Dagens tema

- Hva er kompilering?
- Hvordan foreta syntaksanalyse av et program?
- Hvordan programmere dette i Java?
- Hvordan oppdage feil?

Hva er kompilering?

Anta at vi lager dette lille programmet doble.cflat (kalt *kildekoden*):

```
int n;

int main ()
{
   putchar('?');   n = getint()*2;
   putint(n);  putchar(10);
}
```

Dette programmet kan ikke kjøres direkte på noen datamaskin, men det finnes en x86-kode (kalt **maskinkoden**) som gjør det samme:

```
0000000 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
0000020 02 00 03 00 01 00 00 00 30 83 04 08 34 00 00 00
0000040 28 09 00 00 00 00 00 00 34 00 20 00 08 00 28 00
0000060 1d 00 1a 00 06 00 00 00 34 00 00 00 34 80 04 08
0000100 34 80 04 08 00 01 00 00 00 01 00 00 05 00 00 00
0000120 04 00 00 00 03 00 00 00 34 01 00 00 34 81 04 08
0000140 34 81 04 08 13 00 00 00 13 00 00 00 04 00 00 00
0000160 01 00 00 00 01 00 00 00 00 00 00 00 00 80 04 08
0000200 00 80 04 08 1c 06 00 00 1c 06 00 00 05 00 00 00
0000220 00 10 00 00 01 00 00 00 1c 06 00 00 1c 96 04 08
0000240 1c 96 04 08 0c 01 00 00 14 01 00 00 06 00 00 00
0000260 00 10 00 00 02 00 00 00 30 06 00 00 30 96 04 08
```

⋮

Det er ikke lett å lese slik kode – det går bedre i **assemblerkode** som kan oversettes til **maskinkode** av en **assembler**:

```
        .data
.tmp:   .fill   4                       # Temporary storage
        .globl  n
n:      .fill   4                       # int n;
        .text
        .globl  main
main:   pushl   %ebp                    # Start function main
        movl    %esp,%ebp
        movl    $63,%eax                # 63
        pushl   %eax                    # Push parameter #1
        call    putchar                 # Call putchar
        addl    $4,%esp                 # Remove parameters
        call    getint                  # Call getint
        pushl   %eax
        movl    $2,%eax                 # 2
        movl    %eax,%ecx
        popl    %eax
        imull   %ecx,%eax               # Compute *
        movl    %eax,n                  # n =
```

