DD2488 Kompilatorkonstruktion
Projektrapport

Joel Pettersson joelpet@kth.se

Linus Wallgren linuswa@kth.se

20 maj 2011

Inledning

En kompilator är ett omfattande program vars huvudsakliga uppgift är att översätta programspråk till körbar maskinkod. I den här rapporten presenteras hur vi gick tillväga för att konstruera vår egen kompilator för programspråket MiniJava; vi beskriver kompilatorns kodstruktur och diskuterar bland annat designval och dess konsekvenser samt problem som vi har stött på under arbetets gång.

Metod

Vi utgick ifrån specifikationen av MiniJava som ges i [1] med några små modifikationer av grammatiken. Vårt första steg var att välja ett verktyg för att generera en lexer och en parser, eftersom det är en relativt omständlig uppgift att skriva sådana själv från grunden, vilket dessutom är förknippat med en viss risk för ödesdigra misstag. Vi hade i praktiken att välja på JFlex (lexergenerator) kombinerat med Java CUP (parsergenerator) samt JavaCC där den senare kombinerar lexer- och parsergenereringen. Valet föll slutligen på det första alternativet då Java CUP är en LALR(1)-parser med vilken varje rimligt programmeringsspråk kan analyseras, enligt Appel [1], och således är vanligt förekommande i dessa sammanhang.

Därefter inledde vi arbetet med att skriva en lexerspecifikation för JFlex. Till slut fick vi ut en lexer som kompilerade, men som vi inte riktigt kunde sätta in i ett sammanhang och därmed heller inte kunde testa. Arbetet fortskred ändå genom att vi leta upp några exempel på Java CUP-specifikationer som kunde anpassas till MiniJava. När det var klart insåg vi hur kopplingen mellan lexern och parsern skulle se ut.

Nästa milstolpe bestod av att infoga semantisk åtgärdskod (eng. *semantic action code*) i parsern för att bygga ett abstrakt syntaxträd. Det syntaxträdet – uppbyggt av klasserna i `syntaxtree`-paketet – använde vi senare vid typkontrollen och därefter vid JVM-bytekodgenereringen. Typkontrollen krävde en hel del tillägg av kod, men var ändå relativt rättfram. Därefter återstod alltså själva kodgenereringen, vilken vi gjorde utgående ifrån syntaxträdet med vissa tillägg. Slutligen återstod endast att kontrollera att allting fungerade, vilket det naturligtvis inte gjorde på första försöket, men (förhoppningsvis) väl efter idoga testande och buggletande.

Resultat

I denna del beskriver vi den färdiga kompilatorns uppbyggnad och vilka designval vi har gjort.

Lexer (symbolanalys)

Lexern är den del av kompilatorn som styckar upp indata, det vill säga programkoden, till en ström av symboler (eng. *tokens*). Den lexer som används i vår kompilator är, som nämnts ovan, genererad av JFlex och specificeras i `src/lex/minijava.lex`. Det mesta i specifikationen är väldigt rättframt då det i stort sett endast handlar om att beskriva vilka reguljära uttryck som ska knytas till vilken symbol. Något som ändå vållade vissa bekymmer rörde hanteringen av negativa tal.

Lexern har inte tillräcklig kunskap för att besluta om ett minustecken framför ett tal är en del av symbolen eller om det är en egen symbol – operatorn för subtraktion. Det innebär att vi inte kan veta om ett tal är negativt eller positivt. Det för i sin tur med sig ett problem då det går att representera tal vars absolutvärde är större om talet är negativt än om det är positivt. Det vill säga även om -2^{31} går att representera med ett 32-bitars heltal går det inte att representera 2^{31}; talet kommer alltså bara vara giltigt om det är negativt, vilket vi konstaterat inte går att bedöma i lexern. Av denna anledning hanterar lexern tal som strängar, och överlämnar problemet med negativa tal till parsern.

Parser (syntaxanalys)

Syntaxanalysen behandlar de symboler som genererats av lexern och matchar dem mot grammatiken som består av en samling fördefinierade regler, eller produktioner (eng. *productions*). Produktionerna i sig representerar syntaxen för språket och kommer i slutändan motsvara noder i det syntaxträd som är grunden för resterande del av kompilatorn.

Java CUP är en LALR(1)-parser och är således relativt kraftfull jämfört med till exempel en LR(0)- eller LL(0)-parser, vilket i praktiken innebar att reglerna mer eller mindre direkt motsvarar grammatiken som vi utgick ifrån.

Besökarmönstret

För att undvika alltför många `instanceof` i koden använder vi oss av ett så kallat besökarmönster (eng. *visitor pattern*) då vi traverserar syntaxträ. Be-

sökarmönstret så som vi implementerar det resulterar mer eller mindre i en djupet-först-sökning i syntaxträdet. Sökningen initieras genom att rotnoden via `accept(Visitor v)` bjuder in besökaren som sedan direkt ges tillbaka kontrollen genom att den accepterande noden (rotnoden, i fallet med det initierande besöket) anropar `v.visit(this)`. I `Visitor`-klassen finns en `visit`-metod för vardera nodtyp. På det viset kan vi i varje nod utföra lämpliga uppgifter innan vi fortsätter nedåt genom att anropa `accept(Visitor v)` på den aktuella nodens barn. Exempelvis kan vi skriva ut hur syntaxträdet ser ut genom att låta en besökare i varje nod skriva ut namnet på noden.

