Research Skills 1: Programming
Lesson 5

Erik Tjong Kim Sang
Herman Stehouwer

4 October 2007

Week 1: Variables and number processing

- numeric variables start with a dollar sign: `$yearOfBirth`
- several arithmetic operators are available: `+ - * / % **`
- as well as several functions: `abs() int() rand() sqrt()`
- input can be read from the keyboard: `<STDIN>`

Week 2: Control structures

- Conditional structures: `if (condition) { command }`
- Truth expressions: `and`, `or` and `not`
- Iterative structures: `while ()`, `for(;;)` and `foreach ()`

Week 3: String processing

- Basic string operations: concatenation (`.`); testing (`eq` and others)
- String substitution: `tr///`, `s///` and `//`
- Regular expressions: special tokens like `\s`, `$` and `*`

Week 4: Lists and hashes

- List basics: `@list = (1,2,3)` and `$list[0] = "a"`
- List functions: e.g. `push(),pop()` and `sort()`
- Hash basics: `%hash = qw(key1 value1 key2 value2)`
  `$hash{"key3"} = "value3"`
- Hash functions: e.g. `keys(),values()` and `exists()`

SUBROUTINES

Subroutines: overview

- subroutines are blocks of code with a name
- subroutine references are preceded by `&` like in `&subroutine()`
- subroutines can use a list of parameters (possibly empty)
- example of subroutine definition and call:

```
sub subroutine { print "will print this line\n"; }
&subroutine();
```

Subroutines and Perl module files

- ideally, a subroutine is general and reusable
- subroutines can be stored in separate files: e.g. `file.pm`
- included these in your program like this: `require "file.pm";`
- inclusion from other directories: add `use lib "/directory";`
- note: .pm files need to finish with `1;`
  (to avoid the error message ”did not return a true value”)

Variable scope (1)

A Perl variable with a certain name may refer to different entities in different locations. First example:

```
$a = 0;
print "$a\n"; # 0
sub changeA { $a = 1; }
print "$a\n"; # 0
&changeA();
print "$a\n"; # 1
```

The printed value of `$a` changes because the `$a` in `&changeA()` is a global variable.

Variable scope (2)

Second example:

```
my $a = 0;
print "$a\n"; # 0
sub changeA { my $a = 1; }
print "$a\n"; # 0
&changeA();
print "$a\n"; # 0
```

The printed value of `$a` stays the same because the `$a` in `&changeA()` is a local variable.

Input and output of subroutines

- subroutines store their parameter list in `@_`

```
&subroutine(1,2,3);
sub subroutine { ($a,$b,$c) = @_; ...
```

- they return results in a list

```
sub subroutine {
    return(1,2,3);
}
($a,$b,$c) = &subroutine();
```

Ugly programming

In subroutines, you can change values of variables without naming them:

```
$abc = 1;
sub subr { $_[0] = 2; }
&subr($abc);
print "$abc\n"; # will print 2!
```

Please do not write programs like this. Instead, define the context of each variable by defining them with `my()` (even the global variables).

Input and output of Perl programs

- like subroutines, complete programs can also communicate
- subroutines store parameters in the list `@_`; programs use `@ARGV`
- subroutines output stuff with `return(1)`; programs use `exit(1)`

Example: when a program is called like `perl -w 5.2.pl the`
then the value of `$ARGV[0]` in the program will be `the`

PROGRAMMING EXAMPLE

Programming example: task

Write a program that can translate English sentences to Dutch and the other way around. The translation direction will be determined by an optional command line argument, either `d-e` for Dutch to English translation (default) or `e-d` for the other direction. It is sufficient that the program has a small dictionary of about five words. Assume that unknown words have themselves as a translation.

Divide and conquer

Divide the task in subtasks. It is easier to write software for small well-defined tasks. Here is an example division for the translation task:

- determine translation direction
- repeat forever
  - read text
  - translate
  - print result

Data structures

From this week onwards, we will ask you to define the variables at the start of your program. This will decrease problems caused by undefined variables and misspelled variable names.

```
use strict; # make definition of variables compulsory

my %dict = qw(Jan John
              en and
              Marie Mary
              gingen went
              naar to
              het the);
```

Determine translation direction (1)

Goal: reverse translation dictionary when English to Dutch translation is required (command line argument `e-d`).

