Tema for siste forelesning:

Versjonskontroll

- CVS og Subversion

Kodegenerering

- Funksjoner

Testing

- Ulike testprogrammer

Når flere samarbeider

Når flere jobber sammen, kan man tråkke i beina på hverandre:

1. Per tar en kopi av en kildefil og begynner å rette på den.
2. Kari gjør det samme.
3. Kari blir første ferdig og kopierer filen tilbake.
4. Per blir ferdig og kopierer filen tilbake. Karis endringer går tapt.

Løsningen

Et *versjonskontrollsystem* er løsningen.

De fleste slike systemer er *utsjekkingssystemer* basert på *låsing*:

1. Per ber om og *sjekker ut* (dvs får en kopi av) filen og begynner å rette på den.
2. Kari ber om en kopi, men får den ikke fordi den er *låst*.

Først når Per er ferdig og *sjekker inn* filen, kan Kari få sin kopi.

Fordeler med et slikt utsjekkingssystem:

- Lettforståelig.
- Ganske sikkert.

  (Men hva om Per og Kari begge må rette i to filer hver? Da kan de starte med hver sin fil, men når de er ferdige med den første, finner de at den andre er sjekket ut.)

Ulemper:

- Kari bør kunne få en lese-kopi selv om Per jobber med filen. (Noen systemer tillater det, men ikke alle.)
- Hva om Per glemmer å legge tilbake filen?
- Det burde vært lov for Per og Kari å jobbe på ulike deler av filen samtidig.

Insjekkingssystemer

En bedre løsning er *innsjekkingssystemer*:

- Alle kan når som helst sjekke ut en kopi.
- Ved innsjekking kontrolleres filen mot andre innsjekkinger:
  - Hvis endringene som er gjort, ikke er i konflikt med andres endringer, *blandes* endringene med de tidligere.
  - Ved konflikt får brukeren beskjed om dette og må manuelt klare opp i sakene.

Et scenario

1. Per sjekker ut en kopi av en fil. Han begynner å gjøre endringer i slutten av filen.
2. Kari sjekker ut en kopi av den samme filen. Hun endrer bare i begynnelsen av filen.
3. Per sjekker inn sin kopi av filen.
4. Kari sjekker inn sin kopi, og systemet finner ut at de har jobbet på hver sin del. Innsjekkingen godtas.

Når man er alene

Selv om du jobber alene med et prosjekt, kan det være svært nyttig å bruke et versjonskontrollsystem:

- Man kan enkelt finne frem tidligere versjoner.
- Det kan hende man jobber på flere datamaskiner.

CVS og Subversion

Det mest kjente innsjekkingssystemet er **CVS** («Concurrent Versions System») laget i 1986 av *Dick Grune*. Det er spesielt mye brukt i Unix-miljøer.

For å bøte på noen svakheter i CVS laget firmaet *CollabNet* **Subversion** i 2000. Det ble en del av *Apache* i 2010.

Gratis implementasjoner finnes for alle vanlige operativsystemer; se http://subversion.apache.org/.

Nære og fjerne systemer

Subversion kan operere på to ulike måter:

- Alt skjer i det lokale filsystemet.
- Man kan starte en Subversion-tjener på en maskin og så sjekke inn og ut filer over nettet.

Vi skal gjøre det siste og bruke Ifis Subversion-tjener.

