Dagens tema: Mer av det dere trenger til del 2

- Hvilke klasser trenger vi?
- Testutskrifter
- Versjonskontroll
- 12 gode råd

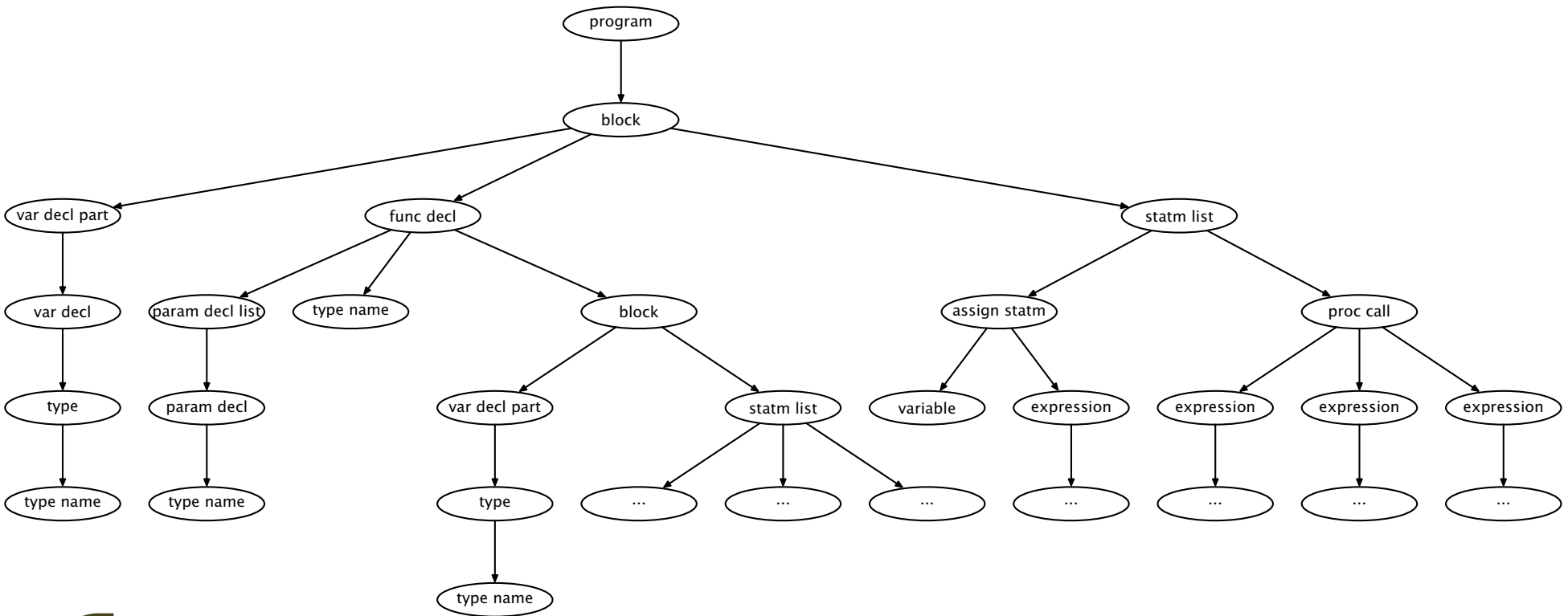
Prosjektet

Vi skal parsere (= syntaksanalysere) dette programmet:

```
program Power2;

var v: integer;

/* pow2: Computes biggest power of 2
         not bigger than x. */
function pow2 (x: integer): integer;
var p2: integer;
begin
   p2 := 1;
   while 2*p2 <= x do p2 := 2*p2;
   pow2 := p2
end;

begin
   v := pow2(1000);
   write('pow2(1000) = ', v, eol)
end.
```

Dagens tema

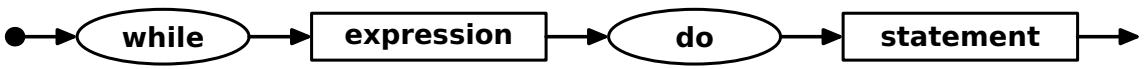
```
program Power2;

var v: integer;

/* pow2: Computes biggest power of 2
         not bigger than x. */
function pow2 (x: integer): integer;
var p2: integer;
begin
   p2 := 1;
   while 2*p2 <= x do p2 := 2*p2;
   pow2 := p2
end;

begin
   v := pow2(1000);
   write('pow2(1000) = ', v, eol)
end.
```



Koblingen mellom grammatikken og klasser

Normalt skal det være én klasse for hver ikke-terminal i grammatikken:

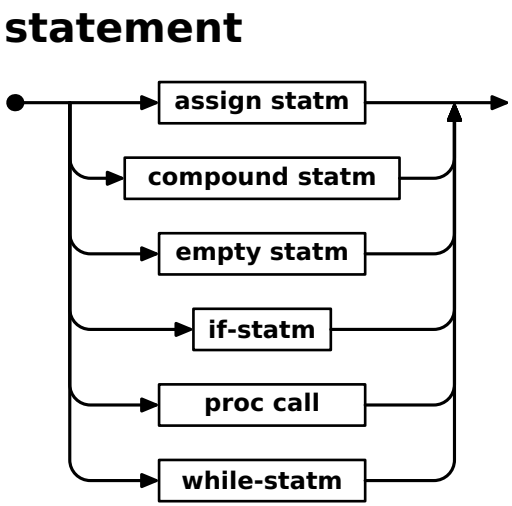
while-statm




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

Valgdefinisjoner

Hva med ikke-terminaler som bare er en samling alternativer?



Dårlig løsning

```
class Statement
        extends PascalSyntax {
    AssignStatm as = null;
    CompundStatm cs = null;
    EmptyStatm es = null;
    IfStatm is =null;
    ProcCall pc = null;
    WhileStatm ws = null;
```

God løsning

```
abstract class Statement
        extends PascalSyntax {
```

Hvor mange klasser trenger vi?

```
abstract class Statement extends PascalSyntax {
    Statement(int lNum) {
        super(lNum);
    }

    static Statement parse(Scanner s) {
        enterParser("statement");

        Statement st = null;
        switch (s.curToken.kind) {
        case beginToken:
            st = CompoundStatm.parse(s);  break;
        case ifToken:
            st = IfStatm.parse(s);  break;
        case nameToken:
            switch (s.nextToken.kind) {
            case assignToken:
            case leftBracketToken:
                st = AssignStatm.parse(s);  break;
            default:
                st = ProcCallStatm.parse(s);  break;
            } break;
        case whileToken:
            st = WhileStatm.parse(s);  break;
        default:
            st = EmptyStatm.parse(s);  break;
        }

        leaveParser("statement");
        return st;
    }
```

LL(1) eller LL(2)

Pascal2100 er ikke LL(1) siden f eks ⟨statement⟩ ikke kan avgjøres bare ved å se på første symbol.

Er så Pascal2100 LL(2)?

Svaret er: Ja, med ett unntak.

La oss se på setningen

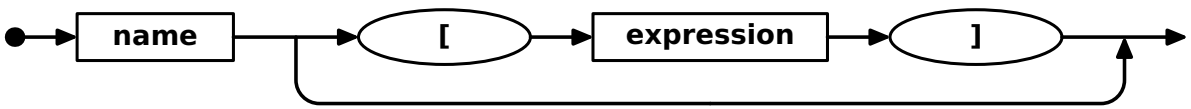
```
v := a
```

Det er en

assign statm

Venstresiden er grei:

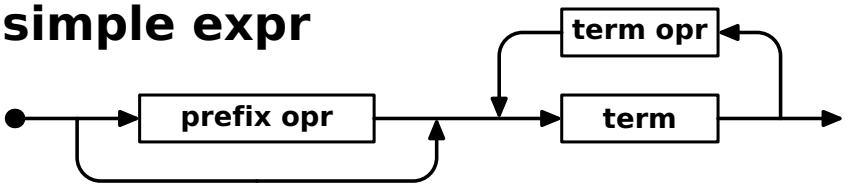
variable



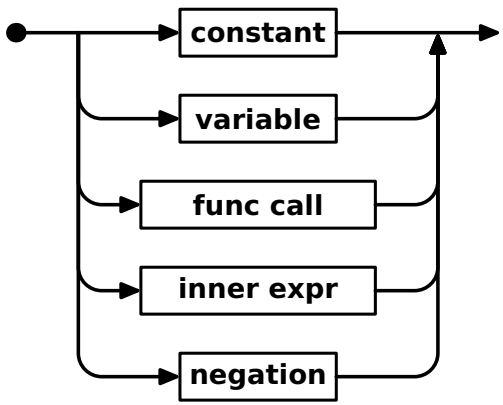
Hva så med høyresiden?

