Kapitel 1

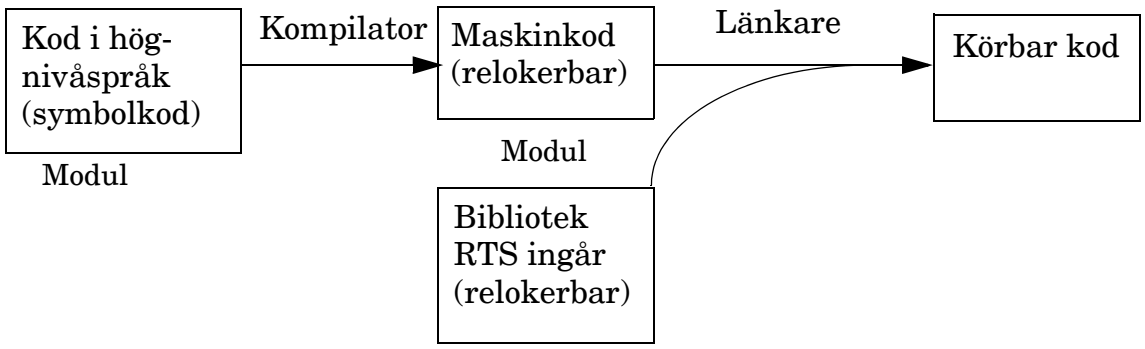
Introduktion till kompilatorer

Kompilatorns uppbyggnad

I detta kapitel ges en introduktion till en kompilators byggnad i stort. Här ges dess generella funktion och de olika delarnas funktion i stort. Sedan kommer de olika huvuddelarna att behandlas i olika kapitel. Man behöver dock känna till vad som finns omkring den "aktuella delen" när den behandlas, därför ges först en kort översikt. När man läser ett kapitel skulle man oftast behöva känna till innehållet i de följande kapitlen. Det kan vara klokt att läsa lite översiktligt en gång, fram tom Exekvering & RTS, för att sedan gå tillbaka och läsa noggrannare.

Kompilatorn och dess omgivning

En kompilators uppgift är att översätta kod i ett högnivåspråk till kod som maskinen kan exekvera. Ibland låter man inte kompilatorn göra hela jobbet utan man producerar resultat i en kod som fortfarande inte går att exekvera, men som andra enkla översättare kan fortsätta på. När vi talar om kompilatorn menar vi ofta ett helt system som behövs för att man skall kunna exekvera kod som är skriven i ett högnivåspråk. Följande figur försöker sätta in kompilatorn och kompilatorsystemet i sitt sammanhang. Kompilatorn



översätter till maskinkod, men vissa vanliga språkkonstruktioner översätts inte till en lång sekvens av maskininstruktioner varje gång den behövs. I stället läggs ut ett anrop av en subrutin som gör vad som behövs. Dessa subrutiner brukar man samla i ett särskilt bibliotek som hör till kompilatorsystemet. Detta bibliotek har jag i figuren kallat RTS som står för Run-Time System. Även andra bibliotek används, tex matematikrutiner ofta gemensamma för flera kompilatorsystem och högnivåspråk. Exempel på rutiner i ett sådant bibliotek är sin, cos och andra vanligt förekommande funktioner. Dessutom används naturligtvis systemets centrala biblioteksrutiner för filhantering och liknande. De flesta kompilatorer tillåter också att man separat kompilerar ett antal moduler (t ex klasser), som sedan länkaren får foga samman. Detta betyder att "modul" i figuren ovan

då kan förekomma en eller flera gånger. Det förekommer också att man inte översätter till maskinkod utan till något på högre nivå som sedan tolkas av ett program vid exekveringen. Man talar då inte egentligen om kompilering, utan om interpretering. Mellanformer finns där man blandar rent kompilerad kod (maskinkod) med interpretering. De flesta java-system är sådana. Normalt är interpretering eller inslag av sådant betydligt långsammare vid exekveringen än rent kompilerade system. I bilden ovan ser vi att alla moduler översätts till maskinkod och sedan bygger länkaren ett körbart program av dessa och rutiner plockade från bibliotek. I en del moderna system görs länkningen efterhand som den behövs under exekvering av programmet. Även kompilering kan då göras modulvis eller rutinvis vid behov. Man talar om en inkrementell kompilering. Här kan man även tänka sig att ändra och kompilera om delar av ett program under exekvering. I denna skrift tas i första hand upp den klassiska bilden av kompilering och länkning utförd innan man exekverar programmet. Det visar sig att denna klassiska grund går bra att stå på även vid konstruktion av moderna inkrementella system.

