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.rusc (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 Rask-kode (kalt *maskinkoden*) som gjør det samme:

```
#! /local/bin/rask

1600000000000006 211000000000000 1600000000009990 0
0 0 331000000000004 303000000000005
211000000000063 1600000000009993 1600000000009992 203010000000000
201000000000002 601030000000001 301000000000003 111000000000003
1600000000009994 211000000000010 1600000000009993 103000000000005
131000000000004 1700000000000000
```

Det er ikke lett å lese slik kode – det går bedre i assemblerkode:

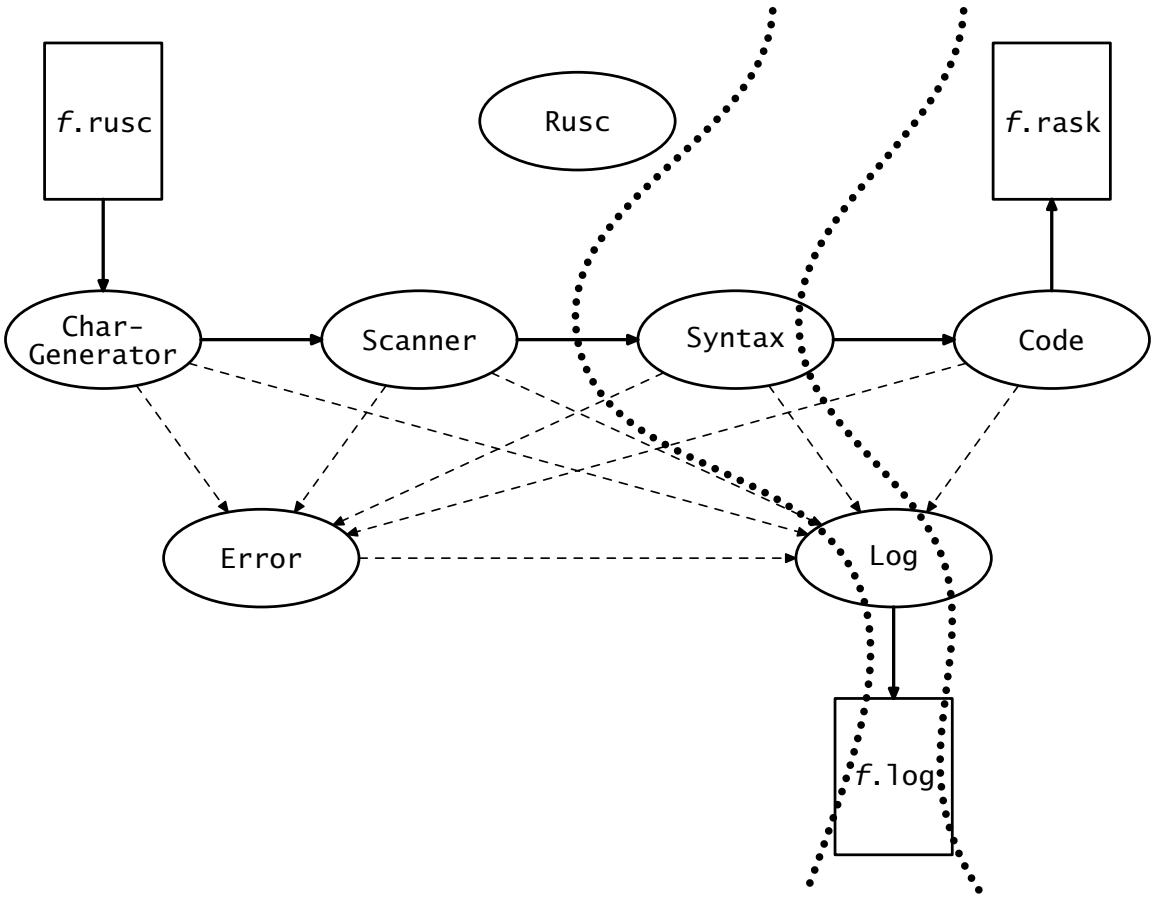
```
Code    0: 1600000000000000  CALL      0  0          0  # main();
Code    1:  211000000000000  SET      11  0          0  #     (0)
Code    2: 1600000000009990  CALL      0  0       9990  # exit
Code    3:                   RES                    1  # int n
Code    4:                   RES                    1  # Return address in main
Code    5:                   RES                    1  # Refuge for R3 in main
Code    6:  331000000000004  STORE    31  0          4  # Save return address
Code    7:  303000000000005  STORE     3  0          5  # Save R3
Code    8:  211000000000063  SET      11  0         63  # R11 = 63
Code    9: 1600000000009993  CALL      0  0       9993  # Call putchar(...)
Code   10: 1600000000009992  CALL      0  0       9992  # Call getint(...)
Code   11:  203010000000000  SET       3  1          0  # Save operand in R3
Code   12:  201000000000002  SET       1  0          2  # R1 = 2
Code   13:  601030000000001  MUL       1  3          1  #   *
Code   14:  301000000000003  STORE     1  0          3  # n =
Code   15:  111000000000003  LOAD     11  0          3  # R11 = n
Code   16: 1600000000009994  CALL      0  0       9994  # Call putint(...)
Code   17:  211000000000010  SET      11  0         10  # R11 = 10
Code   18: 1600000000009993  CALL      0  0       9993  # Call putchar(...)
Code   19:  103000000000005  LOAD      3  0          5  # Restore R3
Code   20:  131000000000004  LOAD     31  0          4  # Restore return address
Code   21: 1700000000000000  RET       0  0          0  # Return from main
--->    0: 1600000000000006                          6  # Fix call to main()
```