expression

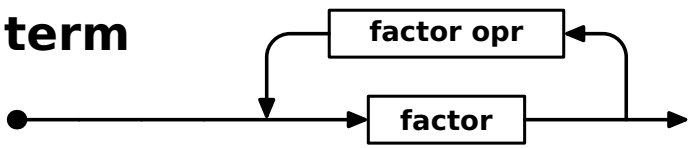
simple expr



factor

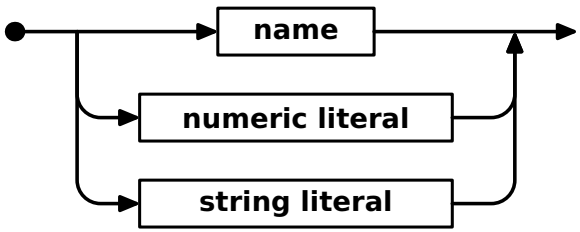


term

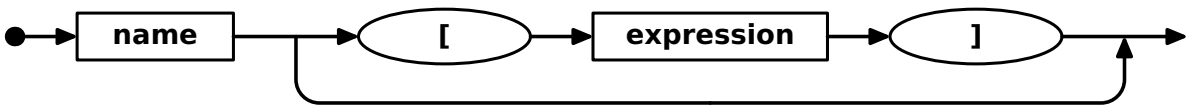


Hva slags ⟨factor⟩ kan **a** være?

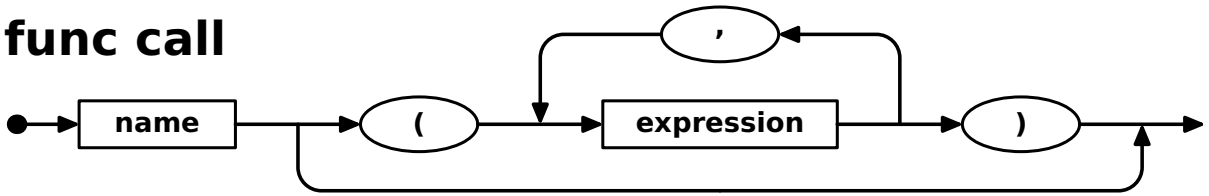
constant



variable



func call



inner expr

negation

Når **a** kan være enten ⟨constant⟩, ⟨variable⟩ eller ⟨func call⟩, hva gjør vi da?

1. Hvis vi har **a[**, vet vi at det er en ⟨variable⟩.
2. Hvis vi har **a(**, vet vi at vi har en ⟨func call⟩.
3. Ellers må vi bare anta at det er en ⟨variable⟩ og så ordne det senere i kompileringen når vi har koplet navn til sine deklarasjoner.

Testutskrifter

Det er lett å gjøre feil når man programmerer noe såpass komplisert som en kompilator. Det lureste er å godta dette og heller finne teknikker for å oppdage feilen.

`-logP` avslører om man gjør riktige valg i jernbanediagrammene. La hver `parse` gi lyd fra seg.

`-logY` sjekker om analysetreet ble riktig ved å skrive det ut etterpå.

`-testparser` slår på begge disse to (og stopper etter parseringen).

Vårt testprogram

```
program Power2;

var v: integer;

/* pow2: Computes biggest power of 2
         not bigger than x. */
function pow2 (x: integer): integer;
var p2: integer;
begin
   p2 := 1;
   while 2*p2 <= x do p2 := 2*p2;
   pow2 := p2
end;

begin
   v := pow2(1000);
   write('pow2(1000) = ', v, eol)
end.
```

```
   1: program Power2;
Parser:   <program>
   2:
   3: var v: integer;
Parser:     <block>
Parser:       <var decl part>
Parser:         <var decl>
Parser:           <type>
Parser:             <type name>
   4:
   5: /* pow2: Computes biggest power of 2
   6:           not bigger than x. */
   7: function pow2 (x: integer): integer;
Parser:             </type name>
Parser:           </type>
Parser:         </var decl>
Parser:       </var decl part>
Parser:       <func decl>
Parser:         <param decl list>
Parser:           <param decl>
Parser:             <type name>
Parser:             </type name>
Parser:           </param decl>
Parser:         </param decl list>
Parser:           <type name>
   8: var p2: integer;
Parser:           </type name>
Parser:           <block>
Parser:             <var decl part>
Parser:               <var decl>
Parser:                 <type>
Parser:                   <type name>
   9: begin
Parser:                   </type name>
Parser:                 </type>
  10:    p2 := 1;
Parser:               </var decl>
Parser:             </var decl part>
Parser:             <statm list>
Parser:               <statement>
Parser:                 <assign statm>
Parser:                   <variable>
Parser:                   </variable>
Parser:                   <expression>
Parser:                     <simple expr>
Parser:                       <term>
Parser:                         <factor>
Parser:                           <constant>
Parser:                             <number liter
```

Implementasjon

Alle parse-metoder må kalle

```
enterParser("while-statm");
```

(eller tilsvarende) ved oppstart og

```
leaveParser("while-statm");
```

ved avslutning; disse er definert i klassen PascalSyntax.

