Perl, continuation

Zdeněk Žabokrtský

Contents:

- intro to functional programming in Perl
- references
- locale
- POD (plain old documentation)
- intro to modules

Functional programming

- computation as evaluation functions
- avoid states
- higher order functions: function takes another function as an argument
- one of the programming paradigms, contrasting imperative programming style
- in Perl: the actual solution is procedural, even though it a simulates functional solution
- good candidates for FP style: sorting, filtering, array transformations (no iteration code needed)
- bad candidates for FP: functions with side effects

Array transformations: map

- Returns an array in which each element underwent a transformation

```
map {BLOCK} @original_array
```

- Example:

```
print (join " ", map { $_**2 } (1..10));
```

- Returns an array of elements which fulfil certain condition (similarly to unix grep)

```
grep {BLOCK} @original_array;
```

- Example:

```
print (join " ", grep {$_ ** 2 > 50} (1..10);
```

List sorting: sort

- Sorting array elements according to a sorting function
  - default lexicographic sort:

    ```
    sort @array
    ```

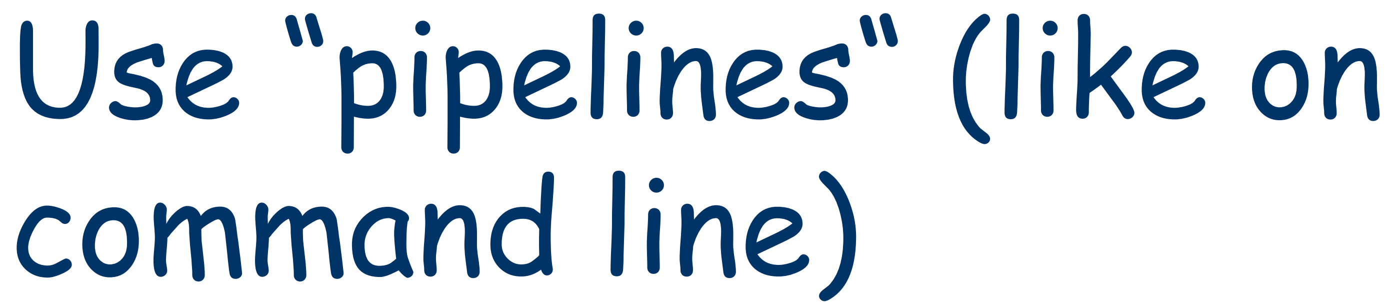
  - customized sort:

    ```
    sort {COMPARING_CODE} @array
    ```

    - predefined variables $a and $b

- Example:

```
print join " ",
   sort { abs($a) <=> abs($b) } (-5..5);
```



Use "pipelines" (like on command line)

```
map {"The price of $_ is $price{$_}." }
    sort {$price{$b}<=>$price{$a}}
       grep {$price{$_} > 10}
         keys %price;
```

- Exercise 1: for a given sentence, print its words longer than three letters sorted according to their length; print its length after each word

- Avoid changing $_ in map/grep/sort
- Avoid using variables called $a and $b

References in Perl

- When you need references in Perl
  - Complex data structures (arrays of arrays, hashes of arrays, cyclic structures etc.)
  - Passing arguments to functions
  - ...
- Reference is a scalar that refers to another scalar, hash or array, or subroutine
- type of reference:
  - ref($reference);
  - values SCALAR, HASH, ARRAY, CODE, REF, GLOB

Creating references

- two ways to create a reference:
  - reference to an existing variable using backslash:
    - `my $h_ref = \%myhash;`
    - `my @a_ref = \@myarray;`
  - creating an anonymous structure
    - anonymous array:

      ```
      my $a_ref = [1,2,3,4];
      ```
    - anonymous hash:

      ```
      my $h_ref = {key1=>10, key2=>20};
      ```
    - anonymous function:

      ```
      my $c_ref = sub {CODE};
      ```

Dereferencing

- two notations:
  - 1) use the reference as if it was the variable's name

```
my @array = @$array_ref;
my %hash = %$hash_ref;
my $scal = $$scalar_ref;
my $array_elem = ${$array_ref}[5];
my $hash_value = ${$hash_ref}{$key};
```

  - 2) arrow operator

```
$element = $array_ref->[1];
$value = $hash_ref->{$key};
```

- Exercise 2:
  - Create a function that takes a hash reference as its argument and prints all its key-value pairs;

Best practices for references

- use anonymous hashes to pass arguments to functions with long or variable set of arguments
- if possible, dereference with arrows (not with $$...)
- use weaken to prevent circular data (memory leaks)

Perl locale

- recall locale: set of parameters specifying user's language and country because of
  - lexicographic ordering
  - character classes in regular expressions
  - case-modification functions, number formatting, etc.

```
use locale;
use POSIX qw(locale_h);
setlocale (LC_ALL,qw(en_US));
print ((join " ", sort qw(cihla chleba))."\n");
setlocale LC_ALL,qw(cs_CZ.UTF8);
print ((join " ", sort qw(cihla chleba))."\n");
```

- Plain Old Documentation is used for most documentation in Perl world
- very simple markup language for writing script/module's documentation directly into the Perl code
- available formatters to plain text, html, man pages etc.
- POD directive comes at the beginning of a line and starts with '=', such as =head1, =head2, =item

- see a sample at

http://en.wikipedia.org/wiki/Plain_Old_Documentation

- use standard templates (boilerplates) for POD: NAME, VERSION, SYNOPSIS, DESCRIPTION, ..., AUTHOR
- you can use module-starter
- place POD at a single place in the file, if possible at its end

Perl modules

- Perl module = a self-contained piece of reusable code, can be included into other Perl scripts or modules
- in Perl, package=module
- two purposes of modules in Perl
  - modularity, encapsulation: modules allow to have separate spaces for variable and function names, so that they are not mixed on a single heap
  - OOP: modules correspond to classes
- each module has a name; the name should be unique
- all variables and functions belong to some package
  - either to the package `main`
  - or in a package defined by the keyword `package`

Perl modules, cont.

- typically, one module corresponds to one .pm file,
- modules are searched for in the directories listed in the PERL5LIB environment variable (separated by colon)
- alternatively, you can
  `use lib '/path'; unshift @INC,'/path';`
- modules can be nested: `MainModule::NestedModule`
- nesting is represented by subdirectories:
- `Module/NestedModule.pm`
- in OO Perl, if modules correspond to classes, then nesting can correspond to class hierarchy

Perl modules, simple example

- module file Greetings.pm

```
package Greetings;
sub hi {print “Hi!\n”};
```

- usage:

```
$ perl -e 'use Greetings; Greetings::hi;'
```

- modules can exist without being in separate files

```
perl -e 'package A;sub hi{print"hi\n"}; package B; A::hi'
```