Opprette et *repository*

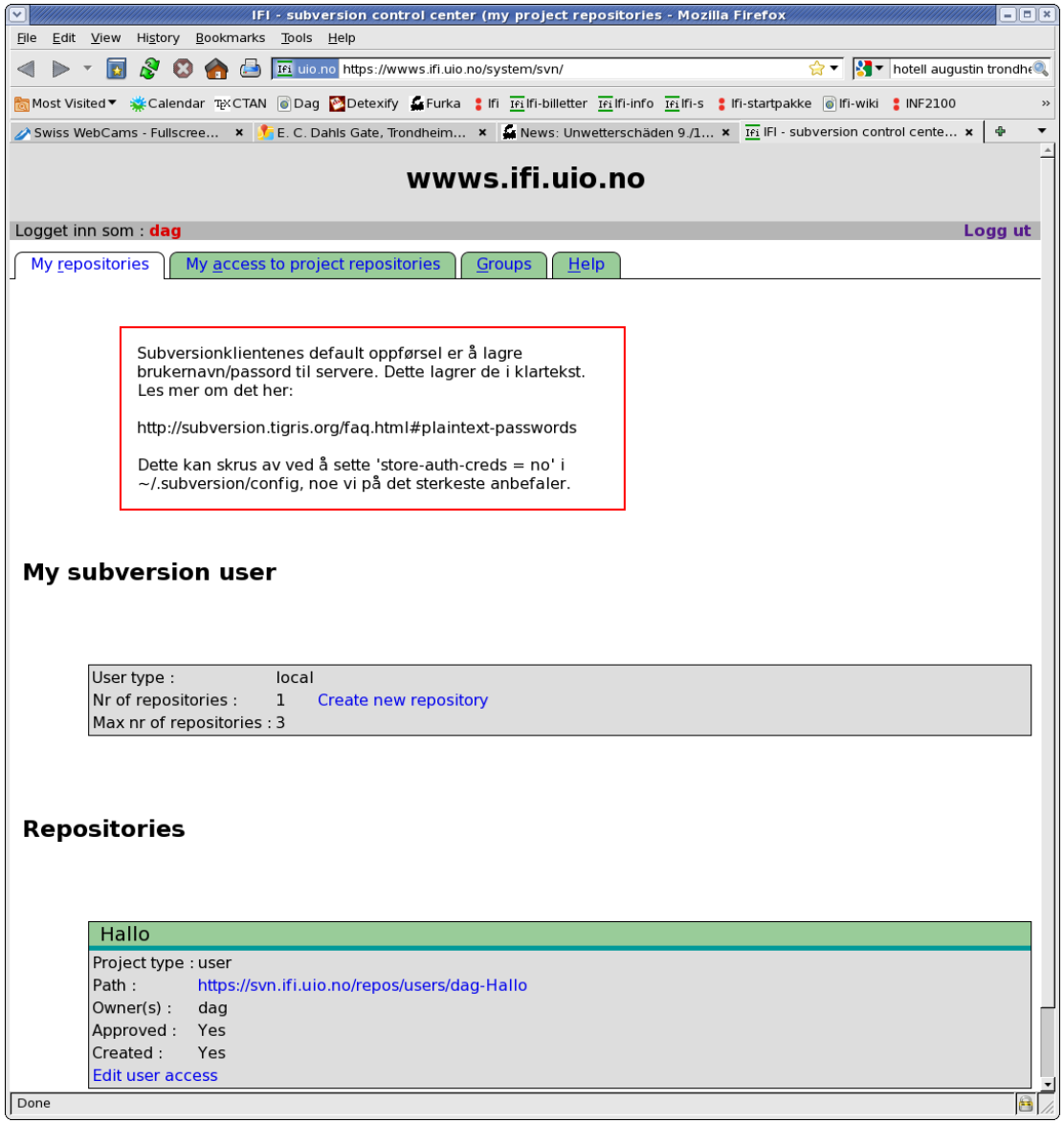
1. Gå inn på nettsiden

   https://wwws.ifi.uio.no/system/svn/

2. Logg inn.
3. Velg «My repositories» og «Create new repository». (I dette eksemplet heter det Hallo.)

   (Alle kan lage inntil tre «repositories».)
4. Hvis det er flere på prosjektet, velg «Edit user access».