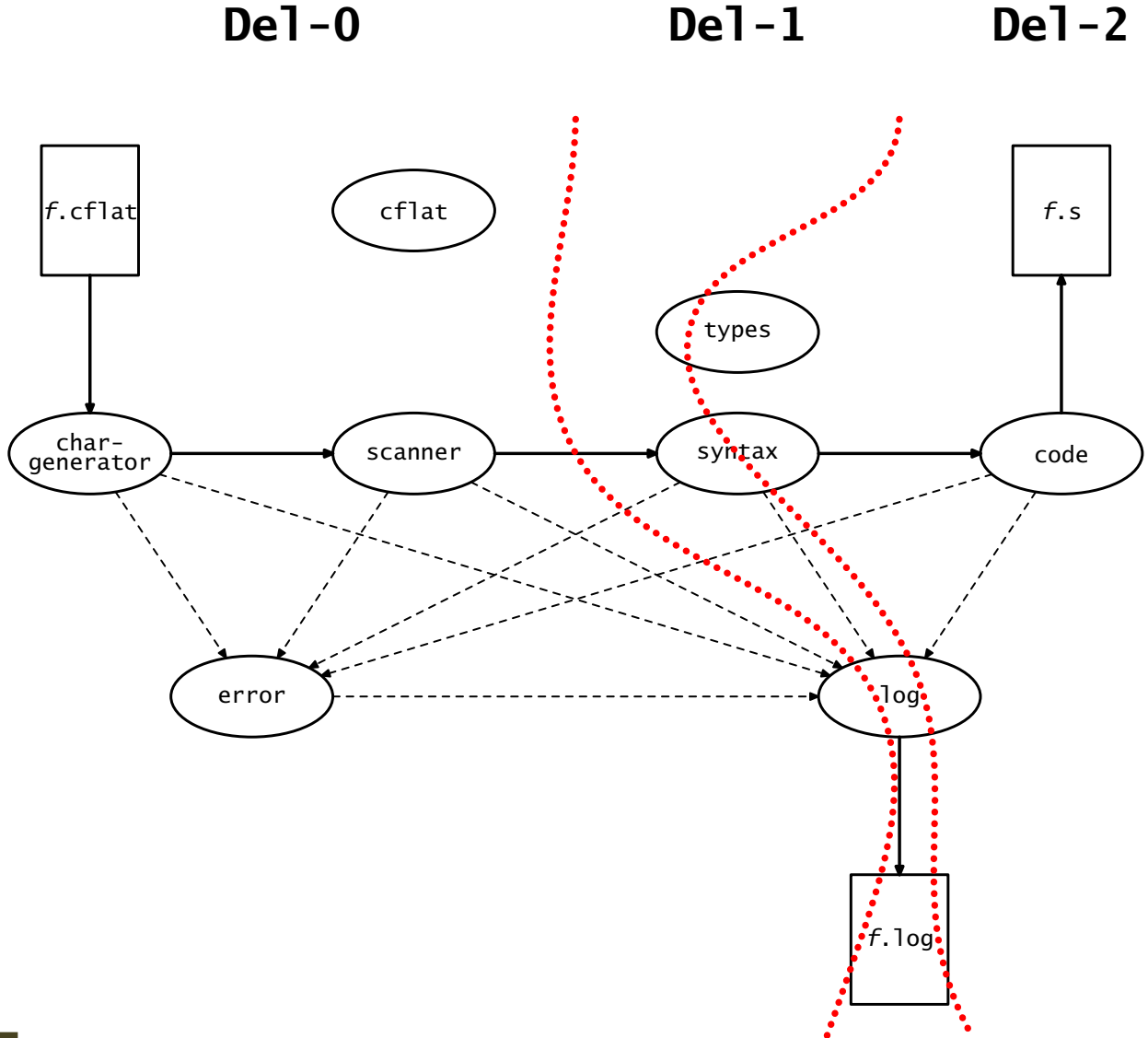
Maskinkode

```
        movl    n,%eax                  # n
        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
```

Kompilatoren

En kompilator leser C♭-koden og lager x86-assemblerkoden.

En slik kompilator skal dere lage.


Del-0
Del-1
Del-2
f.cflat
cflat
f.s
types
char-generator
scanner
syntax
code
error
log
f.log

Programtreet

De færreste programmeringsspråk kan oversettes linje for linje, men det ville vært mulig med C♭.

Det enkleste er likevel å lagre programmet på intern form først.

Det naturlige da er å lage et tre ved å bruke klasser, objekter og pekere. Her er OO-programmering ypperlig egnet.

