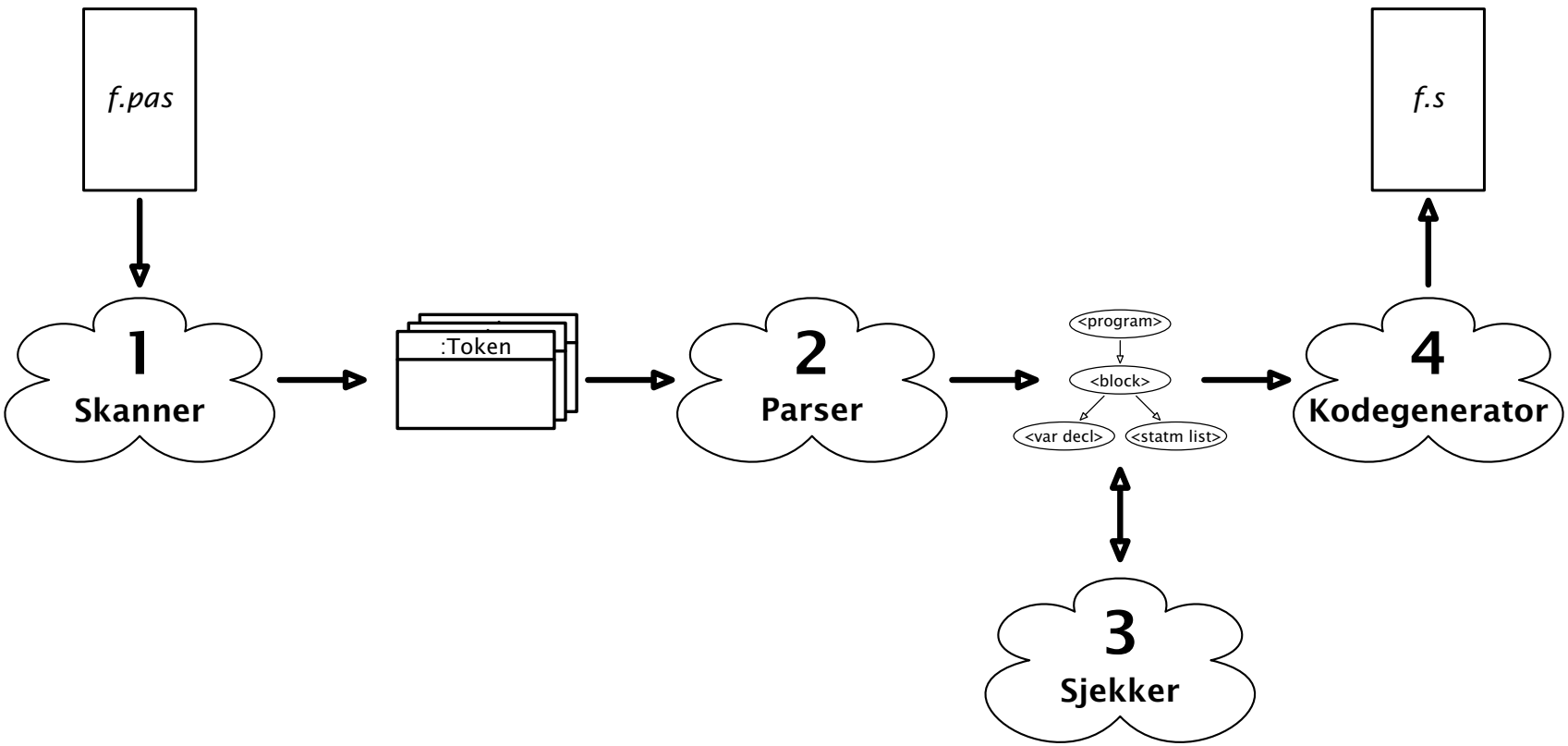
Dagens tema: Sjekking (obligatorisk oppgave 3)

- Navnebinding
- Biblioteket
- Logging
- Riktig bruk av navn
- Typesjekking
- Utregning av konstanter