Typgenomgång

För att generera en karta över vilka identifierare som existerar i koden användes också besökarmönstret. Genom att gå igenom koden och vid varje deklaration, såsom variabel klass och metoddeklarationer, sparade vi undan varje identifierare och dess typ. Datastrukturen som användes är i stort ett träd, där varje nod representerar ett omfång (eng. *scope*). Tanken är att varje omfång ska ha en koppling mellan identifieraren och typen den innehåller. Problemet som uppstår då är metoder, vilka inte riktigt har samma regler som vanliga variabler på grund av överlagring. Med anledning av det hanterar vi metoderna speciellt, medan vanliga identifierare, såsom variabler, sparas i en enkel `HashMap<String, Type>`.

Typ-genomgången ser till att inte samma identifierare deklareras flera gånger inom samma omfång, och även i vissa fall i underliggande nästlade omfång, såsom exempelvis inom ett nytt block.

Typkontroll

Med hjälp av typkopplingen ser vi till att alla tilldelningar och liknande har samma typ i högerled som i vänsterled. Även typkontrollen görs via besöksmönstret med skillnaden att varje steg i trädet returnerar en typen av den noden. På så vis kan man evaluera vilken typ ett komplext uttryck är på ett simpelt vis.

För att hitta alla typ-fel i koden antas hela tiden att varje operation är lyckad, vilket innebär att även om typ-kollen upptäcker ett fel någonstans, returnerar den som om det vore lyckat, för att kunna fortsätta typkontrollen. Det innebär exempelvis att om indata försöker multiplicera två strängar rapporterar vi felet men påstår ändå att multiplikationen resulterade i ett heltal, vilket är att förvänta av en multiplikation.

Kodgenerering

Kodgenereringen sker via besökarmönstret. I var nod genereras instruktioner som sparas i en lista. Varje instruktion innehåller information om vilken bytecode den skall generera samt dess förändring av storleken på operandstacken. Listan med instruktioner gör det enklare att gå igenom och beräkna hur stor gränsen på stacken för var metod skall vara, vilket beräknas efter att all kod genererats. När genereringen är färdig gås listan igenom och varje instruktion skrivs ut till rätt fil, motsvarande klassen instruktionerna befinner sig i.

I varje metod krävs det att alla lokala variabler har ett unikt id. Varje gång en variabel tilldelas slås ID för den identifieraren upp i en karta tillhörande typkopplingen. Ifall variabeln inte tidigare slagits upp genereras ett nytt ID. På så vis får vi en lat användning av variabler vilket innebär att variabeldeklarationerna i sig inte påverkar assemblerkoden vilket i sin tur innebär att icke tilldelade variabler aldrig blir inkluderade i programmet.

Likt identifierarna för de lokala variablerna behövs unika identifierare för alla etiketter (eng. *labels*), som används för hopp (eng. *branches*). En klass med statiska metoder håller koll på de senaste genererade etiketterna för att undvika krockar.

Diskussion

Ett av problemen vi stötte på var hur negativa tal skulle hanteras. Det naturliga är att låta lexern representera dem som tal och låta minustecknet vara en del av talet. Det är tvetydligt ifall minustecknet är en del av talet eller en operator. På grund av detta behövs antingen striktare regler i grammatiken eller en smartare parser som kan hantera den extra komplexiteten. Striktare regler innebär helt enkelt att man skiljer de två fallen åt, exempelvis genom att alltid definiera negativa tal med ett minustecken direkt följt av talet, medan operatorn har ett mellanrum. Likaså kan man göra parsern inteligentare genom att låta den hantera operationer utan explicit operator, utan att parsern får härleda operatorn ifall den högra operanden är negativ. I båda fallen uppstår det problem när man har tal vid gränsen av vad den bakomliggande datatypen kan hantera, iom att negativa och positiva tal har olika stora maximala absolutvärden.

Tack vare att kompilatorn är konstruerad för MiniJava har vi haft möjligheten att begränsa komplexiteten i kompilatorn eftersom språket är såpass begränsat. Detta har inneburit allt ifrån att det är få primitiva datatyper att implementera till att endast behöva implementera en tom standardkonstruktor samt att vi inte var tvungna att hantera klassers eller deras medlemmars

synlighet. Problemet med begränsningarna i MiniJava har å andra sidan inneburit att det är klurigt att skriva testfall, då många vanliga programkonstruktioner inte fungerar.

Eftersom vi i skrivande stund ännu inte har implementerat någon annan backend än JVM finns det heller inga IR-träd, flödesgrafer eller liknande implementerat. Det är nästa steg på vägen till att implementera fler backends. Dock har alla JVM-instruktioner extraherats till ett eget paket för att inte interferera med eventuella extra backends.

Sammanfattningsvis har vi genom olika redan tillgängliga verktyg, på förhand given kod och en hel del egna tillägg konstruerat en egen kompilator för det lilla programspråket MiniJava. Det har givit oss en bättre förståelse för hur programspråk i allmänhet är uppbyggda och kompilatorns viktiga uppgift. Framför allt har det här arbetet fått oss att inse vikten av god modularisering vid större projekt som detta. Eftersom vi redan från början leddes in på rätt spår lyckades vi, i vår mening, relativt väl med uppdelningen i olika moduler, så som lexer, parser och semantikanalyserare, även om paketstrukturen hade mått bra av ytterligare handpåläggning.

Referenser

[1] Appel, A.W. and Palsberg, J. (2002). *Modern compiler implementation in Java,*.