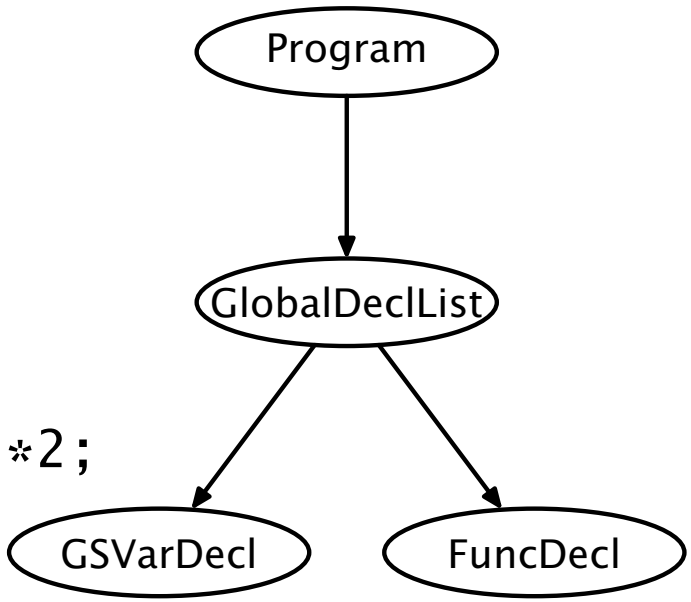
Et Cb-program

Et program består av en samling deklarasjoner og setninger i vilkårlig blanding:

program

Programmet doble.cflat representeres da av

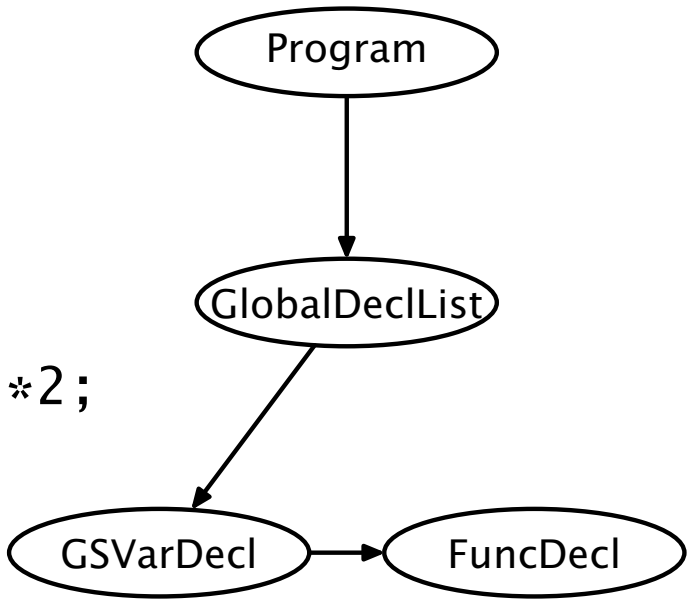
```
int n;

int main ()
{
   putchar('?');   n = getint()*2;
   putint(n);  putchar(10);
}
```



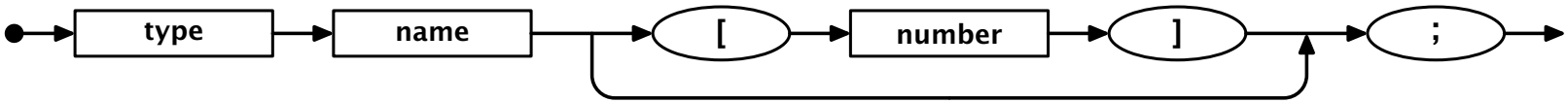
(GSVarDecl = GlobalSimpleVarDecl)

Siden vi skal representere treet som lister, ser det slik ut:

```
int n;

int main ()
{
   putchar('?');   n = getint()*2;
   putint(n);  putchar(10);
}
```