Strukturen til kompilatoren vår



Navnebinding

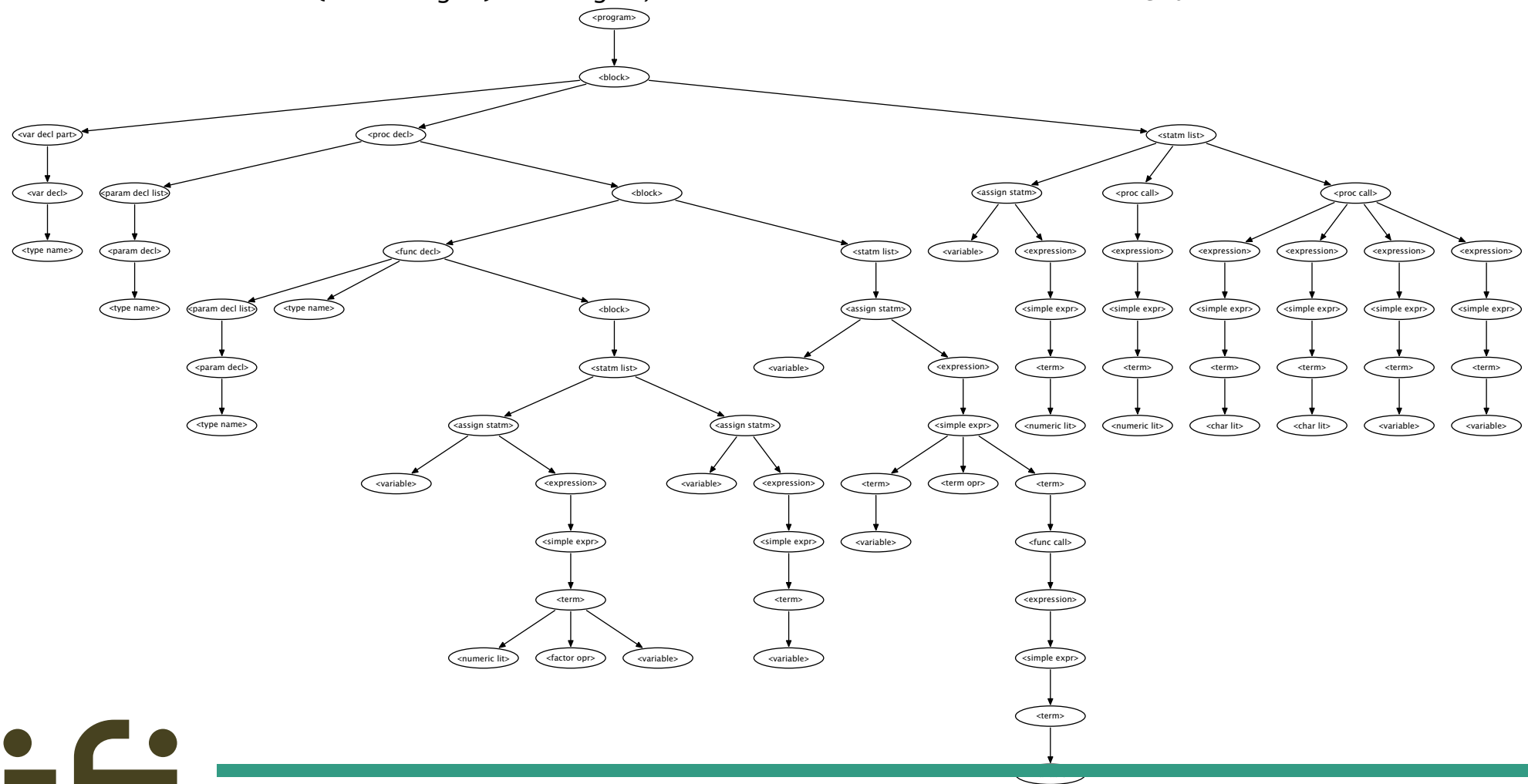
Gitt følgende program:

Oppgaven er: Alle navneforekomster skal bindes til sin deklarasjon.

```
program A;
  var X: Integer;

  procedure A (V: Integer);
    function F (V: Integer): Integer;
    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.
```