Kompilatorn har här endast representerats som en enhet, en svart låda. Det som händer i denna svarta låda kan delas upp i ett antal faser. En grov uppdelning man brukar göra är att tala om en front-end och en back-end. Första delen översätter från

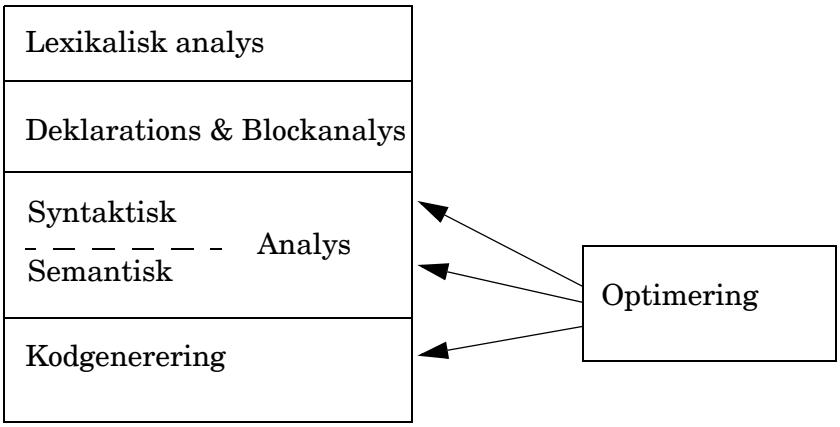
högnivåspråket till ett internt mellanspråk, som sedan back-end översätter till maskinkod. Front-end är ofta oberoende av hur maskinen ser ut och beror endast på högnivåspråket. Back-end däremot är ofta relativt oberoende av högnivåspråk, men naturligtvis starkt beroende av maskin. Det effektivaste är att back-end från mellankoden direkt genererar maskinkod, på relokerbart format. Ofta genererar man dock i stället assemblykod, som man sedan låter systemets assembler översätta till maskinkod. Detta gör man för att en sådan back-end är enklare att konstruera och mycket enklare att avlusa. För att få en maskinoberoende back-end händer det att man genererar C-kod och låter C-kompilatorn översätta denna till maskinkod. I värsta fall genererar då C-kompilatorn assemblykod som sedan översätts av assemblern, en sådan kompilator blir något långsam. Vi skall i fortsättningen anta att vi genererar direkt maskinkod, dock ur principiell kompilatorteknisk synpunkt är skillnaden inte så stor. Oavsett vad vi genererar blir större delen av kompilatorn likadan.

Vi startar med att gå igenom front-end och med genererad kod menar vi då mellankod. Vi håller oss då på en nivå som försöker vara oberoende av maskin och inte alltför beroende av detaljerna i RTS. Även om vi beskriver RTS senare måste man ibland i varje fall ha en aning om vad som händer vid exekveringen och hur RTS är uppbyggt.

Front-end

Ofta delar man upp kompilatorn i ett antal pass där man traditionellt avslutar ett pass innan man startar på nästa. Front-end kan då bli ett eller flera pass och back end normalt ett pass. Vi delar nu i stället upp front-end i ett antal faser. En fas

behöver inte avslutas för hela översättningen innan nästa fas påbörjas. Uppdelning i faser kan göras på olika sätt, här ges ett förslag med för- och nackdelar. Varje fas är här inte på något sätt lika stor som övriga faser. Den syntaktiska och semantiska analysen kan delas upp i en syntaktisk där man bygger ett syntaxträd och en semantisk där man genererar kod från detta träd. Fördelen med detta är att man då har ett extra mellanresultat som man kan låta en optimering titta på. Nackdelen är att man då bygger ytterligare strukturer som ska analyseras, vilket normalt gör kompilatorn långsammare. Optimeringsfaser har angivits vid sidan av, de behövs ju inte för funktionen utan är en



extra putsning för att få effektiv kod. Optimeringar kan komma in på många olika ställen och kan naturligtvis också vävas in direkt i de andra faserna. I det senare fallet brukar man sällan tala om det som optimering utan snarare som effektiv kompilering. Optimering kommer att behandlas i ett särskilt kapitel. I figuren ovan har kodgenereringen en egen fas, men det mesta av den bestäms av den semantiska analysen. Detta betyder att kodgenereringsfasen då "urartar" till att bara bli några metoder använda av den semantiska fasen.