Subversion


IFI - subversion control center (my project repositories - Mozilla Firefox
File Edit View History Bookmarks Tools Help
https://wwws.ifi.uio.no/system/svn/
Most Visited Calendar CTAN Dag Detexify Furka Ifi Ifi-billetter Ifi-info Ifi-s Ifi-startpakke Ifi-wiki INF2100
Swiss WebCams - Fullscree... E. C. Dahls Gate, Trondheim... News: Unwetterschäden 9./1... IFI - subversion control cente...
wwws.ifi.uio.no
Logget inn som : dag
Logg ut
My repositories
My access to project repositories
Groups
Help
Subversionklientenes default oppførsel er å lagre brukernavn/passord til servere. Dette lagrer de i klartekst. Les mer om det her:
http://subversion.tigris.org/faq.html#plaintext-passwords
Dette kan skrus av ved å sette 'store-auth-creds = no' i ~/.subversion/config, noe vi på det sterkeste anbefaler.
My subversion user
User type : local
Nr of repositories : 1 Create new repository
Max nr of repositories : 3
Repositories
Hallo
Project type : user
Path : https://svn.ifi.uio.no/repos/users/dag-Hallo
Owner(s) : dag
Approved : Yes
Created : Yes
Edit user access
Done

Legge inn filer

Så kan vi legge inn mapper. La oss lage en *gren* med mappen Hei som inneholder filen Hello.java:

```
$ cd Hei
$ svn import https://sub.ifi.uio.no/repos/users/dag-Hallo -m "2100demo"
Adding         Hei/Hello.java

Committed revision 1.
```

Opsjonen -m gir en kort beskrivelse av denne grenen.

Sjekke ut filer

Nå kan vi (for eksempel fra en annen datamaskin) hente ut mappen vår:

```
$ svn co https://sub.ifi.uio.no/repos/users/dag-Hallo
A  dag-Hallo/Hello.java
Checked out revision 1.
$ ls -l
drwxr-xr-x    3 dag      ifi-a        4096 2011-11-13 06:46 dag-Hallo
$ ls -la dag-Hallo
total 16
drwxr-xr-x    3 dag      ifi-a        4096 2011-11-13 06:46 .
drwxr-xr-x    3 dag      ifi-a        4096 2011-11-13 06:46 ..
drwxr-xr-x    6 dag      ifi-a        4096 2011-11-13 06:46 .svn
-rw-r--r--    1 dag      ifi-a         500 2011-11-13 06:46 Hello.java
```

Sjekke inn filer

Etter at filen er endret, kan vi sjekke den inn igjen:

```
$ svn commit -m"Enklere kode"
Sending        Hello.java
Transmitting file data .
Committed revision 2.
```

Vi behøver ikke nevne hvilke filer som er endret — det finner Subversion ut selv. (Etter første utsjekking inneholder mappen skjulte opplysninger om repository-et, så det trenger vi ikke nevne mer.)

Andre nyttige kommandoer

svn update . henter inn eventuelle oppdateringer fra repository.

svn info viser informasjon om mappen vår:

```
$ svn info
Path: .
URL: https://sub.ifi.uio.no/repos/users/dag-Hallo
Repository Root: https://sub.ifi.uio.no/repos/users/dag-Hallo
Repository UUID: 8c927215-bc3e-0410-a56f-b2451114731f
Revision: 2
Node Kind: directory
Schedule: normal
Last Changed Author: dag
Last Changed Rev: 2
Last Changed Date: 2011-11-13 07:02:16 +0100 (Sun, 13 Nov 2011)
```

svn diff viser hvilke endringer som er gjort:

```
$ svn diff -r 1:2
Index: Hello.java
===================================================================
--- Hello.java  (revision 1)
+++ Hello.java  (revision 2)
@@ -7,10 +7,9 @@
     Properties prop = System.getProperties();
     String versjon = prop.getProperty("java.version"); // Versjonen
     String koding = prop.getProperty("file.encoding"); // Koding
-    String hei;
+    String hei = "Hallo";

-    hei = "Hallo";
-    hei = hei + ", alle sammen!";
+    hei += ", alle sammen!";
     System.out.println(hei);
     System.out.println("Dette er versjon " + versjon);
     System.out.println("Kodingen er " + koding);
```

Funksjoner

For funksjoner må vi kunne lage kode for

1. funksjonskall
   - parametre
2. funksjonen
   - initiering
   - lokale variabler
   - avslutning
   - resultatverdi
3. return-setningen

