UNIVERSITETET I OSLO
DET MATEMATISK-NATURVITENSKAPELIGE FAKULTET

Mer om Perl

- Filer og bruk av disse
- Lister
- Tabeller
- Søking og regulære uttrykk
- Oppsummering

INF3110/4110

Perl

Fil-operasjoner

Et program som skal foreta tekstbehandling, må kunne håndtere filer på en enkel måte. Det kan Perl.

Åpning av filer for lesning

Operatoren open brukes til dette:

```
open(F, "fil.data");
```

Filvariable (som F) har ingen spesialtegn (som «$») først.

Man bør alltid sjekke om en fil finnes, og den vanlige måten å gjøre dette på i Perl er denne:

```
open(F, "fil.data") or
  die "Kan ikke lese ‘fil.data’.\n";
```

Åpning av filer for skriving

Ved å benytte en «>» foran filnavnet angir vi at filen skal åpnes for skriving:

```
open(FX, ">fil.data") or
  die "Kan ikke lage ‘fil.data’.\n";
```

Å lese fra filer

I Perl leser man nesten alltid filen linje for linje:

```
$linje = <F>;
```

Linjeskilletegnet («\n») følger med på lasset; den kan fjernes med kallet chomp($linje).

Ofte benytter man en løkke som leser linjene inn i $_:

```
while (<F>) {
  print unless $_ eq "\n";
}
```

Denne koden kopierer alle ikketomme linjer.

Å skrive til filer

print-setningen brukes også for å skrive til filer:

```
print FX "En test.\n";
```

Da er det intet komma etter filvariabelen!

Å lese og skrive mot programmer

Man kan lese fra et program i stedet for fra en fil:

```
open(DIR, "ls -l |") or
  die "Klarte ikke kjøre ‘ls’.\n";

while ($fil = <DIR>) { . . . }
close(DIR);
```

Tilsvarende kan man sende utskriften rett til et program:

```
open(PRINTER, "| print -pipe") or
  die "Klarte ikke å kjøre ‘print’.\n";

print PRINTER "Dette er en utskrift.\n";
close(PRINTER);
```

Filer à la Unix-filtre

Unix-filtre som cat, head, sort, grep og andre bruker filer på en spesiell måte:

- cat fil1 fil2 fil3
  vil lese de tre angitte filene i rekkefølge.

- cat
  uten filparametre vil lese fra standard innfil (som vanligvis er tastaturet).

I Perl finnes en egen operator for dette: <>. Perls versjon av cat kan derfor skrives slik:

```
#! /store/bin/perl -w

while (<>) { print; }
exit 0;
```

Dette skjer:

❶ Hver linje leses inn i $_.

❷ print skriver ut linjen.

❸ Linjeskilletegnet («\n») følger med inn i $_ og skrives dermed ut av print.

❹ Til sist avsluttes med statusverdi 0.

Datastrukturer

Perl har kun to typer datastrukturer: *lister* og *tabeller*.

Lister

Perls lister tilsvarer vektorer i Java og C, men er mer fleksible. Antallet elementer behøver ikke angis, og det kan variere over tid. Listevariable har alltid en @ først i navnet sitt:

```
@navn = ("Dag","Anne","Irene","Frøydis");
```

sort sorterer en liste:

sort(@navn) $\overset{\text{gir}}{=\!=\!\Rightarrow}$ ("Anne", "Dag", "Frøydis", "Irene")

shift fjerner første element i listen:

shift(@navn) $\overset{\text{gir}}{=\!=\!\Rightarrow}$ "Dag"

Nå har listen kun tre elementer.

pop fjerner siste elementet i listen:

pop(@navn) $\overset{\text{gir}}{=\!=\!\Rightarrow}$ "Frøydis"

Nå er kun "Anne" og "Irene" i @navn .

INF3110/4110

push legger ett eller flere nye elementer bakerst i listen:

push(@navn,"Margrete") $\overset{\text{gir}}{=\!=\!\Rightarrow}$ ("Anne","Irene","Margrete")

Man kan også be om et vilkårlig element i listen:

$navn[0] $\overset{\text{gir}}{=\!=\!\Rightarrow}$ "Anne"

Nummereringen starter med 0.

NB! Her skal det brukes $ siden det er snakk om *ett element*.
Hakeparentesene ([...]) angir at det er snakk om en liste.