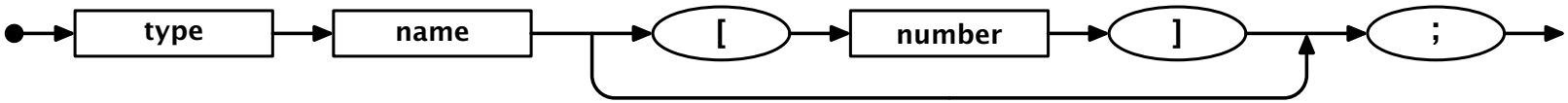
var decl



En GlobalSimpleVarDecl må inneholde data om

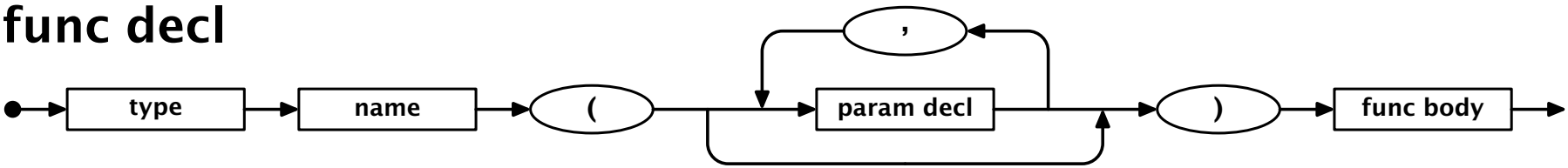
- variabelens type,
- dens navn og
- antall elementer (om den er en array).

var decl



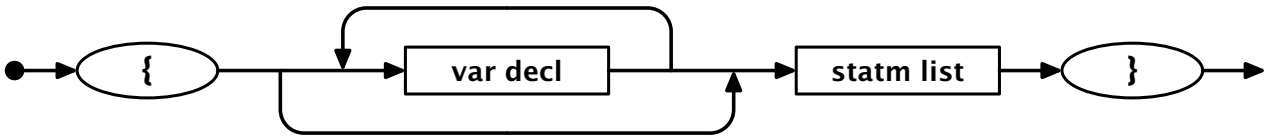

```
abstract class Declaration ... {
    String name;
    Type type;
      :

    Declaration(String n) {
        name = n;
    }
-----------------------------------------------
abstract class VarDecl extends Declaration {
    VarDecl(String n) {
        super(n);
    }
-----------------------------------------------
class GlobalSimpleVarDecl extends VarDecl {
    GlobalSimpleVarDecl(String n) {
        super(n);
    }
```

func decl



En FuncDecl må inneholde opplysninger om funksjonstypen, navnet, parametrene og innmaten.