Et eksempel

Den enklest mulige funksjonen (og et kall på den) ser slik ut:

```
int f() {
}

int main () {
   f();
}
```

```
        .globl  f
f:      pushl   %ebp                    # Start function f
        movl    %esp,%ebp
.exit$f:
        movl    %ebp,%esp
        popl    %ebp
        ret                             # End function f
```

```
        .globl  main
main:   pushl   %ebp                    # Start function main
        movl    %esp,%ebp
        call    f                       # Call f
.exit$main:
        movl    %ebp,%esp
        popl    %ebp
        ret                             # End function main
```

Kallet

Kallet skjer enkelt ved å generere en call.

(Parametre venter vi litt med.)

Initiering i funksjonen

Funksjonens navn angis som en vanlig merkelapp. (Siden alle funksjoner er globale, må vi ha med en .globl-spesifikasjon.)

Vi må starte med å initiere %EBP-registeret til å peke på parametrene og lokale variabler (etter å ha tatt vare på den gamle verdien):

```
        .globl  f
f:      pushl   %ebp                    # Start function f
        movl    %esp,%ebp
```

Avslutning av funksjonen

Ved retur må vi gjenopprette %EBP- og %ESP-registrene før tilbakehoppet med en ret-instruksjon:

```
.exit$f:
        movl    %ebp,%esp
        popl    %ebp
        ret                                     # End function f
```

Alle funksjoner får en navnelapp pga return-setningene.

Lokale variabler

Lokale variabler legges på stakken, og *vi* må beregne adressen deres:

```
int f1 ()
{
   int a;
   int b;
   int c;

   a = 1;
   b = 2;
   c = 3;
}
```

```
        .globl  f1
f1:     pushl   %ebp                    # Start function f1
        movl    %esp,%ebp
        subl    $12,%esp                # Get 12 bytes local data space
        movl    $1,%eax                 # 1
        movl    %eax,-4(%ebp)           # a =
        movl    $2,%eax                 # 2
        movl    %eax,-8(%ebp)           # b =
        movl    $3,%eax                 # 3
        movl    %eax,-12(%ebp)          # c =
.exit$f1:
        movl    %ebp,%esp
        popl    %ebp
        ret                             # End function f1
```

Følgende gjelder ved lokale variabler i funksjoner:

1. Start med en «offset»-teller $o = 0$.
2. For alle lokale variabler v:
   1. $o = o + v$.dataSize()
   2. v får «navnet» $\boxed{-o(\%ebp)}$.
3. Vi setter av plass til *alle* lokale variablene i en funksjon med

   subl $o,%esp

   (Vi dropper instruksjonen om det ikke er noen lokale variabler.)
4. Plassen frigjøres automatisk ved return når vi gjenoppretter %ESP.

Dette gjelder også for lokale vektorer:

```
double f2 ()
{
   double x;
   double y[10];
   double z;

   x = -1;
   y[0] = -2;
   z = -3;
}
```

```
        .globl  f2
f2:     pushl   %ebp                    # Start function f2
        movl    %esp,%ebp
        subl    $96,%esp                # Get 96 bytes local data space
        movl    $-1,%eax                # -1
        movl    %eax,.tmp
        fildl   .tmp                    #   (double)
        fstpl   -8(%ebp)                # x =
        movl    $0,%eax                 # 0
        pushl   %eax
        movl    $-2,%eax                # -2
        leal    -88(%ebp),%edx
        popl    %ecx
        movl    %eax,.tmp
        fildl   .tmp                    #   (double)
        fstpl   (%edx,%ecx,8)           # y[...] =
        movl    $-3,%eax                # -3
        movl    %eax,.tmp
        fildl   .tmp                    #   (double)
        fstpl   -96(%ebp)               # z =
        fldz
.exit$f2:
        movl    %ebp,%esp
        popl    %ebp
        ret                             # End function f2
```

Parametre

Vi kan gi funksjonen parametre:

```
int fd (int da, double db)
{
  double va;  int vb;
}

double two;

int main ()
{
   fd(1, two);
}
```

Konvensjonen er at parametrenes verdi overføres på stakken.

