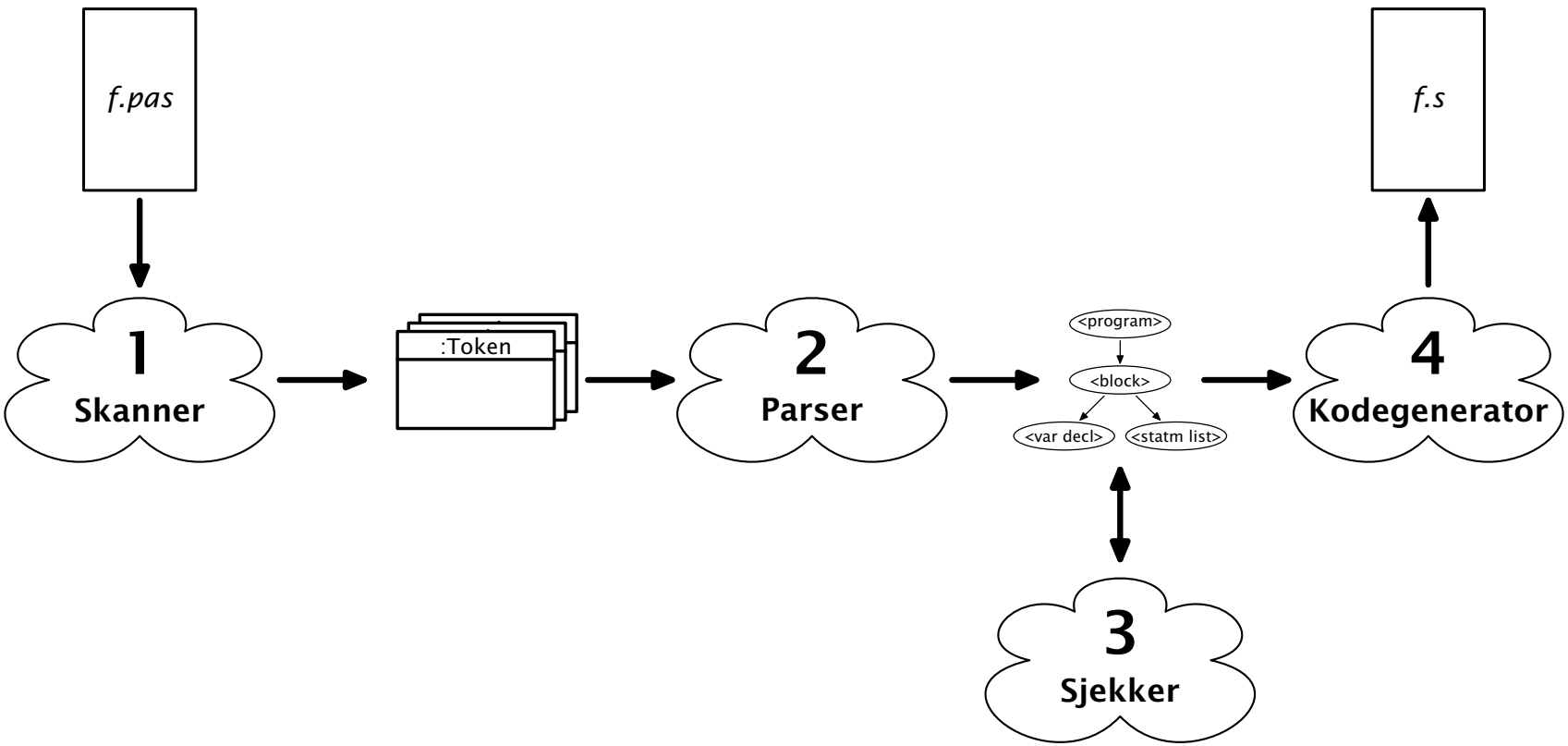
Dagens tema: Sjekking

- Navnebinding (obligatorisk oppgave 3)
  - Biblioteket
  - Logging
- Riktig bruk av navn (frivillig)
- Typesjekking (frivillig)

Strukturen til kompilatoren vår



Navnebinding

Gitt følgende program:

Oppgaven er: Alle navneforekomster skal bindes til sin deklarasjon.

```
program A;
var X: Integer;

procedure A (V: Integer);
type A = Integer;

   function F (V: A): A;
   begin
      X := 2*V;
      F := X
   end; { F }

begin
   X := X + F(V)
end; { A }

begin
   X := 1;
   A(10);
   write('X = ', X, eol)
end.
```

