Dagens tema: Resten av det dere trenger til del 2

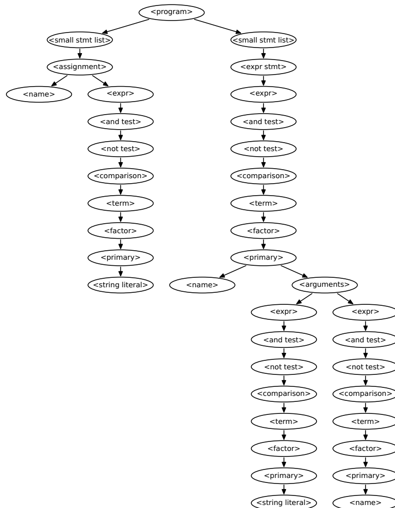
- Kort oppfriskning:
  - Hva er parsering?
  - Hvordan programmerer vi det?
- Testutskrifter
- 12 gode råd

Prosjektet

Vi skal parsere (= syntaksanalysere) programmer:

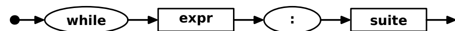
- Vi skal sjekke om det er korrekt (grammatikalsk).
- Vi skal bygge opp syntakstreet.

```
# En hyggelig hilsen
navn='Dag'
print ("Hei,",navn)
```



Dette programmeres ved å legge inn en parse-metode i hver klasse som representerer en ikke-terminal.

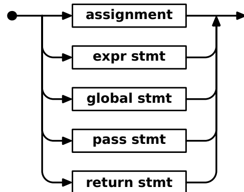
while stmt




```
class AspWhileStmt extends AspCompoundStmt {
    AspExpr test;
    AspSuite body;

    AspWhileStmt(int n) {
        super(n);
    }


    static AspWhileStmt parse(Scanner s) {
        AspWhileStmt aws = new AspWhileStmt(s.curLineNum());
        skip(s, whileToken);  aws.test = AspExpr.parse(s);
        skip(s, colonToken);  aws.body = AspSuite.parse(s);
        return aws;
    }
```

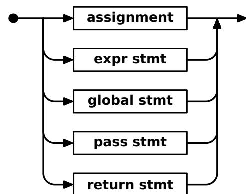
Veien er (nesten) alltid klar

Ved å se på nåværende symbol (`Scanner.curToken()`) kan vi avgjøre hvilken vei vi skal gå:

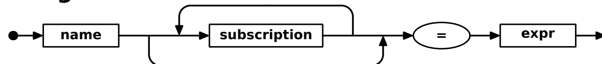


		Eksempel
`globalToken`	⟨global stmt⟩	`global a`
`passToken`	⟨pass stmt⟩	`pass`
`returnToken`	⟨return stmt⟩	`return x+1`

Men hva med **nameToken**? Hva hvis en **AspSmallStmt.parse** finner et nameToken?



Det er da to muligheter:



Begge kan begynne med et ⟨name⟩.

Løsning: Se etter = i setningen (metoden `Scanner.anyEqualToken`):

```
public boolean anyEqualToken() {
    for (Token t: curLineTokens) {
        if (t.kind == equalToken) return true;
        if (t.kind == semicolonToken) return false;
    }
    return false;
}
```

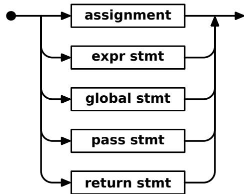
(Dette er én av grunnene til at skanneren leser en hel linje av gangen og finner alle Token-ene i den.)

Eksempel

- `a[4*x+1] = 17`
  er et ⟨assignment⟩.
- `print("Svaret er", x)`
  er et ⟨expr stmt⟩.

Dette er det eneste unntaket i Asp. I alle andre sammenhenger vet parseren hva den skal gjøre.

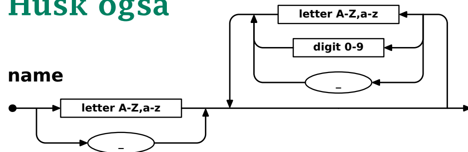
Veien er nesten alltid gitt!



