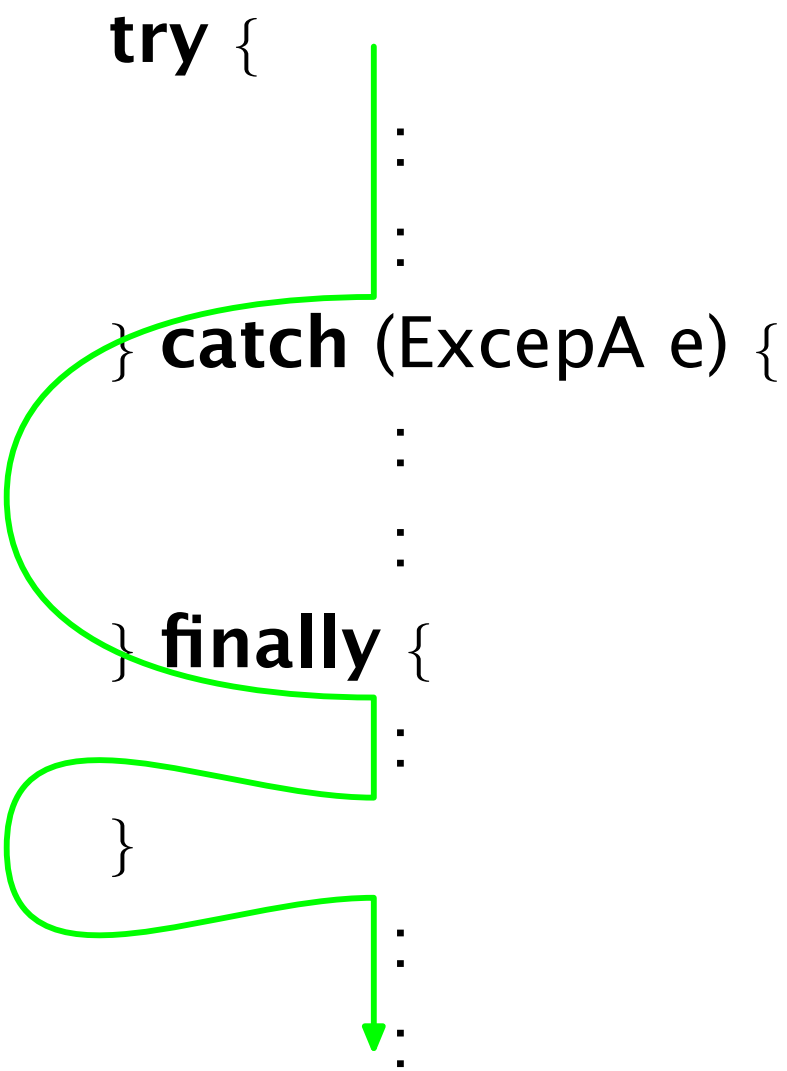
Dagens tema

- Mer om `try-catch-finally`
- Hva er kompilering?
- Hvordan foreta syntaksanalyse av et program?
- Hvordan programmere dette i Java?
- Hvordan oppdage syntaksfeil?