1. Før kallet må parametrene legges på stakken.

   int: pushl
   double: subl+fstpl

NB!

Parametrene må legges på stakken i *omvendt* rekkefølge!

2. Under utførelsen av funksjonen er parametrene som vanlige variabler med «navnene» **8(%ebp)**, **12(%ebp)**, **16(%ebp)** etc.
3. Ved retur skal resultatverdien ligge

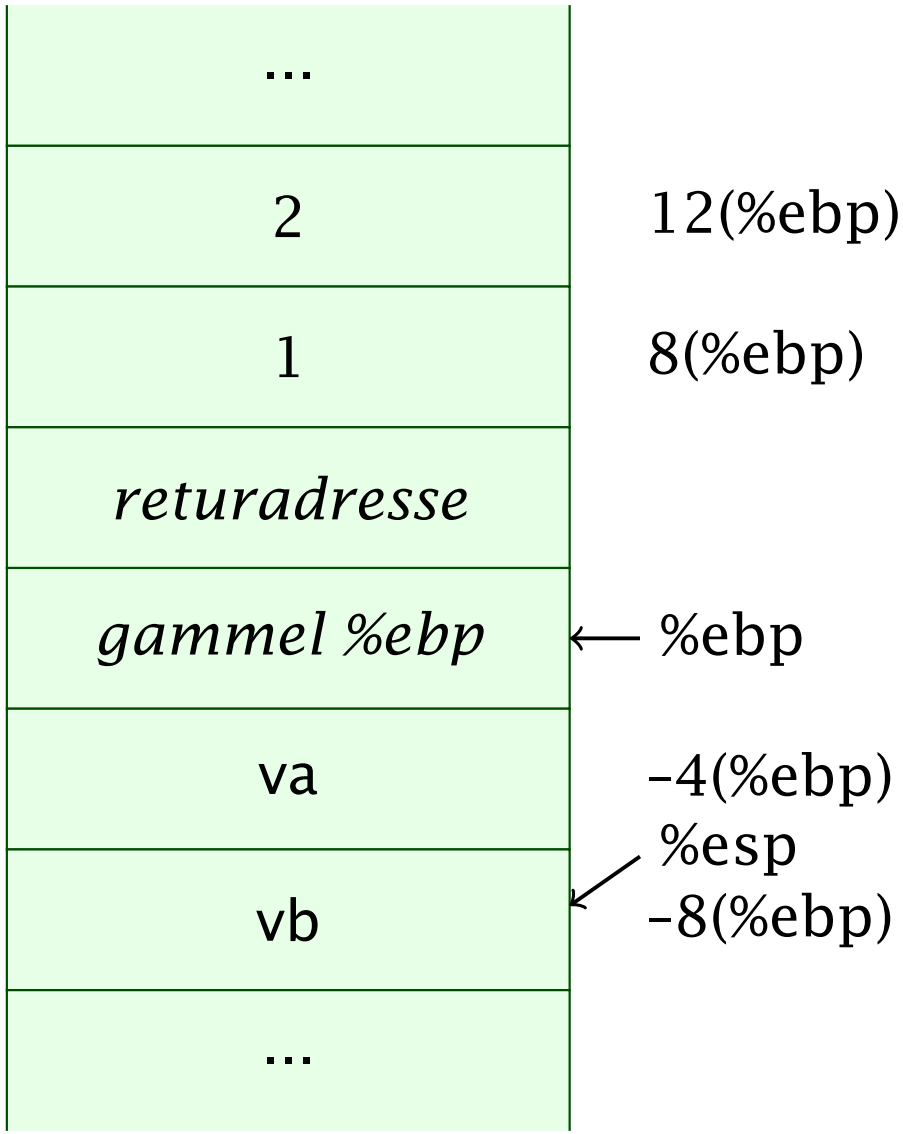
   int: %EAX
   double: på flyttallsstakken

4. Etter retur fra funksjonen må parametrene fjernes fra stakken igjen.

Slik beregnes adressene til parametrene:

1. Start med en «offset»-teller $o = 8$.
2. For alle parametre p:
   1. p får «navnet» $\boxed{o(\%ebp)}$.
   2. $o = o + p.\text{dataSize}()$
3. Siden det er kalleren som legger parametrene på stakken, trenger ikke funksjonen gjøre det.

