

Dagens tema:

Kodegenerering

- Introduksjon
- Enkle variable
- Uttrykk
- Tilordning
- Litt mer kompliserte setninger med betingelser

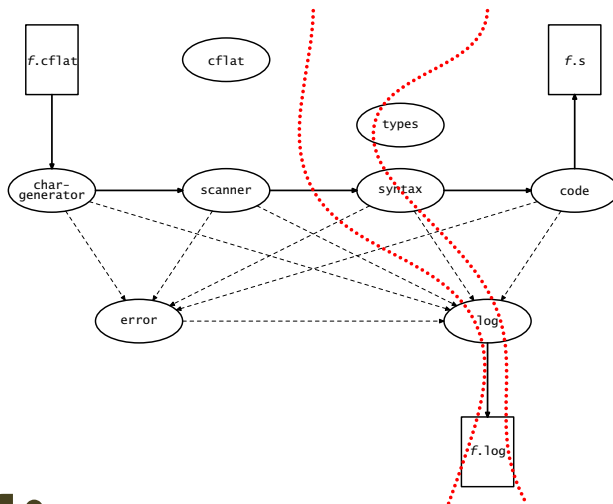
(Alt om kodegenerering unntatt funksjoner.)

Prosjektoversikt

Del-0

Del-1

Del-2



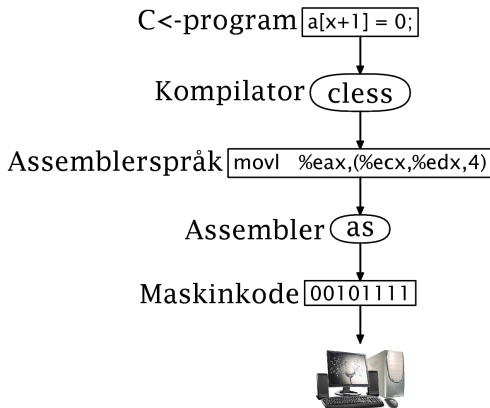
Formålet

Vårt oppdrag er å lage en kompilator:

Inndata er

trerepresentasjonen av C-programmet laget i del 1.

Utdata er en fil med x86 assemblerkode.

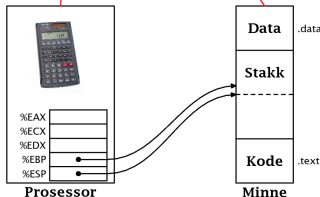
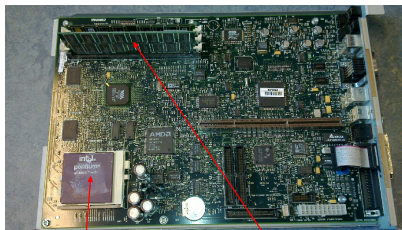


Hvordan ser datamaskinen vår ut?

Datamaskinen vår

Minnet inneholder tre former for data:

- **Data** inneholder globale variabler.
- **Stakken** inneholder parametre og lokale variabler.
- **Koden** er programmet (som tall).



Hvordan ser datamaskinen vår ut?

movl	⟨v ₁ ⟩,⟨v ₂ ⟩	Flytt ⟨v ₁ ⟩ til ⟨v ₂ ⟩.
cdq		Omform 32-bits %EAX til 64-bits %EDX:%EAX.
leal	⟨v ₁ ⟩,⟨v ₂ ⟩	Flytt <i>adressen</i> til ⟨v ₁ ⟩ til ⟨v ₂ ⟩.
pushl	⟨v⟩	Legg ⟨v⟩ på stakken.
popl	⟨v⟩	Fjern toppen av stakken og legg verdien i ⟨v⟩.
addl	⟨v ₁ ⟩,⟨v ₂ ⟩	Addér ⟨v ₁ ⟩ til ⟨v ₂ ⟩.
subl	⟨v ₁ ⟩,⟨v ₂ ⟩	Subtraher ⟨v ₁ ⟩ fra ⟨v ₂ ⟩.
imull	⟨v ₁ ⟩,⟨v ₂ ⟩	Multipliser ⟨v ₁ ⟩ med ⟨v ₂ ⟩.
idivl	⟨v⟩	Del %EDX:%EAX med ⟨v⟩; svar i %EAX.
call	⟨lab⟩	Kall funksjonen i ⟨lab⟩.
ret		Returner fra funksjonen.
cmpl	⟨v ₁ ⟩,⟨v ₂ ⟩	Sammenligning ⟨v ₁ ⟩ og ⟨v ₂ ⟩.
jmp	⟨lab⟩	Hopp til ⟨lab⟩.
sete	⟨v⟩	Sett ⟨v⟩=1 om =, ellers ⟨v⟩=0.
setne	⟨v⟩	Sett ⟨v⟩=1 om ≠, ellers ⟨v⟩=0.
setl	⟨v⟩	Sett ⟨v⟩=1 om <, ellers ⟨v⟩=0.
setle	⟨v⟩	Sett ⟨v⟩=1 om ≤, ellers ⟨v⟩=0.
setg	⟨v⟩	Sett ⟨v⟩=1 om >, ellers ⟨v⟩=0.
setge	⟨v⟩	Sett ⟨v⟩=1 om ≥, ellers ⟨v⟩=0.