I `main.LogFile` finnes den egentlige implementasjonen:

```
public void enterParser(String name) {
    if (doLogParser) {
        noteParserInfo(name);  ++parseLevel;
    }
}


public void leaveParser(String name) {
    if (doLogParser) {
        --parseLevel;  noteParserInfo("/"+name);
    }
}

private void noteParserInfo(String name) {
    String logLine = "Parser:   ";
    for (int i = 1;  i <= parseLevel;  ++i) logLine += "  ";
    writeLogLine(logLine + "<" + name + ">");
}
```

Korrekt parsering av treet sjekkes enkelt ved å regenerere det (såkalt «pretty-printing»):

Original

```
program Power2;

var v: integer;

/* pow2: Computes biggest power of 2
         not bigger than x. */
function pow2 (x: integer): integer;
var p2: integer;
begin
   p2 := 1;
   while 2*p2 <= x do p2 := 2*p2;
   pow2 := p2
end;

begin
   v := pow2(1000);
   write('pow2(1000) = ', v, eol)
end.
```

Produsert av `-logY`

```
program power2;
var
  v: integer;

function pow2 (x: integer): integer;
var
  p2: integer;
begin
  p2 := 1;
  while 2 * p2 <= x do
    p2 := 2 * p2;
  pow2 := p2
end; {pow2}

begin
  v := pow2(1000);
  write('pow2(1000) = ', v, eol)
end.
```

Et eksempel

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


    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;
    }
```

Alle klasser som utvider `SyntaxUnit`, må redefinere den virtuelle metoden `prettyPrint`:

```
@Override void prettyPrint() {
    Main.log.prettyPrint("while ");  expr.prettyPrint();
    Main.log.prettyPrintLn(" do");  Main.log.prettyIndent();
    body.prettyPrint();  Main.log.prettyOutdent();
}
```

I `main.LogFile` finnes noen nyttige metoder:

```
private String prettyLine = "";
private int prettyIndentation = 0;


public void prettyPrint(String s) {
    if (prettyLine.equals("")) {
        for (int i = 1;  i <= prettyIndentation;  i++)
            prettyLine += "  ";
    }
    prettyLine += s;
}

public void prettyPrintLn(String s) {
    prettyPrint(s);  prettyPrintLn();
}

public void prettyPrintLn() {
    writeLogLine(prettyLine);
    prettyLine = "";
}

