

Obligatorisk Innlevering 2

INF5110 - Kompilatorteknikk

Våren 2017

Frist **07.05.2017 23:59**

Dette er den andre av to oppgaver våren 2017. Den bygger videre på det som er gjort i den første innleveringen.

Hensikten med oppgaven

Hensikten er å få enda mer praktisk erfaring med hva som gjøres i en kompilator og hvordan dette kan programmeres:

- Gjøre syntaktisk analyse og bygge et abstrakt syntakstre (oblig 1)
- Anvendelse av et abstrakt syntakstre
- Typesjekk, sjekking av navn og blokkstrukturer i programmer
- Kodegenerering til en egen bytekode (litt forenklet variant av Javas bytekode) fra et abstrakt syntakstre og litt om interpretering av bytekode

Oppgaven

Del to av den obligatoriske oppgaven bygger på den første delen, og går ut på å sjekke at kravene til statisk semantikk som er beskrevet i språknotatet [1] overholdes. Resultatene fra kjøring av en utlevert "test suite" (et sett av automatiske tester) skal leveres. I tillegg skal det også leveres en listing av bytekoden for programmet **RunMe.cmp**.

- Ta utgangspunkt i klassene dere lagde i innlevering 1 for å generere abstrakte syntakstrær. Dere kan gjøre forandringer hvis det trengs. Kode for semantisk sjekk og kodegenerering legges i metoder i disse klassene.
- Kompilatoren skal returnere en verdi som forteller om programmet ble godkjent (**0**), om det var syntaksfeil (**1**) eller om det var semantikkfeil (**2**).
- Dersom man finner feil under kompilering skal det gis en kort melding på skjerm om hva som gikk galt.
- Bytekode skal genereres ved bruk av et eget bibliotek (se pakken **bytecode.*** under **./src** og underpakkene [2]). Legg merke til at det er begrensninger i bytekoden (bl.a. mht. blokknivåer), slik at det ikke er mulig å generere bytekode for alle eksemplene i katalogen test (se lenger ned).

Virtuell maskin og bytekode

Den virtuelle maskinen og bytekode er grundig beskrevet i notat om bytekode og interpreter. Legg merke til at bibliotekprosedyrene ikke skal ha en bindestrek i navnene (i notatet er dette tilfellet). F.eks. er **printint()** riktig. Dessuten refereres det til funksjoner (**func**) mens vi forholder oss til prosedyrer (**proc**).

Det er laget en pakke med klasser for å lage bytekode. For å lage bytekode, opprettes et objekt av klassen **CodeFile**. Dette objektet benyttes for å legge til variabler, klasser, metoder og bytekodeinstruksjoner. Når alt er lagt til, kan man hente ut bytekoden med **getBytecode()** som lager en array med **bytes (byte[])**:

```
CodeFile codeFile = new CodeFile();

// Her bygges bytekoden opp ...
byte[] bytecode = codeFile.getBytecode();
DataOutputStream stream = new DataOutputStream( new FileOutputStream( "Navnet på
    fil for bytekode" ) );
stream.write( bytecode );
stream.close();
```

Bytekoden er stack-basert og har 35 instruksjoner. Bytekoden er ikke like uttrykkskraftig som språket **Compila17**, derfor er reglene for programmene dere skal generere bytekode for forskjellige fra de dere skal implementere i semantikksjekken. Forskjellene er:

- Det er ikke blokknivåer. Altså må alle klasser være deklartert på det ytterste nivået. Det er heller ikke prosedyrer inne i prosedyrer.
- Den støtter ikke referansetyper (ref og deref).
- Det er også brukt litt andre navn, og eksempler som ikke stemmer helt med beskrivelsen av **Compila17**:
 - Klasser kalles strukter, og deklarerer med nøkkelordet "struct".
 - Typen er angitt før variabler og parametre, i stedet for etter som i **Compila17**.
 - Prosedyrer kalles i notat funksjoner, og deklarerer der med nøkkelordet "func" i stedet for "proc" som vi bruker.

Merk at det ikke er noe krav at man sjekker at programmet som det skal genereres kode for følger disse reglene, man kan for enkelhets skyld bare anta at programmer som kommer som input til kodegenerator følger disse reglene.

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

Under følger et kort eksempel på hvordan man bygger opp bytekoden til et enkelt program med en global variabel og en enkel prosedyre med to parametere (**float** og **Complex**)

samt en lokal variabel (**int**). Metoden printer ut **float**-parameteren og returnerer.