Hvordan ser datamaskinen vår ut?

faddp	Adder to flyttall.
fdivp	Divider ett flyttall på et annet.
fildl <v>	Legg heltall i <v> på stakken som flyttall.
fistpl <v>	Lagre flyttall som int i <v>.
fldl <fv>	Legg flyttall på stakken.
fldz	Legg 0,0 på stakken.
fmlp	Multipliser to flyttall.
fstpl <fv>	Lagre flyttall som double i <fv>.
fstps <fv>	Lagre flyttall som float i <fv>.
fsubp	Subtraher ett flyttall fra et annet.

Anta at vi har C_b-koden

$v = 1 + 2;$

Disse x86-instruksjonene gjør dette:

```

movl    $1,%eax    # %EAX=1 %ECX=? stack=...
pushl   %eax       # %EAX=1 %ECX=? stack=... 1
movl    $2,%eax    # %EAX=2 %ECX=? stack=... 1
movl    %eax,%ecx  # %EAX=2 %ECX=2 stack=... 1
popl    %eax       # %EAX=1 %ECX=2 stack=...
addl    %ecx,%eax  # %EAX=3 %ECX=2 stack=...
movl    %eax,v     # v = 3
    
```

Husk!

Det finnes mange mulige kodebiter som gjør det samme. I kompendiet står angitt nøyaktig hvilke som skal brukes.

Hvordan implementere kodegenerering

Det beste er å følge samme opplegg som for å sjekke programkoden:

Kodegenerering

Legg en metode `genCode` inn i alle klasser som representerer en del av C_b-programmet (dvs er subklasse av `SyntaxUnit`).

Metoden 'genCode'

```
class WhileStatm extends Statement {
    Expression test = new Expression();
    StatmList body = new StatmList();

    @Override void check(DeclList curDecls) {
        :
    }

    @Override void genCode(FuncDecl curFunc) {
        :
    }

    @Override void parse() {
        :
    }

    @Override void printTree() {
        :
    }
}
```



Konvensjoner

Kodegenerering blir mye enklere om vi setter opp noen fornuftige konvensjoner:

- Alle int-beregninger skal ende opp i %EAX.
- Alle double-beregninger skal legges på flyttallsstakken.
- %ECX og %EDX er hjelperegistre.
- Hovedstakken (aksessert via %ESP) er til
 - mellomresultater av int-verdier
 - funksjonskall (neste uke)
- Flyttallsstakken er til
 - mellomresultater av double-verdier

Hva slags variabler har vi?

Variabler

Det finnes fem sorter variabler i C^b:

	Enkle	Array-er	
Globale	✓	✓	
Lokale	✓	✓	(De blå tar vi neste uke.)
Parametre	✓		

De kan være int eller double.

De kan forekomme i tre ulike situasjoner:

Deklarasjon	int v;
Tilordning	v = 1;
Bruk	if (v>0) ...

