KTH Datavetenskap och kommunikation

Lärare

Kursledare och föreläsare är Mikael Goldmann. Han kan nås med e-post »migo+primp@nada.kth.se» eller på rum 1444 på institutionen. Telefonnumret dit är 790 68 13 och han har under kursens gång mottagning enligt överenskommelse.

Kurslitteratur och innehåll

Kursens mål är att ge teorin bakom och principerna för konstruktion och implementation av kompilatorer och andra översättare, samt detaljförståelse för mekanismerna i programspråk och hur en kompilator kan realisera dem.

Kursboken är *Andrew W. Appel: Modern Compiler Implementation in Java, second edition*. Köp inte den första utgåvan av boken! Kursen behandlar kapitel 6–12 i boken och förutsätter avslutad laborationsdel i kursen 2D1373, Artificiella språk och syntaxanalys, anno 2004 eller senare.

Registrering

Som vanligt behöver du dels se till att din studievägledning lägger in ditt kursval i Ladok och dels registrera dig i CSCs resultatrapporteringssystem res. Det senare gör du genom att ge kommandot `res checkin primp06` på någon av CSCs Unixdatorer.

Dessutom är det bra att ge kommandot `course join primp06` vilket dels ger dig login-meddelanden som är relevanta för kursen och dels ger dig en länk till kursens hemsida från din ehen webbsida med aktiva kurser. När du är klar med kursen ger du kommandot `course leave primp06` för att slipp ytterligare login-meddelanden och ta bort länken.

Examination och betygssättning

Kursen ger fyra poäng och innehåller två olika moment, laborationer respektive muntlig tentamen, på två poäng vardera. Genom att göra vissa laborationer (se nedan) aspirerar ni på ett visst betyg. Om ni blir godkänd på den muntliga tentamen fastställs det betyget. Observera att kapitel 6–12 ingår i kursen oavsett om ni gjort samtliga laborationer, och att kraven vid temtamen anpassas efter det betyg ni aspirerar på.

Det är naturligtvis viktigt att det klart och tydligt framgår vad ni har gjort i era lösningar. Ni ska därför skriva en kort rapport, en till två A4-sidor, som lämnas till kursledaren, antingen på papper eller som e-post. Rapporten ska innehålla en beskrivning av hur kursledaren ska göra för att kompilera och testköra programmet på 5–10 indata som visar att programmet fungerar som det ska, samt en övergripande redogörelse för hur ni har löst uppgiften. Lämpligen läggs en kopia av källkoden – som ska vara lättläst och välkommenterad – i en katalog som kursledaren får läs- och skrivrättigheter i.

Den skrivna rapporterna måste lämnas in senast två dagar innan ni går upp på muntlig tentamen.

Laborationer

Appel går igenom ett antal kompilatormoduler i sin bok; kapitel 1 är en uppvärmning och kapitel 2–11 innehåller en modul vardera. Kapitel 2–5 behandlas i kursen 2D1373, Artificiella språk och syntaxanalys. Det finns två huvudspår i kursen vad gäller laborationer; båda dessa spår beskrivs i detalj nedan. Den *lätta vägen*, som bara kan ge betyg 3 för KTH-studenter och betyg G för SU-studenter, innebär att generera klassfiler som kan köras på Javas virtuella maskin. Syftet med den lätta vägen

är att ge dem som satsar på betyg 3 möjlighet att konstruera en kompilator som faktiskt skapar körbara program. Den *rätta vägen*, som kan ge betyg 3–5 respektive G och VG, syftar till att generera ett färdigt körbart program för SPARC-arkitekturen eller någon annan »riktig» processor.

Det finns en modul som ger dig tillgång till användbara kommandon (javacc, jflex, javacup och jasmin) och som dessutom adderar modulen »gnu». Ladda modulen med kommandot `module add /info/primp06/module/modprimp`.

Det är tillåtet att arbeta i grupper om högst två personer.

Muntlig tentamen

Den muntliga tentamen går ut på att kursledaren ställer frågor på kapitel 6–12 i kursboken samt på era laborationslösningar. De kapitel i boken som svarar mot de laborationer ni har gjort ska ni vara väl förtrogna med. Förutom konkreta frågor om sina egna lösningar ska man alltså kunna svara även på sådant som rör given kod i de olika laborationerna, bakgrund och lösningsmetoder. Appel diskuterar normalt flera olika lösningsmetoder i varje kapitel. Endast en av dem används på laborationen, men ni ska naturligtvis kunna redogöra för hur alla metoderna fungerar. Vissa kapitel i boken har ett litet laborationsmoment; där är det naturligtvis större tyngd på de teoretiska bitarna.

Ni behöver bara översiktlig kunskap om de kapitel som svarar mot labbar ni ej gjort. Ni ska förstå vad den fas av kompileringsprocessen som beskrivs i kapitlen har för syfte och hur det går till i stora drag.