Klassen **Complex** blir også definert. Legg spesielt merke til at alle deklarasjonene blir 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 videre oppover. Man må

også fortelle den virtuelle maskinen hva som er **main**-prosedyren. Legg merke til at bibliotekprosedyrer som benyttes må legges til (riktig nok uten innhold) - se hvordan **printfloat** har blitt lagt til.

```
// Kode for å lage example.bin:
CodeFile codeFile = new CodeFile();
codeFile.addProcedure( "printfloat" );
codeFile.addProcedure( "Main" );
codeFile.addVariable( "myGlobalVar" );
codeFile.addProcedure( "test" );
codeFile.addStruct( "Complex" );

CodeProcedure printFloat = new CodeProcedure( "printfloat", VoidType.TYPE,
    codeFile );
printFloat.addParameter( "f", FloatType.TYPE );
codeFile.updateProcedure( printFloat );

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( "printfloat" ) ) );
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 til filen ./code-examples/example.bin
```

Resultatet (listingen) av dette blir (ved å kjøre biten med kode ovenfor og så kjøre kommandoen

java runtime.VirtualMachine -1 ./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
```

Patch og test suite

Det er laget en patch til koden fra Oblig 1 som er tilgjengelig fra kurssidene [3] og den 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 **0**, **1** eller **2**, som beskrevet 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.cmp`).
- 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`).

Dette ligger i filen `./build.xml.patch`.

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

Etter det kan testene kjøres med `ant test` og dere kan kompilere `RunMe` med `ant`

`compile-runme` og liste ut bytekoden 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.

Sjekkliste for del to

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

Typekonvertering: `int` is-a `float`, `int` kan forfremmes til `float` i aritmetiske uttrykk, returverdier og som aktuelt argumentuttrykk i prosedyrekall

- 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
- Andre prosedyredeklarasjoners returverdi er:
 - ikke noe, eller
 - type i symboltabellen (predefinert eller brukerdefinert)
- Stmts
 - AssignStmt
 1. Variabelen på venstresiden må være deklareret
 2. Typene på venstre side og høyre side må være like (etter evt. typekonvertering)
 - ReturnStmt: Typen til uttrykket skal stemme overens med prosedyrens deklarererte type. Merk også særtilfellet 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`, eller `bool`.
 - Binære uttrykk (felles for alle operatorer bortsett fra negasjon) må ha lik type (evt. etter 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 deklarerert som prosedyre
 2. Kallet må ha samme antall aktuelle parametre som formelle parametre
 3. De aktuelle parametrene må ha samme type som (eller typer som er mulig å tilordne til samme type som) de formelle parametrene
 4. Bruk av referansetyper (**ref** og **deref**) må stemme overens med deklarasjonene
 - Var
 1. Hvis objektvariabel, må variabelen være definert i klassen som objektet
 - er instansiert fra
 2. Navnet må være deklarerert

Gjennomføring og levering

Gruppene fra innlevering 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 (altså hele din mappestruktur), herunder:
 - JFlex-koden for skanneren
 - CUP-koden for en av grammatikkene
 - Java-koden for å bygge syntakstrær, semantikksjekking og bytekodegenerering
 - Byggeskript
- Instruksjoner for bygging og kjøring
- Utskrift fra kjøring med test suiten (**ant test**)
- Utskrift av listing av bytekoden til eksemplet **RunMe.cmp** (**ant list-runme**)

Fristen for levering er satt til **07.05.2017 23:59**. Ved sykdom må dispensasjon fås fra administrasjonen, samt at den foreløpige versjonen av innleveringen må leveres hvis sykdomsperioden overskrider fristen. Se for øvrig: <http://www.mn.uio.no/ifi/studier/admin/obliger/oblig-retningslinjer.html>.

Krav for å få bestått:

- **RunMe.cmp** kompiles og kjøres korrekt.
- Minst 32 av de 42 testene evalueres riktig.

- Alle punktene under seksjonen Gjennomføring og Levering er tatt hensyn til.
- Løsningen kompilerer og kjører på en UiO-maskin (test dette!).

Besvarelsen skal levers via Devilry. Filnavnet skal være [deres_brukernavn]_inf5110_oblig2.

Referanser

- [1] <http://www.uio.no/studier/emner/matnat/ifi/INF5110/v17/languagespec/>
- [2] http://www.uio.no/studier/emner/matnat/ifi/INF5110/v17/obligns/inf5110-9110_bytecode.pdf
- [3] http://www.uio.no/studier/emner/matnat/ifi/INF5110/v17/obligns/INF5110-2017_oblig2_patch.zip