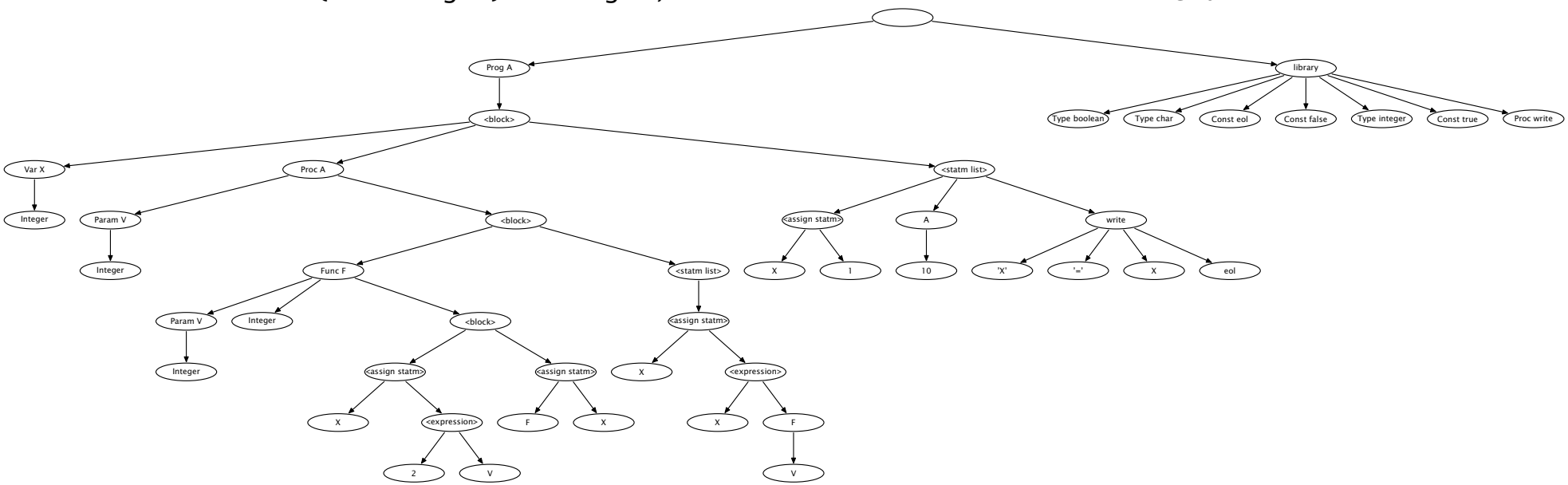
Syntakstreet

```
program A;
  var X: Integer;

  procedure A (V: Integer);
    function F (V: Integer): Integer;
    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 se på et forenklet bilde av syntakstreet:

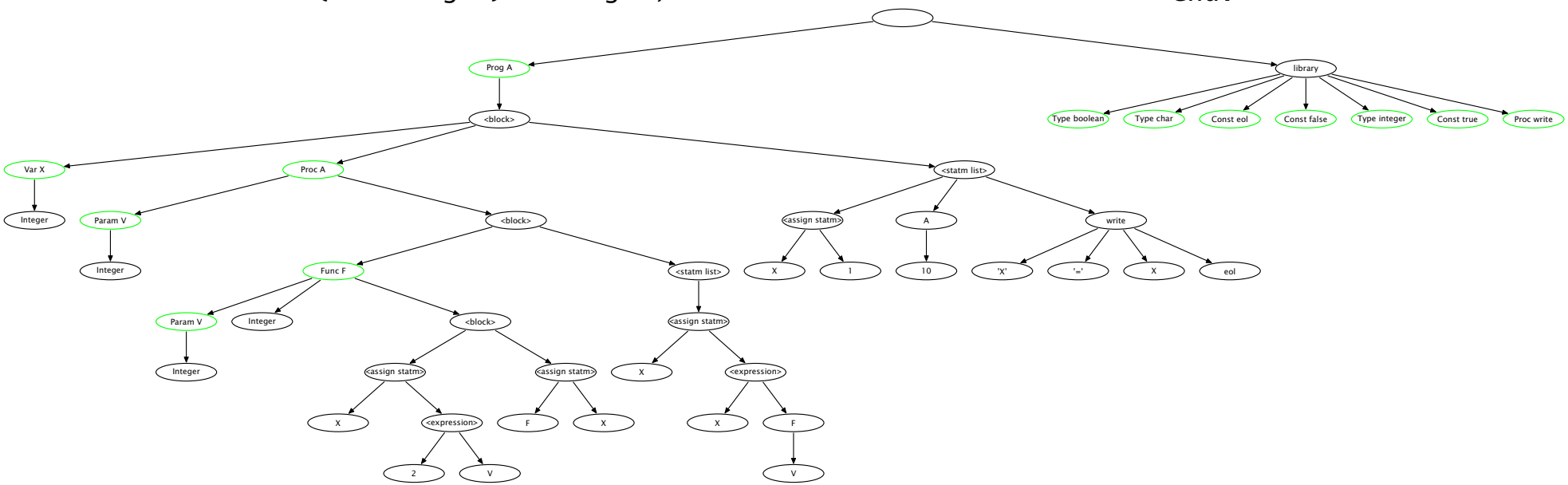
```
program A;
  var X: Integer;

  procedure A (V: Integer);
    function F (V: Integer): Integer;
    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.