Siste element i listen @navn har indeksen $#navn:

$navn[$#navn] $\overset{\text{gir}}{=\!=\!\Rightarrow}$ "Margrete"

Kontekst

Alle vanlige Perl-operatorer kan brukes i to sammenhenger (såkalte *kontekster*):

Skalar kontekst er når det forventes én verdi (tall eller tekst), som i

$$\$leng = @navn \overset{gir}{\Longrightarrow} 3$$

Listekontekst er når det forventes en liste:

$$@sn = reverse(sort(@navn)) \overset{gir}{\Longrightarrow} ("Margrete","Irene","Anne")$$

(Operatoren reverse snur en liste.)

Effekten av en operator vil altså avhenge av i hvilken sammenheng den brukes. Vi kan tenke oss at det er to beslektede operatorer med samme navn.

Noen ganger kan dette ha en uventet effekt. I setningen

```
push(@liste, <F>);
```

er andre parameter i listekontekst, så *alle* resterende linjer av F vil bli lest inn og lagt inn i @liste!

Konklusjon: Les læreboken, særlig hvis det skjer overraskende ting.

Tabeller

Tabeller («hash»-er) er som lister, men indeksen (nøkkelen) er en tekst i stedet for et heltall. Tabellnavn starter alltid med en %:

```
%alder = ( "Dag" => 49, "Anne" => 43 );

print "Dag er ", $alder{"Dag"}, " år.\n";
```

vil skrive «Dag er 49 år.» Her er det krøllparentesene ({. . .}) som angir at det er snakk om en tabell.

Det er lett å sette inn nye elementer:

```
$alder{"Irene"} = 18;
```

Nyttige operasjoner for tabeller er følgende:

keys gir en liste med alle tabellens nøkler i vilkårlig rekkefølge.

values gir en liste med alle verdiene.

Utskrift av en tabell

For å skrive ut en tabell, må vi gå gjennom den element for element:

```
foreach (sort(keys(%alder))) {
  print "$_ er $alder{$_} år.\n.";
}
```

Dette gir følgende utskrift:

```
Anne er 43 år.
Dag er 49 år.
Irene er 18 år.
```

Eksempel
Én liste og én tabell er alltid definert:

@ARGV inneholder programmets parametre.

%ENV inneholder alle omgivelsesvariablene.

Unix-programmet printenv skriver ut definerte omgivelsesvariable:

```
> printenv USER
dag
> printenv PRINTER
lucida
```

I Perl kan vi skrive en forbedret versjon arg som kan håndtere mer enn én parameter:

```
#! /store/bin/perl -w

foreach (@ARGV) {
    print "$_ = $ENV{$_}\n" if $ENV{$_};
}
exit 0;
```

Den brukes slik:

```
> arg USER XX PRINTER
USER = dag
PRINTER = lucida
```

Søking i tekst

Den aller sterkeste siden av Perl er mekanismene for søking i tekst. Det er mulig å søke i henhold til enkle eller svært kompliserte regulære uttrykk med operatoren m (for «match»).

$$\$var =\sim m/Page:/; \overset{gir}{\Longrightarrow} 1$$

hvis teksten i $var inneholder tegnene "Page:".

Regulære uttrykk

Et **regulært uttrykk** er et mønster som det kan søkes etter. Det kan bestå at en fast tekst som i m/Page:/ der mønsteret er på fem tegn.

"On Page 4 we find"	Nei, intet «:»
"On page: 4 we see"	Nei, «p» ikke «P»
"there. Page: Here we"	Ja
"Page: 4 4"	Ja

Spesielle tegn i regulære uttrykk

Ofte er ikke en fast tekst nok. Vi kan for eksempel være interessert i at vårt mønster skal stå helt først i teksten.

- Tegnene «^» og «$» angir henholdsvis starten og slutten av teksten. Hvis mønsteret er m/^Page:/, vil kun siste tekst på forrige ark bli godtatt.

- Tegnet «.» angir et vilkårlig tegn. Anta at mønsteret er m/^.age:$/, vil følgende bli godtatt:

"Page: 4 4"	Nei (ikke sist)
"Page:"	Ja
"page:"	Ja
"Tage:"	Ja
"Page="	Nei (ikke «:»)

- Hvis vi ønsker å sjekke et tegn som normalt tolkes som spesialtegn (som «^», «$» eller «.»), kan vi sette en «\» foran.

  m/\$/ vil sjekke om teksten inneholder et dollartegn.