Generering av globale variable

```
abstract class Declaration extends SyntaxUnit {
    String name, assemblerName;
    Type type;
    boolean visible = false;
    Declaration nextDecl = null;
    :
}
```

```
abstract class VarDecl extends Declaration {
    :
}
```

```
class GlobalSimpleVarDecl extends VarDecl {
    GlobalSimpleVarDecl(String n) {
        super(n);
        assemblerName = (Cflat.underscoredGlobals() ? "_" : "") + n;
    }

    @Override void genCode(FuncDecl curFunc) {
        Code.genVar(assemblerName, true, declSize(),
            type.typeName()+" "+name+";");
    }

    :
}
```



Metoden Code.genVar vil sette av plass til globale variabler:

```
public static void genVar(String name, boolean global,
                          int nBytes, String comment) {
    if (! generatingData) {
        codeFile.println("          .data");
        generatingData = true;
    }

    if (global)
        codeFile.println("          .globl " + name);

    printLabel(name, false);
    codeFile.printf(".fill    %-24d", nBytes);

    if (comment.length() > 0) {
        codeFile.print("# " + comment);
    }
    codeFile.println();
}
```



Setninger

I kompendiet finner vi kodeskjemaer for alle setningene.

Assignment

$\langle iv \rangle = \langle ie \rangle$	⇒	<pre> ⟨Beregn ⟨ie⟩ med svar i %EAX⟩ movl %eax,⟨iv⟩ </pre>
$\langle ia \rangle[\langle ie_1 \rangle] = \langle ie_2 \rangle;$	⇒	<pre> ⟨Beregn ⟨ie₁⟩ med svar i %EAX⟩ pushl %eax ⟨Beregn ⟨ie₂⟩ med svar i %EAX⟩ leal ⟨ia⟩,%edx popl %ecx movl %eax,(%edx,%ecx,4) </pre>

Assignment av flyttall

$\langle fv \rangle = \langle fe \rangle$	\Rightarrow	<pre> (Beregn $\langle fe \rangle$ med svar på f-stakken) fstpl $\langle fv \rangle$ </pre>
---	---------------	--

Noen ganger må vi konvertere fra int til float:

$\langle fv \rangle = \langle ie \rangle;$	\Rightarrow	<pre> (Beregn $\langle ie \rangle$ med svar i %EAX) movl %eax, .tmp fildl .tmp fstpl $\langle fv \rangle$ </pre>
--	---------------	--

(Legg merke til hjelpevariabelen `.tmp` – den må alltid være der.)

Et forslag til implementering:

```

class AssignStatm extends Statement {
    Assignment assignment = new Assignment();

    @Override void genCode(FuncDecl curFunc) {
        assignment.genCode(curFunc);
    }

    :
}

class Assignment extends SyntaxUnit {
    Variable lhs = new Variable();           /* "Left hand side" */
    Expression rhs = new Expression();      /* "Right hand side" */

    @Override void genCode(FuncDecl curFunc) {
        if ((det dreier seg om et array-element)) {
            // To be written...
        } else {
            rhs.genCode(curFunc);
            lhs.declRef.genStore(rhs.valType);
        }
    }

    :
}

```



Hint

Alle `VarDecl` bør ha en `genStore`- og `genStoreArray`-metode for å lage kode for tilordning til seg selv; de *kan* se slik ut:

```
abstract class VarDecl extends Declaration {
  VarDecl(String n) {
    super(n);
  }

  void genStore(Type valType) {
    if (type==Types.doubleType && valType==Types.doubleType) {
      Code.genInstr("", "fstpl", assemblerName, name+" =");
    } else if (type==Types.doubleType && valType==Types.intType) {
      Code.genInstr("", "movl", "%eax,"+Code.tmpLabel, "");
      Code.genInstr("", "fildl", Code.tmpLabel, " (double)");
      Code.genInstr("", "fstpl", assemblerName, name+" =");
    } else if (type==Types.intType && valType==Types.doubleType) {
      Code.genInstr("", "fistpl", assemblerName, name+" = (int)");
    } else if (type==Types.intType && valType==Types.intType) {
      Code.genInstr("", "movl", "%eax,"+assemblerName, name+" =");
    } else {
      Error.panic("Declaration.genStore");
    }
  }
}
```

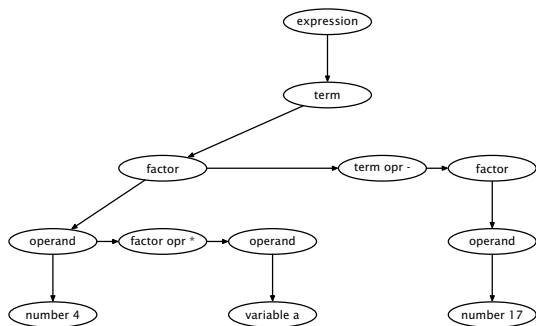


Hvordan lagres uttrykk?