public void prettyIndent() {
    prettyIndentation++;
}

public void prettyOutdent() {
    prettyIndentation--;
}
```

Prosjektet del 2

Dere skal

1. implementere en parser for Pascal2100 med en klasse for hver ikke-terminal,
2. skrive en egnet metode `parse` i hver av disse klassene slik at grammatikkfeil blir oppdaget og syntakstreet bygget samt
3. sørge for logging à la **`-logP`** og **`-logY`**.

Til hjelp finnes

~inf2100/oblig/test/ inneholder diverse testprogrammer.

~inf2100/oblig/feil/ inneholder programmer med feil.

Parseren din bør kunne håndtere disse programmene.

Når flere samarbeider

Når flere jobber sammen, kan man tråkke i beina på hverandre:

1. Per tar en kopi av en kildefil og begynner å rette på den.
2. Kari gjør det samme.
3. Kari blir første ferdig og kopierer filen tilbake.
4. Per blir ferdig og kopierer filen tilbake. Karis endringer går tapt.

Løsningen

Et *versjonskontrollsystem* er løsningen.

De fleste slike systemer er *utsjekkingssystemer* basert på *låsing*:

1. Per ber om og *sjekker ut* (dvs får en kopi av) filen og begynner å rette på den.
2. Kari ber om en kopi, men får den ikke fordi den er *låst*.

Først når Per er ferdig og *sjekker inn* filen, kan Kari få sin kopi.

Fordeler med et slikt utsjekkingssystem:

- Lettforståelig.
- Ganske sikkert.

  (Men hva om Per og Kari begge må rette i to filer hver? Da kan de starte med hver sin fil, men når de er ferdige med den første, finner de at den andre er sjekket ut.)

Ulemper:

- Kari bør kunne få en lese-kopi selv om Per jobber med filen. (Noen systemer tillater det, men ikke alle.)
- Hva om Per glemmer å legge tilbake filen?
- Det burde vært lov for Per og Kari å jobbe på ulike deler av filen samtidig.

Insjekkingssystemer

En bedre løsning er *innsjekkingssystemer*:

- Alle kan når som helst sjekke ut en kopi.
- Ved innsjekking kontrolleres filen mot andre innsjekkinger:
  - Hvis endringene som er gjort, ikke er i konflikt med andres endringer, *blandes* endringene med de tidligere.
  - Ved konflikt får brukeren beskjed om dette og må manuelt klare opp i sakene.

Et scenario

1. Per sjekker ut en kopi av en fil. Han begynner å gjøre endringer i slutten av filen.
2. Kari sjekker ut en kopi av den samme filen. Hun endrer bare i begynnelsen av filen.
3. Per sjekker inn sin kopi av filen.
4. Kari sjekker inn sin kopi, og systemet finner ut at de har jobbet på hver sin del. Innsjekkingen godtas.

