Obligatorisk oppgave 1

INF5110 - Kompilatorteknikk
Våren 2012
Frist mandag 19. mars

1. Hensikten med oppgaven

Tanken bak denne oppgaven er at man skal få litt praktisk erfaring med følgende:

- Bruke skannings- og parseringsverktøy, i dette tilfellet JFlex og CUP.
- Omarbeide en grammatikk fra én form (utvidet BNF) til en annen (BNF som passer verktøyet).
- Kunne kontrollere presedens og assosiativitet på to måter:
    1. Ved å lage en entydig grammatikk som gir riktig presedens/assosiativitet
    2. Gi en flertydig grammatikk og styre presedens og assosiativitet ved passelige kommandoer i CUP.
- Definere nodeklasser til et abstrakt syntaks-tre og bruke parserverktøyet til å bygge trær som representerer innleste programmer.
- I oblig 1 skal vi bygge opp det abstrakte syntakstreet og deretter skrive det ut med en entydig parentetisk form som angir presedens og assosiativitet.

2. Verktøy

Dere skal implementere kompilatoren i Java, ved hjelp av skannergeneratoren JFlex og parsergeneratoren CUP.

Dere skal bruke bruke Ant, Maven, GNU make eller lignende verktøy til bygging av kildekoden slik at hele innleveringen kan bygges og testes med noen få kommandoer. Prekoden har et oppsett dere kan utvide.

3. Oppgaven

Oppgaven går ut på å lage et program, ved hjelp av JFlex og CUP, som skal sjekke at programmer i språket Simpila er syntaktisk korrekte. Språket er definert i et eget notat. Merk at det i oblig 1 ikke er nødvendig å sjekke at programmer er semantisk korrekte. Man kan trygt ignorere alt som står i notatet om Simpila vedrørende sjekking av semantikk og kjøretidsforhold. Det blir først relevant for oblig 2.

4. Syntakstre

Under parseringen av språket skal det bygges opp et abstrakt syntakstre. Node-klasser må defineres for dette treet med en passelig subklasse-struktur. Treet bygges under parseringen ved hjelp av aksjonskode og returverdier fra produksjoner i en CUP-grammatikk. Ikke-terminaler i grammatikken trenger tilsvarende klasser i nodehierarkiet.

Objektorienterte språk egner seg godt til bygging av abstrakte parseringstrær. Det gir naturlige forhold mellom hierarkiet av noder. For eksempel vil et if-statement være en subklasse av et generelt statement, som typisk vil være en subtype av en generell nodeklasse.
Metoder i klassene kan da brukes ved traversering og bruk av treet.

Legg merke til at bare det *abstrakte* syntaks-treet skal lages og skrives ut. For produksjoner av typen ”stmt -> if_stmt” skal det altså ikke lages noen ny node.

5. Utskrift av syntakstre

Når man har bygget opp et abstrakt syntaks-tre, skal dette skrives ut i prefiksform. Hver node i det abstrakte treet skal representeres med ett par parenteser, og hvert "barn" av denne noden skal inngå i mellom disse parentesene. Rett etter første parentes kommer navnet på noden (som angitt i grammatikken), fulgt av attributtene (barna) til noden.

Eksempelfiler:

Canonical.d - input til kompilatoren (eksempel gitt lenger ned).

Canonical.ast - output fra kompilatoren (eksempel gitt lenger ned).

Formatering er valgfritt i utskriften, men det kreves en viss ryddighet og struktur.

6. To grammatikker

Oppgaven skal løses med to forskjellige grammatikker:

1. En grammatikk for Simpila skal ha en entydig syntaks uttrykt med BNF alene. Presedens og assosiativitet skal være innebygget i grammatikken.
2. Den andre skal være flertydig og skal ikke ha innebygget presedens og prioritet. Den resulterende grammatikken vil da trenge egne presedens og assosiativitetsregler definert i CUP for å håndtere konfliktene som oppstår. Dette vil vanligvis gjøre grammatikken kortere og mer konsis.

6.1 Sammenlikne de to parserne

Undersøk og karakteriser konfliktene i den opprinnelige grammatikken og forklar hvordan du løste dem. Undersøk også hvor mange tilstander CUP-automaten får for de to grammatikkene. Diskutér også om valg av grammatikk har noe å si for hvor greit det er å definere nodene og bygge syntakstreet.

Merk:
Det er tilstrekkelig å bygge og skrive ut syntakstreet fra én av grammatikkfilene. Den andre grammatikken trenger derfor ingen aksjonskode, bare CUP-grammatikk som kan generere et program som sjekker om et Simpila-program er syntaktisk riktig.

7. Leksikalsk analyse

Leksikalsk analyse skal utføres med JFlex, som leverer ett og ett token til parseren ved kall på metoden ”next_token();”. Hovedoppgavene til skanneren blir å gjenkjenne identifikatorer og konstanter, samt å hoppe over kommentarer, linjeskift og blanke tegn. Hvordan man benytter JFlex og CUP sammen er beskrevet under emnet *Working together - JFlex and CUP* i manualen til JFlex.

Det vesentlige ved integrasjonen er interfacet java_cup.runtime.Scanner som må implementeres av scanneren. JFlex implementerer dette interfacet automatisk når man bruker

%cup-switchen i JFlex. Ved levering av et token fra skanneren vil det være av typen Symbol og metoden Symbol.value benyttes for å levere tekster eller andre objekter fra scanneren til parseren.

8. Feilhåndtering

Dersom det oppdages en syntaktisk feil i programmet skal man bare stoppe analysen av Simpila-programmet, og melde at det er funnet en feil. Det er ikke krav om noen spesielt intelligent feilmelding (men det er morsomt om man kan få til noe her).