Kompilatorns uppgift är ju att analysera strängar i den inmatade koden och att sedan beroende av analysens resultat generera kod som när den exekveras får det resultat som specificerats för just de strängarna som utgör satser i högnivåspråket. För att kunna göra detta måste vi således ha ett sätt att definiera utseendet av de strängar som hör till ett högnivåspråk och vilken betydelse de ska ha. Vad gäller strängarnas utseende, den sk syntaxen, definieras den nästan undantagslöst med hjälp av en grammatik. Den formella definitionen av grammatik ges senare. När man väl konstruerat språket med de egenskaper man vill att det ska ha försöker man naturligtvis beskriva detta med en grammatik som gör det så enkelt som möjligt att konstruera en kompilator. Normalt delar man upp beskrivningen i två delar, först beskriver man sk lexem och sedan hur dessa sätts samman till strängar i språket. Lexemen, de lexikaliska enheterna är grundläggande begrepp som identifierare, tal, operatorer och andra specialtecken. Ett lexem kallas ofta också för token. Hur lexemen byggs upp av bokstäver och andra tecken beskrivs oftast i form av regulära uttryck som formellt beskrivs i ett senare kapitel. Man skiljer ofta också på den grammatik man använder för att till en användare av språket förmedla hur man skall konstruera program, samt på den grammatik en kompilatorkonstruktör behöver för att implementera språket. Det är oftast för svårt eller omöjligt att ge en grammatik som beskriver språket och som samtidigt har sådana goda egenskaper att det är enkelt att konstruera en kompilator från den. Detta löser man ofta genom att man delar upp språket och använder flera olika grammatiker för att beskriva vad man

vill. Varje fas av kompilatorn kan motsvara en egenskap hos språket, beskrivet med en grammatik.

I alla beskrivningar av en kompilator brukar man starta med en lexikalisk analys. Denna delar upp insträngen i ett antal enkla enheter, lexem, som motsvarar orden eller skiljetecken i vanligt naturligt språk. Hur detta ska göras kan beskrivas med en synnerligen enkel grammatik, med goda egenskaper, dvs med en grammatik som gör att man kan analysera strängar ur ett sådant "språk" med en mycket enkel och effektiv metod. Detta beskrivs senare i teoridelen av kompendiet

Nästa steg i min uppdelning i faser har jag ännu inte sett i litteraturen. I denna fas gör man en enkel transformering av källkoden, på så sätt att man tar reda på blockstruktur och deklarationer. Det är en fas som inte gör så särskilt mycket, men som i den senare syntaktiska och semantiska analysen är till stor hjälp, då vissa komplicerade beroende redan har tagits om hand av denna analys. Det betyder att man i den fortsatta analysen redan har information om vad variabler och andra identifierare står för. Detta gäller naturligtvis särskilt språk med hierarkisk blockstruktur av något slag. Redan Pascal men i ännu högre grad Simula och Ada är språk av denna typ. I C som har en flat struktur har detta inte så stor betydelse, här ligger ju inte blocken kapslade inne i andra block. I C definieras alla rutiner på toppnivån och procedurer i procedurer finns inte. Denna fas delar således upp inkoden i block, håller ordning på blockens inbördes förhållande samt analyserar deklarationer som hänförs till rätt block. Detta betyder att ett token som inmatning till den fortsatta analysen kan ha mer information än vi hade före deklarations och strukturanalysen. När man skiljer ut denna fas från den syntaktiska fasen uppnår man att man slipper beskriva det komplicerade förhållandet mellan block, deklarationer och de exekverbara satsernas syntax i en och samma grammatik.

Nästa fas gör en syntaktisk analys av en sekvens av tokens som efter de tidigare faserna har en betydligt enklare grammatik än den vi utgick ifrån. Här kan man bygga ett abstrakt syntaxträd eller ett härledningsträd av den typ som beskrivs i teoridelen av denna skrift. Den rena syntaxanalysen är förhållandevis enkel, men vi vill ju till denna också knyta en semantisk analys, dvs vi ska också försöka få fram vad som menas med den struktur vi finner.

Nästa fas är att från syntaxträdet generera mellankod. Naturligtvis behöver inte dessa faser separeras från varandra med ett verkligt representerat syntaxträd emellan. Man kan generera kod direkt från analysen med ett träd som inte finns representerat på annat sätt än av kompilatorns konstruktion. Nackdelen med att inte representera trädet är att mycket optimering kan ske i ett sådant träd, som då måste finnas i sin helhet eller i varje fall som ett stort delträd.

Resultatet av front-end är mellankod, hur ser då denna ut? Man kan ju tänka sig ett högnivåspråk som mellankod, eller maskinkod som mellankod, båda valen är ju naturligtvis olämpliga. Oftast väljer man ett assemblybetonat språk där man dock tagit bort beroendet av en viss maskin. Man ser ofta mellankoden som kod för en kraftfull fiktiv maskin, som man sedan implementerar. Om man betraktar dagens java-översättare (t ex i jdk) så producerar dessa en sk "bytekod" som man sedan interpreterar. Denna bytekod kan man naturligtvis betrakta som ett slags mellankod. I stället för att göra en back-end som genererar kod kan man förstås direkt

interpretera mellankoden. Detta brukar dock betyda betydligt långsammare exekvering, en faktor fem till tio brukar anges. Det finns moderna java-översättare som efterhand som det är lämpligt översätter denna bytekod till normal binär objektkod, komersiella java-översättare som kompilerar på traditionellt sätt finns numera också.