Avbrudd

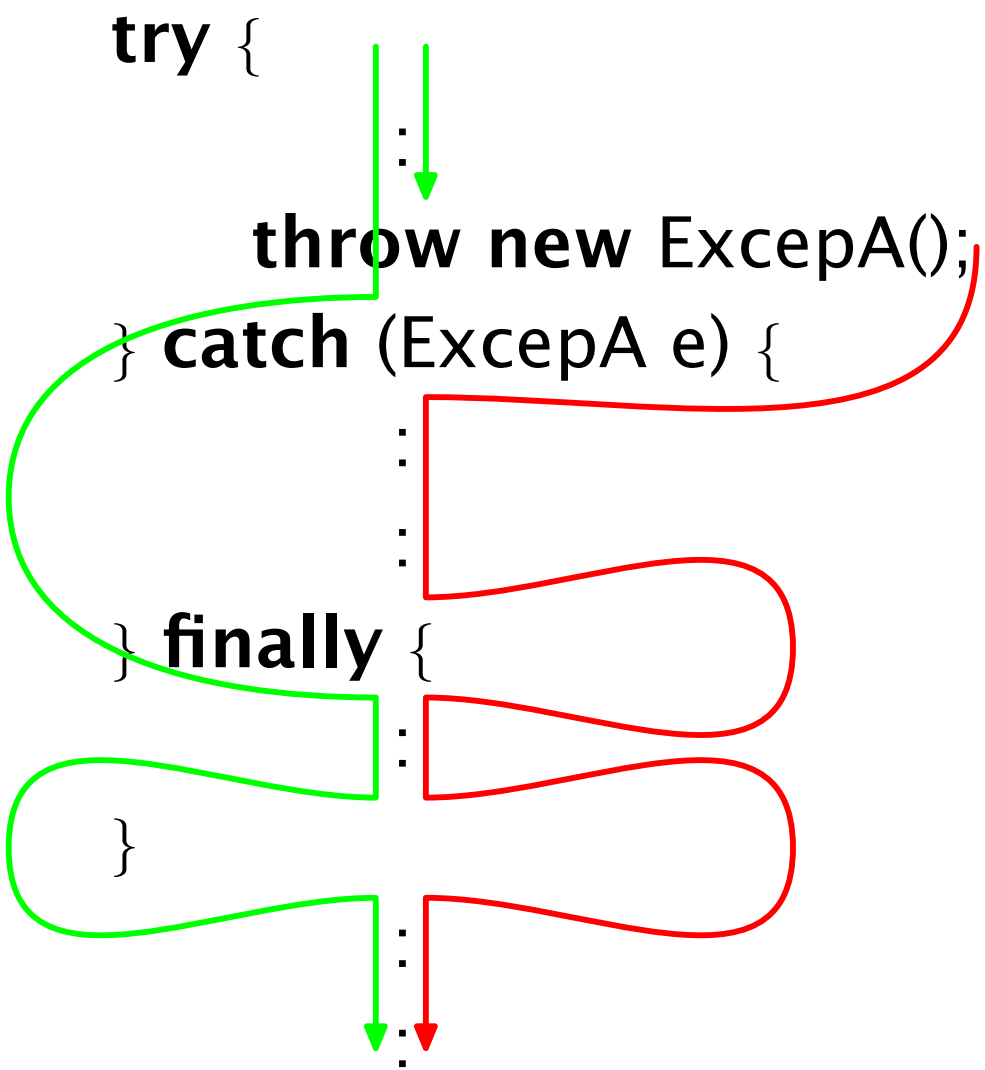
Uten avbrudd utføres `finally`-delen.



