Oblig 2 - Simpila

INF5110 - Kompilatorteknikk

Våren 2012

Dette er den andre av to oppgaver våren 2012. Den bygger videre på det som er gjort i oblig 1.

1 Hensikten med oppgaven

Hensikten er å få enda mer praktisk erfaring med hva som gjøres i en kompilator, nemlig:

- Analyse av et programs statiske semantikk.
- Anvendelse av et abstrakt syntakstree.
- Typesjekking, sjekking av bruk av navn og blokkstrtuktur i språk.
- Kodegenerering til en bytekode fra et abstrakt syntakstre og litt om interpretering av bytekode.

2 Oppgaven

Del to av den obligatoriske oppgaven bygger på den første delen, og går ut på å implementere kravene til statisk semantikk, beskrevet i språknotatet.[1] Resultatene fra kjøring av en testsuite skal leveres. I tillegg skal det også leveres en listing av bytekoden for eksempelprogrammet *RunMe.d*.

- Ta utgangspunkt i et abstrakt syntakstre og sjekke om treet oppfyller alle kravene gitt av språkets semantikk.
- Gjøre om på det treet som ble laget i oblig 1 om nødvendig og lage en oversiktlig kode som benytter abstrakte klasser og funksjoner.
- Dersom man finner feil, gi en kort melding, som skrives ut på skjermen.
- Returnere en verdi fra kompilatoren som forteller om koden var godkjent (0), om det var syntaksfeil (1) eller om det var semantikkfeil (2).

[1] `http://www.uio.no/studier/emner/matnat/ifi/INF5110/v12/obliger/Simpila-2012-2.pdf`

- Generere byte-kode ved hjelp av det utdelte biblioteket for byte-kode (Se pakken *bytecode.** og underpakkene og spesielt klassen *CodeFile*). Legg merke til at det er begrensninger i byte-koden, slik at det **ikke** er mulig å generere byte-kode for alle eksemplene i katalogen *test*. Les mer om det under Virtuell maskin og byte-kode.

3 Virtuell maskin og byte-kode

Den virtuelle maskinen og byte-kode er grundig beskrevet i notat om bytecode og interpreter.[2]

Det er laget en pakke med klasser for å lage byte-kode. For å lage byte-kode, opprettes et objekt av klassen `CodeFile`, hvor variabler, klasser, metoder m.m. kan legges til. Når alt er lagt til, kan man hente ut byte-koden med `getBytecode()` som lager en array med bytes (`byte[]`). F.eks. se på dette skallet:

```
CodeFile codeFile = new CodeFile();

// Her bygges bytekoden opp ...

byte[] bytecode = codeFile.getBytecode();
DataOutputStream stream = new DataOutputStream(
   new FileOutputStream ("Navnet på filen her ..."));
stream.write(bytecode);
stream.close();
```

Bytekoden er stack-basert og har ca. 30 instruksjoner.

Byte-koden er ikke like uttrykkskraftig som språket Simpila, derfor er reglene for programmene dere skal generere byte-kode for forskjellige fra de dere skal implementere i semantikksjekken. Forskjellen er:

- Det er ikke blokknivåer. Altså må alle klasser være deklarert på det øverste nivået og det er ikke prosedyrer inne i prosedyrer.
- Det er ikke referanseparametere, så basisparameterene overfører by-value og objektvariablene er pekerverdier og overføres også by-value. Altså, det er på samme måte som i Java.
- Det er også brukt litt andre navn:
  - Klasser kalles strukter (de inneholder kun data).

Et eksempel på et program som følger de begrensede reglene er `./code-examples/RunMe.d`.

Her er et kort eksempel på hvordan man bygger opp byte-koden til et enkelt program med en global variabel og en enkel prosedyre med to parametere (float og Complex) og en lokal variabel (int). Metoden printer ut float-parameteren og returnerer. Klassen Complex blir også definert. Legg spesielt merke til at alle deklarasjonene blir

[2] `http://www.uio.no/studier/emner/matnat/ifi/INF5110/v12/obliger/bytecode-interpreter.pdf`

definert først og oppdatert senere. Legg også merke til at parameterne får nummer fra 0 og oppover og at variabler inne i prosedyren blir nummerert etter det. Man må også fortelle den virtuelle maskinen hva som er main-prosedyren. Dette vil altså være kode som for eksempel er spredt rundt i det abstrakte syntakstreet, hvor hver node har ansvar for sine egne instruksjoner.

```
// Lage example.bin:
CodeFile codeFile = new CodeFile();
codeFile.addProcedure("Main");
codeFile.addVariable("myGlobalVar");
codeFile.addProcedure("test");
codeFile.addStruct("Complex");

CodeProcedure main = new CodeProcedure("Main", VoidType.TYPE, codeFile);
main.addInstruction(new RETURN());
codeFile.updateProcedure(main);

codeFile.updateVariable("myGlobalVar", new RefType(codeFile.structNumber(
    "Complex")));

CodeProcedure test = new CodeProcedure("test", VoidType.TYPE, codeFile);
test.addParameter("firstPar", FloatType.TYPE);
test.addParameter("secondPar", new RefType(test.structNumber("Complex")))
    ;
test.addInstruction(new LOADLOCAL(test.variableNumber("firstPar")));
test.addInstruction(new CALL(test.procedureNumber("print_float")));
test.addInstruction(new RETURN());
codeFile.updateProcedure(test);

CodeStruct complex = new CodeStruct("Complex");
complex.addVariable("Real", FloatType.TYPE);
complex.addVariable("Imag", FloatType.TYPE);
codeFile.updateStruct(complex);

codeFile.setMain("Main");

byte[] bytecode = codeFile.getBytecode();
// ... Lagre i filen ./code-examples/example.bin
```