En kompilator

En kompilator leser RusC-koden og lager Rask-koden.

En slik kompilator skal dere lage.

Del-0 **Del-1** **Del-2**



Programtreet

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

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.

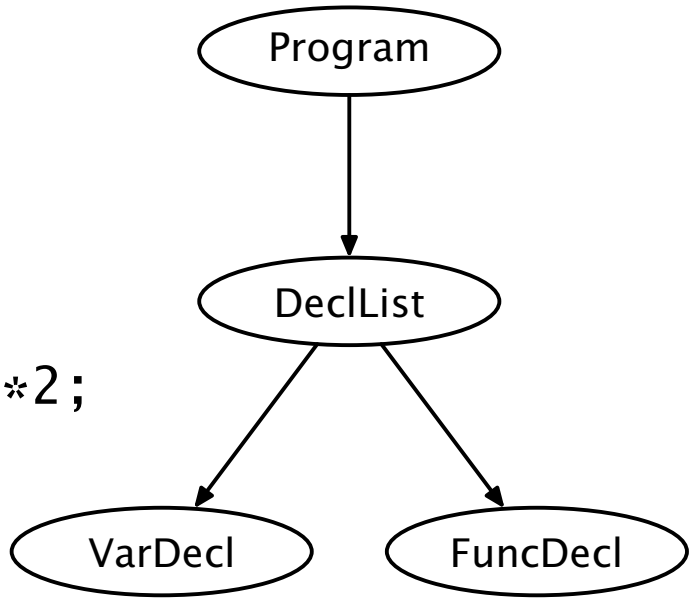
Et RusC-program

Et program består av en samling deklarasjoner og en samling setninger:

program

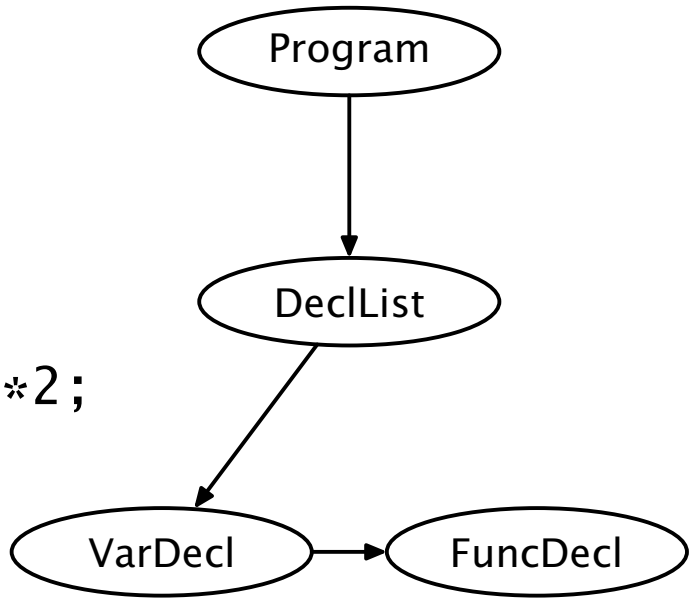
Programmet doble.rusc representeres da av

```
int n;

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

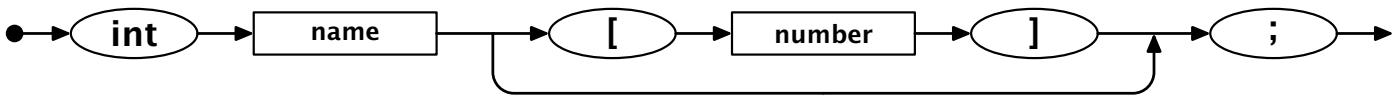


Siden vi skal representere treet som lister, ser det slik ut:

```
int n;

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

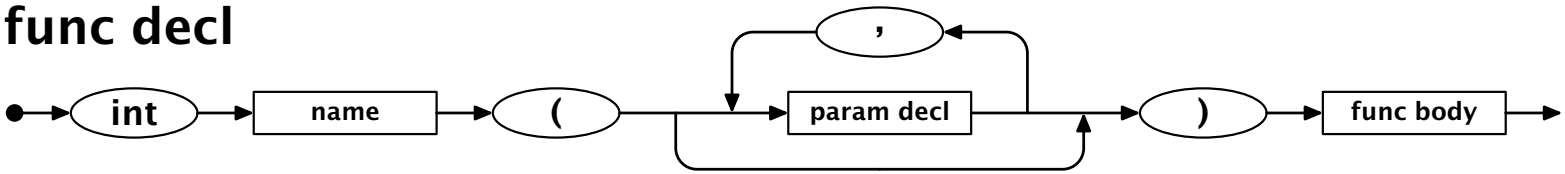


var decl



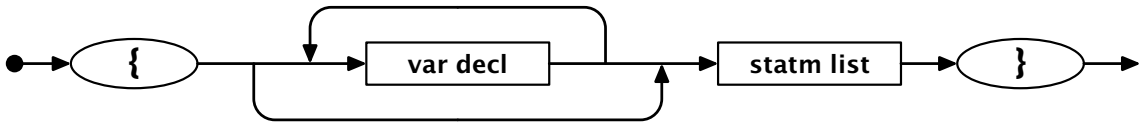
En VarDecl må inneholde data om variabelens navn og — om den er en array — hvor mange elementer den har.

func decl



En FuncDecl må inneholde opplysninger om parametrene og innmaten.