Man kan inte gå upp på den muntliga tentamen förrän man har lämnat in laborationsrapporterna.

Ordinarie tillfälle för muntlig tentamen är är januari 2006, tid och plats enligt överenskommelse. Det kommer dessutom att finnas möjlighet till restredovisning i början av period 4. Om ni inte tenterar vid något av dessa tillfällen så kontakta kursledaren. Detta är sista gången kursen ges.

Laborationsanvisningar

Laborationsmomentet i kursen 2D1373, Artificiella språk och syntaxanalys, behandlar från och med våren 2004 de första fem kapitlen i Appels bok. Laborationerna i den här kursen bygger vidare på dessa kapitel och förutsätter att man har en lösning till de programmeringsupgifter som beskrivs i dessa kapitel.

Den lätta vägen

Er laborationsuppgift – som givet godkänd muntlig tentamen ger betyg 3 för KTH-studenter respektive betyg G för SU-studenter – är att med utgångspunkt från er lösning till de första fem laborationerna konstruera en kompilator som genererar assemblerkod för Javas virtuella maskin (JVM). Denna assemblerkod assembleras sedan med verktyget *jasmin*, som finns installerat i kurskatalogen, till en körbar klassfil. För att åstadkomma detta måste ni först göra kapitel 6 i kursboken och sedan skriva en kodgenerator.

Om ni vill kan ni istället kompilera till Microsofts .NET-plattform, men kurskatalogen innehåller inga färdiga lösningar till kaitel 6 för denna plattform.

Aktiveringsposter

I kurskatalogen finns implementationer av det som Appel kallar Frame-objekt respektive Access-objekt för Javas virtuella maskin. Det finns också Record-objekt – motsvarigheten till Frame-objekt för klasser. Gränssnitten i dessa implementationer är inte samma som de som beskrivs i boken. Detta beror helt enkelt på att kodgenerering för Javas virtuella maskin kräver att dessa objekt innehåller mer information än de behöver innehålla för »vanlig» kodgenereing. Ni måste anpassa den semantiska analysen (kapitel 5) så att varje klass kopplas ihop med ett objekt som implementerar gränssnittet frame.VMRecord, varje metod kopplas ihop med ett objekt som implementerar frame.VMFrame och varje variabel kopplas ihop med ett objekt som implementerar frame.VMAccess. Lämpligen görs detta genom att modifiera symboltabellen. Variablerna i en klass ges VMAccess-objekt genom att metoden allocField() anropas i klassens VMRecord-objekt. På samma sätt ges parametrar och lokala variabler i metoder VMAccess-objekt genom att metoderna allocFormal() och allocLocal() anropas i metodens VMFrame-objekt. Beroende på hur ni har löst de första fem kapitlen i boken kan ni komma att behöva ändra koden i kurskatalogen i olika stor utsträckning.

Kodgenerering

När symboltabellen är modifierad är det förmodligen enklast att generera assemblerkod direkt från syntaxträdet genom att använda Visitor-mönstret.

Javas virtuella maskin är stackbaserad, vilket innebär att alla beräkningar sker på en stack. Till exempel adderas två tal genom att först lägga dem på stacken och sedan ut-

föra en instruktion som lägger ihop de två översta talen på stacken, dvs tar bort de två översta talen på stacken, adderar dem och lägger resultatet på stacken. Förutom stacken finns det också 256 lokala variabler som innehåller funktionens inparametrar och lokala variabler. Det som Appel kallar *view shift* implementeras automatiskt av den virtuella maskinen. Varje funktion har sin egen stack. Vid inhopp till en funktion är funktionens stack tom; dessutom ligger de inkommande parametrarna i ordning i de första lokala variablerna. Vid returhopp från en funktion läggs returvärdet på den anropande funktionens stack.

På kursens hemsida finns utdata från kursledarens kompilering av programmet Factorial.java. Där kan ni få information om vissa delar av JVM-assemblerspråket. Mer information finns i dokumentationen till *jasmin*, som finns både på nätet och i kurskatalogen, och i boken *The Java Virtual Machine Specification*, som finns på webben i elektronisk form och på Nadas bibliotek som en klassisk bok. Man kan också lära sig en del genom att disassemblera färdiga klassfiler med verktyget *javap*. Kom dock ihåg att utmatningen från *javap* tyvärr inte har riktigt samma utseende som indata till *jasmin* ska ha. Ni kan till sist även få inspiration från kapitel 7 i boken. Visserligen behandlar det kapitlet en annan typ av kodgenerering, men det går att använda idéerna därifrån för att göra effektivare kodgenerering även för virtuella maskiner.