First attempt:

```
sub detTrMod { if ($ARGV[0] eq "e-d") {%dict = reverse(%dict);} }
```

Main problem of this code: error detection.

Determine translation direction (2)

```
# determine translation direction from first argument
sub detTrMod {
    if (defined $ARGV[0] and $ARGV[0] eq "e-d") {
       # English - Dutch required:  reverse dictionary
       %dict = reverse(%dict);
    } elsif (not defined $ARGV[0] or $ARGV[0] ne "d-e") {
       # argument neither e-d nor d-e
       print "usage:  perl -w translate.pl e-d|e-d\n";
       exit(1);
    }
    # remove direction from argument list
    shift(@ARGV);
}
```

Translation code

There are two ways to implement a subroutine for translating a sentence:

- first it translates the first word, then the second word and so on (iteration)
- first it translates the first word and then it calls itself for translating the rest of the sentence (recursion)

Translation by iteration

```
sub translate {
    my @translation = ();
    my $word;
    foreach $word (@_) {
       if (defined($dict{$word})) { # known word
          push(@translation,$dict{$word});
       } else { # unknown word
          push(@translation,$word);
       }
    }
    return(@translation);
}
```

Translation by recursion

```
sub translate {
    if (not @_) { return(); } # nothing to translate
    else {
       my ($word,@rest) = @_;
       if (defined($dict{$word})) { # known word
          return($dict{$word},&translate(@rest));
       } else { # unknown word
          return($word,&translate(@rest));
       }
    }
}
```

Main program body (1)

```
my $text = "start text; will be ignored";
my @text = (); # input text
my @translated; # translated text
my $translated; # translated text

&detTrMod(); # determine translation direction
while (defined($text) and $text ne "") {
    print "> ";
    $text = <STDIN>;
    if (defined $text) { chomp($text); }
    if (defined($text) and $text ne "") {
```

Main program body (2)

```
        # remove non-word characters
        $text =~ tr/[a-zA-Z0-9 ]//cd;
        # convert string $text to list @text
        @text = split(/\s+/,$text);
        # translate words
        @translated = &translate(@text);
        # convert translated word list to string
        $translated = join(" ",@translated);
        print "$translated\n";
    }
}
```

EXERCISES WEEK 4

Exercise results week 4

mark	1	2	3	4	5
9.8	♣	♣	♣	♠	♣
9.6	♠	♣	♣	♣	♣
9.5	♠	♣	♠	♠	♠
9.2	♠	♣	♣	♠	♣
6.9	♣	♣	♠		
6.9	♠	♣	♣		
6.4	♣	♠	♠		
5.9	♣	♣	♠		
5.3	♣	♣	♠		

mark	1	2	3	4	5
5.3	♣	♣	♠		♠
5.1	♠	♣			♠
4.8	♠	♠	♠		
4.8	♠	♠	♠		
4.3	♠	♣			
3.7	♣	♠	♠		
3.2	♠	♠			
2.7	♣				
1.6	♠	♠	♠		

♣ = perfect; ♠ = one or more errors

Common problem in the exercise solutions

Some functions change their argument (like `chomp`) while others leave it unchanged and return results (like `lc`).

argument change	**returns result**	**both**
chomp(), push()	exists(), defined(), lc()	pop(), shift()
delete(), unshift()	sort(), join(), values()	splice()
	reverse(), keys(), split()	

So most functions require additional actions for the collection of results.

Programming tips

- start with making a top-level program description
- use comments to explain the code
- use newlines to separate unrelated blocks of code
- use indentations for optional command blocks
- use meaningful variable names
- test your programs with option -w: `perl -w program.pl`
- use extra print statements to check values of variables

START WITH EXERCISES AT
http://ifarm.nl/erikt/perl2007/