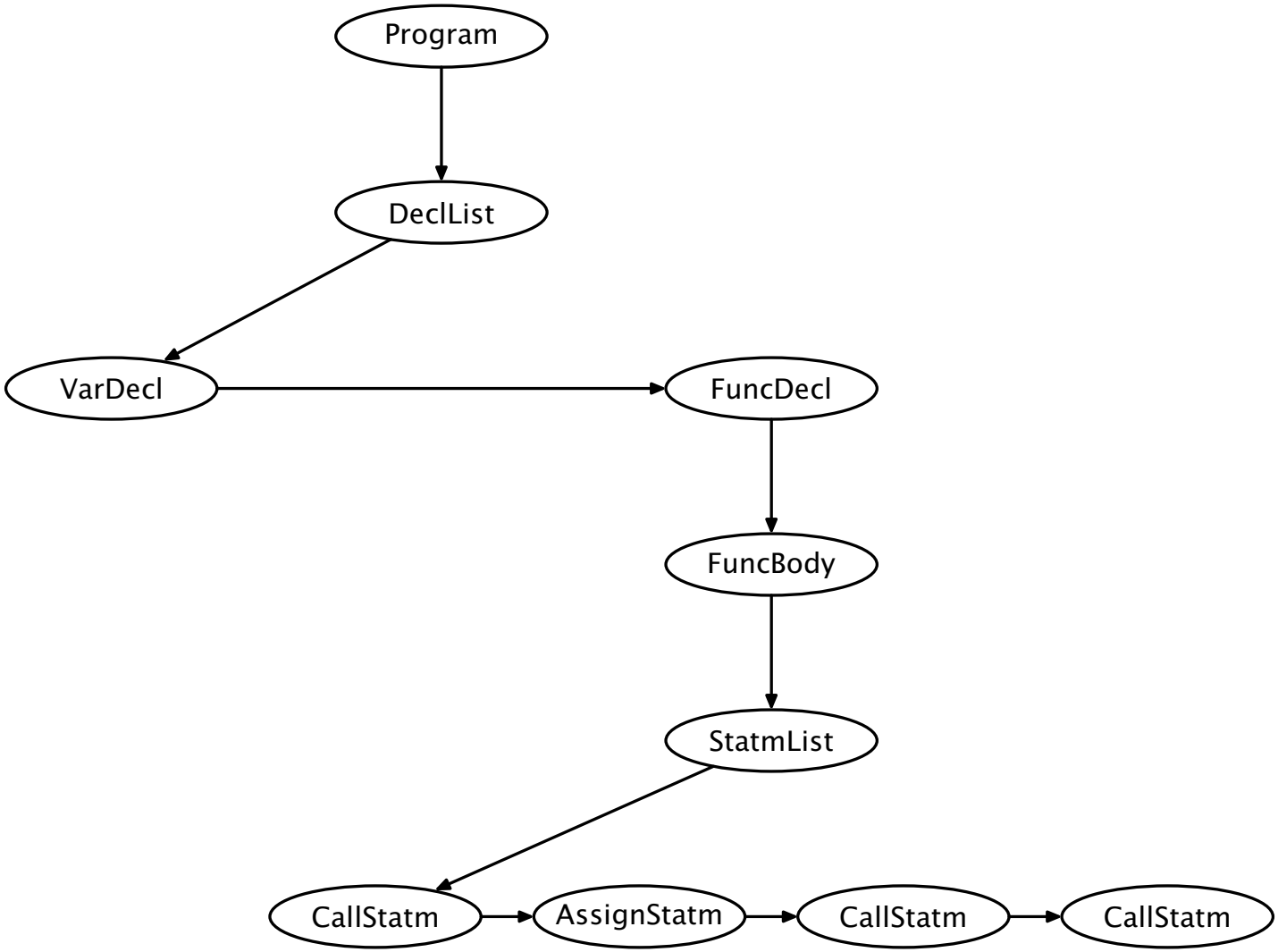
func body



En FuncBody inneholder lokale int-deklarasjoner og setninger.

Eksemplet vårt

```
int n;

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


Program
DeclList
VarDecl
FuncDecl
FuncBody
StatmList
CallStatm
AssignStatm
CallStatm
CallStatm

Syntaksanalyse

På skolen hadde vi grammatikkanalyse hvor vi fant subjekt og predikat:

Mannen ga piken 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 klarere definisjon.
- Programmeringsspråk er laget for å kunne analyseres rimelig enkelt.

RusC-kompilatoren har i hvert fall disse klassene:

```
[SyntaxUnit]
  DeclList
  [Declaration]
    FuncDecl
    VarDecl
  [ExprElement]
    FunctionCall
    Operator
  Expression
  Program
  SimpleExpr
  [Statement]
    EmptyStatm
    ForStatm
    IfStatm
    ReturnStatm
    WhileStatm
  StatmList
```

(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

Utifra dette vet vi:

- Først kommer symbolet while.
- Så kommer en (.
- Så kommer en *expression*.
- Etter den kommer en).
- Deretter kommer en {.
- I klammene kommer *statm-list*.
- 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 RusC-program:

```
public static WhileStatm parse() {
  ⟨Sjekk at vi har lest while⟩
  ⟨Sjekk at vi har lest (⟩
  ⟨Analysér Expression⟩
  ⟨Sjekk at vi har lest )⟩
  ⟨Sjekk at vi har lest {⟩
  ⟨Analysér 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?