Avbrudd

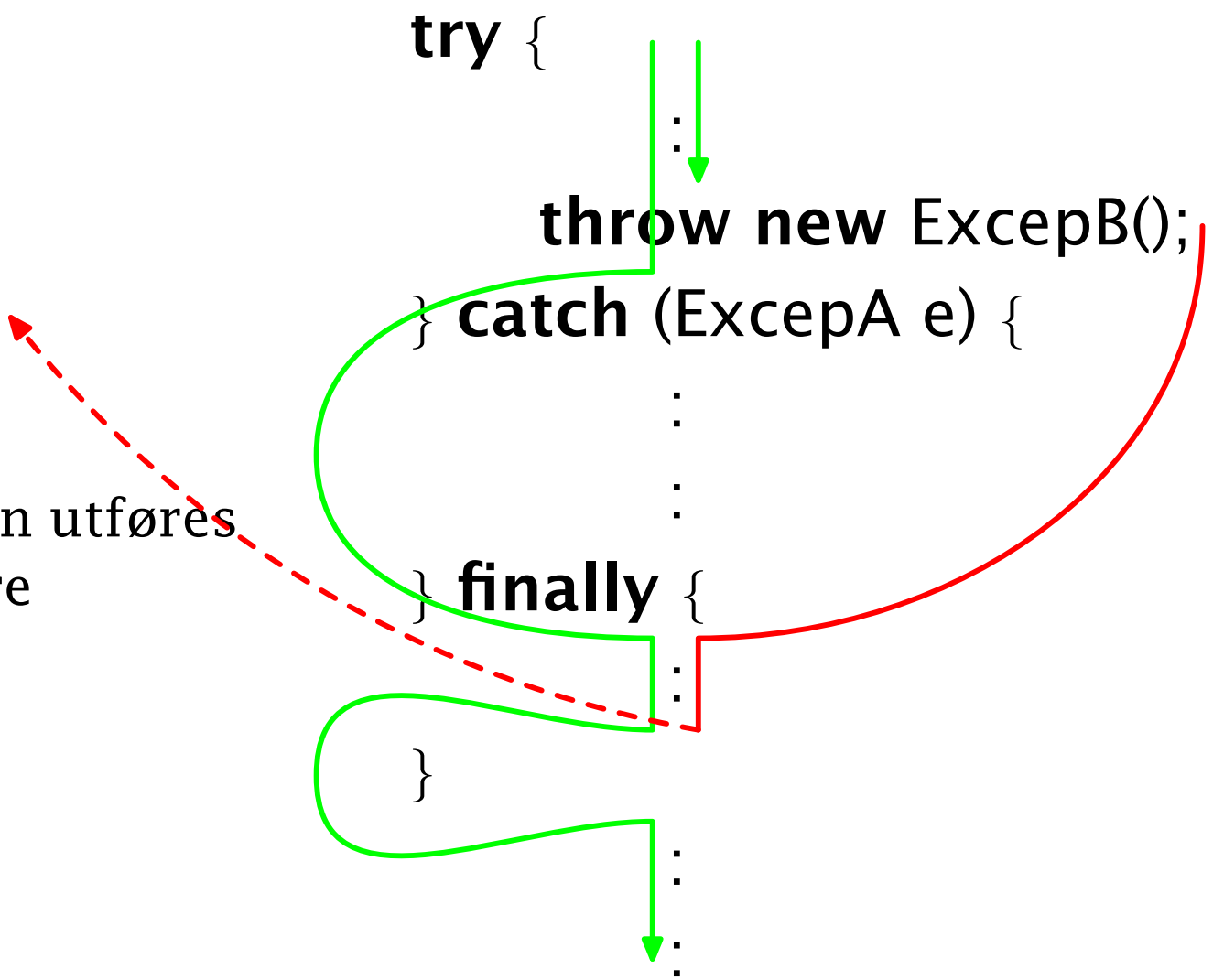
Uten avbrudd utføres `finally`-delen.

Også ved avbrudd utføres `finally`-delen.