Uttrykk

Hvordan lager vi kode for et uttrykk som

$$4 * a - 17?$$



Operander

Det er ganske enkelt å lage kode som legger en global enkel int-variabel i %EAX-registeret eller en double-variabel på flyttallsstakken:

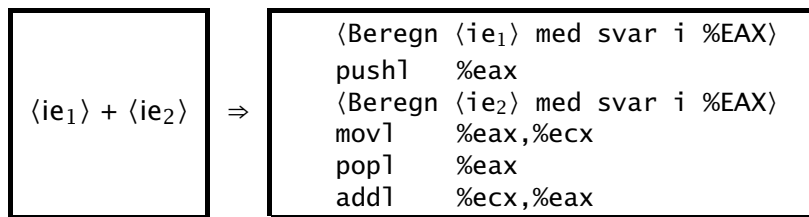
<code><iv></code>	⇒	<code>movl <iv>,%eax</code>
<code><fv></code>	⇒	<code>fldl <fv></code>

Det er like enkelt for int-konstanter (tall eller tegn).

<code><n></code>	⇒	<code>movl \$<n>,%eax</code>
------------------------	---	------------------------------------

(\$-tegnet angir at det er snakk om en tallkonstant.)

Aritmetisk int-operatorer



- Ved at alle int-operander er innom %EAX blir det enklere å skrive kodegenereringen.
- Divisjon og sammenligninger er litt vanskeligere; se tabellene i kompendiet.
- Hvorfor må $\langle ie_1 \rangle$ legges på stakken?
- double-operatorene fordrer helt annen kode.

While-setningen med int-test

```
while (<ie>) {  
    <S>  
}
```

⇒

```
.Ln1: <Beregn <ie> med svar i %EAX>  
        cmp1    $0,%eax  
        je      .Ln2  
        <S>  
        jmp     .Ln1  
  
.Ln2:
```

While-setningen med double-test

```
while (<fe>) {
    <S>
}
```

⇒

```
.Ln1: <Beregn <fe> med svar på f-stak
      fstps    .tmp
      cml     $0,.tmp
      je      .Ln2
      <S>
      jmp     .Ln1

.Ln2:
```

Siden de to variantene er like med unntak av kode for testen, er det nyttig med

```
abstract public class Type {
    abstract public int size();
    abstract public String typeName();

    public void genJumpIfZero(String jumpLabel) {}
}
```

```
public class Types {
    public static BasicType doubleType, intType;

    public static void init() {
        doubleType = new BasicType() {
            @Override public void genJumpIfZero(String jumpLabel) {
                Code.genInstr("", "fstps", Code.tmpLabel, "");
                Code.genInstr("", "cml", "$0,"+Code.tmpLabel, "");
                Code.genInstr("", "je", jumpLabel, "");
            }
        };
    }
};
```



Lokale navnelapper

Når vi skal lage slike hopp, trenger vi stadig nye navnelapper. Dette kan vi få fra Code-modulen:

```
private static int numLabels = 0;

public static String getLocalLabel() {
    return String.format(".L%04d", ++numLabels);
}
```


Hele koden for WhileStatm

```
class WhileStatm extends Statement {
    Expression test = new Expression();
    StatmList body = new StatmList();

    @Override void genCode(FuncDecl curFunc) {
        String testLabel = Code.getLocalLabel(),
            endLabel = Code.getLocalLabel();

        Code.genInstr(testLabel, "", "", "Start while-statement");
        test.genCode(curFunc);
        test.valType.genJumpIfZero(endLabel);
        body.genCode(curFunc);
        Code.genInstr("", "jmp", testLabel, "");
        Code.genInstr(endLabel, "", "", "End while-statement");
    }
}
```

Et eksempel

```

while (a < 10) {
    a = a + 1;
}

.L0001:                                # Start while-statement
    movl    a,%eax                      # a
    pushl   %eax
    movl    $10,%eax                    # 10
    popl    %ecx
    cmpl   %eax,%ecx
    movl    $0,%eax
    setl   %al                          # Test <
    cmpl   $0,%eax
    je     .L0002
    movl    a,%eax                      # a
    pushl   %eax
    movl    $1,%eax                    # 1
    movl    %eax,%ecx
    popl    %eax
    addl   %ecx,%eax                    # Compute +
    movl    %eax,a                      # a =
    jmp    .L0001
.L0002:                                # End while-statement

```