Resultatet (listingen) av dette blir (Ved å kjøre biten med kode ovenfor og så kjøre komandoen `java runtime.VirtualMachine -l ./code-examples/example.bin`)

```
Loading from file: ./code-examples/example.bin
Variables:
0: var Complex myGlobalVar
Procedures:
0: func void Main()
    0: return
1: func void test(float 0, Complex 1)
    0: loadlocal 0
    1: call print_float {100}
    2: return
```

```
Structs:
0: Complex
    0: float
    1: float
Constants:
STARTWITH: Main
```

4 Testsuite

Det er laget en patch til det prosjektet som ble delt ut og som dere har bygd på i oblig 1. Den er tilgjengelig fra kurssidene[3] og består av følgende:

- En ny Compiler-klasse (`.\src\compiler\Compiler.java`). Den inneholder et skall som brukes av testen. Den forutsetter at metoden `compile()` returnerer en `int` 0, 1 eller 2, som nevnt tidligere.
- En hjelpeklasse til testen (`.\src\test\FileEndingFilter.java`).
- Klassen som utfører testen (`.\src\test\Tester.java`).
- En katalog med filer det testes mot (`./tests/`). Filene med navn som inneholder `fail` skal gi semantikkfeil (2). Ingen av filene skal gi syntaksfeil (1).
- Et testprogram for den virtuelle maskinen (`./code-examples/RunMe.d`).
- Noen linjer som kan legges til build-filen for å
  (1) kalle testen (*compile-test*, *test*).
  (2) kompilere og kjøre eksemplet RunMe (*compile-runme*, *list-runme*, *run-runme*).
  De ligger i filen `./build.xml.patch`.

Plasser filene slik at katalogene ligger i prosjektmappen deres (Pass på å **legge til** innholdet i `Compiler.java` og `build.xml` **uten å skrive over** de filene dere har).

Etter det kan testen kjøres med `ant test` og dere kan kompilere RunMe med `ant compile-runme` og liste ut byte-koden med `ant list-runme`.

Klassen Tester kaller klassen Compiler for alle testene i katalogen `./tests/`. Det skrives ut en linje for hver test, samt en oppsummering.

5 Sjekkliste for del to

Under følger en sjekkliste for semantikken (*merk* at det kan være flere krav, les også språknotatet) i Simpila:

- Multiple deklarasjoner av ett navn i samme skop må detekteres.

[3] `http://www.uio.no/studier/emner/matnat/ifi/INF5110/v10/obliger/INF5110-Oblig2-patch.tar.gz`

- Typekonvertering: int is-a float, int kan forfremmes til float i aritmetiske uttrykk, returverdier, og som aktuelt argumentuttrykk i funksjonskall.
- Main-prosedyren
  1. Prosedyredeklarasjonen til `Main`, må finnes på øverste blokknivå av programmet.
  2. Signaturen må matche `proc Main()`, d.v.s. ingen argumenter og ingen returverdi.
- Prosedyredeklarasjoners returverdi er:
  - ikke noe, eller
  - type i symboltabellen (predefinert eller brukerdefinert)
- Stmts
  - AssignStmt
    1. Variabelen på venstresiden må være deklarert.
    2. Typene på venstre side og høyre side må være like, etter evt. typekonvertering.
  - ReturnStmt: Typen til uttrykket skal stemme overens med prosedyrens deklarerte type. Merk også særtilfelle uten returverdi, da blir argumenter til `return`-setningen en feil.
  - WhileStmt: må ha en betingelse av typen `bool`.
  - IfStmt: må ha en betingelse av typen `bool`.
- Exps
  - NewExp: Navnet som instansieres må være en definert klasse.
  - Literals: Må være av en de forhåndsdefinerte typene i språket: float, int, string, bool eller null.
  - Binære uttrykk (felles for alle operatorer bortsett fra negasjon) må ha lik type, etter evt. typeforfremming, på begge sider av operatoren.
  - Aritmetiske uttrykk: som for binære uttrykk, men typene må også begrenses til aritmetiske (int eller float).
  - Logiske uttrykk (med "**&&**", "||" eller "**not**"): operandene må være av typen bool.
  - CallStmt
    1. Navnet som kalles må være deklarert som funksjon.
    2. Kallet må ha samme antall aktuelle parametre som formelle parametre.
    3. De aktuelle parametrene må ha samme type (eller mulig å tilordne) som de formelle parametrene.

4. Bruk av referanser (ref") må stemme overens med prosedyredeklarasjonen.

- Var
  1. Hvis objektvariabel, må varibelen være definert i klassen som objektet er instansiert fra.
  2. Navnet må være deklarert.

6 Gjennomføring og levering

Gruppene fra oblig 1 beholdes. Det som skal leveres er:

- En forside med navn og brukernavn til de som leverer besvarelsen.
- En kort rapport om gjennomføringen med forklaring av designet.
- All kode som trengs for å bygge og kjøre parserne, herunder:
  - JFlex-koden for skanneren
  - CUP-koden for en av grammatikkene
  - Java-koden for syntakstreet, semantikksjekking og byte-kode-generering
  - Byggeskript: build.xml eller Makefile
- Instruksjoner for bygging og kjøring.
- Utskrift fra kjøring med testesuiten (`ant test`).
- Utskrift av listing av byte-koden til eksemplet RunMe.d (`ant list-runme`).

6.1 Frist

Fristen er onsdag 2. mai.

6.2 Levering

Besvarelsen leveres som ett pakket filarkiv til gruppelærer Magnus Haugom Christensen magnushc@ifi.uio.no. Hvis ønskelig kan også subversion brukes, da må dere legge til brukernavn "magnushc"med leserettigheter, og sende url til repository i mail.

Sist oppdatert 27. mars 2012