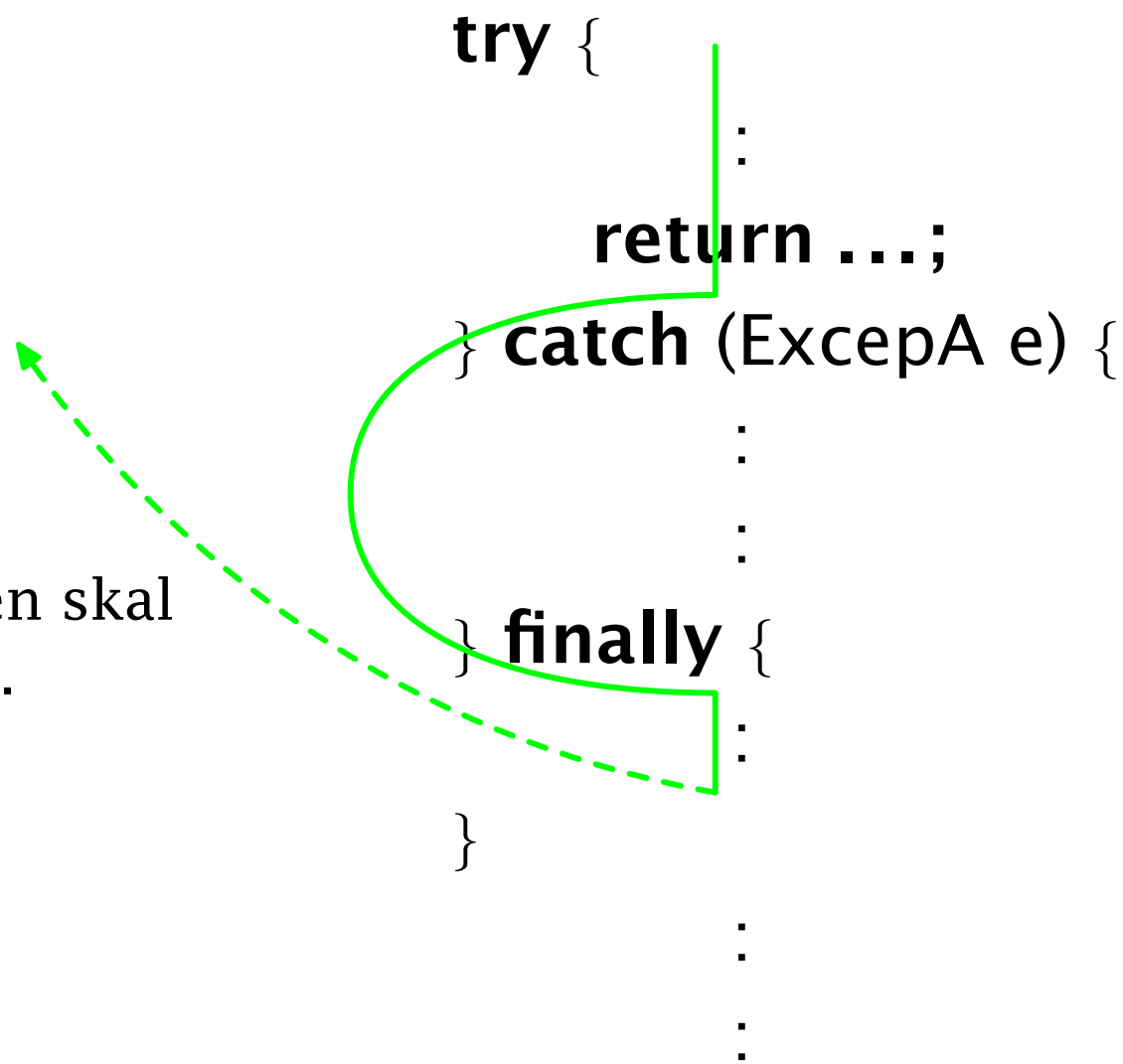
Avbrudd

`finally`-delen utføres også for andre avbrudd.



Avbrudd

`finally`-delen skal *alltid* utføres.



Hva er kompilering?

Anta at vi lager dette lille programmet `mini.pas` (kalt *kildekoden*):

```
/* Et minimalt Pascal-program */
program Mini;
begin
   write('x');
end.
```