9. Frist

Fristen for levering er satt til mandag den 19. mars.

10. Gjennomføring og levering

Man kan arbeide alene eller parvis. Det som skal leveres er:

- En forside med navn og brukernavn til de som leverer besvarelsen.
- En kort rapport om gjennomføringen: forklaring av designet (Syntakstre, osv) og konfliktene i grammatikken og løsningene i de to grammatikkvariantene. Se beskrivelsen ovenfor.
- All kode som trengs for å bygge og kjøre parserne, herunder:
- JFlex-koden for skanneren
- CUP-koden for de to syntaksvariantene
- Java-klassene for syntakstreet
- Byggeskript: build.xml, pom.xml eller Makefile
- Instruksjoner for bygging og kjøring
- Utskrift fra kjøring med Canonical.d som input

10.1 Levering

Besvarelsen leveres til gruppelæreren på mail. Husk å legg alle data i en egen mappe før pakking. Hvis ønskelig kan også subversion brukes, da må dere legge til brukernavn ”magnushc” med leserettigheter, og sende url til repository i mail (mer om dette på verktøyspresentasjonen 01.03).

11. Ressurser

Det viktigste her er det som står i læreboka i kap 2.6 og 5.5 (men der refereres det til de tilsvarende verktøyene Lex og Yacc),

11.1 JFlex- og Cup-dokumentasjon

- JFlex – hovedsiden[1]
- JFlex - User's Manual[2]: Det nyttigste er kapitlet *Lexical Specifications*.
- CUP – hovedsiden[3]

1 http://jflex.de/
2 http://jflex.de/manual.html
3 http://www2.cs.tum.edu/projects/cup/

- CUP - User's Manual[4]: Det nyttigste er kapittel 2 *Specification Syntax*, men man får kanskje mest ut av eksemplene.

12. Canonical.d

```
class Complex {
   var float Real;
   var float Imag;
}

proc Swap(ref int a, ref int b) {
   var int tmp;
   tmp := a;
   a := b;
   b := tmp;
}

proc ret Complex Add(Complex a, Complex b) {
   var Complex retval;
   retval := new Complex;
   retval.Real := a.Real + b.Real;
   retval.Imag := a.Imag + b.Imag;
   return retval;
}

proc ret int Max(int a, int b) {
   if a > b then {
      return a;
   }
   return b;
}

proc Main() {
   proc ret float Square(float val) {
      return val ** 2.0;
   }
   var float num;
   num := 6.480740;
   print_float( num );
   print_str( " squared is " );
   print_float( Square( num ) );
   return;
}
```

13. Canonical.ast

```
(PROGRAM
   (CLASS_DECL (NAME Complex)
      (VAR_DECL (TYPE float) (NAME Real))
      (VAR_DECL (TYPE float) (NAME Imag))
   )
   (PROC_DECL (TYPE void) (NAME Swap)
      (PARAM_DECL ref (TYPE int) (NAME a))
      (PARAM_DECL ref (TYPE int) (NAME b))
```

4 http://www.cs.princeton.edu/~appel/modern/java/CUP/manual.html

```
   (VAR_DECL (TYPE int) (NAME tmp))

   (ASSIGN_STMT
      (NAME tmp)
      (NAME a)
   )
   (ASSIGN_STMT
      (NAME a)
      (NAME b)
   )
   (ASSIGN_STMT
      (NAME b)
      (NAME tmp)
   )
)
(PROC_DECL (TYPE Complex) (NAME Add)
   (PARAM_DECL (TYPE Complex) (NAME a))
   (PARAM_DECL (TYPE Complex) (NAME b))

   (VAR_DECL (TYPE Complex) (NAME retval))

   (ASSIGN_STMT
      (NAME retval)
      (NEW (TYPE Complex))
   )
   (ASSIGN_STMT
      ( . (NAME retval) (NAME Real))
      (ARIT_OP +
         ( . (NAME a) (NAME Real))
         ( . (NAME b) (NAME Real))
      )
   )
   (ASSIGN_STMT
      ( . (NAME retval) (NAME Imag))
      (ARIT_OP +
         ( . (NAME a) (NAME Imag))
         ( . (NAME b) (NAME Imag))
      )
   )
   (RETURN_STMT (NAME retval))
)
(PROC_DECL (TYPE int) (NAME Max)
   (PARAM_DECL (TYPE int) (NAME a))
   (PARAM_DECL (TYPE int) (NAME b))

   (IF_STMT (REL_OP >
      (NAME a)
      (NAME b)
   )
      (
         (RETURN_STMT (NAME a))
      )
   )
   (RETURN_STMT (NAME b))
)

(PROC_DECL (TYPE void) (NAME Main)

   (PROC_DECL (TYPE float) (NAME Square)
      (PARAM_DECL (TYPE float) (NAME val))
```

```
            (RETURN_STMT (ARIT_OP **
               (NAME val)
               (FLOAT_LITERAL 2.0)
            ))
         )
         (VAR_DECL (TYPE float) (NAME num))

         (ASSIGN_STMT
            (NAME num)
            (FLOAT_LITERAL 6.480740)
         )

         (CALL_STMT (NAME print_float)
            (ACTUAL_PARAM (NAME num))
         )
         (CALL_STMT (NAME print_str)
            (ACTUAL_PARAM (STRING_LITERAL " squared is "))
         )
         (CALL_STMT (NAME print_float)
            (ACTUAL_PARAM (CALL_STMT (NAME Square)
               (ACTUAL_PARAM (NAME num))
               ))
         )
         (RETURN_STMT)
      )
   )
```