Når man er alene

Selv om du jobber alene med et prosjekt, kan det være svært nyttig å bruke et versjonskontrollsystem:

- Man kan enkelt finne frem tidligere versjoner.
- Det kan hende man jobber på flere datamaskiner.

CVS og Subversion

Det mest kjente innsjekkingssystemet er **CVS** («Concurrent Versions System») laget i 1986 av *Dick Grune*. Det er spesielt mye brukt i Unix-miljøer.

For å bøte på noen svakheter i CVS laget firmaet *CollabNet* **Subversion** i 2000. Det ble en del av *Apache* i 2010.

Gratis implementasjoner finnes for alle vanlige operativsystemer; se http://subversion.apache.org/.

Nære og fjerne systemer

Subversion kan operere på to ulike måter:

- Alt skjer i det lokale filsystemet.
- Man kan starte en Subversion-tjener på en maskin og så sjekke inn og ut filer over nettet.

Vi skal gjøre det siste og bruke Ifis Subversion-tjener.

Opprette et *repository*

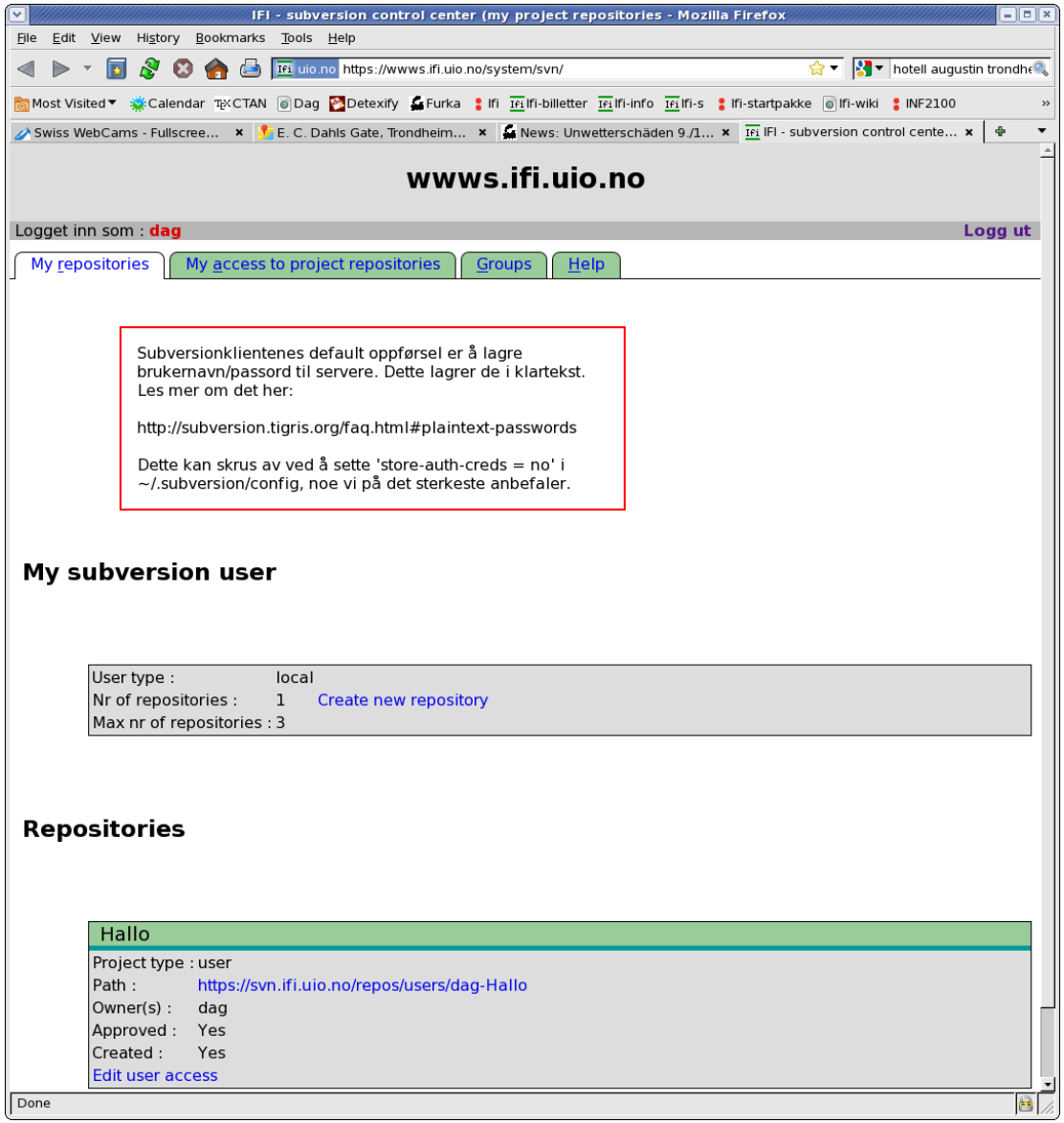
1. Gå inn på nettsiden

   https://wwws.ifi.uio.no/system/svn/

2. Logg inn.
3. Velg «My repositories» og «Create new repository». (I dette eksemplet heter det `Hallo`.)

   (Alle kan lage inntil tre «repositories».)
4. Hvis det er flere på prosjektet, velg «Edit user access».

Subversion


IFI - subversion control center (my project repositories - Mozilla Firefox
File Edit View History Bookmarks Tools Help
https://wwws.ifi.uio.no/system/svn/
Most Visited Calendar CTAN Dag Detexify Furka Ifi Ifi-billetter Ifi-info Ifi-s Ifi-startpakke Ifi-wiki INF2100
Swiss WebCams - Fullscree...
E. C. Dahls Gate, Trondheim...
News: Unwetterschäden 9./1...
IFI - subversion control cente...
wwws.ifi.uio.no
Logget inn som : dag
Logg ut
My repositories
My access to project repositories
Groups
Help
Subversionklientenes default oppførsel er å lagre brukernavn/passord til servere. Dette lagrer de i klartekst. Les mer om det her:
http://subversion.tigris.org/faq.html#plaintext-passwords
Dette kan skrus av ved å sette 'store-auth-creds = no' i ~/.subversion/config, noe vi på det sterkeste anbefaler.
My subversion user
User type : local
Nr of repositories : 1 Create new repository
Max nr of repositories : 3
Repositories
Hallo
Project type : user
Path : https://svn.ifi.uio.no/repos/users/dag-Hallo
Owner(s) : dag
Approved : Yes
Created : Yes
Edit user access
Done

Legge inn filer

Så kan vi legge inn mapper. La oss lage en *gren* med mappen `Hei` som inneholder filen `Hello.java`:

```
$ cd Hei
$ svn import https://sub.ifi.uio.no/repos/users/dag-Hallo -m "2100demo"
Adding         Hei/Hello.java

Committed revision 1.
```

Opsjonen `-m` gir en kort beskrivelse av denne grenen.

Sjekke ut filer

Nå kan vi (for eksempel fra en annen datamaskin) hente ut mappen vår:

```
$ svn co https://sub.ifi.uio.no/repos/users/dag-Hallo
A  dag-Hallo/Hello.java
Checked out revision 1.
$ ls -l
drwxr-xr-x    3 dag      ifi-a        4096 2011-11-13 06:46 dag-Hallo
$ ls -l -a dag-Hallo
total 16
drwxr-xr-x    3 dag      ifi-a        4096 2011-11-13 06:46 .
drwxr-xr-x    3 dag      ifi-a        4096 2011-11-13 06:46 ..
drwxr-xr-x    6 dag      ifi-a        4096 2011-11-13 06:46 .svn
-rw-r--r--    1 dag      ifi-a         500 2011-11-13 06:46 Hello.java
```

Sjekke inn filer

Etter at filen er endret, kan vi sjekke den inn igjen:

```
$ svn commit -m"Enklere kode"
Sending        Hello.java
Transmitting file data .
Committed revision 2.
```

Vi behøver ikke nevne hvilke filer som er endret — det finner Subversion ut selv. (Etter første utsjekking inneholder mappen skjulte opplysninger om repository-et, så det trenger vi ikke nevne mer.)

Andre nyttige kommandoer

`svn update .` henter inn eventuelle oppdateringer fra repository.

`svn info` viser informasjon om mappen vår:

```
$ svn info
Path: .
URL: https://sub.ifi.uio.no/repos/users/dag-Hallo
Repository Root: https://sub.ifi.uio.no/repos/users/dag-Hallo
Repository UUID: 8c927215-bc3e-0410-a56f-b2451114731f
Revision: 2
Node Kind: directory
Schedule: normal
Last Changed Author: dag
Last Changed Rev: 2
Last Changed Date: 2011-11-13 07:02:16 +0100 (Sun, 13 Nov 2011)
```

`svn diff` viser hvilke endringer som er gjort:

```
$ svn diff -r 1:2
Index: Hello.java
===================================================================
--- Hello.java  (revision 1)
+++ Hello.java  (revision 2)
@@ -7,10 +7,9 @@
     Properties prop = System.getProperties();
     String versjon = prop.getProperty("java.version"); // Versjonen
     String koding = prop.getProperty("file.encoding"); // Koding
-    String hei;
+    String hei = "Hallo";

-    hei = "Hallo";
-    hei = hei + ", alle sammen!";
+    hei += ", alle sammen!";
     System.out.println(hei);
     System.out.println("Dette er versjon " + versjon);
     System.out.println("Kodingen er " + koding);
```

Om man ikke vil ta i bruk Ifis Subversion-tjener, finnes det mange (ofte gratis) alternativer på nettet, som Bitbucket, GitHub og andre.

Husk bare at dette prosjektet ikke får ligge åpent noe sted.

Konklusjon

De få timene man bruker på å lære seg et versjonskontrollsystem, betaler seg raskt.

Råd nr 1: Forstå problemet!

Forstå hva du skal gjøre *før* du begynner å programmere.

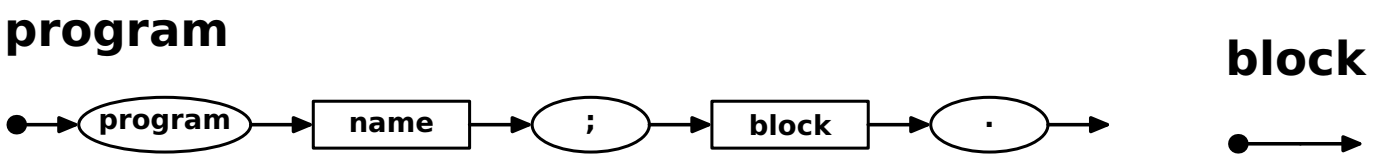
- Skriv noen korte kodesnutter i Pascal2100.
- Tegn syntakstrærne deres.
- Studér eksemplet med språket E i øvelsesoppgavene.

NB!

På dette stadium kan man samarbeide så mye man ønsker!

Råd nr 2: Start med noe enkelt!

Ingen bør forvente at de kan skrive all koden og så bare virker den. Start med et enkelt programmeringsspråk



og få det til å virke først. Utvid etter hvert. Sjekk hver utvidelse før du går videre.

Råd nr 3: Ikke sitt og stirr på koden!

Når programmet ikke virker:

1. Se på *siste versjon* av programkoden.
2. Siden du arbeider i små steg er feilen sannsynligvis i de siste linjene du endret.
3. Hvis du ikke har funnet feilen i løpet av fem minutter, gå over til *aktiv feilsøking*.

Råd nr 4: Les testutskriftene!

P-utskriftene forteller hvilke `parse`-metoder som kalles.

Anta at vi har programmet

```
program Test2;
function pow2 (x: integer): integer;
begin
   pow2 := 2*x+1
    :
```

Programmet gir en feilmelding i linje 2:

```
Expected a : but found a (!
```

Hva er galt?

Husk testutskriftene!

Svaret kan vi kanskje finne i **P**-utskriften:

```
   1: program Test2;
Parser:   <program>
   2: function pow2 (x: integer): integer;
Parser:     <block>
Parser:       <func decl>
Parser:         <type name>
```

Utskriften skulle kanskje ha startet

```
   1: program Test2;
Parser:   <program>
   2: function pow2 (x: integer): integer;
Parser:     <block>
Parser:       <func decl>
Parser:         <param decl list>
Parser:           <param decl>
Parser:             <type name>
Parser:             </type name>
Parser:           </param decl>
Parser:         </param decl list>
Parser:         <type name>
```

Husk testutskriftene!

Y-utskriften viser en «pen» utskrift av det genererte treet.
Anta at vi har det samme testprogrammet