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 50 83 04 08 34 00 00 00
0000040 54 11 00 00 00 00 00 00 34 00 20 00 09 00 28 00
0000060 1e 00 1b 00 06 00 00 00 34 00 00 00 34 80 04 08
0000100 34 80 04 08 20 01 00 00 20 01 00 00 05 00 00 00
0000120 04 00 00 00 03 00 00 00 54 01 00 00 54 81 04 08
0000140 54 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 94 06 00 00 94 06 00 00 05 00 00 00
0000220 00 10 00 00 01 00 00 00 08 0f 00 00 08 9f 04 08
0000240 08 9f 04 08 18 01 00 00 1c 01 00 00 06 00 00 00
0000260 00 10 00 00 02 00 00 00 14 0f 00 00 14 9f 04 08
```

⋮

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

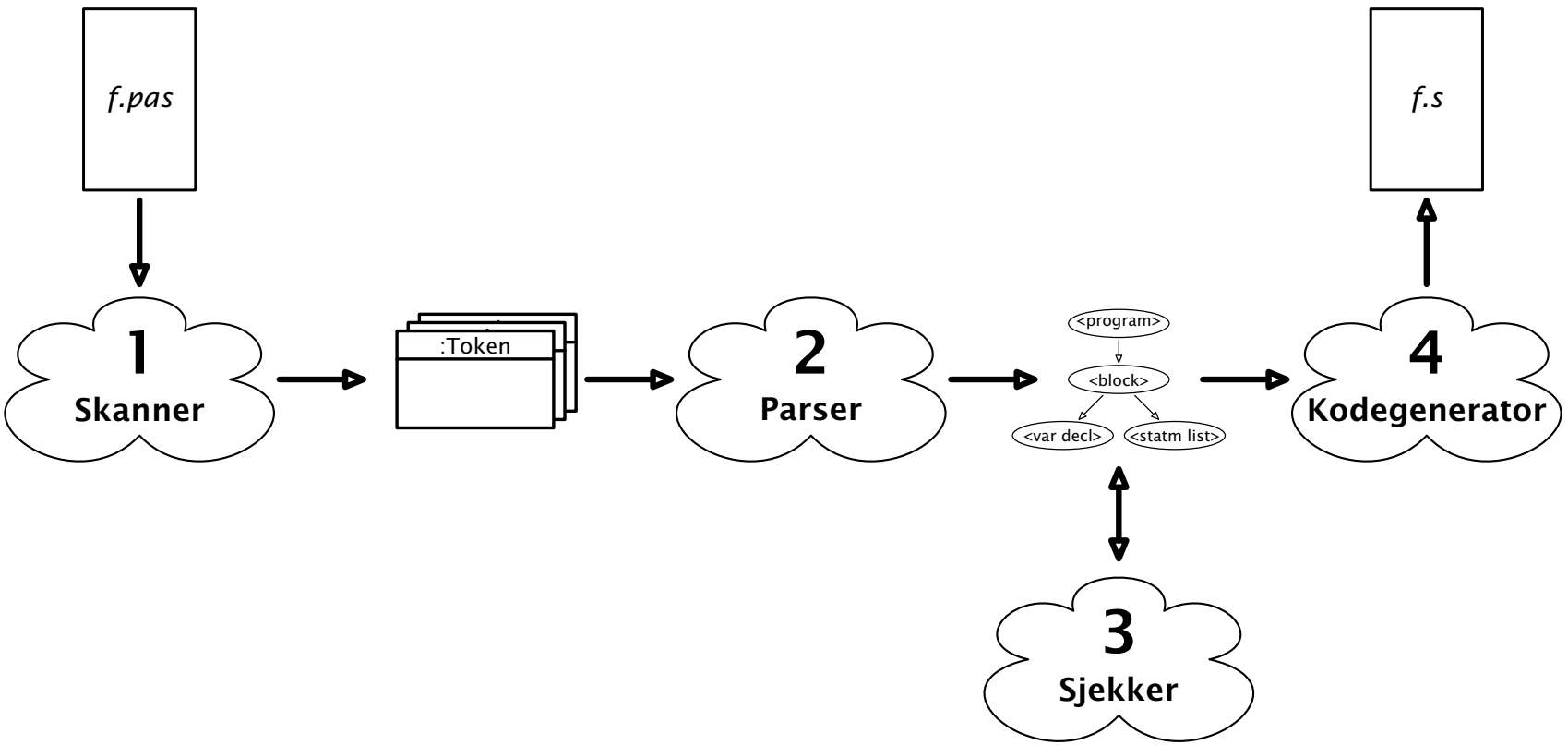
```
# Code file created by Pascal2016 compiler 2016-08-31 13:59:31
        .globl  main
main:
        call    prog$mini_1             # Start program
        movl    $0,%eax                 # Set status 0 and
        ret                             # terminate the program
prog$mini_1:
        enter   $32,$1                  # Start of mini
        movl    $120,%eax               #   'x'
        pushl   %eax                    # Push next param.
        call    write_char
        addl    $4,%esp                 # Pop param.
        leave                           # End of mini
        ret
```

Kompilatoren

En kompilator leser Pascal2016-koden og lager x86-assemblerkoden.

En slik kompilator skal dere lage.

Strukturen til kompilatoren vår



Programtreet

De færreste programmeringsspråk kan oversettes linje for linje, men det ville vært mulig med Pascal2016.

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 Pascal2016-program

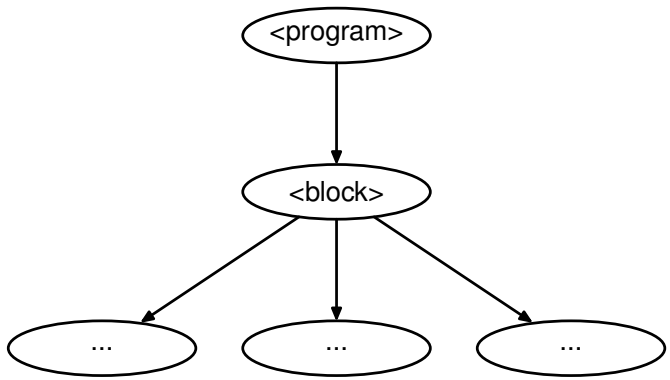
Et program består av en navngitt blokk:

program

Et tre er den beste representasjonsformen

Programmet `mini.pas` representeres da av

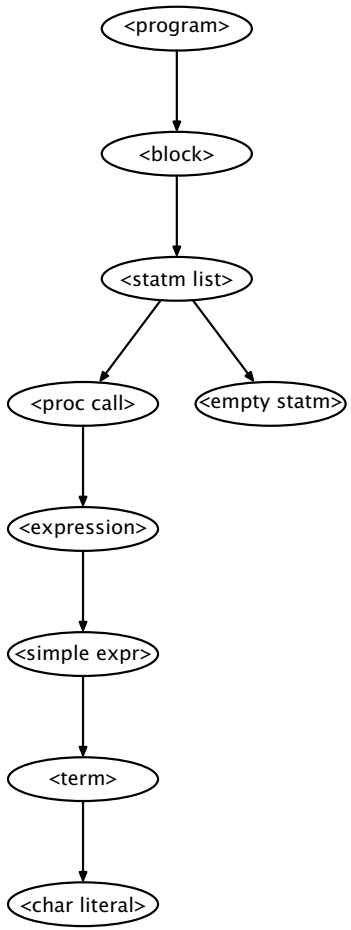
```
/* Et minimalt Pascal-program */
program Mini;
begin
   write('x');
end.
```



Et tre er den beste representasjonsformen

Det fulle programmet `mini.pas` ser slik ut:

```
/* Et minimalt Pascal-program */
program Mini;
begin
   write('x');
end.
```



Slike trær skal vi lage.

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**.

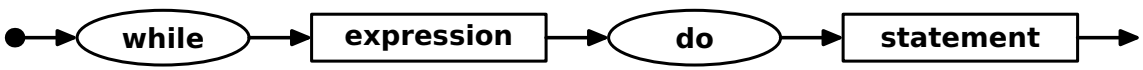
Heldigvis: Analyse av programmeringsspråk er enklere enn naturlige språk:

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

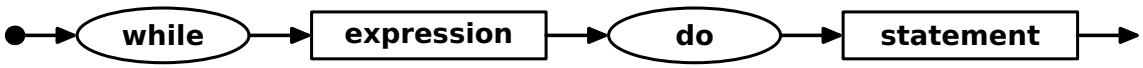
Grammatikk

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 *expression*.
3. Etter den kommer en do.
4. Til sist kommer et *statement*.

Programmering av syntaksanalyse

Utifra jernbanediagrammet kan vi lage en skisse for en metode som analyserer en `while`-setning i et Pascal2016-program:

```
class WhileStatm extends Statement {
    :

    static WhileStatm parse(Scanner s) {
      ⟨Sjekk at vi har lest while⟩
      Expression.parse(s);
      ⟨Sjekk at vi har lest do⟩
      Statement.parse(s);
    }
}
```

Stort sett gjør vi to ting:

- Terminaler (i rundinger) sjekkes.
- Ikke-terminaler (i firkanter) overlates til sine egne metoder for analysering.

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

Er det så enkelt?

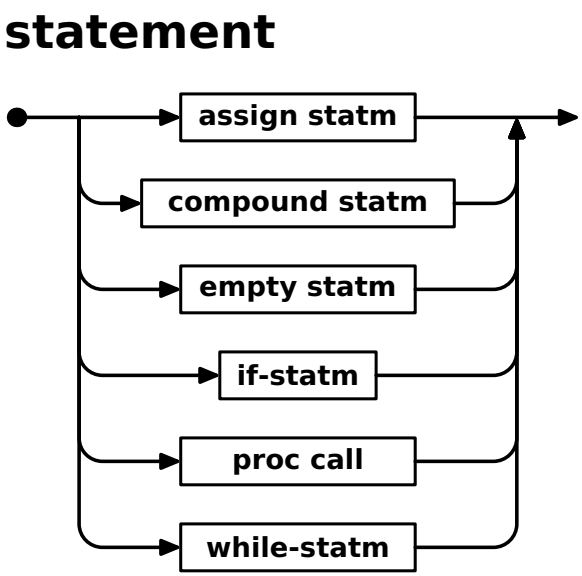
Mange programmeringsspråk (som Java og Pascal men ikke C og C++) er designet slik at denne teknikken kalt **recursive descent** fungerer.

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. Pascal2016 er LL(2).[1]

[1] Det finnes ett unntak; det skal vi se på neste uke.

Veien er alltid klar



Hvordan programmerer vi dette i Java?

Husk at målet med analysen er tofoldig:

1. Vi skal sjekke at programmet er riktig.
2. Vi skal bygge opp et korrekt syntakstre.

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

- Hver ikke-terminal i jernbanediagrammet implementeres av en Java-klasse.[2]
- Hver av disse klassene får en metode

