Tema for siste forelesning:

Kodegenerering

- Funksjoner

Testing

- Ulike testprogrammer

Kompilering av programsystemer

- make
- ant

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();
}
```

```
        .text
        .globl  f
f:      enter   $0,$0                   # Start function f
.exit$f:
        leave
        ret                             # End function f
---------------------------------------------------------
        call    f                       # Call f
```

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 funksjonen og sette av plass til lokale variabler (første parameter til enter):

```
        .text
        .globl  f
f:      enter   $0,$0                   # Start function f
```

Avslutning av funksjonen

Ved retur må vi rydde opp i funksjonen før tilbakehoppet med en ret-instruksjon:

```
.exit$f:
        leave
        ret                                     # End function f
```

Alle funksjoner får en navnelapp pga return-setningene.

Lokale variabler

Lokale variabler ligger 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:     enter   $12,$0                  # Start function f1
        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:
        leave
        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. $o = o + v.\text{dataSize}()$
   2. v får «navnet» $\boxed{-o(\%\text{ebp})}$.
3. Instruksjonen enter setter av plass til alle de lokale variablene.
4. Plassen frigjøres med instruksjonen leave.

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:     enter   $96,$0                  # Start function f2
        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:
        leave
        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

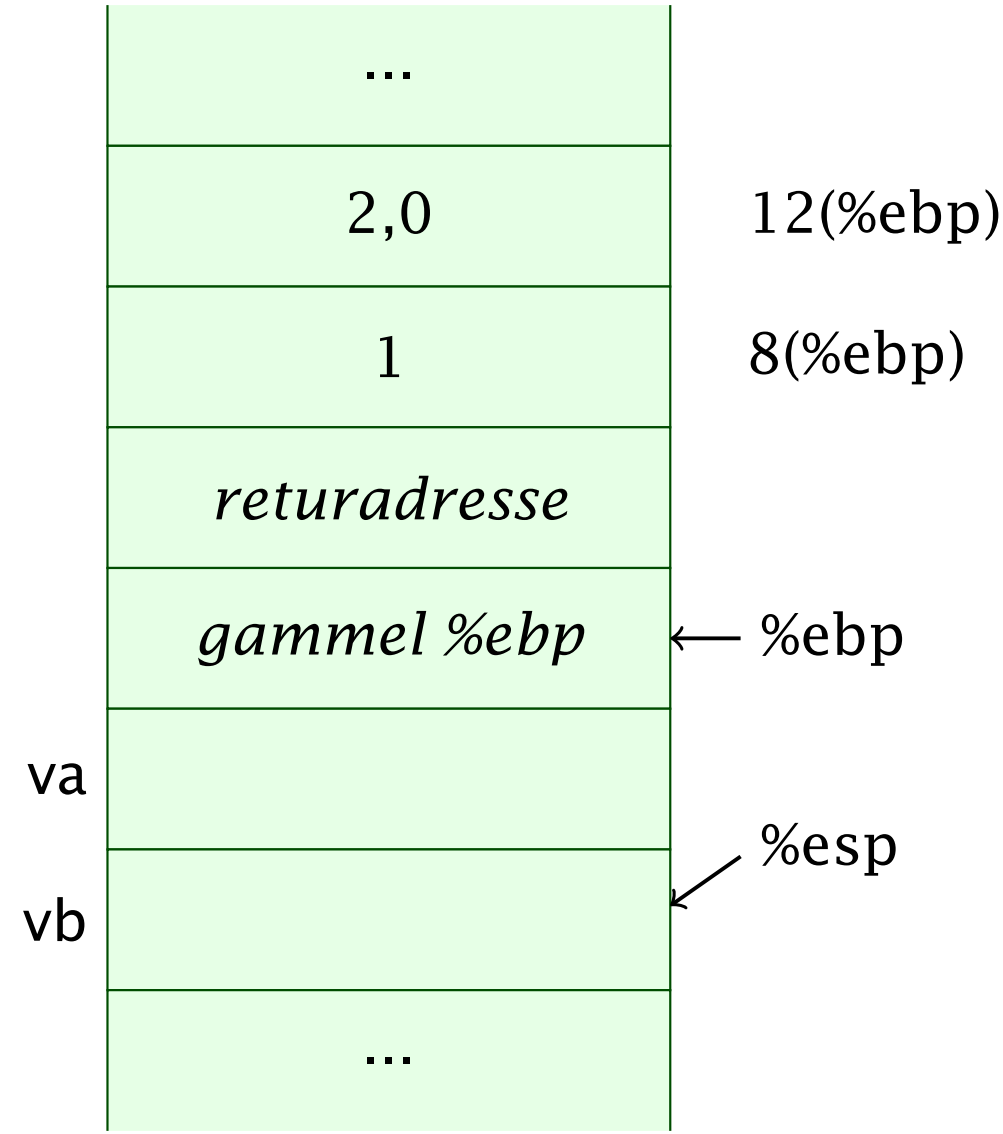
   int: %EAX
   double: på flyttallsstakken

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:     enter   $0,$0                   # Start function f2
.exit$f2:
        leave
        ret                             # End function f2
        .globl  main
main:   enter   $0,$0                   # Start function main
        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:
        leave
        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) {
  :
```

Hva om vi glemmer return?

