

## Corpustaaikunde 2009: Perl

Erik Tjong Kim Sang  
Gosse Bouma  
22 April 2009

### Previous topics

- variables (names start with \$) and arithmetic (+ - \* / % \*\*)
- reading input from the keyboard: <STDIN>
- conditional structures, truth expressions, iterative structures
- strings and regular expressions
- lists and hashes
- subroutines

### Topics of this week

- subroutines (example)
- working with files
- error messages and testing

## PROGRAMMING EXAMPLE (SUBROUTINES)

### 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

We will 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: change direction of 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: complain
        print "usage: perl -w translate.pl e-d|d-e\n";
        exit(1);
    }
    # remove direction from argument list
    shift(@ARGV);
}

```

Informatiekunde

8

**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)

Informatiekunde

9

**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);
}

```

Informatiekunde

10

**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));
        }
    }
}

```

Informatiekunde

11

**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 "") {

```

Informatiekunde

12

**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";
}
}

```

Informatiekunde

13

**FILE HANDLING****File handling: overview**

- file formats: character encodings
- file operations: open, read, write, close
- other topics: file handles, system commands
- programming tips: error handling and testing

Informatiekunde

15

### File encodings

We work with text files which can be encoded in several ways:

- ASCII: seven-bit encoding covering the usual keyboard characters
- ISO 8859-1: eight-bit encoding which includes diacritics
- UTF-8: 32-bit encoding covering characters from many languages

### Document structure

Apart from characters, documents also contain different structures: headings, paragraphs, lists, tables, ... which can also be defined in different ways:

- with human-readable text codes, like in HTML
- with binary codes, like in Word's DOC format

### File operations: opening

- `open(INFILE,"myfile")`: reading
- `open(OUTFILE,">myfile")`: writing
- `open(OUTFILE,">>myfile")`: appending
- `open(INFILE,"someprogram |")`: reading from program
- `open(OUTFILE,"| someprogram")`: writing to program
- `opendir(DIR,"mydir")`: open directory

### File handles

- INFILE and OUTFILE mentioned above are file handles
- these are file variables: conventionally with CAPITAL names
- there are three special file handles: STDIN, STDOUT and STDERR
- STDIN is the familiar input handle; other two are for output
- use STDERR for sending error messages to

### File operations: reading

- `$a = <INFILE>`: read a line from INFILE into \$a
- `@a = <INFILE>`: read all lines from INFILE into @a
- `$a = readdir(DIR)`: read a filename from DIR into \$a
- `@a = readdir(DIR)`: read all filenames from DIR into @a
- `read(INFILE,$a,$length)`: \$length characters from INFILE into \$a

### File operations: writing and closing

- `print OUTFILE "text"`: write some text in OUTFILE
- `close(FILE)`: close a file
- `closedir(DIR)`: close a directory

### More file instructions (1)

- `binmode(HANDLE)`: change file mode from text to binary
- `unlink("myfile")`: delete file myfile
- `rename("file1","file2")`: change name of file file1 to file2
- `mkdir("mydir")`: create directory mydir
- `rmdir("mydir")`: delete directory mydir

### More file instructions (2)

- `chdir("mydir")`: change the current directory to mydir
- `system("command1")`: execute command command1
- `die("message")`: exit program with message message
- `warn("message")`: warn user about problem message
- Example: `open(INFILE,"myfile")` or `die("cannot open myfile!")`

### Examples of print and printf

```
print <<"EOF";
this text will be printed
the fact that it spans four lines
is no problem
variable $x will be evaluated as well
EOF

# this will print "1 1.234 1.23\n"
printf "%d %-7s %4.2f\n",1.234,1.234,1.234;
```

## ERROR MESSAGES AND TESTING

### Error messages and warnings

After you have made a plan for your program and written the first version of the code, you need to check if your code is working properly.

Perl can assist you in this task by providing error messages for instructions that are invalid and warnings for code that it finds suspicious.

In order to benefit most from these diagnostic messages, always start your programs with `use strict` and always run Perl with the warning option: `perl -w program.pl`

### Example program (version 1)

```
# program.pl
use strict;
$a = 1
$b = 2
$c = $a/$b
printf "%d", $c

# example run
$ perl -w program.pl
Scalar found where operator expected at line 4, near "$b"
(Missing semicolon on previous line?)
syntax error at program.pl line 4, near "$b "
```

### Example program (version 2)

```
# program.pl
use strict;
$a = 1;
$b = 2;
$c = $a/$b;
printf "%d", $c;

# example run
$ perl -w program.pl
Global symbol "$c" requires explicit package name at line 5.
Global symbol "$c" requires explicit package name at line 6.
Execution of program.pl aborted due to compilation errors.
```

### Example program (version 3)

```
# program.pl
use strict;
my $a = 1;
my $b = 2;
my $c = $a/$b;
printf "%d", $c;

# example run
$ perl -w program.pl
0$
```

### Example program (version 4)

```
# program.pl
use strict;
my $a = 1;
my $b = 2;
my $c = $a/$b;
printf "%d\n", $c;

# example run with debugger
$ perl -d program.pl
Loading DB routines from perl5db.pl version 1.27
Editor support available.

Enter h or 'h h' for help, or 'man perldebug' for more help.
```

### Example program (debugging version 4)

```
main::(program.pl:3): my $a = 1;
DB<1> n
main::(program.pl:4): my $b = 2;
DB<1> n
main::(program.pl:5): my $c = $a/$b;
DB<1> n
main::(program.pl:6): printf "%d\n", $c;
DB<1> p$a
1
DB<3> p$c
0.5
```

### Example program (version 5)

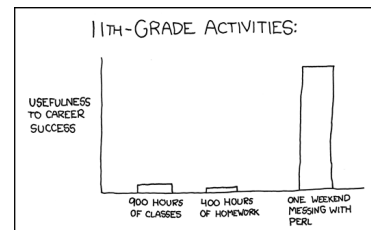
```
# program.pl
use strict;
my $a = 1;
my $b = 2;
my $c = $a/$b;
printf "%3.1f\n", $c;

# example run
$ perl -w program.pl
0.5
$
```

### About testing your code

- use arguments for passing data to subroutines
- use `return()` for passing data out of subroutines
- this way you can test each subroutine separately
- always test boundary cases: no input, single input (lowest/highest)

**THE END**



Source: <http://kkcd.com/519/>