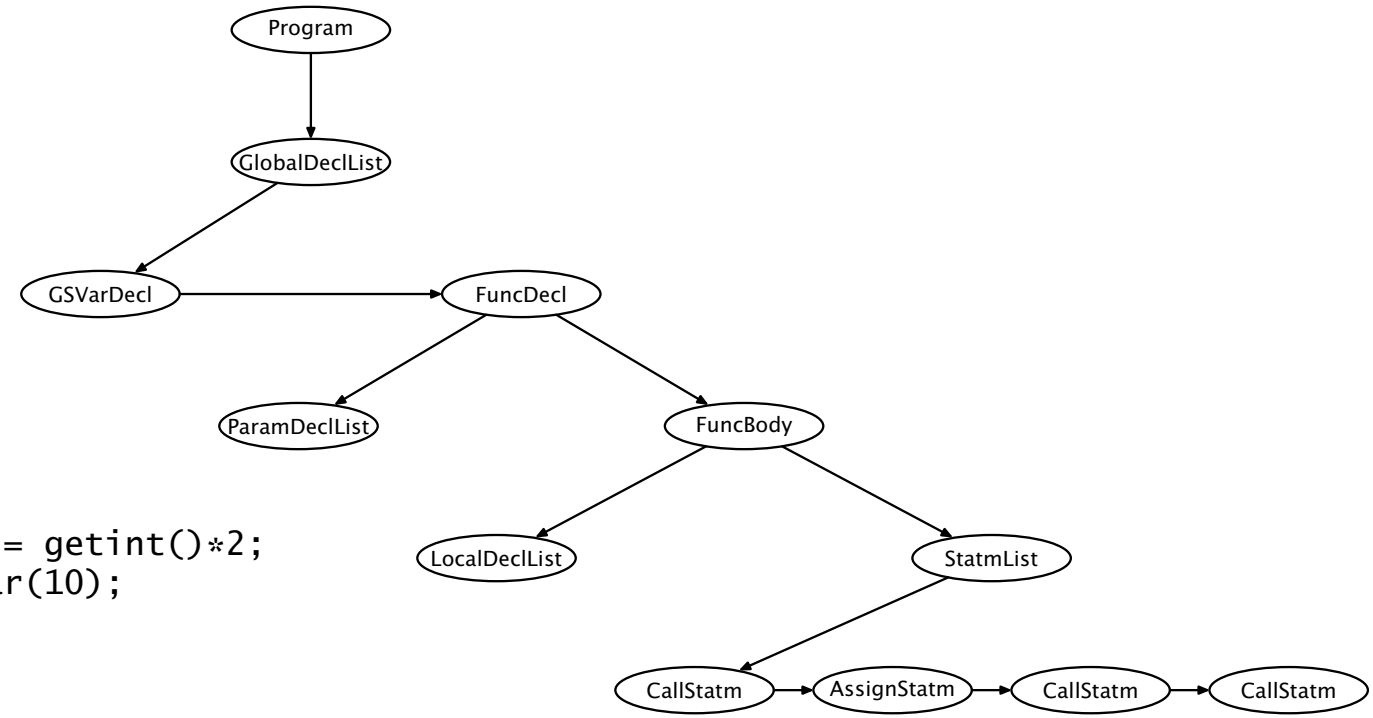
func body



En FuncBody inneholder lokale variabeldeklarasjoner og setninger.

Eksemplet vårt

```
int n;

int main ()
{
   putchar('?');   n = getint()*2;
   putint(n);  putchar(10);
}
```



Syntaksanalyse

På skolen hadde vi grammatikkanalyse hvor vi fant subjekt, predikat, indirekte og direkte objekt:

Faren ga datteren en ball.

(Det er ikke alltid like enkelt:

Fanger krabber så lenge de orker.)

Syntaksanalyse er på samme måte å finne hvilke språkelementer vi har og bygge **syntakstreet**.

Heldigvis: Analyse av programmeringsspråk er enklere enn naturlige språk:

- Programmeringsspråk har en klar og entydig definisjon i jernbanediagrammer eller tilsvarende.
- Programmeringsspråk er laget for å kunne analyseres rimelig enkelt.

C♭-kompilatoren har i hvert fall disse klassene[1] der alle subklassene til SyntaxUnit representerer et **metasymbol** (et jernbanediagram):

```
Cflat
CharGenerator
Code
Error
Log
Scanner
Syntax
[SyntaxUnit]
  [DeclList]
    GlobalDeclList
    LocalDeclList
    ParamDeclList
  [Declaration]
    FuncDecl
    [VarDecl]
      GlobalArrayDecl
      GlobalSimpleVarDecl
      LocalArrayDecl
      LocalSimpleVarDecl
      ParamDecl
    ExprList
    [Operand]
      Expression
      FunctionCall
      Number
      Variable
    [Operator]
      RelOperator
    Program
    [Statement]
      EmptyStatm
      IfStatm
      WhileStatm
    StatmList
    Term
  Token
  [Type]
    ArrayType
    [BasicType]
  Types
```

[1] Klasser i parentes er abstrakte.

Grammatikk

Grammatikken (i form av jernbanediagrammene) er et ypperlig utgangspunkt for å analysere et program og bygge opp syntakstreet:

while-statm

while-statm

Utifra dette vet vi:

1. Først kommer symbolet while.
2. Så kommer en (.
3. Så kommer en *expression*.
4. Etter den kommer en).
5. Deretter kommer en {.
6. I klammene kommer *statm-list*.
7. Helt til sist kommer en }.

Programmering i Java

Utifra jernbanediagrammet kan vi lage en skisse for en metode som analyserer en while-setning i et C♭-program:

```
class WhileStatm extends Statement {
    ⋮

    @Override void parse() {
      ⟨Sjekk at vi har lest while⟩
      ⟨Sjekk at vi har lest (⟩
      ⟨Analyser Expression⟩
      ⟨Sjekk at vi har lest )⟩
      ⟨Sjekk at vi har lest {⟩
      ⟨Analyser StatmList⟩
      ⟨Sjekk at vi har lest }⟩
    }
}
```

Stort sett gjør vi to ting:

- Symboler (i rundinger) sjekkes.
- Meta-symboler (i firkanter) overlates til sine egne metoder for analysering.