Stakken etter kallet og initieringen:



```
double one;

int f2 (int a,
        double b,
        int c)
{}

int main ()
{
   one = 1;
   f2(0, one, 2);
}
```

```
        .globl  one
one:    .fill   8                       # double one;
        .text
        .globl  f2
f2:     pushl   %ebp                    # Start function f2
        movl    %esp,%ebp
.exit$f2:
        movl    %ebp,%esp
        popl    %ebp
        ret                             # End function f2
        .globl  main
main:   pushl   %ebp                    # Start function main
        movl    %esp,%ebp
        movl    $1,%eax                 # 1
        movl    %eax,.tmp
        fildl   .tmp                    #   (double)
        fstpl   one                     # one =
        movl    $2,%eax                 # 2
        pushl   %eax                    # Push parameter #3
        fldl    one                     # one
        subl    $8,%esp
        fstpl   (%esp)                  # Push parameter #2
        movl    $0,%eax                 # 0
        pushl   %eax                    # Push parameter #1
        call    f2                      # Call f2
        addl    $16,%esp                # Remove parameters
.exit$main:
        movl    %ebp,%esp
        popl    %ebp
        ret                             # End function main
```

Navn i generert kode

I koden vi genererer, vil navn på variabler og funksjoner se litt anderledes ut enn i C♭-programmet:

- Globale navn (både variabler og funksjoner) beholder C♭-navnet.
  - På Windows og Mac får navnet en «_» foran.
- Parametre får «navn» på formen «*n*(%ebp)» (der $n > 0$).
- Lokale variabler får samme «navn» som parametre, men $n < 0$.

Declaration.name er navnet i C♭-koden.

Declaration.assemblerName er navnet slik det skal se ut i generert kode.

return-setningen

return skal gjøre følgende:

1. Beregne resultatverdien slik at den ligger i %EAX-registeret eller på toppen av flyttallsstakken.
2. Hoppe (med en jmp-instruksjon) til avslutningen .exit$*xxxx*.

Men hva heter den funksjonen jeg er i?

Løsningen er å la genCode ha en parameter som angir funksjonen:

```
@Override void genCode(FuncDecl curFunc) {
   :
```

Et siste eksempel

```
int pot2 (int x) {
   int p2;   p2 = 1;
   while (p2 < x) {
      p2 = 2*p2;
   }
   return p2;
}

int main () {
   putint(pot2(getint()));
   putchar(10);
}
```

Programmet kjøres slik:

```
$ cflat -logB demo.cflat
$ ./demo
200
256
```

Navnebindinger (opsjonen –logB)

```
1   int pot2 (int x) {
2      int p2;   p2 = 1;
3      while (p2 < x) {
4         p2 = 2*p2;
5      }
6      return p2;
7   }
8
9   int main () {
10     putint(pot2(getint()));
11     putchar(10);
12  }
```

```
Binding: Line 2: p2 refers to declaration in line 2
Binding: Line 3: p2 refers to declaration in line 2
Binding: Line 3: x refers to declaration in line 1
Binding: Line 4: p2 refers to declaration in line 2
Binding: Line 4: p2 refers to declaration in line 2
Binding: Line 6: p2 refers to declaration in line 2
Binding: Line 10: putint refers to declaration in the library
Binding: Line 10: pot2 refers to declaration in line 1
Binding: Line 10: getint refers to declaration in the library
Binding: Line 11: putchar refers to declaration in the library
Binding: main refers to declaration in line 9
```