- Konstruksjonene «\d», «\w» og «\s» angir henholdsvis et siffer, et alfanumerisk tegn (en bokstav, et siffer eller «_») og et tegn med luft (blank, tabulator, linjeskift el). Med mønsteret m/^Page:\s\d\s\d$/ vil følgende bli godtatt:

"Page: 4 7"	Ja
"Page:0 0"	Nei (ikke blank etter «:»)
"Page: 3 17"	Nei (mer enn ett siffer)
"Page: 3 8 "	Nei (blank sist)

- Tegnene «*», «+» og «?» angir repetisjoner:
  - * 0 eller flere ganger
  - + 1 eller flere ganger
  - ? 0 eller 1 gang
- Parenteser kan brukes til å gruppere deler av mønsteret.
- Tegnet «|» angir alternativer.

Mønsteret vårt blir da
m/^(Side|Page):\s*(\d+)\s+(\d+)\s*$/.

Sideeffekter

Etter et vellykket søk vil $1 inneholde det som passet i første parentessett, $2 det andre, osv. Dermed kan man plukke frem akkurat de deler av teksten man ønsker:

```
if (m/^(Side|Page):\s*(\d+)\s+(\d+)\s*$/) {
  $norsk = ($1 eq "Side");
  $num1 = $2;
  $num2 = $3;
}
```

Eksempel

Regulære uttrykk er meget kraftige; man kan uttrykke mye med få tegn. Et engelsk desimaltall kan for eksempel forekomme i flere former:

"2", "3.", "-0.7", ".32",

men ikke "" (tom), ".", "1.2.3" eller "2.a". Et passende regulært uttrykk er

m/^-?(\d+\.?\d*|\.\d+)$/

Alternativ koding

I et språk uten regulære uttrykk, må slikt kodes ved å gi en algoritme:

```
static boolean isNumber (String s) {
    int i = 0, len = s.length();

    if (i<len && s.charAt(i)=='-') ++i;
    if (i<len && s.charAt(i)=='.') {
        ++i;
        if (i >= len) return false;
        while (i<len && Character.isDigit(s.charAt(i))) ++i;
    } else {
        if (i>=len || !Character.isDigit(s.charAt(i))) return false;
        while (i<len && Character.isDigit(s.charAt(i))) ++i;
        if (i<len && s.charAt(i)=='.') ++i;
        while (i<len && Character.isDigit(s.charAt(i))) ++i;
    }
    return i==len;
}
```

Men . . .

noen språk (som Java) har tatt høyde for slikt på annen måte:

```
static boolean isNumber2 (String s) {
    try {
        Float.parseFloat(s);
    } catch (NumberFormatException e) { return false; }
    return true;
}
```

Søking

Som nevnt benyttes m-operatoren med et regulært uttrykk til søking. Hvilken tekst det skal søkes i, angis med =~:

```
if ($t =~ m/per/i) { . . . };
```

(Modifikatoren i angir at det ikke skal skilles på små og store bokstaver.)

Hvis det skal søkes i $_, trenger man ikke nevne den:

```
if (m/per/i) { . . . };
```

Man kan til og med droppe m-en om man vil det!

Endring

Når man søker etter et mønster, er man ofte interessert i å bytte dette ut med en annen tekst. Dette gjøres med operatoren s (for «substitute»):

```
$t =~ s/Per/Kari/;
```

Funksjoner

Det er enkelt å lage funksjoner i Perl. Parametre overføres i listen @_. Lokale variable deklareres med my (men $_ med local).

```
sub Sum {
    my $res = 0;
    local $_;

    foreach (@_) { $res += $_; }
    return $res;
}
```

Funksjoner kan stå omtrent hvor som helst i koden, men plasseres vanligvis sist.

Funksjoner kalles med & foran navnet:

$$\&Sum(1,2,3,7) \overset{\text{gir}}{\Longrightarrow} 13$$

Min personlige vurdering

Sterke sider ved Perl

Det jeg liker best ved Perl er følgende:

- Moro å programmere (men smaken kan variere).
- Lett å programmere avansert tekstbehandling.
- Alt man trenger for å kommunisere med operativsystemet (spesielt Unix).
- Ingen begrensninger på linjelengde ol.
- Ingen problemer med ulike implementasjoner.
- God dokumentasjon med svært mange *nyttige* eksempler.
- Stor og hjelpsom brukergruppe.