. . . og dermed har problemet nærmest løst seg selv!

Er det så enkelt?

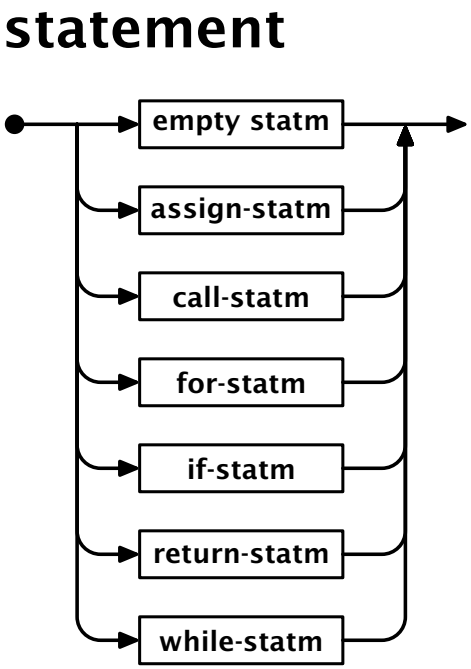
Mange programmeringsspråk (som C♭ og Pascal men ikke Java, C og C++) er designet slik at denne teknikken kalt **recursive descent** alltid fungerer.

Et analyseprogram for et LL(1)-språk er aldri i tvil om hvilken vei gjennom programmet som er den rette.

Ved analyse av LL(2)-språk må man av og til se ett symbol fremover.

Ved analyse av LL(3)-språk (som C♭) må man av og til se to symboler fremover.

Veien er alltid klar



Samarbeid med Scanner

Hvordan sikrer vi at symbolstrømmen fra Scanner er i fase med vår parsering?

Det beste er å vedta noen regler som alle parse-metodene *må* følge:

1. Når man kaller parse, skal første symbol være lest inn!
2. Når man returnerer fra en parse, skal første symbol *etter* konstruksjonen være lest!

Plassering av metodene

Husk at målet med analysen er tofoldig:

1. Vi skal sjekke at programmet er riktig.
2. Vi skal bygge opp syntakstreet.

Det er naturlig å koble analysemetoden til den klassen som skal inngå i syntakstreet.

- Hvert meta-symbol i diagrammet implementeres av en Java-klasse.[2]
- Hver av disse klassene får en virtuell metode

  ```
  @Override void parse() { ... }
  ```

  som kan analysere «seg selv».

Dette er OO-programmering.

[2] Dette er bare nesten riktig — mer om dette siden.

Feilsjekking

Sjekken på syntaksfeil er svært enkel:

Hvordan finne feil?

Hvis neste symbol ikke gir noen lovlig vei i diagrammet, er det en feil.

while-statm


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


    @Override void parse() {
        Log.enterParser("<while-statm>");

        Scanner.readNext();
        Scanner.skip(leftParToken);
        test.parse();
        Scanner.skip(rightParToken);
        Scanner.skip(leftCurlToken);
        body.parse();
        Scanner.skip(rightCurlToken);

        Log.leaveParser("</while-statm>");
    }
```

Husk

I Scanner-modulen har vi

```
public static void check(Token t) {
    if (curToken != t)
        Error.expected("A " + t);
}

public static void skip(Token t) {
    check(t);  readNext();
}
```

og i Error-modulen har vi

```
public static void expected(String exp) {
    error(Scanner.curLine,
          exp + " expected, but found a " + Scanner.curToken + "!");
}
```

Oppsummering

Vi har vært gjennom

- Hva kompilering er
- Hvordan foreta en syntaksanalyse av et program
- Hvordan programmere dette objektorientert
- Hvordan oppdage feil