Det er lov å droppe return-setningen. Da kan funksjonen returnere hvilken verdi den vil.

Men . . .

Flyttallsstakken kommer i ulage om en double-funksjon ikke returnerer noen verdi.

Løsning

Vi avslutter alle double-funksjoner med en fldz som legger 0,0 på flyttallsstakken.

- Hvis funksjonen utfører en return, hopper vi over fldz.
- Hvis ikke, vil fldz legge 0,0 på stakken.

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:           enter   $4,$0                   # Start function pot2
                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:
        leave
        ret                             # End function pot2
```

Et eksempel

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

```
        .globl  main
main:   enter   $0,$0                   # Start function main
        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:
        leave
        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:

http://inf2100.at.ifi.uio.no/oblig/test/ inneholder ni testprogrammer:

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

http://inf2100.at.ifi.uio.no/oblig/feil/Del-*/ har 19 småprogrammer med feil, som

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

Programmet make

Det er mange praktiske problemer forbundet med programmering av et større program:

- Hvordan holde orden på et stort program når det er mange separatkompilerte filer?
- Hvis én fil endres, hvilke filer må da kompileres på nytt?
- Hvordan sikre at alle de riktige opsjonene er med ved hver kompilering?
- Hvordan unngå at brukeren skriver seg i hjel på lange kommandolinjer?

Den vanligste Unix-løsningen er *make!*

Brukeren lager en fil med navn makefile eller Makefile. Den ser slik ut:

F_0: F_1 F_2 ... F_n
⟨Tab⟩*Kommandolinje*
⟨Tab⟩*Kommandolinje*
⋮

Dette betyr:

1. Filen F_0 er «avhengig» av filene $F_1, F_2, \ldots, F_n$. Det betyr at hvis F_0 er *eldre* enn noen av disse, må F_0 regenereres.
2. Kommandolinjene må da utføres for å generere F_0. Disse linjene *må* starte med en ⟨Tab⟩.

Programmet make gjør altså følgende:

1. Først leses filen makefile eller Makefile.
2. Så bygger make opp en oversikt (nærmere bestemt en graf) over hvilke filer som avhenger av hvilke.

   Utgangspunktet (roten) for denne grafen oppgis som parameter til make; ellers tas første fil.

   Denne grafen angir hvilken rekkefølge filene skal sjekkes (og eventuelt regenereres) i. For Java-programmer spiller ikke dette noen rolle, men av og til er dette vesentlig.

Nå kan hver fil sjekkes i riktig rekkefølge. Hvis

1. filen ikke finnes, eller
2. noen av filene den er avhengig av ikke finnes, eller
3. filen er eldre enn noen av de den er avhengig av,

blir den regenerert. Brukeren får beskjed om hvilke kommandoer som utføres.

Et eksempel

Denne bruker jeg for å kompilere dagens forelesning:

```
demo.s:         demo.cflat
        cflat -logB demo.cflat
f2.s:           f2.cflat
        cflat f2.cflat
f4.s:           f4.cflat
        cflat f4.cflat

clean:
        rm -f *.log *.s
```

make-operasjoner

Det fine med make er at man også kan legge inn *handlinger* (som «make clean» på forrige ark).

```
$ make clean
rm -f *.log *.s
```

Oppsummering om make

- make er meget nyttig til å holde orden på programsystemer med flere filer.
- Makefile-er er meget enkle å skrive.
- Det dokumenterer hva som må gjøres.
- Et Unix-system har tusener av Makefile-er.

Men . . .

- make er veldig Unix-orientert.

  Tilbudet av kommandoer, kommandonavnene, parameternotasjon og mye annet er forskjellig fra Unix til andre operativsystemer.

Så . . .

Programmet ant ble laget for

- støtte kompilering av Java-programmer
- kunne fungere uendret under ulike operativsystemer
- benytte et mer standisert spesifikasjonsspråk (i XML)

I kurset kompilerer vi med denne **build.xml**:

```
<?xml version="1.0"?>
<project name="Cflat" default="jar">

  <target name="clean" description="Fjern genererte filer">
    <delete dir="classes"/>
    <delete file="Cflat.jar"/>
  </target>

  <target name="compile" description="Kompiler Java-programmet">
    <mkdir dir="classes"/>
    <javac srcdir="." destdir="classes" includeantruntime="false"/>
  </target>

  <target name="jar" depends="compile" description="Lag en JAR-fil">
    <jar destfile="Cflat.jar">
      <fileset dir="classes" includes="**/*.class"/>
      <manifest>
        <attribute name="Main-Class" value="no.uio.ifi.cflat.cflat.Cflat"/>
      </manifest>
    </jar>
  </target>
</project>
```

Om dere likte kurset

har dere flere kurs innen samme felt ved Ifi:

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