Kompilatorkonstruktion VT 14

Robin Engström, roen@kth.se
Casper Winsnes, cwinsnes@kth.se

Maj 2014

1 Introduktion

Den här rapporten beskriver hur vi under vårterminen 2014 skapade en kompilator i kursen Komp14. Kompilatorn är byggd för ett språk som i kursen kallades för Minijava[1], vilket på ett simplet sätt kan beskrivas som en delmängd av Java. Vissa tillägg kunde göras, främst för att få språket att mer likna riktiga Java. Det gick även att välja vilken (eller vilka) CPU backend (eller JVM) som man skulle kompilera mot. Vår kompilator kompilerar mot JVM med hjälp av Jasmin, som kan ses som ett assemblerspråk för JVM.

2 Metod

Vi började att skriva vår kompilator med hjälp av verktyget javacc, vilket är ett av de verktyg som rekommenderas av läroboken. Javacc var dock väldigt svåranvänt så vi letade efter andra system för grammatikutveckling och insåg snabbt att det fanns nyare verktyg för grammatikkonstruktion som är mycket bättre och mindre begränsande än javacc. De största begränsningarna upplevde vi fanns i kravet på en massa hjälpklasser för att beskriva varje enskild del av grammatiken så det kändes inte som det mest praktiska att använda för stora projekt.

Grunden i vår kompilator är istället uppbyggd med hjälp av verktyget antlr4[2]. Det är i antlr4 som vi har beskrivit vår grammatik och sedan låtit antlr4 själv skapa nödvändiga filer för lexer och parser. Antlr4 skapar ett snytaxträd åt oss och ger oss bra verktyg för att kunna stega igenom syntaxträdet.

Vi går igenom vårt syntaxträd 3 gånger och gör följande saker vid respektive genomgång:

1. Sparar alla klasser, metoder och variabler i datastrukturer samt kollar efter namnkrockar.
2. Typcheckar. Vi går igenom alla statementsöch "expressionsför att kolla så att alla typer stämmer överens med vad som förväntas.

[1] http://www.csc.kth.se/utbildning/kth/kurser/DD2488/komp14/project/grammar14v1b.pdf
[2] http://www.antlr.org/index.html

3. Skapar en jasminfil för varje klass och skriver jasminkoden.

Den största nackdelen med att ha använt antlr4 är att det inte finns ett uppenbart sätt att gå igenom hela syntaxträdet utan att göra massor med null-checkar överallt. Vår kod har därför blivit ganska svåröverskådlig med väldigt många ifsatser, och har man inte arbetat med den väldigt mycket med den är det kanske inte helt uppenbart vad alla metoder gör.

3 Kodöverblick

3.1 Kompilatorns viktigaste filer

- *javagrammar.g4* som beskriver hela grammatiken. Ska nya nyckelord läggas till eller prioriteringsordning för nyckelord ändras måste det göras i denna fil och kompileras med programmet antlr4. Filer som genereras av antlr4 (alla med namn på formen javagrammar.*
.java) bör ej röras.

- *JVMMain.java*, i vilken mainklassen kan hittas, strukturerar upp kompilatorn så att allt som ska göras görs i rätt ordning. Denna gör inget annat än att delegera arbetet till andra klasser.

- *ClassSymbol.java, MethodSymbol.java, VariableSymbol.java* beskriver de grundläggande egenskaperna hos klasser, metoder och variabler i ett minijavaprogram. Dessa används mest för att underlätta lagring av information mellan kompilatorsteg. Dessa kan utvecklas på för att undvika viss kodrepetition i andra delar av programmet.

- *SymbolRecorder.java* går igenom minijavaprogrammet och sparar alla klasser och deras egenskaper för att man senare ska kunna använda detta som referens, t.ex. om man behöver veta vad en viss metod returnerar för typ.

- *StatementValidator.java* går igenom minijavaprogrammet och kontrollerar att de uttryck som finns i programmet inte bryter mot de regler som sätts upp av minijava. Den viktigaste metoden i denna klass är getTypeFromExp som går igenom en exp för att ta reda på vilken typ som genereras av expen.

- *JasminTranslator.java* översätter den skriva minijavakoden till Jasmin "assembly". De viktigaste metoderna i denna klass är *handleStmt* som går igenom ett stmt (statement) och ser till att rätt kod genereras för att hantera detta och evaluateExp som gör liknande saker för en exp (expression).

3.2 Kompilatorns viktigaste metoder

- *StatementValidator/getTypeFromExp* går igenom ett expression, och kollar sedan rekursivt så att alla expressions i vårt expression går stämmer

överens med förväntad typ. Metoden returnerar sedan typen av vår expression i form av en *String*.

- *StatementValidator/getTypeFromId* hämtar en typ från ett id, så länge det id vi skickar med antingen är en lokal variabel i metoden vi för tillfället besöker eller ett fält i klassen som vi besöker. Returnerar en *String* som beskriver typen, t.ex. *int* eller *MyClass*.

- *StatementValidator/getVarFromId* gör samma sak som metoden ovan bara att den returnerar en *VariableSymbol* istället för en *String* som beskriver typen.