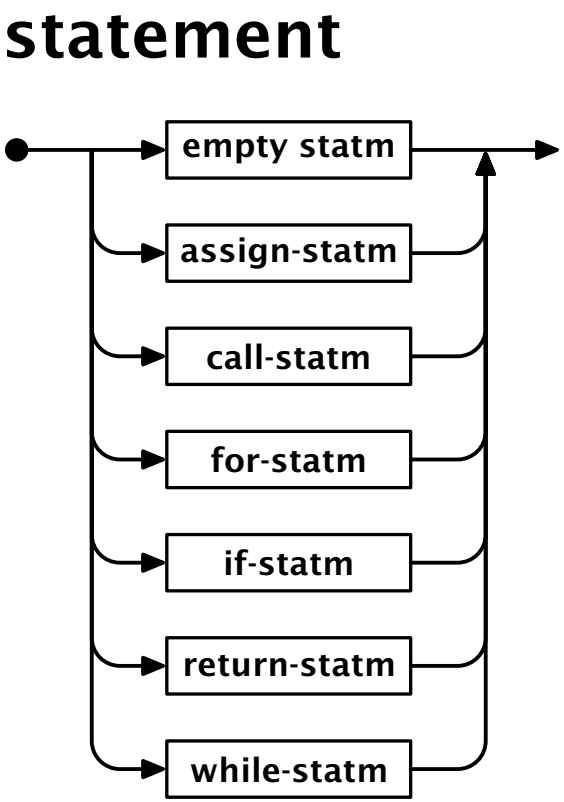
Mange programmeringsspråk (som RusC og Pascal men ikke Java, C og C++) er designet slik at denne teknikken kalt «recursive descent» alltid fungerer.

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 RusC) 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:

- Vi skal sjekke at programmet er riktig.
- 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.
- Hver av disse klassene får en metode

  ```
  public static xxx parse () { ... }
  ```

  som kan analysere «seg selv».

Dette er OO-programmering.

Vanlige variable i klasser

Vanlige variable oppstår når et objekt opprettes. Det kan derfor være vilkårlig mange av dem.

static-variable

Disse ligger i «selve klassen» så det vil alltid være nøyaktig én slik.

Et eksempel

```
class Item {
    private static int total = 0;
    public int id;

    public Item() { id = ++total; }
}

class RunItem {
    public static void main(String arg[]) {
        Item a = new Item(), b = new Item();

        System.out.println("a.id = "+a.id);
        System.out.println("b.id = "+b.id);
    }
}
```

Vanlige metoder i klasser

Vanlige metoder ligger logisk sett i det enkelte objektet. Når de refererer til variable, menes variable i samme objekt eller static-variable i klassen.

static-metoder

Disse ligger logisk sett i «selve klassen». De kan derfor kalles før noen objekter er opprettet men de kan bare referere til static-variable i samme klasse.

```
class BinTreeNode {
    private static BinTreeNode root = null;
    private BinTreeNode l_sub, r_sub;
    public int value;

    BinTreeNode(int v) { value = v;  l_sub = r_sub = null; }

    public static void insert(BinTreeNode p) {
        if (root == null) root = p;
        else root.insertNode(p);
    }

    private void insertNode(BinTreeNode p) {
        if (value <= p.value) {
            if (l_sub == null) l_sub = p;
            else l_sub.insertNode(p);
        } else {
            if (r_sub == null) r_sub = p;
            else r_sub.insertNode(p);
        }
    }
}
```

```
class RunBinTree {
    public static void main(String arg[]) {
        BinTreeNode.insert(new BinTreeNode(17));
        BinTreeNode.insert(new BinTreeNode(-4));
        BinTreeNode.insert(new BinTreeNode(3));
    }
}
```

Feilsjekking

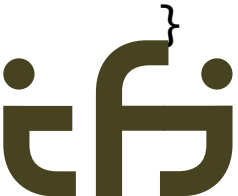
Sjekken på syntaksfeil er svært enkel:

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;
    StatmList body;

    public static WhileStatm parse(DeclList decls) {
          :
        WhileStatm w = new WhileStatm();
        Scanner.readNext();
        Scanner.skip(Token.leftParToken);
        w.test = Expression.parse(decls);
        Scanner.skip(Token.rightParToken);
        Scanner.skip(Token.leftCurlToken);
        w.body = StatmList.parse(decls);
        Scanner.skip(Token.rightCurlToken);
          :
        return w;
    }
    :
}
```



Husk

I Scanner-modulen har vi

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

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

Oppsummering

Vi har vært gjennom

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