Syntakstreet

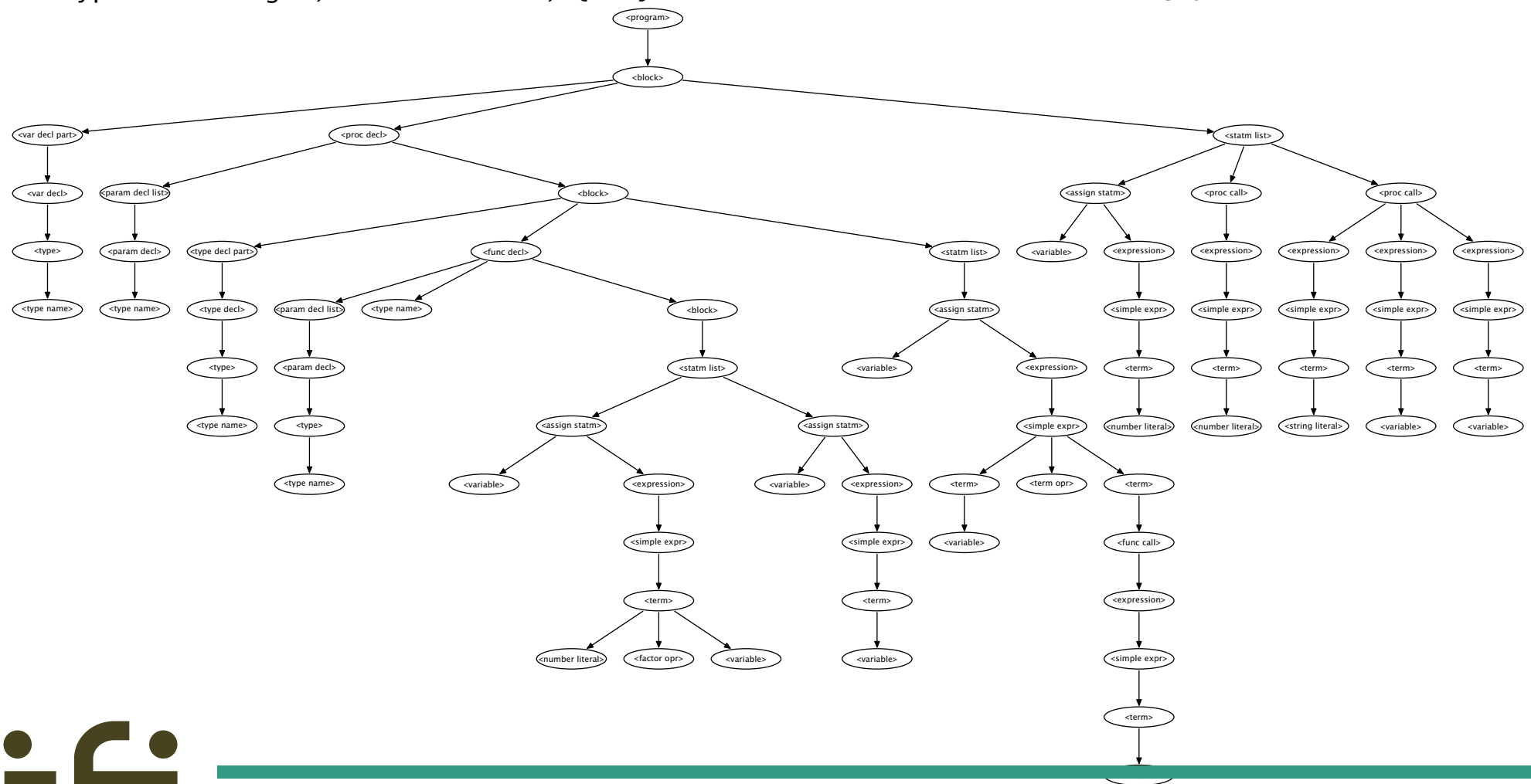
```
program A;
var X: Integer;

procedure A (V: Integer);
type A = Integer;
```

```
function F (V: A): A;
begin
   X := 2*V;
   F := X
end; { F }
```

```
begin
   X := X + F(V)
end; { A }
```

```
begin
   X := 1;
   A(10);
   write('X = ', X, eol)
end.
```



La oss forenkle syntakstreet:

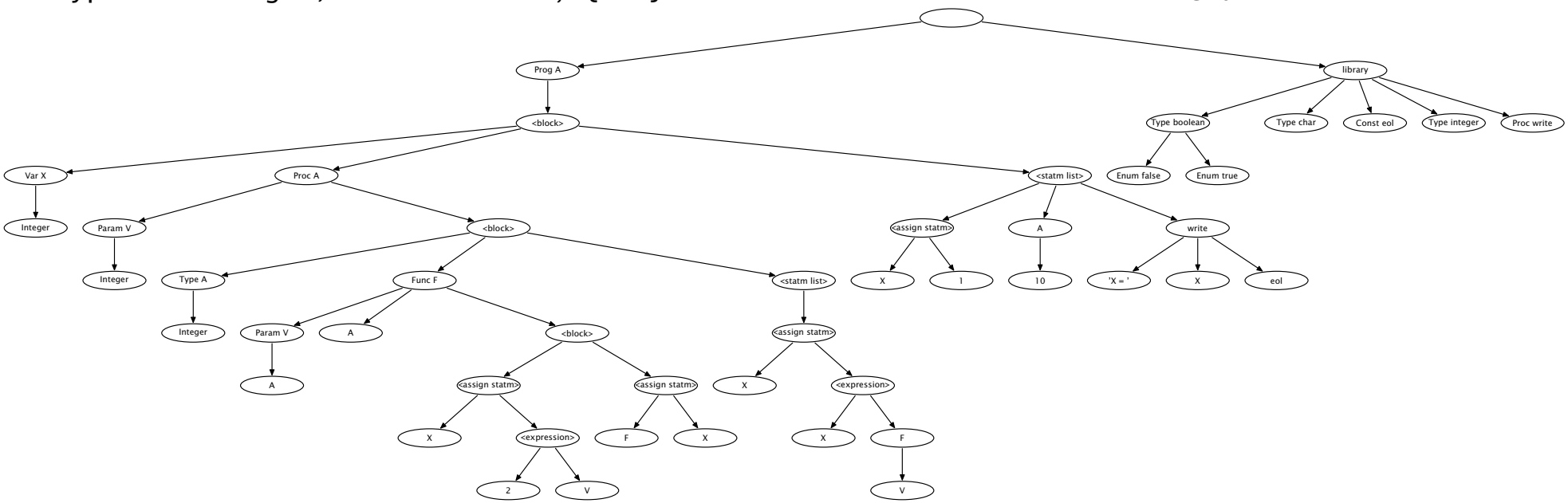
```
program A;
var X: Integer;

procedure A (V: Integer);
type A = Integer;
```

```
function F (V: A): A;
begin
   X := 2*V;
   F := X
end; { F }
```

```
begin
   X := X + F(V)
end; { A }
```

```
begin
   X := 1;
   A(10);
   write('X = ', X, eol)
end.
```



Hvilke noder har vi?

Grønne noder er deklarasjoner.

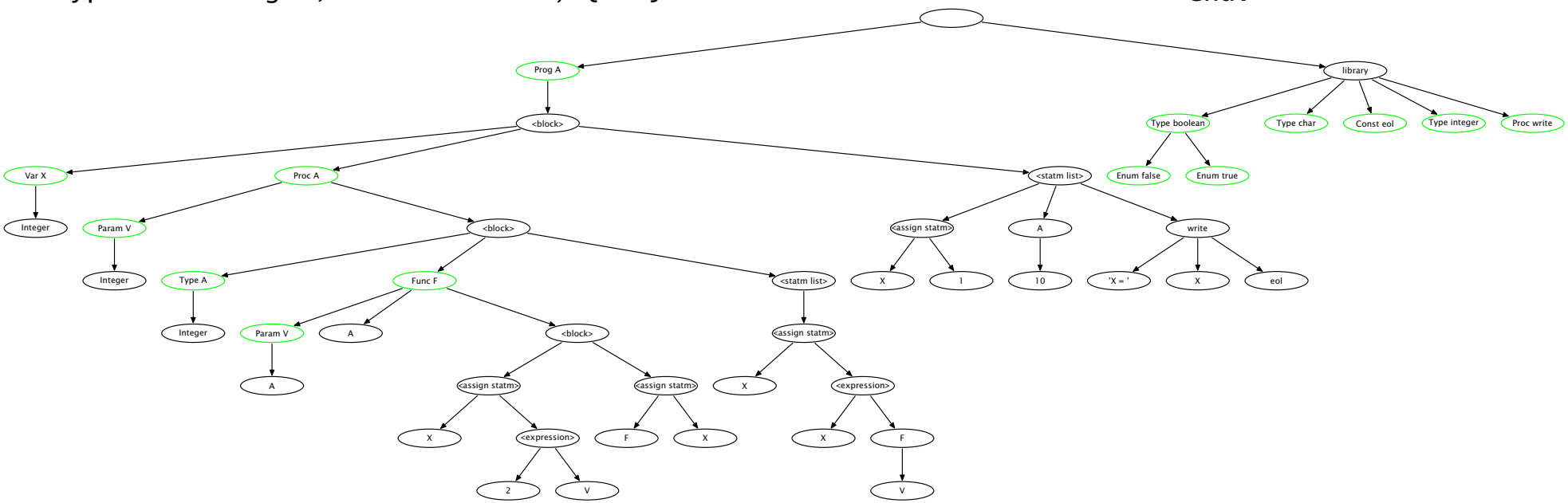
```
program A;
var X: Integer;

procedure A (V: Integer);
type A = Integer;
```

```
function F (V: A): A;
begin
   X := 2*V;
   F := X
end; { F }
```

```
begin
   X := X + F(V)
end; { A }
```

```
begin
   X := 1;
   A(10);
   write('X = ', X, eol)
end.
```



Hvilke noder har vi?