  ```
  static xxx parse(Scanner s) { ... }
  ```

  som kan analysere «seg selv» og returnere et objekt som representerer «seg selv».

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

Pascal2016-kompilatoren *kan* ha disse klassene til å representere **ikke-terminaler**. (Abstrakte klasser er satt i hakeparenteser og innrykk markerer subklasser.)

```
CodeFile
LogFile
Main
[PascalSyntax]
- Block
- - Library
- ConstDeclPart
- Constant
- Expression
- [Factor]
- - FuncCall
- - InnerExpr
- - Negation
- - [UnsignedConstant]
- - - CharLiteral
- - - NamedConst
- - - NumberLiteral
- - Variable
- [Operator]
- - FactorOperator
- - PrefixOperator
- - RelOperator
- - TermOperator
- ParamDeclList
- [PascalDecl]
- - ConstDecl
- - ParamDecl
- - ProcDecl
- - - FuncDecl
- - Program
- - TypeDecl
- - VarDecl
- SimpleExpr
- [Statement]
- - AssignStatm
- - CompoundStatm
- - EmptyStatm
- - IfStatm
- - ProcCallStatm
- - WhileStatm
- StatmList
- Term
- [Type]
- - ArrayType
- - BoolType
- - CharType
- - IntType
- - TypeName
- VarDeclPart
Scanner
Token
TokenKind
```

Statiske deklarasjoner

Vanlige variabler i klasser

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

`static`-variabler

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

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);
    }
}
```