Den rätta vägen

Er laborationsuppgift är att konstruera en kompilator som genererar assemblerkod för SPARC-arkitekturen. Givet godkänd muntlig tentamen sätts betyg enligt följande för KTH-elever: Avklarade kapitel 6–8 ger betyg 3; kapitel 6–9 ger betyg 4; kapitel 6-11 ger betyg 5. För SU-elever gäller: Avklarat kapitel 6–8 ger betyg G; kapitel 6–10 ger betyg VG.

Ni får i princip kompilera till vilken processor ni vill. Kurskatalogen innehåller dock dellösningar endast för SPARC-arkitekturen. Appel tillhandahåller lösningar till kapitel 8 på kursbokens hemsida.

Aktiveringsposter

I kurskatalogen finns implementationer av det som Appel kallar Frame-objekt respektive Access-objekt för SPARC. Det finns också Record-objekt – motsvarigheten till Frame-objekt för klasser. Gränssnitten i dessa implementationer är, väsentligen, de som beskrivs i boken, den enda skillnaden är att Frame-objektets konstruktor har bytts ut mot en metod allocFormal(). Ni måste anpassa den semantiska analysen (kapitel 5) så att varje klass kopplas ihop med ett objekt som implementerar gränssnittet frame.Record, varje metod kopplas ihop med ett objekt som implementerar frame.Frame och varje variabel kopplas ihop med ett objekt som implementerar frame.Access. Lämpligen görs detta genom att modifiera symboltabellen. Variablerna i en klass ges Access-objekt genom att metoden allocField() anropas i klassens Record-objekt. På samma sätt ges parametrar och lokala variabler i metoder Access-objekt genom att metoderna allocFormal() och allocLocal() anropas i metodens Frame-objekt. Beroende på hur ni har löst de första fem kapitlen i boken kan ni komma att behöva ändra koden i kurskatalogen i olika stor utsträckning.

Mer information om aktiveringsposterna i SPARC-arkitekturen finns till exempel i kapitel 3 i skriften *System V Application Binary Interface—SPARC Processor Supplement, Third Edition*. Den finns på Suns webbplats »http://www.sparc.com/standards.html», dokumentet kallas där »psABI 3.0».

Intermediärkod

Intermediärkoden, som Appel kallar trädkod, genereras på det sätt som beskrivs i kapitel 7. Den mer avancerade versionen av generering – med klasserna Cx, Ex och Nx – är i princip en lat evaluering av deluttryck i trädkoden. I MiniJava har man inte något problem med att veta om ett deluttryck ska tolkas som ett uttryck eller en sats och därför kan man först tycka att det är meningslöst att använda den mer avancerade trädkodsgeneratorn. Man kan dock vinna mycket i elegans på att använda klasser härledda från Cx-klassen för att implementera semantiken hos if-satser och booleska uttryck, så det är nog värt ansträngningen.

Intermediärkoden optimeras sedan med hjälp av algoritmerna som beskrivs i kapitel 8. Appel tillhandahåller implementationer av dessa algoritmer på bokens hemsida.

Assemblerkod

Det är främst två saker man måste tänka på vid kodgeneringen: SPARC-processorn har dels *branch delay slots* och dels *registerfönster*. Med *branch delay slots* menas att vid en hopp-instruktion utförs alltid instruktionen efter, oavsett om man hoppar eller inte. Enklast är att alltid låta varje hopp följas av instruktionen `nop`. Vid retur från funktionsanrop kan man dock optimera lite; se nedan.

Med *registerfönster* menas att det som Appel kallar *view shift* i princip implementeras automatiskt av processorn. Man lägger utgående parametrar i *utregistren* `%o0`–`%o5` och när man anropar en funktion ser processorn au-

tomatiskt till att dessa värden hamnar i *inregistren* `%i0`–`%i5`, att man har åtta fräscha *lokala register* `%l0`–`%l7`, och att man har nya utregister där man kan lägga nya utparametrar. Registren `%i6` respektive `%o6` är frame- respektive stackpekare; de har i assemblerspråket också de alternativa namnen `%fp` respektive `%sp`. Register `%i7` innehåller returadressen vid subrutinanrop. För att returnera ett värde från en funktion lägger man det i `%i0`, det återfinns då i `%o0` i den anropande funktionen. Dessutom finns åtta globala register `%g0`–`%g7` där `%g0` alltid ska innehålla noll. För att allt detta ska fungera måste man lägga in en instruktion som talar om hur stor den aktuella aktiveringsposten är i prologen till varje funktion. Om aktiveringsposten till exempel är 96 bytes ska den instruktionen vara

```
save    %sp, -96, %sp
```

Stackpekaren måste alltid ha ett värde som är en multipel av åtta. Vidare måste man återställa stackpekaren när man returnerar från funktionen. Det gör man lämpligen genom att utnyttja en *delay slot:*

```
ret
restore
```

En sammanställning av teknisk dokumentation om SPARC-arkitekturen finns på URL »http://www.users.qwest.net/~eballen1/sparc.tech.links.html». Där finns pekare till ett antal mer eller mindre detaljerade beskrivningar av SPARC-assembler. En lämplig början kan vara *CS 341Lab manual*, vill man kan man sedan läsa mer om hur *view shift* och registerfönster fungerar i dokumentet *Understanding stacks and registers in the Sparc architecture(s)*. Det finns dessutom en bra och kortfattad sammanfattning, som innehåller det mesta man behöver veta, på URL »http://www2.ics.hawaii.edu/~chin/331/SPARCnut.pdf».