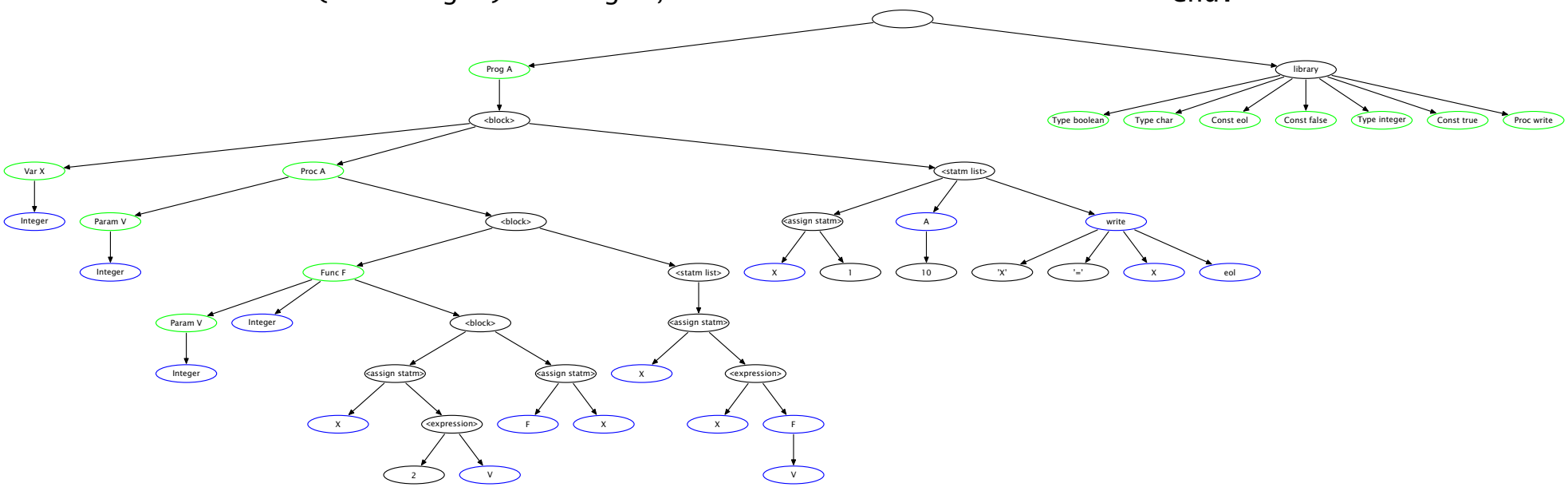
```
program A;
  var X: Integer;

  procedure A (V: Integer);
    function F (V: Integer): Integer;
    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?

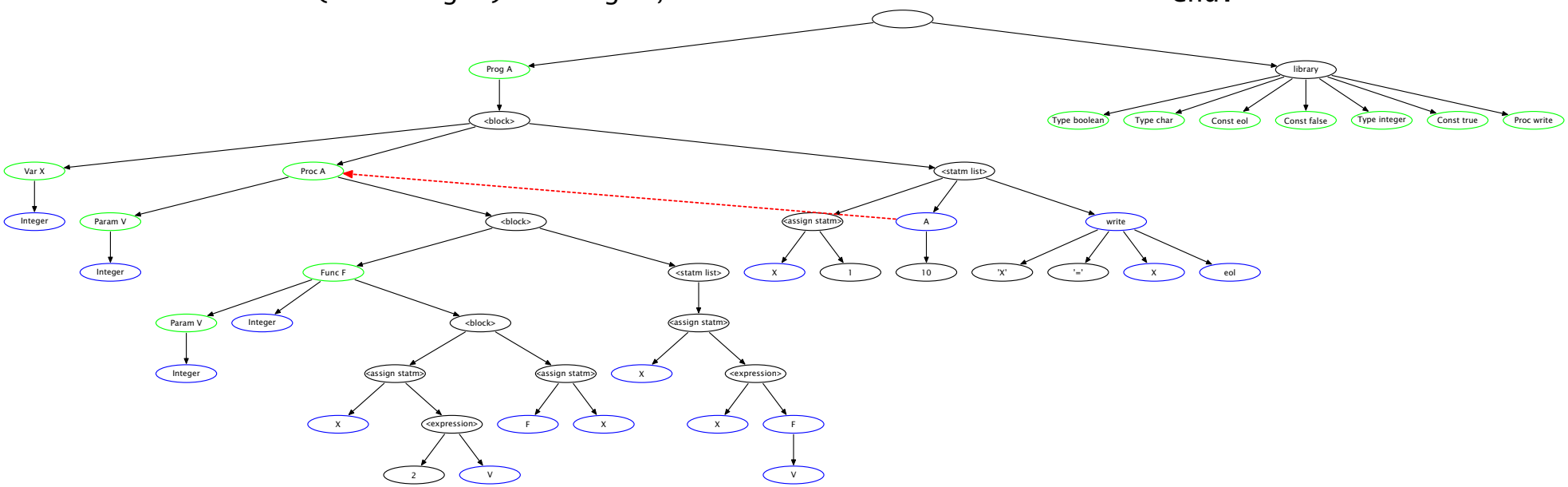
Blå noder er navneforekomster.