Blå noder er navneforekomster.

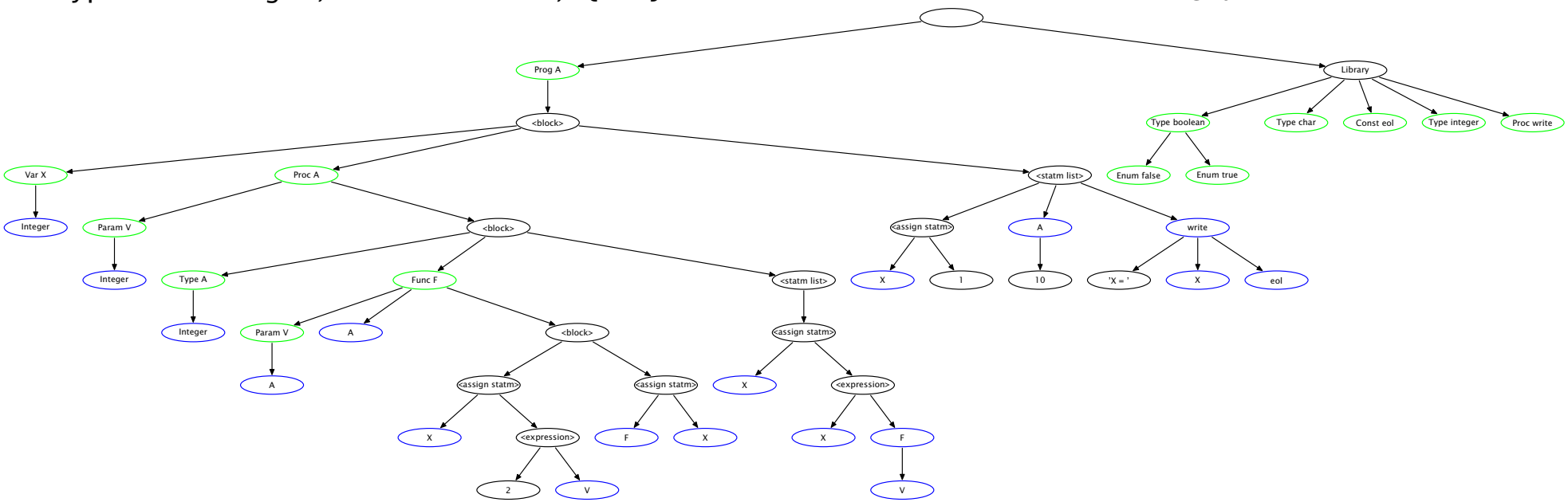
```
program A;
var X: Integer;

procedure A (V: Integer);
type A = Integer;
```

```
function F (V: A): A;
begin
   X := 2*V;
   F := X
end; { F }
```

```
begin
   X := X + F(V)
end; { A }
```

```
begin
   X := 1;
   A(10);
   write('X = ', X, eol)
end.
```



En del av dem skal bindes slik:

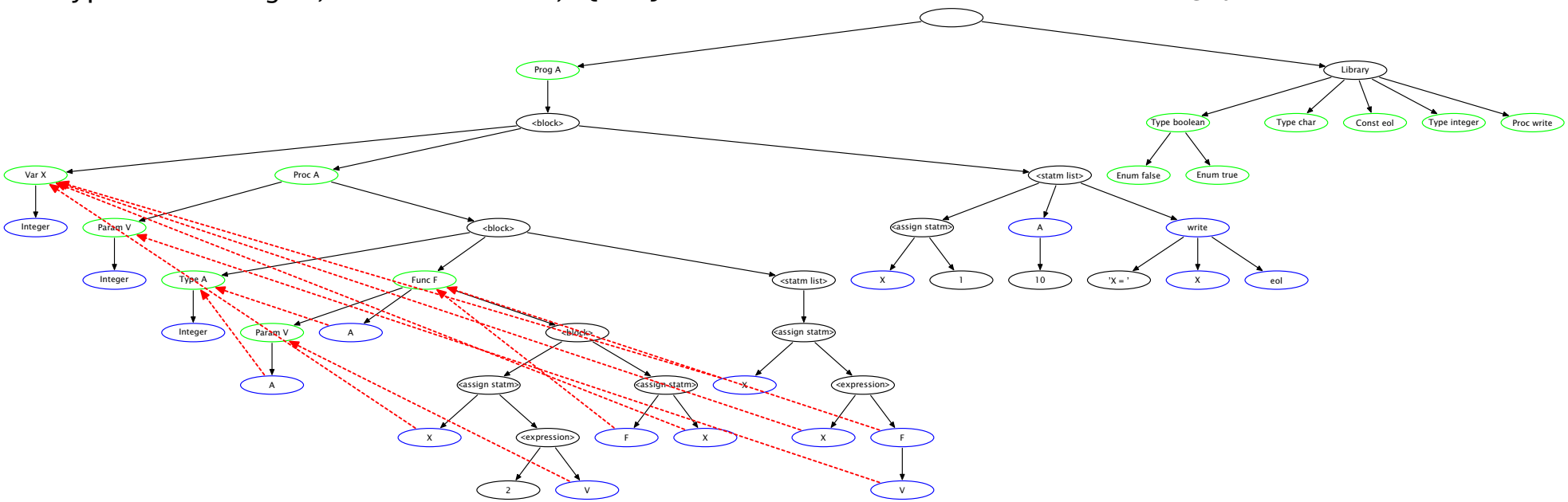
```
program A;
var X: Integer;

procedure A (V: Integer);
type A = Integer;
```

```
function F (V: A): A;
begin
   X := 2*V;
   F := X
end; { F }
```

```
begin
   X := X + F(V)
end; { A }
```

```
begin
   X := 1;
   A(10);
   write('X = ', X, eol)
end.
```



De andre skal bindes slik:

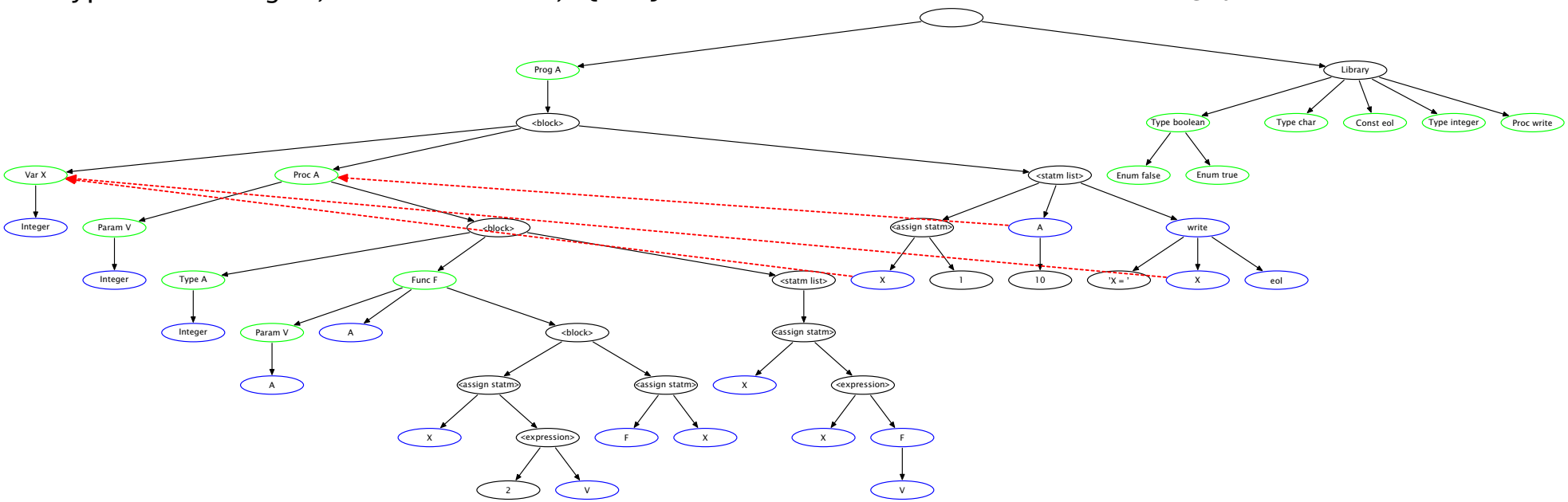
```
program A;
var X: Integer;

procedure A (V: Integer);
type A = Integer;
```

```
function F (V: A): A;
begin
   X := 2*V;
   F := X
end; { F }
```

```
begin
   X := X + F(V)
end; { A }
```

```
begin
   X := 1;
   A(10);
   write('X = ', X, eol)
end.
```



Hvordan kan foreta navnebindingen?