Ett annat mycket bra sätt att lära sig skriva assemblerprogram är att kompilera små C-program och titta på assemblerkoden. Om man anger väljaren `-S` till gcc produceras en assemblerfil som motsvarar det program man kompilerar. Jämför gärna resultatet med respektive utan optimering påslagen. Prova också Suns C-kompilator (som hör till Sun ONE Studio-sviten och finns installerad på Nada), den genererar inte alltid samma kod som gcc. På kursens hemsida finns utdata från kursledarens kompilering av programmet Factorial.java. Där kan ni också få lite impulser vad gäller kodgenereringen. Referenslitteratur finns på Suns webbplats »http://www.sparc.com/standards.html» för den intresserade.

Registerallokering

Avslutningsvis implementeras registerallokering enligt beskrivningen i Appels bok. Ni behöver varken implementera *spilling* eller *coalescing*, om ni inte vill förstås. Förhoppningsvis räcker de lokala registren tillsammans med lediga in- och utregister för att registerallokeraren ska bli nöjd.

Om ni har tid och lust, prova gärna att lägga alla lokala variabler i register (dvs Temp-objekt) och implementera sedan både *spilling* och *coalescing*. Det borde ge väsentligt bättre assemblerkod än lösningen i kurskatalogen (som lägger alla lokala variabler på stacken).

Sätta ihop hela klabbet

I kapitel 12, som inte är obligatoriskt men som jag rekommenderar att ni gör ändå, sätts så det hela ihop. I kurskatalogen finns ett standardbibliotek skrivet i C. Använd inte standardbiblioteket som finns på kursbokens hemsida, det svarar mot första utgåvan av kursboken. Kompilera detta, och se till att er kompilator assemblerar den genererade assemblerkoden och länkar ihop den med standardbiblioteket.

Rekommenderad studiemetod

Det är framför allt viktigt att komma igång i tid med arbetet. Det finns kod i kurskatalogen som hjälper er att komma igång med kapitel 6. Varje modul i kompilatorn ska gå att konstruera hyfsat oberoende av de övriga. Försök gärna ha så klara abstraktionsbarriärer som möjligt mellan modulerna. Det går naturligtvis inte att separera modulerna helt och hållet, men ju mer självständiga de blir, desto bättre.

Det kan vara bra att lägga in gott om spårutskrifter i programmet. För att kursledaren ska kunna kontrollera era lösningar är det dessutom viktigt att varje fas i kompileringen genererar någon slags utmatning som gör troligt att ni har löst uppgiften korrekt. Från kapitel 5 kan man till exempel skriva ut symboltabellen; från kapitel 6 är det kopplingarna mellan variabler i programmet och det som Appel kallar *Frame.Access* som är intressanta; från kapitel 7 och 8 är det lämpligt att skriva ut den genererade trädkoden; och så vidare.

Det är inte nödvändigt att följa Appels instruktioner slaviskt. I själva verket är det ofta så att hans exempel på kodsnuttar visar en idé snarare än ett färdigt program som går att kompilera. Ni ska alltså inte se boken som ett »labb-pek» utan istället bearbeta informationen i boken och ur detta konstruera en lösning.

Eftersom laborationsuppgifterna är tämligen omfattande rekommenderar jag att ni använder något versions-

hanteringssystem för källkoden. Under Unix är systemen RCS och CVS vanliga, men ni kan naturligtvis använda något annat också. CVS blir tillgängligt genom »module add cvs» på CSC; mer information om CVS finns på »http://www.cvshome.org/».

Eftersom kursen är en läskurs är inga terminalövningar schemalagda. Istället fungerar kursens nyhetsgrupp »nada.kurser.primp» som diskussionsforum. Viss information finns också på kursens hemsida »http://www.csc.kth.se/utbildning/kth/kurser/2D1375/» samt i kurskatalogen »/info/primp06/». Kursbokens webbsida är »http://uk.cambridge.org/resources/052182060X/». Den innehåller bland annat skelett för vissa av modulerna i kompilatorn. Det går alltid bra att kontakta kursledaren, antingen genom besök eller e-post, med frågor rörande kursen.