```
program A;
  var X: Integer;

  procedure A (V: Integer);
    function F (V: Integer): Integer;
    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.
```



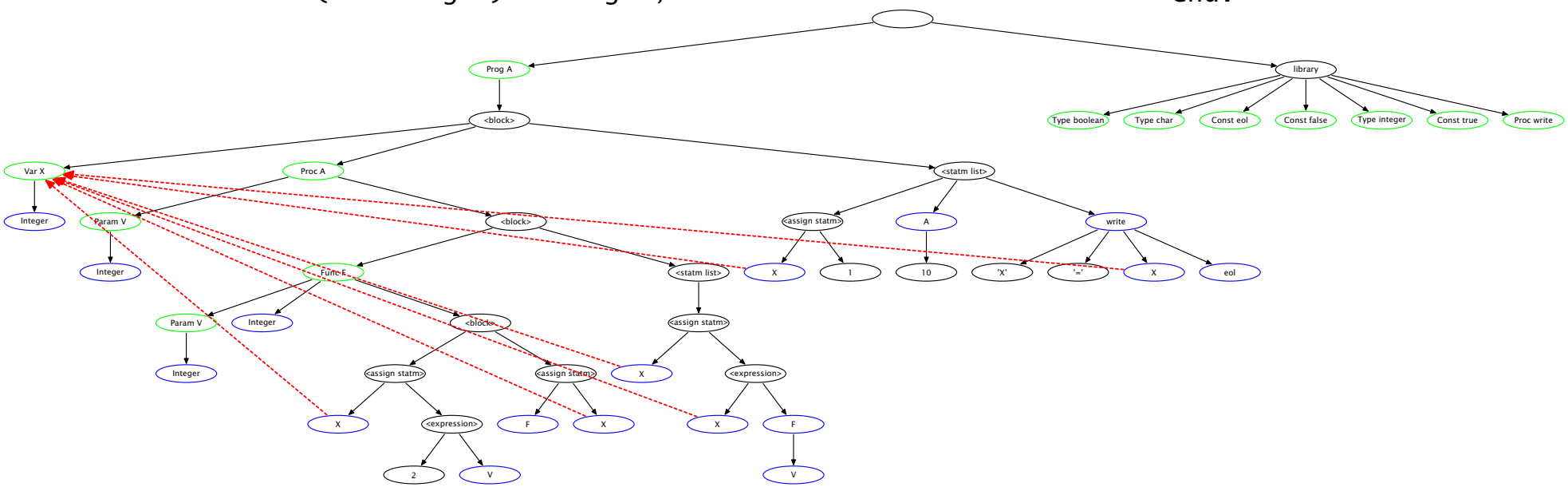
Forekomsten av a bindes slik:

```
program A;
  var X: Integer;

  procedure A (V: Integer);
    function F (V: Integer): Integer;
    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.
```



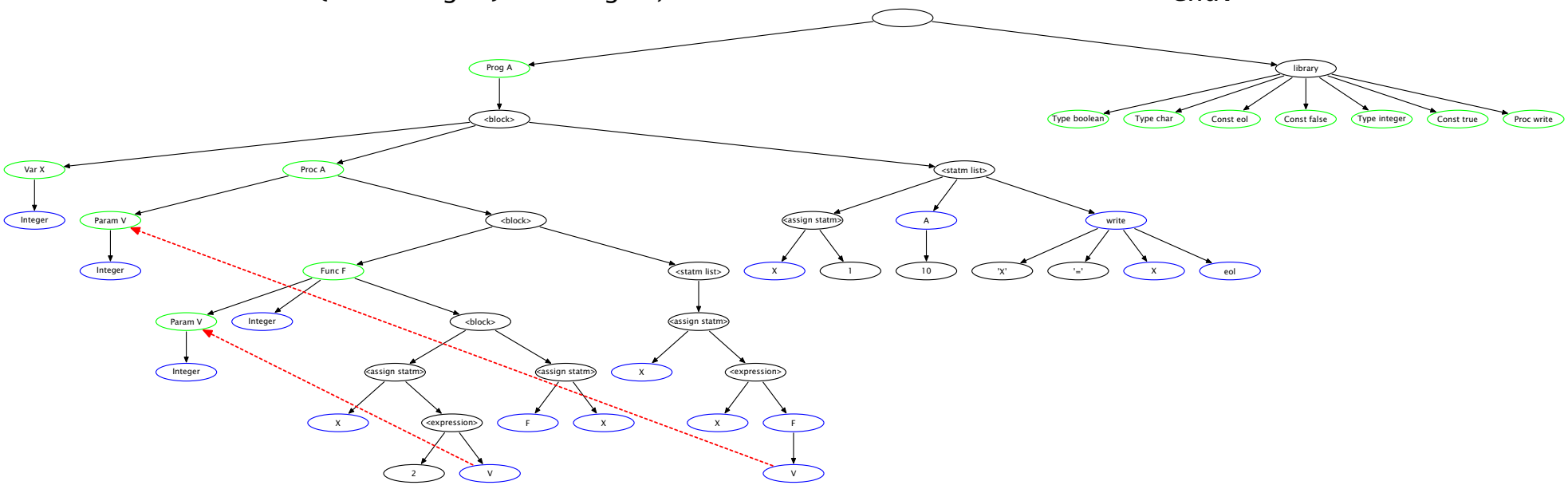
Navnet x bindes slik:

```
program A;
  var X: Integer;

  procedure A (V: Integer);
    function F (V: Integer): Integer;
    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.
```



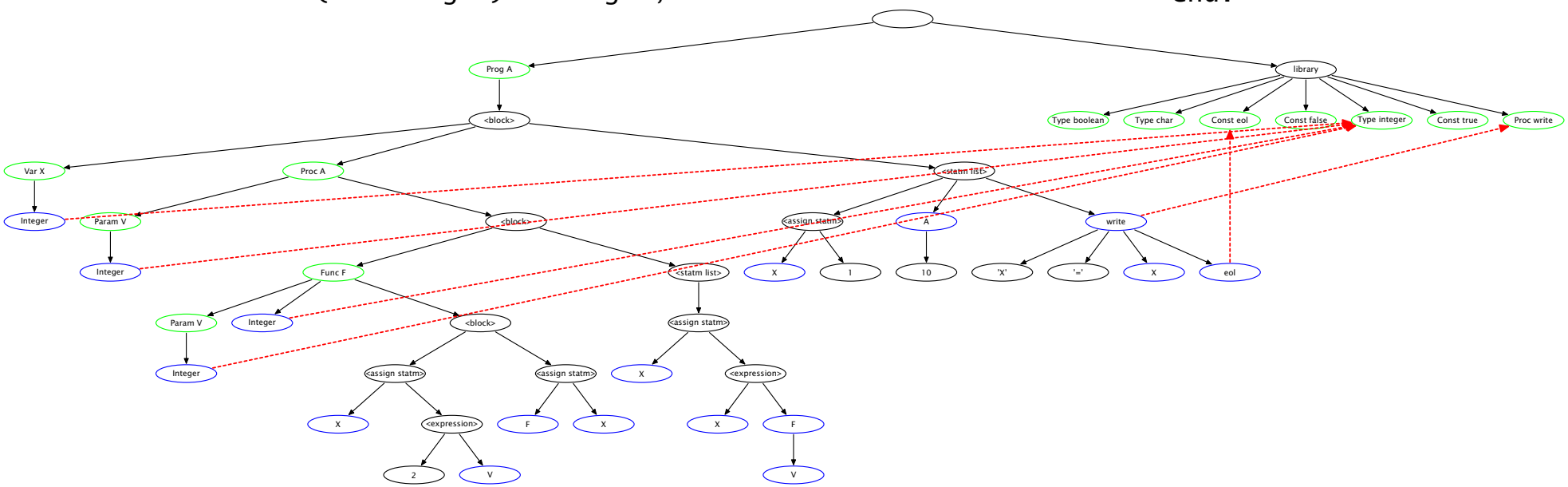
Navnet v må bindes til den riktige deklarasjonen:

```
program A;
  var X: Integer;

  procedure A (V: Integer);
    function F (V: Integer): Integer;
    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.
```



Predefinerte navn bindes til biblioteket:

```
program A;
  var X: Integer;

  procedure A (V: Integer);
    function F (V: Integer): Integer;
    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 vi foreta navnebindingen?

Alle deklarasjonene er i en ⟨block⟩:

- konstanter (`const pi = 3;`)
- variabler (`var teller: integer;`)
- funksjoner (`function f: char; ...`)
- prosedyrer (`procedure p; ...`)
- parametre

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.