- Perl er rask!

Eksempel
Finn antall sider i PostScript-filer, dvs antall linjer med «%%Page:».†

grep er et Unix-søkeprogram:

```
grep -c '^%%Page:' filer
```

Tid: 0,06 sekunder.

Perl trenger syv linjer:

```
#! /store/bin/perl

$N = 0;
while (<>) {
    ++$N if /^%%Page:/o;
}
print "$N\n";
exit 0;
```

Tid: 2,00 sekunder.

† Til testingen ble det brukt tyve filer på tilsammen 96 Mbyte; de inneholdt totalt 3 608 210 linjer.

Java trenger ca 24 linjer:

```
import java.io.*;

class Pages {
    public static void main (String arg[]) {
        long sum = 0;
        for (int i = 0;  i < arg.length;  ++i)
            sum += nPages(arg[i]);
        System.out.println("Sum = " + sum);
    }

    static long nPages (String fName) {
        long res = 0;
        try {
            BufferedReader f =
                new BufferedReader(new FileReader(fName));
            String line = f.readLine();

            while (line != null) {
                if (line.startsWith("%%Page:")) ++res;
                line = f.readLine();
            }
            f.close();
        } catch (Exception e) {
            System.err.println("Read error: "+e);
        }
        return res;
    }
}
```

Tid: 3,00 sekunder.

C trenger ca 21 linjer:

```
#include <stdio.h>

int n_pages (char *f_name)
{
  FILE *f = fopen(f_name,"r");
  char line[2048];
  int res = 0;

  while (fgets(line, 2048, f)) {
    if (strncmp(line,"%%Page:",7) == 0) ++res;
    while (line[strlen(line)-1] != '\n') {
      if (! fgets(line, 2048, f)) return res;
    }
  }
  return res;
}

int main (int argc, char *argv[]) {
  int i, sum = 0;
  for (i = 1;  i < argc;  ++i)
    sum += n_pages(argv[i]);
  printf("%d\n", sum);
  return 0;
}
```

Tid: 1,30 sekunder.

Ulemper ved Perl

Perl bryter de fleste reglene for språkdesign:

- Perl er blitt veldig stort; i dette kurset har jeg beskrevet anslagsvis 25% av språket. (Men disse 25% brukes i 95% av koden de fleste skriver.)

  Eksempel
  I uttrykk er operatorene fordelt på 24 nivåer.

- Det er veldig mange spesialtilfeller.

  Eksempel
  I testen

  ```
  while (<F>) ...
  ```

  plasseres leste linje i $_, men ikke i

  ```
  if (<F>) ...
  ```

- Det er for lett å skrive uleselige programmer i Perl.

Eksempel
I programmet psfilter som plukker ut tekst fra en PostScript-fil, finnes følgende kode:

```
while (<>) {
    $AnyFound = 0;
    s/\\(/\x1b/g;  s/\\)/\x1c/g;  s/\\\/\x1d/g;
    s/\)[lmnoqrst]\(//g if $Dvips;

    while (s/^([^(]*)\(([^()]*(\([^()]*\))?[^()]*)\)//) {
        $T = $2;
        $T =~ s/\\(\d{1,3})/sprintf("%c",oct($1))/eg;
        print " " x length($1), " $T ";  $AnyFound = 1;
    }
    print "\n" if $AnyFound || $Keep;
}
```

Dette er typisk «bruk og kast»-kode.

- Mange operatorer har en uventet effekt:

Eksempel
I programmet

```
#! /store/bin/perl -w

$X = $Y = "004";
print "Ja\n" if $X == 1;

print "X = ", $X, ", Y = ", $Y, "\n";
++$X;  ++$Y;
print "X = ", $X, ", Y = ", $Y, "\n";

exit 0;
```

brukes $X i en test. Dette medfører at ++$X får en annen effekt:

```
X = 004, Y = 004
X = 5, Y = 005
```

Konklusjon

- Perl er et meget nyttig språk til mange problemer.
- Overraskende mange problemer lar seg løse usedvanlig lett med Perl.
- Jeg ville ikke skrevet store programmer (dvs >1000 linjer) i Perl.
- Når man programmere Perl, blir man alltid overrasket over hva som skjer.
  - Løsningen er å bygge programmer i små steg.
  - Test og test igjen i hvert steg.