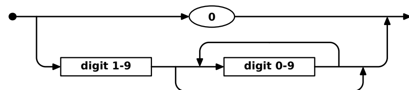

```
abstract class AspSmallStmt extends AspSyntax {
    AspSmallStmt(int n) {
        super(n);
    }
    static AspSmallStmt parse(Scanner s) {
        AspSmallStmt as = null;
        TokenKind cur = s.curToken().kind;
        if (cur == globalToken)
            as = AspGlobalStmt.parse(s);
        else if (cur == passToken)
            as = AspPassStmt.parse(s);
        else if (cur == returnToken)
            as = AspReturnStmt.parse(s);
        else if (cur == nameToken && s.anyEqualToken())
            as = AspAssignment.parse(s);
        else
            as = AspExprStmt.parse(s);

        return as;
    }
}
```

Litt av jobben er allerede gjort av skanneren

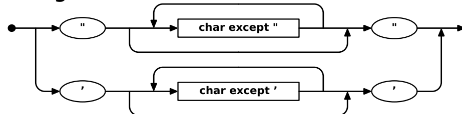
Husk også



integer literal



string literal



Disse ikke-terminalene er allerede tatt hånd om av skanneren og blitt et `nameToken`, `stringToken`, `integerToken` eller `floatToken`.

Testutskrifter

Det er lett å gjøre feil når man programmerer noe såpass komplisert som en parser. 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).

-logP

```
# En hyggelig hilsen
navn='Dag'
print ("Hei,",navn)
```

```
<program>
   1:
   2: # En hyggelig hilsen
   3: navn='Dag'
  <stmt>
    <small stmt list>
      <small stmt>
        <assignment>
          <name>
          </name>
          <expr>
            <and test>
              <not test>
                <comparison>
                  <term>
                    <factor>
                      <primary>
                        <atom>
                          <string literal>
                          </string literal>
                        </atom>
                      </primary>
                    </factor>
                  </term>
                </comparison>
              </not test>
            </and test>
          </expr>
        </assignment>
      </small stmt>
    </small stmt list>
  </stmt>
   4: print ("Hei,",navn)
  <stmt>
    <small stmt list>
      <small stmt>
        <expr stmt>
          <expr>
            <and test>
              <not test>
                <comparison>
                  <term>
                    <factor>
                      <primary>
                        <atom>
                          <name>
                          </name>
                        </atom>
                        <primary suffix>
```

Implementasjon

Alle parse-metoder må kalle

```
enterParser("and test");
```

(eller tilsvarende) ved oppstart og

```
leaveParser("and test");
```

ved avslutning.

Metodene er definert i `AspSyntax` (superklassen for alle parserobjektene):

```
protected static void enterParser(String nonTerm) {
    Main.log.enterParser(nonTerm);
}

protected static void leaveParser(String nonTerm) {
    Main.log.leaveParser(nonTerm);
}
```

I `main.LogFile` finnes metodene som foretar loggingen:

```
public class LogFile {
    private int parseLevel = 0;

    public void enterParser(String nonTerm) {
        writeParseInfo(nonTerm);
        ++parseLevel;
    }

    public void leaveParser(String nonTerm) {
        --parseLevel;
        writeParseInfo("/" + nonTerm);
    }

    private void writeParseInfo(String nonTerm) {
        if (! doLogParser) return;

        String indent = "";
        for (int i = 1;  i <= parseLevel;  ++i)
            indent += "  ";
        writeLogLine(indent + "<" + nonTerm + ">");
    }
```

En kilde til forvirring

Med `-logP` og `-testparser` kommer loggutskriftene fra to kilder:

1. skanneren, som leser linje for linje
2. parseren, når `parse`-metoder kalles og avsluttes.

Det betyr at utskriftene ikke alltid er synkronisert:

```
   8:      if word[i1] != word[i2]: return False
                 <stmt>
                   <compound stmt>
                     <if stmt>

                               :
                         </small stmt list>
                       </suite>
   9:      i1 = i1 + 1;  i2 = i2 - 1
                     </if stmt>
                   </compound stmt>
                 </stmt>
                 <stmt>
                   <small stmt list>
```

Dette må vi leve med.

Sjekke korrekt lagring av programmet

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

Original

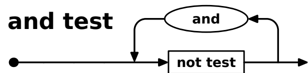
```
# En hyggelig hilsen
navn='Dag'
print ("Hei,",navn)
```

Produsert av -logY

```
navn = "Dag"
print("Hei,", navn)
```

Legg merke til at vi ikke får en identisk kopi av originalen.

