Get the max out of it ...

```
sub max {
    if ( $_[0] > $_[1] ) {
        return $_[0];
    }
    else {
        return $_[1];
    }
}

my $maximum = &max( 16, 12, 72 );
# Houston, we have a problem ...
```

... by maximal flexibility

```
sub max {
  my $current_max = shift @_;  # 1st argument

  foreach ( @_ ) {             # for all others:
    if ( $_ > $current_max ) { # greater number?
      $current_max = $_;
    }
  }

  return $current_max;         # the maximum!
}

$maximum = &max( 16, 12, 72, 4, 23, 15, 8, 42 );
# Houston, we have marvelous programmers!
```

Control Flow

Conditional Statements

Loops

Special Variables

Subroutines

OO-Perl

Packages and Modules

procedural vs. object-oriented

Classes and Objects in Perl

Examples From the SFX World

Packages

- The namespace of a global (i.e. not `my`-)variable is the **package** wherein it occurs first.
- At the start of any program the **current package** is `main` by default.
- In order to make another package *`package_name`* the current package you simply have to declare:

```
package package_name;
```

- Global variables in any other than the current package always continue to be accessible via their fully qualified identifier *`package_name::variable_name`*.

Modules

- A **module** is a package that is declared in a file the path name of which „matches“ the package name and ends on the extension `.pm`. Example:

  The path name of the target parser `WWW::Excite` is

  …`/lib/Parsers/TargetParser/WWW/Excite.pm`

- Basic idea: The code within a module can be **reused** completely or in parts by other (e.g. your!) programs.
- Modules are the **key concept for object-oriented programming in Perl** as „classes“ are nothing else than special modules.

use-Statement

- Modules are imported in a Perl program as follows: Selected parts of it by means of

```
use module_name list_of_identifiers;
```

or „as completely as possible“ by means of

```
use module_name;
```

- Behind the scenes the identifiers of the **variables and subroutines** of the specified module are included in the current package and, thus, become available for reuse.

Procedural Programming

- The procedural programming paradigm focuses on the **„what is going on“ aspect** of a given workflow.
- Input and output data are regarded as parameters of this workflow.
- Usually the final procedural program **only works for input and output data with certain properties**.
- For other input and output data which don‘t have these properties the program needs to be **modified** or even rewritten from scratch.
- These are fairly acceptable restrictions as long as we deal with relatively small programs only.

Object-Oriented Programming

- The object-oriented programming paradigm focuses on the **„who is taking part“ aspect** of a workflow.
- Input and output data are **objects** of certain **classes**, i.e. with a certain set of properties (**attributes**) significant to the workflow, and they come along with all the **methods** to manipulate these properties.
- Objects of derived classes inherit attributes / methods from their „parent classes“ but they may extend or even overload them.
- Following these concepts results in clearly defined, stable interfaces and hierarchically structured code which is **extremely reusable** also in other workflows!

Classes and Objects in Perl

Class.pm

```
package Demo::Class;

sub new {
    my $self = shift @_; # eq 'Demo::Class'
    my $hash_ref = {
        'att_A' => value_of_att_A,
        'att_B' => value_of_att_B,
        # ... and so on
    };
    return bless( $hash_r
}

sub method {
    # body of a method
}
```

script.pl

```
package Completely::Different;

use Demo::Class;

my $d   = Demo::Class->new();
:
$demo->method( list_of_args );
```

Examples from the SFX World (I)

- **ContextObject**:

```
my $ctx_obj = …
:
my $issn   = $ctx_obj->{ 'rft.issn'   };
my $year   = $ctx_obj->{ 'rft.year'   };
my $volume = $ctx_obj->{ 'rft.volume' };
my $issue  = $ctx_obj->{ 'rft.issue'  };
my $spage  = $ctx_obj->get( 'rft.spage' );
my $base   = $ctx_obj->parse_param( 'url' );
```

- **TargetParser**: object with at least one service method, e.g. `$tp->getFullTxt( $ctx_obj )`

Examples from the SFX World (II)

SourceParsers are objects with up to 3 public methods:

- **`parsePrivateID`** – analyses / processes the value of the ContextObject attribute **`rft_dat`** (or **`pid`**, resp.)
- **`fetchRecord`** – queries an external database (e.g. PubMed) for the record with the ID (e.g. PMID) that was returned by the method **`parsePrivateID`**
- **`parseRecord`** – analyses / processes the record that was fetched by the method **`fetchRecord`** and sets or corrects ContextObject attribute values accordingly.

OO-Perl

Packages and Modules

procedural vs. object-oriented

Classes and Objects in Perl

Examples From the SFX World

THANKS For Your Attention!

```
my @questions = <STDIN>;
```