Generert kode

```
int pot2 (int x) {
   int p2;   p2 = 1;
```

```
                .data
.tmp:           .fill   4                       # Temporary storage
                .text
                .globl  pot2
pot2:           pushl   %ebp                    # Start function pot2
                movl    %esp,%ebp
                subl    $4,%esp                 # Get 4 bytes local data
                movl    $1,%eax                 # 1
                movl    %eax,-4(%ebp)           # p2 =
```

```
    while (p2 < x) {
        p2 = 2*p2;
    }
    return p2;
}
```

```
.L0001:                                 # Start while-statement
        movl    -4(%ebp),%eax           # p2
        pushl   %eax
        movl    8(%ebp),%eax            # x
        popl    %ecx
        cmpl    %eax,%ecx
        movl    $0,%eax
        setl    %al                     # Test <
        cmpl    $0,%eax
        je      .L0002
        movl    $2,%eax                 # 2
        pushl   %eax
        movl    -4(%ebp),%eax           # p2
        movl    %eax,%ecx
        popl    %eax
        imull   %ecx,%eax               # Compute *
        movl    %eax,-4(%ebp)           # p2 =
        jmp     .L0001
.L0002:                                 # End while-statement
        movl    -4(%ebp),%eax           # p2
        jmp     .exit$pot2              # Return-statement
.exit$pot2:
        movl    %ebp,%esp
        popl    %ebp
        ret                             # End function pot2
```

```
int main () {
   putint(pot2(getint()));
   putchar(10);
}
```

```
        .globl  main
main:   pushl   %ebp                    # Start function main
        movl    %esp,%ebp
        call    getint                  # Call getint
        pushl   %eax                    # Push parameter #1
        call    pot2                    # Call pot2
        addl    $4,%esp                 # Remove parameters
        pushl   %eax                    # Push parameter #1
        call    putint                  # Call putint
        addl    $4,%esp                 # Remove parameters
        movl    $10,%eax                # 10
        pushl   %eax                    # Push parameter #1
        call    putchar                 # Call putchar
        addl    $4,%esp                 # Remove parameters
.exit$main:
        movl    %ebp,%esp
        popl    %ebp
        ret                             # End function main
```

Hva med rekursive funksjoner?

Vårt opplegg med å legge parametre, returadresse og lokale variabler på stakken gjør at rekursive funksjoner ikke er noe problem.

```
int fak (int n)
{
   if (n <= 1) { return 1; }
   return n * fak(n - 1);
}
```

Biblioteksfunksjonene

Hvordan tar vi oss av standardfunksjonene exit, getchar etc?

- Vi må legge inn funksjonsnavnene i kompilatoren slik at
  - vi vet at de er deklarert og at
  - de brukes riktig (dvs har korrekt antall parametre og riktig type).
- Ellers kan vi benytte dem fritt i assemblerkoden vi lager; den eksekverbare koden til disse funksjonene hentes av gcc med -L/local/share/inf2100 -lcflat eller fra C-biblioteket. Vi trenger altså ikke skrive standardfunksjonene.

Testing

Alle programmer må testes.

NB!

De beste testprogrammene skriver du selv!

- Tenk på noe som kan gå galt.
- Skriv et testprogram som sjekker akkurat dette.

Det finnes også ferdiglagde testprogrammer:

~inf2100/oblig/test/ inneholder ni testprogrammer:

*.cflat	C♭-programmet
*.s	Generert assemblerkode
*.input	Data til testprogrammet (for noen)
*.res	Utskrift fra testprogrammet

~inf2100/oblig/feil/Del-*/ har 19 småprogrammer med feil, som

```
int main (int argc)  /* main skal ikke ha parametre! */
{
   putchar('!');   putchar(10);
}
```

Om dere likte kurset

har dere flere kurs innen samme felt:

INF2270 lærer dere om hvordan en datamaskiner er bygget opp og hvordan de programmeres i assemblerkode.

INF3110 introduserer dere for mange flere programmeringsspråk (og litt mer om kompilering)

INF5110 gir en grundig innføring i hvordan man skriver en ekte kompilator