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 riktig 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.

Denna rapport ger en överblick över hur kompilatorn är uppbyggd och diskuterar våra val av lösningar under byggandets gång.

2 Metod

Grunden i vår kompilator är uppbyggd med hjälp av verktyget Antlr4[2] som kan ses som ett alternativ till JavaCC. Det är i Antlr4 som vi har beskrivit vår grammatik och sedan låter vi Antlr4 själv skapa nödvändiga filer för lexer och parser. Antlr4 skapar även syntaxträd och ger ett enkelt och strukturerat sätt att stega igenom syntaxträdet med hjälp av klassen *ParseTreeWalker*

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 uttryck för att kolla så att alla typer stämmer överens med vad som förväntas.
3. Skapar en jasminfil för varje klass och skriver jasminkoden.

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

Jasmin kräver att alla metoder ges ett maximalt antal platser på stacken, vilket skrivs ut i assemblerfilerna tillsammans med metoderna. Dessa räknade vi ut genom att helt enkelt hålla koll på vilket stackdjup vi teoretiskt skulle ha samtidigt som vi skriver ut assemblerinstruktionerna till filerna. När metoden är slut så hämtar vi det värdet som det teoretiska stackdjupet hade som mest. I fallet att vi hoppar i koden, (t.ex. vid en if-sats) så blir det automatiskt den vägen som kräver högst stackdjup som räknas.

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 Antlr4. Filer som genereras av Antlr4 (alla med namn på formen javagrammar.*.java) är genererade av Antlr4 och 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 ta in namnet på .java-filen som ska kompileras och delegerar sedan arbetet till andra klasser.

- *ClassSymbol.java, MethodSymbol.java, VariableSymbol.java* beskriver de grundläggande egenskaperna hos klasser, metoder och variabler i ett minijavaprogram. Dessa skapas under den första genomgången av syntaxträdet för att sedan användas vid den andra och tredje genomgången. En *ClassSymbol* innehåller datastrukturerer som lagrar fält (*VariableSymbol*) samt metoder (*MethodSymbol*). MethodSymbol innehåller i sin tur datastrukturer för parametrar till metoden (*VaraibleSymbol*) samt lokala variabler definerade i metoden (*VariableSymbol*)

- *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. Den kontrollerar även att det inte sker några namnkrockar i klassnamn, metodnamn, fältnamn och variabelnamn. Denna klass används i det första genomgången av syntaxträdet

- *StatementValidator.java* går igenom minijavaprogrammet och typcheckar alla uttryck. Den viktigaste metoden i denna klass är getTypeFromExp som går igenom ett uttryck för att ta reda på vilken typ som genereras av den. Denna klass används i den andra genomgången av syntaxträdet.

- *JasminTranslator.java* översätter den skriva minijavakoden till Jasmin "assembly". De viktigaste metoderna i denna klass är *handleStmt* som går igenom ett uttryck (Stmt) och ser till att rätt kod genereras för att

hantera dem, samt *evaluateExp* som gör samma sak för våra så kallade ”exps” eller ”expressions”. Denna klass används i den tredje och sista genomgången av syntaxträdet.

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 statements som vi hittar i syntaxträdet. Att vi inte använder *enterStmt* i denna klass är för att vi kommer behöva lägga till så kallade ”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 ökar eller minskar värdet för stackdjupet vid varje tillägg av instruktioner, sedan har vi 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

Det största misstaget vi gjorde under utveckligen var nog det vi gjorde direkt i början. Vi ansåg oss helt enkelt vara kungarna av KTH när vi hoppade in i projektet och började skriva en lexer och en parser helt själva utan att ha utforskat kurshemsidan allt för mycket. Efter några hundra rader kod så fick vi nys om att JavaCC fanns och dessutom rekommenderades av kursansvarig. Upprymda och glada som vi var över upptäkten så började vi direkt att definiera vår grammatik i JavaCC. Några dagar efter det så bytte vi igen, den gången blev det Antlr4[3]. Antlr4 löste också våra diskussioner om hurvida vi skulle ta den "hackiga" vägen och skriva i VIM eller orka sätta upp ett projekt i IntelliJ eftersom att det fanns ett fungerande plugin för Antlr4 till IntelliJ.

Förutom våra två omstarter i början av projektet så hade vi även en mindre omstart i typcheckinfasen. Vid det tillfället så hade vi inte något strukturerat sätt att lagra information om klasser, metoder och variabler på utan vi förlitade oss på simpla strängar i mängder av Hashmaps. Vår första (och för tillfället enda) genomgång av syntaxträdet blev istället till två stycken. Vi valde då att istället göra klasser för att hantera den sortens data samt att låta den första genomgången av syntaxträdet fokusera på att enbart läsa in data och skapa ovan nämnda klasser. Efter omstruktureringen så blev det (för en stund) lättare att hänga med i koden, men nu i efterhand så hade vi nog kunnat strukturera om det igen.

Angående intressanta buggar så hade vi nog inga. Vi hade däremot en mängd triviala buggar som för mesta delen uppkommit efter slarvfel. Den absolut sista buggen vi fixade hade att göra med att vi inte räknade med 2 platser på stacken efter ett metodanrop som returnerar en *long*. Detta är visserligen en trivial bugg men det tog väldigt många timmar av felsökande innan vi kom på att vi kunde skriva ut vårt teoretiska stackdjup som kommentarer i våra genererade jasminfiler. När detta väl var gjort så var det väldigt lätt att hitta fram till stället där det teoretiska stackdjupet stack ner och blev negativt.

5 Hur man använder kompilatorn

Kompilatorn är uppdelad i två delar på sådant vis att ena delen skapar jasminfiler 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>.

[3]Anledningen till att vi bytte till Antlr4 går att läsa i diskussionen.

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 man i stort sätt kunde tänka på samma sätt som för 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

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 mer användarvänliga. 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.

En stor fördel med Antlr4 jämfört med JavaCC är att Antlr4 hanterar vänsterrekusion automatiskt så länge den inte är indirekt. Antlr4 skapar också en väldigt användbar klass kallad (javagrammar)BaseListener som möjliggör en gång genom hela syntaxträdet. T.ex. så anropas metoden *enterClassdecl* om vi kommer till en klassdeklaration och då kan vi bestämma precis vad vi 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 Antlr4 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 det radnummer i .java-filen som de förekommer i.

Den största nackdelen med att ha använt Antlr4 är att det inte finns ett uppenbart sätt att gå igenom hela syntaxträdet på utan att göra massor med null-checkar överallt. Vår kod har därför blivit ganska svåröverskådlig med många if-satser och har man inte arbetat mycket med projektet så är det kanske inte helt uppenbart vad alla metoder gör.