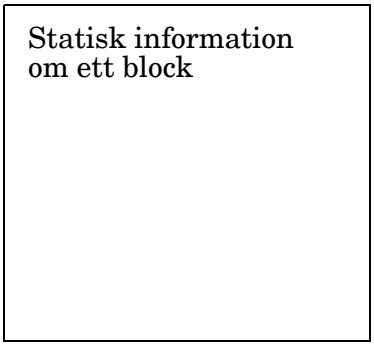
Back-end

Denna har till uppgift att överföra mellankoden till maskinkod. Av tradition har denna fas betraktats som helt maskinberoende, det är ju maskinkod som skall produceras. Tänker man närmre efter inser man att även om slutresultatet är maskinkod så är stora delar av back end oberoende av maskinkodens exakta format. Man skall hålla reda på adresser, externa namn, genererade bitmönster och mycket mer. Större delen av detta beror inte på vilka bitmönster man genererar. Mycket av koden här är av tråkig och detaljrik bitfifflarkaraktär. Här kan man använda en del programgenererande verktyg, som från en beskrivning av vad som skall göras genererar programkod som gör det. Detta beskrivs längre fram. Den genererade koden är naturligtvis starkt beroende av hur man har byggt upp sitt exekveringssystem, sitt RTS. Back-end och RTS är således starkt förenade med varandra.

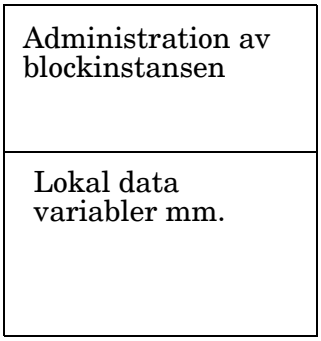
RTS

När man genererar kod måste man också bestämma sig för hur exekvering av ett program skall gå till. Hur allokerar man minne, hur görs subrutinanrop, vilka kontroller görs och mycket mera. Minnesallokering i ett hierarkiskt uppbyggt språk med objekt, såsom Simula, är betydligt mer komplicerat än i ett enklare språk som C. Detta medför att komplexiteten hos ett RTS skiljer sig avsevärt från språk till språk. Vet man inte huvuddragen av RTS utseende vet man inte heller vilken kod man skall generera. Detaljer om RTS kommer i ett senare kapitel, men något om strukturen kan behövas redan från början för att vi skall kunna förstå vissa detaljer i front-end. En sådan sak är hur man hanterar minnesutrymme för data. I de flesta språk vi behandlar finns blocket som en viktig enhet. Ett block kan vara en procedur eller ett stycke avgränsad kod med lokala variabler (*begin deklarationer; satser end*). När man exekverar ett sådant block måste man ha ett utrymme i minnet för blockadministration och deklarerade lokala variabler. Denna information är dynamisk, dvs den hör till blockets exekvering och kan vara olika vid olika exekveringar av blocket, dessutom ändras den under exekveringens gång. Detta minnesutrymme organiseras i form av en sk aktiveringspost, oftat förkortat AR (Activation Record). Till ett block hör också information som inte förändras under körning, denna läggs i ett utrymme som brukar kallas template. En template innehåller statisk information. Ser vi närmre på exekveringen av en procedur, som ju kan vara rekursiv, inser man att man kan ha flera exekveringar av procedurblocket körande samtidigt. Varje sådan exekvering måste naturligtvis ha sitt eget utrymme för lokala variabler. Man säger att man har flera instanser av blocket. Detta betyder att man har flera AR men endast en template hörande till blocket. Man kan också se det som att blocket har en template med statisk information och varje instans av blocket har en aktiveringspost med den dynamiska informationen. Hur vi sedan hanterar administrationen av AR och templates behandlar vi i detalj i kapitlet om exekvering och RTS. Normalt allokerar man utrymme för AR i en stack, men i språk med objekt räcker inte detta. Man måste då ha ett minnesutrymme med en mer komplicerad minneshantering, bl a lumpsamling (Garbage Collection). Vad vi behöver veta vid deklarationshanteringen är att minnesutrymme för variabler allokeras i aktiveringsposten, att man redan tidigt i kompilatorn

vet var i en aktiveringspost detta utrymme kommer att ligga, men att man inte vet var aktiveringsposten kommer att finnas förrän under exekveringen av objektkoden. Mer komplicerade förhållande kan förekomma, men detta beskrivs först senare. I



Template (en per block)



Aktiveringspost (en per blockinstans)

språk som java där man kan deklarera attribut och metoder som statiska (static) och där övriga attribut och metoder är dynamiska, dvs både instansvariabler och sk klassvariabler, kan man ana hur detta lämpligen är implementerat.