- *JasminTranslator/handleStmt* hanterar alla stmts som vi hittar i syntaxträdet. Att vi inte använder *enterStmt* i denna klass är för att vi kommer behöva lägga till labels och hoppa om vi kommer till en if eller while-sats. I de fallen så vill vi evaluera våra expressions i en viss ordning som inte kan uppnås om vi hanterar dem när de kommer. Om vi har att göra med en tilldelning så antar handleStmt att värdet vi vill tilldela en viss variabel ligger överst på stacken och lägger sedan till jasmininstruktioner för att ta det värdet och lägga i den lokala variabeln eller vektorn. Se vidare information om detta i *evaluateExp*.

- *JasminTranslator/evaluateExp* hanterar alla expressions som vi hittar i metoden *handleStmt*. Metodens främsta funktion är att skriva jasminkod så att vår expression evalueras och sedan lägga resultatet på stacken. Den returnerar också en *String* som beskriver vilken typ vi har att göra med då den informationen är nödvändig vid val av instruktion.

- *JasminTranslator/incStack* har kanske ett lite vilseledande namn eftersom att metoden både kan användas till att öka och minska värdet på stacken. Huvudfunktionen av denna metod är att förenkla uträknandet av det maximala stackdjupet som behövs i en viss funktion. Detta görs då vi adderar eller minskar värdet för stackdjupet vid varje tillägg av instruktioner, sedan har metoden koll på vad stackdjuet har vart som maximalt. Detta maximala stackdjup hämtas sedan i metoden *exitMethoddecl* som skriver ut vår stacklimit innan metoden stängs.

4 Buggar och misstag under utvecklingen

Vi hade inga riktigt intressanta buggar under utvecklingen av kompilatorn men felsökningen kring de fel vi hade var väldigt intressanta då vi många gånger fick gå igenom den genererade jasminkoden. Då det inte alltid var så lätt att återskapa felen utan att blanda in väldigt många delar av grammatiken var vi flera gånger tvugna att gå igenom den genererade jasminkoden och förstå var det brast i den för att kunna fixa problemen med kompilatorn.

De största misstagen vi gjorde var framförallt att vi började skriva utan att ha någon egentlig plan för att beräkna stackdjup eller hålla koll på de variabler

som finns i programmet. Vi kontrollerade inte heller i förtid hur Jasmin fungerar eller för den delen hur JVM fungerar i bakgrunden varför vi flera gånger blev väldigt förvånande över hur programmet faktiskt fungerar. Detta gjorde också att vi flera gånger behövt skriva om saker i onödan. I slutändan ledde också detta till att vi inte hann med att skriva kod för en annan backend än JVM så till nästa gång ser vi till att läsa dokumentationen mycket noggrannare innan vi börjar.

5 Hur man använder kompilatorn

Kompilatorn är uppdelad i två delar på sådant vis att ena delen skapar jasmin-filer som sedan kan göras till klassfiler med hjälp av programvaran Jasmin.

För att köra kompilatorn används kommandot

```
java -cp <pathtodir>/mjc.jar mjc.JVMMain <filename>
```

följt av

```
java -jar <pathtodir>/jasmin.jar *.j
```

vilket kommer generera de klassfiler som beskrivs i filen med namnet ¡filename¿.

6 Diskussion

6.1 Utbyggnader

Vi valde att ha följande utbyggnader av Minijava:

- If utan else
- Long
- Alla jämförelseoperationer

Anledningen till att vi valde de utbyggnaderna var för att vi i början av projektet ansåg att de skulle vara lätta att implementera. I efterhand så kan vi säga att så även var fallet till stor del.

If utan else var inga problem alls att implementera, man skulle nästan kunna påstå att If **med** else är svårare eftersom att det kräver två labels istället för en. Jämförelseoperationerna var inte heller särskilt svåra att implementera eftersom att det var så lika den jämförelseoperationen som var obligatorisk (<).

Long var däremot en klurigare utbyggnad eftersom att det krävde väldigt mycket extra kod. För alla jämförelseoperationer så behövde man ta hänsyn till att ett av jämförelsevärderna kunde vara en *int* och den andra en *long* vilket krävde konvertering av *int* till *long*. Dessutom så behövde vi ta hänsyn till att en *long* kräver två platser på stacken.

6.2 Antlr4

En stor fördel med antlr4 jämfört med javacc är att antlr4 hanterar vänsterrekusion automatiskt sålänge den inte är indirekt. Antlr4 skapar också en väldigt användbar klass kallad (javagrammar)BaseListener som möjliggör en DFS genom hela syntaxträdet. T.ex. så anropas metoden *enterClassdecl* om vi kommer till en klassdeklaration och då kan man bestämma precis vad man ska göra, som t.ex. att spara klassnamnet i en datastruktur. Efter *enterClassdecl* så går vi in och kollar variabeldeklarationer och metoder innan vi kommer till slutet av klassen och då anropas metoden *exitClassdecl* där vi kan välja att göra ytterligare saker som t.ex. att spara klasfilen.

Ytterligare en väldigt bra funktion med antlr är att vi får klasserna *grammatiknamn.grammatikregel**Context*** som skickas med till våra *enter* och *exit* metoder. Dessa kan ses som noder i vårt syntaxträd och de går att använda för att få fram allt från de underliggande noderna till radnummer i .java-filen som de förekommer i.