```
program Test2;
function pow2 (x: integer): integer;
begin
   pow2 := 2*x+1
    :
```

Hvis **Y**-utskriften er

```
program Test2;
function pow2 (x: integer): integer;
begin
   pow2 := 2
    :
```

så vet vi at uttrykk ikke lagres riktig
(eller at det er feil i **Y**-utskriften ☺).

Råd nr 5: Lag egne testutskrifter

Her er feilaktig kode fra `Statement.parse`:

```
    static Statement parse(Scanner s) {
        enterParser("statement");

        Statement st = null;
        switch (s.curToken.kind) {
        case beginToken:
            st = CompoundStatm.parse(s);  break;
        case ifToken:
            st = IfStatm.parse(s);  break;
        case nameToken:
            switch (s.curToken.kind) {
            case assignToken:
            case leftBracketToken:
                st = AssignStatm.parse(s);  break;
            default:
                st = ProcCallStatm.parse(s);  break;
            } break;
        case whileToken:
            st = WhileStatm.parse(s);  break;
        default:
            st = EmptyStatm.parse(s);  break;
        }

        leaveParser("statement");
        return st;
    }
```

Anta at vi oppdager at `Assignstatm.parse` aldri blir kalt, selv om vi har en tilordningssetning.

Mitt råd er å legge inn noe à la

```
while (...) {
  System.out.println("Statement.parse: " +
    "curToken er " + s.curToken.identify() +
    " og nextToken er " + s.nextToken.identify());
```

før første `switch` så vi er sikre på hva de er.

Råd nr 6: Behold testutskriftene!

Når feilen er funnet, bør man la testutskriften forbli i koden. Man kan få bruk for den igjen.

Derimot bør man kunne slå den av eller på:

- Det mest avanserte er å bruke opsjoner på kommandolinjen:

```
java -jar pascal2100.jar -debugS testprog.pas
```

- Det fungerer også godt å benytte statusvariable:

```
static boolean debugS = true;
       :
if (debugS) {
  System.out.println("...");
}
```

Råd nr 7: Mistro din egen kode!

Det er altfor lett å stole på at ens egen kode andre steder er korrekt.

Løsningen er å legge inn *assertions* som bare sjekker at alt er som det skal være. Java støtter dette.

Stol ikke på det du har skrevet!

```
assert s.curToken.kind == whileToken:
  "While-setning starter ikke med 'while'!";
```

NB!

Husk å kjøre med

```
java -ea -jar pascal2100.jar ...
```

for å slå på mekanismen.

Råd nr 8: Sjekk spesielt på `Scanner.readNextToken()`!

Det er lett å kalle `readNextToken()` for ofte eller for sjelden.
Her er reglene som `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.

Vær spesielt oppmerksom der du har forgreninger og løkker i jernbanediagrammet.

Råd nr 9: Ta kopier daglig eller oftere!

Programmering er mye prøving og feiling. Noen ganger må man bare glemme alt man gjorde den siste timen.

Det finnes systemer for versjonskontroll som man bør lære seg før eller siden. En «fattigmannsversjon» er:

1. Ta en kopi av Java-filen hver gang du starter med å legge inn ny kode.
2. Ta uansett en kopi hver dag (om noe som helst er endret).

Råd nr 10: Fordel arbeidet!

Dere er to om jobben. Selv om begge må kjenne til hovedstrukturen, kan man fordele programmeringen.

Forslag

1. Én tar deklarasjoner og typer.
2. Én tar setninger og uttrykk.

Men . . .

- Snakk ofte sammen.
- Planlegg hvordan dere bruker filene så ikke den ene ødelegger det den andre har gjort.

Råd nr 11: Bruk hjelpemidlene

Spør gruppelærerne!
De er tilgjengelige under gruppetimene og svarer på e-post til andre tider.

Orakel

De siste ukene før oblig-fristene vil gruppelærerne bare jobbe med å svare på spørsmål.

Les kompendiet
Stoffet er forklart med flere detaljer enn det er mulig på forelesningene.

Råd nr 12: Start *nå*!

Det kan ta fra 20 til 100 timer å programmere del 2. Det er vanskelig å anslå dette nøyaktig på forhånd.

Påtrengende spørsmål

Det er 20 arbeidsdager til 21. oktober. Hvor mange timer per dag blir det?