Deler av `Block` kan se slik ut:

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

    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);
        }
        :
```

Leting etter navn

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

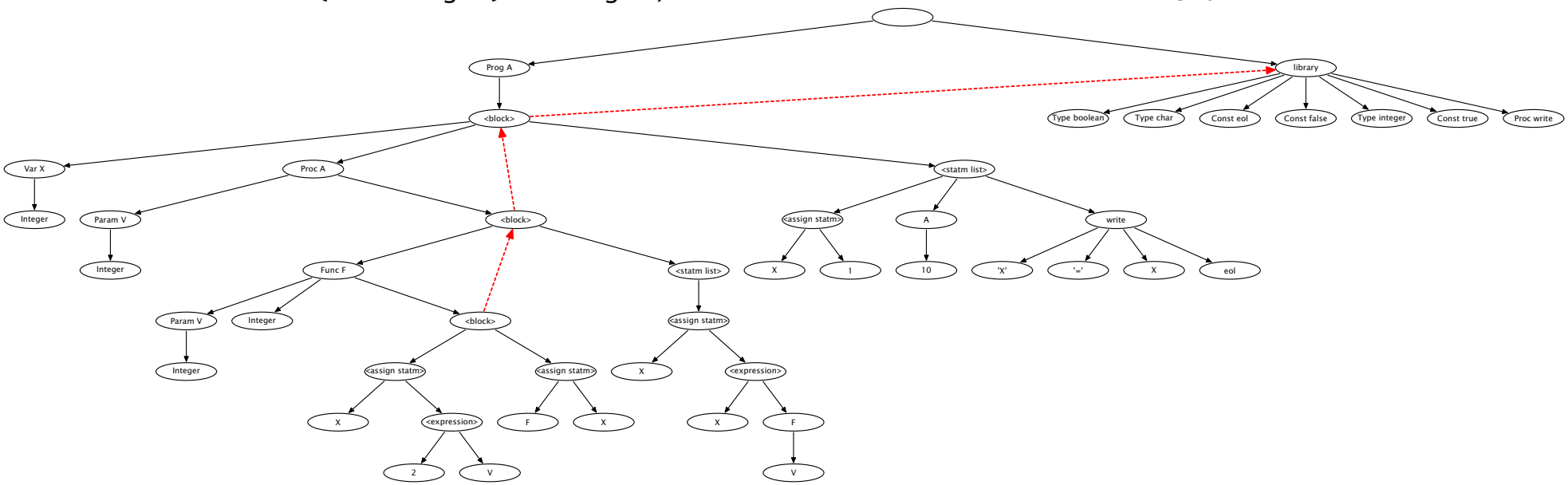
Eksempel

```
class ProcCallStatm extends Statement {
    String procName;
    ArrayList<Expression> actParams = new ArrayList<>();
    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);
    function F (V: Integer): Integer;
    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. Da er det greit å initiere den med innholdet av biblioteket.

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

```
Binding on line 2: integer was declared as <type decl> integer in the library
Binding on line 4: integer was declared as <type decl> integer in the library
Binding on line 5: integer was declared as <type decl> integer in the library
Binding on line 5: integer was declared as <type decl> integer in the library
Binding on line 7: x was declared as <var decl> x on line 2
Binding on line 7: v was declared as <param decl> v on line 5
Binding on line 8: f was declared as <func decl> f on line 5
Binding on line 8: x was declared as <var decl> x on line 2
Binding on line 12: x was declared as <var decl> x on line 2
Binding on line 12: x was declared as <var decl> x on line 2
Binding on line 12: f was declared as <func decl> f on line 5
Binding on line 12: v was declared as <param decl> v on line 4
Binding on line 16: x was declared as <var decl> x on line 2
Binding on line 17: a was declared as <proc decl> a on line 4
Binding on line 18: write was declared as <proc decl> write in the library
Binding on line 18: x was declared as <var decl> x on line 2
Binding on line 18: eol was declared as <const decl> eol in the library
```

Navnebruk

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

```
procedure P;
begin
  P := 17
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`)

`checkWhetherFunction` for funksjonskall

`checkWhetherProcedure` for prosedyrekall

`checkWhetherValue` for uttrykk

Typesjekking

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

```
var A: Integer;
    B: Char;
    C: array [1..'z'] 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.
I pakken `types` finnes det fem klasser:

`Type`	er en abstrakt superklasse
`ArrayType`	er for arrayer brukeren deklarerer
`BoolType`	er for standardtypen `Boolean`
`CharType`	er for standardtypen `Char`
`IntType`	er for standardtypen `Integer`

NB!