Alle deklarasjonene er i en ⟨block⟩:

- konstanter (`const pi = 3;`)
- typer (`type decimal = 0..9;`)
- enum-konstanter (`type farge = (R, G, B);`)
- variabler (`var teller: 1..2;`)
- funksjoner (`function f: char; ...`)
- prosedyrer (`procedure p; ...`)
- parametre (om ⟨block⟩ i funksjon eller prosedyre)

Et forslag

La klassen `Block` inneholde en oversikt over alle dens deklarasjoner, for eksempel i form av en `HashMap<String,PascalDecl>`.

En slik struktur ar en ekstra fordel: Det er enkelt å sjekke om noen navn er deklarert flere ganger i samme blokk. Dette skal i så fall gi en feilmelding.

Sjekkingen

For å sjekke hele programmet, må vi skrive en rekursiv metode

```
void check(Block curScope, Library lib)
```

som traverserer det. (Parametrene kommer vi tilbake til senere.)

Deler av Block kan se slik ut:

```
public class Block extends PascalSyntax {
    ConstDeclPart constDeclPart = null;
        :
    HashMap<String,PascalDecl> decls = new HashMap<String,PascalDecl>();

    void addDecl(String id, PascalDecl d) {
        if (decls.containsKey(id))
            d.error(id + " declared twice in same block!");
        decls.put(id, d);
    }

    @Override void check(Block curScope, Library lib) {
        if (constDeclPart != null) {
            constDeclPart.check(this, lib);
            for (ConstDecl cd: constDeclPart.constants) {
                addDecl(cd.name, cd);
            }
        }
        :
```

Leting etter navn

1. Det enkleste først: Anta at navnet finnes i den lokale blokken: Da sender vi med en peker til den som parameter til check.

Eksempel

```
class ProcCallStatm extends Statement {
    String procName;
    ArrayList<Expression> actParams = new ArrayList<Expression>();
    ProcDecl procRef;

    @Override void check(Block curScope, Library lib) {
        PascalDecl d = curScope.findDecl(procName,this);
            :
        procRef = (ProcDecl)d;
            :
```

2. Hvis deklarasjonen ikke er lokal, kan vi finne den ved å lete i ytre skop. Derfor bør `Block` inneholde en peker `Block outerScope` som peker på blokken utenfor. Den kan initieres av `check`.

Leting i ytre skop

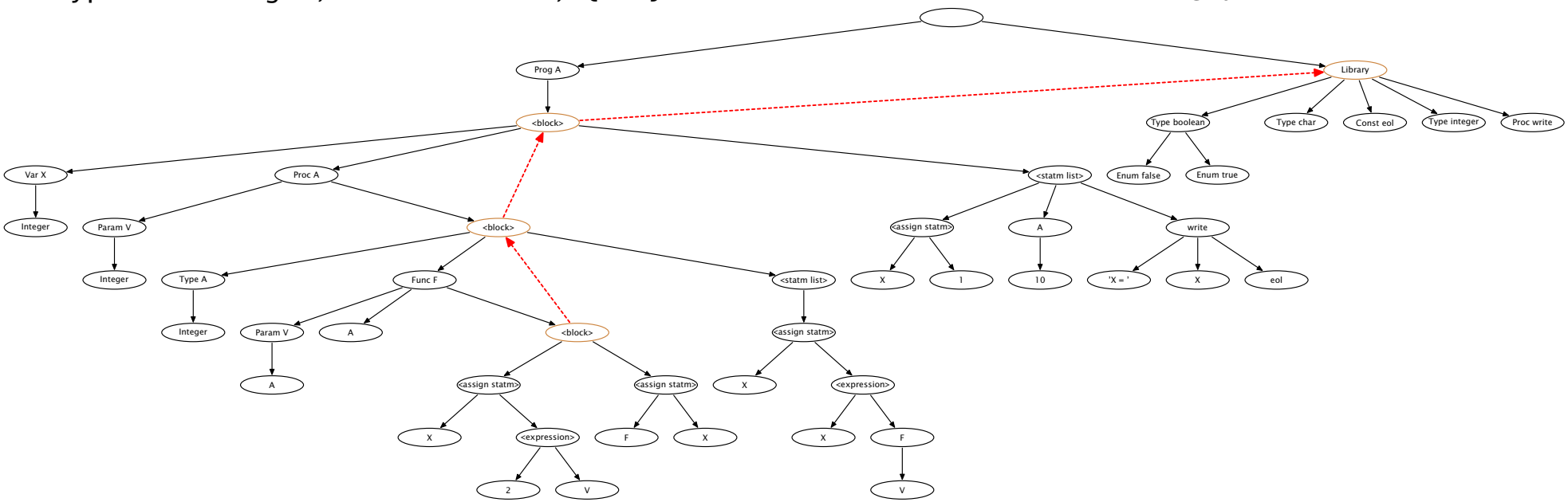
```
program A;
var X: Integer;

procedure A (V: Integer);
type A = Integer;
```

```
function F (V: A): A;
begin
   X := 2*V;
   F := X
end; { F }
```

```
begin
   X := X + F(V)
end; { A }
```

```
begin
   X := 1;
   A(10);
   write('X = ', X, eol)
end.
```