Et komplett eksempel




```
class AspAndTest extends AspSyntax {
    ArrayList<AspNotTest> notTests = new ArrayList<>();
    static AspAndTest parse(Scanner s) {
        enterParser("and test");

        AspAndTest aat = new AspAndTest(s.curLineNum());
        while (true) {
            aat.notTests.add(AspNotTest.parse(s));
            if (s.curToken().kind != andToken) break;
            skip(s, andToken);
        }

        leaveParser("and test");
        return aat;
}
```

Som alle andre subklasser av AspSyntax, må AspAndTest redefinere den virtuelle metoden prettyPrint:

```
@Override
void prettyPrint() {
    int nPrinted = 0;

    for (AspNotTest ant: notTests) {
        if (nPrinted > 0)
            prettyWrite(" and ");
        ant.prettyPrint();  ++nPrinted;
    }
}
```

-logY

I `main.LogFile` finnes noen nyttige metoder:

```
public class LogFile {
    private String prettyLine = "";
    private int prettyIndentation = 0;
    public void prettyWrite(String s) {
        if (prettyLine.equals("")) {
            for (int i = 1;  i <= prettyIndentation;  i++)
                prettyLine += "  ";
        }
        prettyLine += s;
    }

    public void prettyWriteLn(String s) {
        prettyWrite(s);  prettyWriteLn();
    }

    public void prettyWriteLn() {
        if (doLogPrettyPrint)
            writeLogLine(prettyLine);
        prettyLine = "";
    }

    public void prettyIndent() {
        prettyIndentation++;
    }

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

Husk

Formålet med `prettyPrint` er å kunne sjekke at programmet er korrekt lagret i syntakstreet.

Det er derfor ikke så viktig om blanke, innrykk og linjeskift er nøyaktig som i referanseinterpreten. Det viktige er at alle elementene fra programmet er med.

Prosjektet del 2

1. Dere skal implementere en parser for Asp 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 og
3. sørge for logging à la **`-logP`** og **`-logY`**.

~inf2100/oblig/obligatorisk/ inneholder de fire programmene som må fungere.

Til hjelp finnes

~inf2100/oblig/test/ inneholder diverse testprogrammer.
~inf2100/oblig/feil/ inneholder programmer med feil.

Parseren din bør kunne håndtere disse programmene også.

Råd nr 1: Forstå problemet!

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

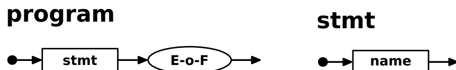
- Skriv noen korte kodesnutter i Asp.
- Tegn syntakstrærne deres.
- Studer eksemplet med språket E i øvelsesoppgavene.

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

```
def pow (x, p):
   if p == 0:
      :
```

Interpreten vår gir en feilmelding i linje 1:

```
Expected a name but found a ,!
```

Hva er galt?

Husk testutskriftene!

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

```
<program>
   1: def pow (x, p):
  <stmt>
    <func def>
      <name>
      </name>
      <name>
      </name>
Asp parser error on line 1:
    Expected a name but found a ,!
```

Utskriften skulle ha startet

```
<program>
   1: def pow (x, p):
  <stmt>
    <func def>
      <name>
      </name>
      <name>
      </name>
      <name>
      </name>
      <suite>
```

Husk testutskriftene!

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

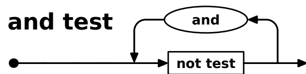
```
def pow (x, p):
   if p == 0:
      :
```

Hvis **Y**-utskriften er

```
def pow ():
   if p == 0:
      :
```

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

Råd nr 5: Lag egne testutskrifter



Her er feilaktig kode fra `AspAndTest.parse`:

```
static AspAndTest parse(Scanner s) {
    enterParser("and test");

    AspAndTest aat = new AspAndTest(s.curLineNum());
    while (true) {
        aat.notTests.add(AspNotTest.parse(s));
        if (s.curToken().kind != ampToken) break;
        skip(s, ampToken);
    }

    leaveParser("and test");
    return aat;
}
```

Anta at vi oppdager at `AspAndTest.parse` ikke fungerer med **and**.

Mitt råd er å legge inn noe à la

```
System.out.println("AspAndTest.parse: " +
  "curToken er " + s.curToken());
```

før og etter kallet på `AspNotTest.parse`.

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 asp.jar -debugS testprog.asp
```

- Det fungerer også godt å benytte statusvariabler:

```
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.

Én løsning 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 asp.jar ...
```

for å slå på mekanismen.

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

Det er lett å kalle disse to metodene for ofte eller for sjelden.

Her er reglene som `parse`-metodene *må* følge:

1. Når man kaller `parse`, skal `Scanner.curToken()` gi oss dets første symbol!
2. Når man returnerer fra en `parse`, skal `Scanner.curToken()` gi oss første symbol *etter* konstruksjonen!

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.

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.

Bruk nettforumet *Discourse*
Her kan du få svar både fra studenter og lærere.

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

Les prekoden
Det finnes en god del nyttige hjelpefunksjoner der.

Råd nr 12: Start *nå*!

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

Påtrengende spørsmål

Det er 15 arbeidsdager til 11. oktober. Hvor mange timer per dag blir det?