Nå har vi *to* klasser Type: **`parser.Type`** og **`types.Type`**.
Bruk pakkeprefikset når du mener den som ikke er i samme pakke.

Alt som kan ha type (dvs ⟨expression⟩, ⟨simple expr⟩ etc) definerer et element

```
types.Type type;
```

Her settes elementets type inn av den enkelte check-metode; her er vist for ⟨expression⟩.

```
@Override void check(Block curScope, Library lib) {
    leftOp.check(curScope, lib);
    type = leftOp.type;

    if (rightOp != null) {
        rightOp.check(curScope, lib);
        String oprName = relOpr.opr.kind.toString();
        type.checkType(rightOp.type, oprName+" operands", this,
            "Operands to "+oprName+" are of different type!");
        type = lib.booleanType;
    }
}
```

I klassen `types.Type` deklareres en metode

```
public void checkType(Type tx, String op, PascalSyntax where, String mess) {
    Main.log.noteTypeCheck(this, op, tx, where);
    if (this != tx)
        where.error(mess);
}
```

Parametrene er:

`tx` er typen som «vår» type skal sammenlignes med

`op` er en angivelse av hvordan typen brukes

`where` angir hvor i programmet typen forekommer

`message` inneholder meldingen som skal gis om det er typefeil

(For `types.ArrayType` er testen mer komplisert, så den er redefinert.)

Metoden `Main.log.noteTypeCheck` gir logging av typesjekkingen om vi angir opsjonen **`-logT`**.

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

```
Type check left * operand on line 7: type Integer vs type Integer
Type check right * operand on line 7: type Integer vs type Integer
Type check := on line 7: type Integer vs type Integer
Type check := on line 8: type Integer vs type Integer
Type check param #1 on line 12: type Integer vs type Integer
Type check left + operand on line 12: type Integer vs type Integer
Type check right + operand on line 12: type Integer vs type Integer
Type check := on line 12: type Integer vs type Integer
Type check := on line 16: type Integer vs type Integer
Type check param #1 on line 17: type Integer vs type Integer
```

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);
        expr.type.checkType(lib.booleanType, "while-test", this,
            "While-test is not Boolean.");
        body.check(curScope, lib);
    }
```

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

Konstantverdier kan regnes ut av kompilatoren

Konstanter

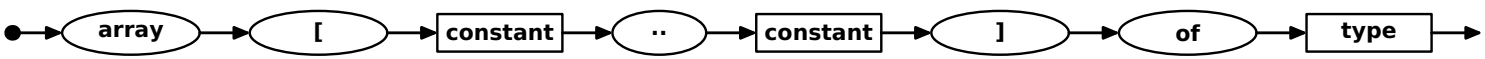
Kompilatoren må vite hva alle konstanter er allerede under kompileringen, for eksempel for å sette av plass.

```
var A: array [ 1 .. 10 ] of Boolean;
```

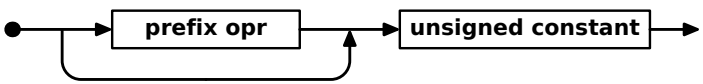
Noen ganger krever det litt beregning:

```
const size = 45;
var Ax: array [ -1 .. +size] of Char;
```

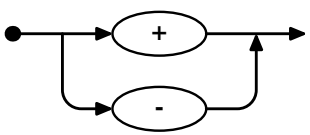
array-type



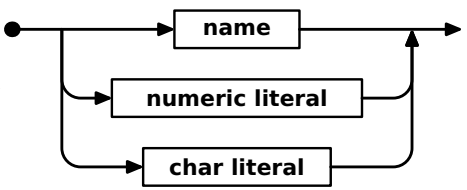
constant



prefix opr



unsigned constant



Konstantverdier kan regnes ut av kompilatoren

Alle elementer i syntakstreet som inneholder en konstant (dvs `CharLiteral`, `Constant`, `ConstDecl` etc), deklarerer en «`int constVal`» og denne kan beregnes av `check` som allikevel går gjennom hele treet.

```
class Constant extends PascalSyntax {
    PrefixOperator prefix = null;
    UnsignedConstant uConst;
    types.Type type;
    int constVal;


    @Override void check(Block curScope, Library lib) {
        uConst.check(curScope, lib);
        type = uConst.type;
        constVal = uConst.constVal;
        if (prefix != null) {
            String oprName = prefix.opr.kind.toString();
            uConst.type.checkType(lib.integerType, "Prefix "+oprName, this,
                "Prefix + or - may only be applied to Integers.");
            if (prefix.opr.kind == subtractToken)
                constVal = -constVal;
        }
    }
```