Resultatet

```
a.id = 1
b.id = 2
```

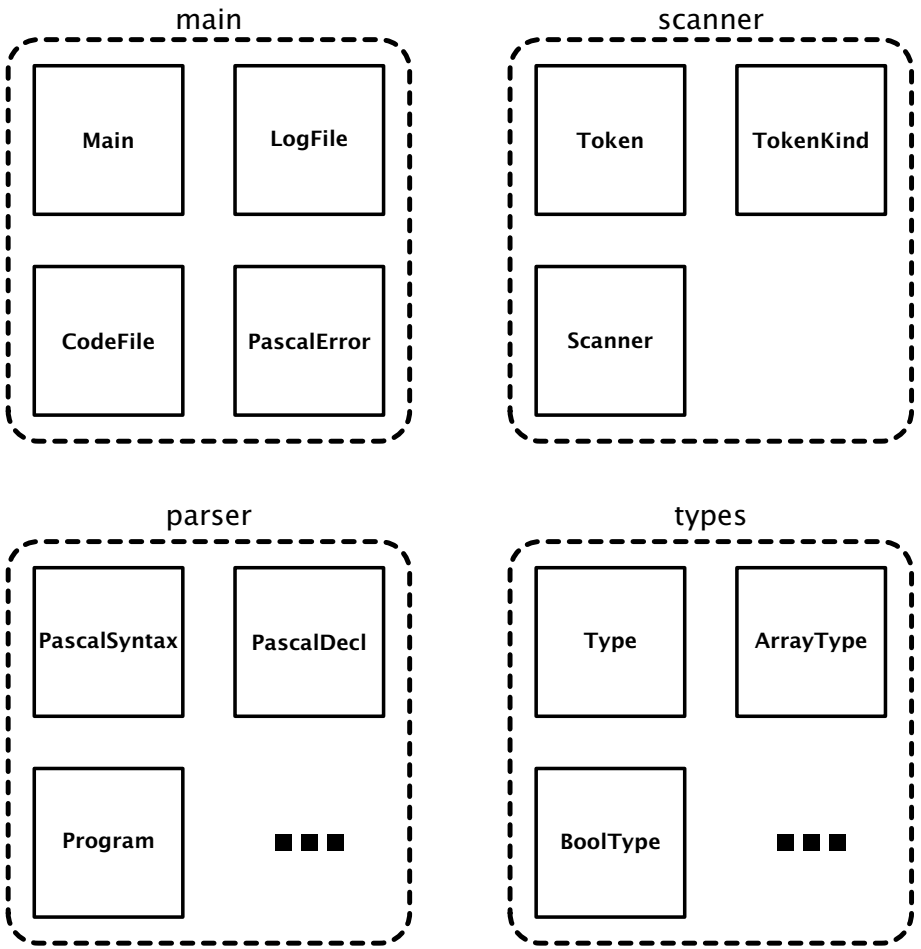
Vanlige metoder i klasser

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

Prosjektet vårt i Java



Rot-klassen

Vi lar alle parser-klassene være subklasser av PascalSyntax.

```
package parser;

import main.*;

public abstract class PascalSyntax {
    public int lineNum;

    PascalSyntax(int n) {
        lineNum = n;
    }

    boolean isInLibrary() {
        return lineNum < 0;
    }

    abstract void check(Block curScope, Library lib);
    abstract void genCode(CodeFile f);
    abstract public String identify();
    abstract void prettyPrint();

    public void error(String message) {
        Main.error("Error at line " + lineNum + ": " + message);
    }
```

Alle deklarasjonene er også subklasser av `PascalDecl`:

```
package parser;

public abstract class PascalDecl extends PascalSyntax {
    String name, progProcFuncName;
    int declLevel = 0, declOffset = 0;
    types.Type type = null;

    PascalDecl(String id, int lNum) {
        super(lNum);
        name = id;
    }
```

Klassen Program kan da se slik ut:

```
package parser;

import main.*;
import scanner.*;
import static scanner.TokenKind.*;

/* <program> ::= 'program' <name> ';' <block> '.' */

public class Program extends PascalDecl {
    Block progBlock;

    Program(String id, int lNum) {
        super(id, lNum);
    }


    @Override public String identify() {
        return "<program> " + name + " on line " + lineNum;
    }
```

```
public static Program parse(Scanner s) {
    enterParser("program");
    s.skip(programToken);
    s.test(nameToken);

    Program p = new Program(s.curToken.id, s.curLineNum());
    s.readNextToken();
    s.skip(semicolonToken);
    p.progBlock = Block.parse(s);  p.progBlock.context = p;
    s.skip(dotToken);

    leaveParser("program");
    return p;
}
```

Husk

I `scanner.Scanner` har vi

```
public void test(TokenKind t) {
    if (curToken.kind != t)
        testError(t.toString());
}

public void testError(String message) {
    Main.error(curLineNum(),
               "Expected a " + message +
               " but found a " + curToken.kind + "!");
}

public void skip(TokenKind t) {
    test(t);
    readNextToken();
}
```

og i `main.Main` har vi

```
public static void error(String message) {
    log.noteError(message);
    throw new PascalError(message);
}
```

Samarbeid med skanneren

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!

Feilsjekking

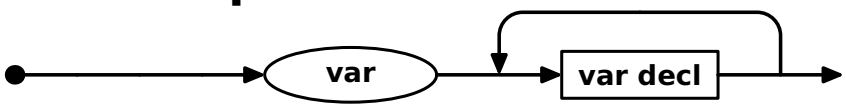
Sjekken på syntaksfeil er svært enkel:

Hvordan finne feil?

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

Et eksempel

var decl part



var decl


```
var v : integer = 5 ;
```

Parsering av While-setning

```
package parser;

import main.*;
import scanner.*;
import static scanner.TokenKind.*;

/* <while-statm> ::= 'while' <expression> 'do' <statement> */

class WhileStatm extends Statement {
    Expression expr;
    Statement body;

    WhileStatm(int lNum) {
        super(lNum);
    }

    @Override public String identify() {
        return "<while-statm> on line " + lineNum;
    }

    static WhileStatm parse(Scanner s) {
        enterParser("while-statm");

        WhileStatm ws = new WhileStatm(s.curLineNum());
        s.skip(whileToken);

        ws.expr = Expression.parse(s);
        s.skip(doToken);
        ws.body = Statement.parse(s);

        leaveParser("while-statm");
        return ws;
    }
}
```