Klassen `Block` kan da ha en metode `findDecl`:

```
PascalDecl findDecl(String id, PascalSyntax where) {
    PascalDecl d = decls.get(id);
    if (d != null) {
        Main.log.noteBinding(id, where, d);
        return d;
    }

    if (outerScope != null)
        return outerScope.findDecl(id,where);

    where.error("Name " + id + " is unknown!");
    return null;  // Required by the Java compiler.
}
```

Biblioteket

Noen navn som `integer` og `write` er predefinert. Hvordan bør vi håndtere dem?

Løsning
Lag et «kunstig» `Block`-objekt med disse predefinerte deklarasjonene og legg det ytterst. Da vil de bli funnet om ikke brukeren har deklarert noe med samme navn.

Hint
Lag en subklasse `Library` av `Block` for dette formålet.

Biblioteket

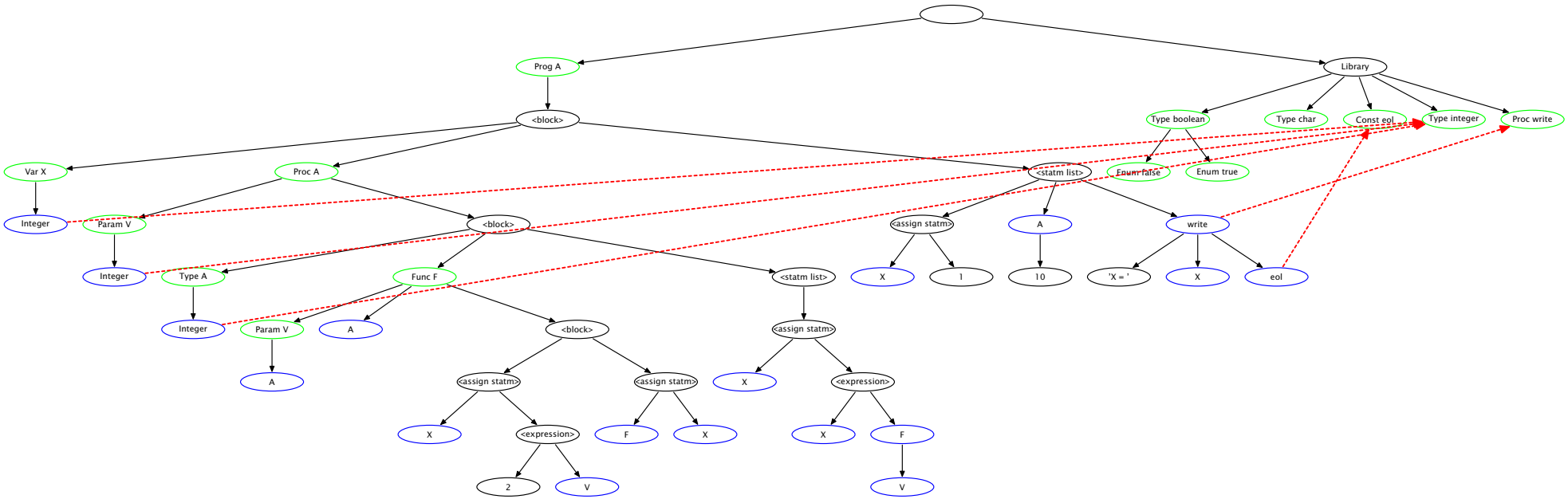
```
program A;
var X: Integer;

procedure A (V: Integer);
type A = Integer;
```

```
function F (V: A): A;
begin
   X := 2*V;
   F := X
end; { F }
```

```
begin
   X := X + F(V)
end; { A }
```

```
begin
   X := 1;
   A(10);
   write('X = ', X, eol)
end.
```



Kontroll

For å sjekke navnebindingen brukes opsjonen **-logB**. Kallet på `Main.log.noteBinding` i `Block.findDecl` gir oss den informasjonen vi trenger.

```
 1   program A;
 2   var X: Integer;
 3
 4   procedure A (V: Integer);
 5   type A = Integer;
 6
 7      function F (V: A): A;
 8      begin
 9         X := 2*V;
10         F := X
11      end; { F }
12
13   begin
14      X := X + F(V)
15   end; { A }
16
17   begin
18      X := 1;
19      A(10);
20      write('X = ', X, eol)
21   end.
```

```
Binding on line 2: integer was declared in <type decl> in the libra
Binding on line 4: integer was declared in <type decl> in the libra
Binding on line 5: integer was declared in <type decl> in the libra
Binding on line 7: a was declared in <type decl> on line 5
Binding on line 7: a was declared in <type decl> on line 5
Binding on line 9: x was declared in <var decl> on line 2
Binding on line 9: v was declared in <param decl> on line 7
Binding on line 10: f was declared in <func decl> on line 7
Binding on line 10: x was declared in <var decl> on line 2
Binding on line 14: x was declared in <var decl> on line 2
Binding on line 14: x was declared in <var decl> on line 2
Binding on line 14: f was declared in <func decl> on line 7
Binding on line 14: v was declared in <param decl> on line 4
Binding on line 18: x was declared in <var decl> on line 2
Binding on line 19: a was declared in <proc decl> on line 4
Binding on line 20: write was declared in <proc decl> in the librar
Binding on line 20: x was declared in <var decl> on line 2
Binding on line 20: eol was declared in <const decl> in the library
```

Navnebruk (frivillig)

Etter å ha funnet hvor et navn er deklarert, må en kompilator sjekke at det brukes rett, for eksempel at vi *ikke* har

```
const C = 5;
procedure P;
begin
  C := 10 + P;
end; {P}
```

Hvordan sjekke dette?

Det er mange måter å sjekke navnebruken på; jeg skal vise at oo-programmering kan gjøre dette enkelt og oversiktlig.

1. Deklarer en virtuell funksjon i klassen `PascalDecl`:

```
abstract void checkWhetherAssignable(PascalSyntax where);
```

2. I alle deklarasjoner som *kan* stå til venstre i en tilordning (f eks `VarDecl` og `FuncDecl`), implementeres denne som en tom metode:

```
@Override void checkWhetherAssignable(PascalSyntax where) {}
```

3. I alle andre deklarasjoner lager vi i stedet en

```
@Override void checkWhetherAssignable(PascalSyntax where) {
    where.error("You cannot assign to a constant.");
}
```

Ved alle navneforekomster der det skal skje en tilordning (f eks i `AssignStatm`), kan vi bruke denne metoden:

```
class AssignStatm extends Statement {
    Variable var;
    Expression expr;

    @Override void check(Block curScope, Library lib) {
        var.check(curScope, lib);
        var.varDecl.checkWhetherAssignable(this);
        expr.check(curScope, lib);
          :
```

Hvilke `checkWhether`-metoder trenger vi?

Selv har jeg brukt disse:

`checkWhetherAssignable` for tilordning (i `AssignStatm` og funksjons- og prosedyrekall)

`checkWhetherFunction` for funksjonskall

`checkWhetherProcedure` for prosedyrekall

`checkWhetherValue` for uttrykk

Typesjekking (frivillig)

Det er også viktig å sjekke at programmereren overholder typereglene og ikke skriver slikt som

```
type Index = 1 .. 'z';
var A: Integer;
    B: Char;
    C: array [Index] of Boolean;
begin
  if A then begin
    B := C + 1;
       :
```

Hvordan kan dette implementeres?

Det er også her mange mulige måter å ordne seg på; jeg vil vise én som minner om `checkWhether`:

1. I klassen Type deklareres en virtuell metode

   ```
   abstract void checkType(Type tx, PascalSyntax where, String message);
   ```

   Parametrene er:

   `tx` er typen som «vår» type skal sammenlignes med
   `where` angir hvor i programmet typen forekommer
   `message` inneholder meldingen som skal gis om det er typefeil

2. Hver subklasse til Type må da implementere en fornuftig checkType. I EnumType kan den se slik ut:

```
class EnumType extends Type {
    ArrayList<EnumLiteral> values = new ArrayList<EnumLiteral>();


    @Override void checkType(Type tx, PascalSyntax where, String message) {
        if (tx == this)
            return;  // OK
        else if (tx instanceof TypeName)
            checkType(((TypeName)tx).namedRef, where, message);
        else
            where.error(message);
    }
```

Nå har vi testapparatet vi trenger. I `WhileStatm` kan vi for eksempel skrive

```
class WhileStatm extends Statement {
    Expression expr;
    Statement body;


    @Override void check(Block curScope, Library lib) {
        expr.check(curScope, lib);
        Main.log.noteTypeCheck("while", expr.type, this);
        expr.type.checkType(lib.booleanType, this,
            "While-test is not Boolean.");
        body.check(curScope, lib);
    }
```

(Her ser vi at parameteren `lib` til check-metoden er nyttig.)