Kapitel 2

Front end

Konstruktion av front-end

Detta kapitel behandlar de olika delarna i en klassisk front-end av en kompilator. Här behandlas således en lexikalisk analys, en deklarations och strukturanalys samt en syntaktisk och semantisk analys. Här behandlas både klassisk “recursive descent” och operator precedens.

Lexikalisk analys

Den lexikaliska analysen gör man lämpligen i form av en funktion som hämtar ett lexem eller som man normalt säger ett token i taget. Ett token motsvarar närmast vad man i ett naturligt språk skulle kalla ett ord. Nästa fas av kompilatorn betraktar sedan inmatningen som en sekvens av tokens, som analyseras enligt någon grammatik där enskilda detaljer i lexem inte längre finns med. Ett litet exempel från något språk i Algolfamiljen.

if sum>20 then sum:=sum+2 else sum:=30;

Denna sats representeras av följande sekvens av tokens: *if, sum, >, 20, then, sum, :=, sum, +, 2, else, sum, :=, 30 och ;*. Det kan dessutom vara lämpligt att den lexikaliska analysatorn inte endast lämnar ifrån sig ett token som en sträng av tecken, utan att man lägger mer information i ett token. Dessutom vill man inte i resten av kompilatorn representera ett token med sin definierande sträng, en enkel kod för identifikation av tex en variabel blir effektivare. I sekvensen ovan ser vi ett token definierat av strängen sum flera gånger, det är naturligtvis väsentligt att känna igen att detta är samma token varje gång. Ser vi på de token som är identifierare används dessa antingen som ett reserverat ord (*if*) eller som namn på någonting (*sum*), ofta en variabel. För att se att sum är samma ord varje gång får vi på något sätt hålla reda på dessa strängar och vilka token de identifierar. Några föreslår att man för att klara detta lägger upp strängen i en symboltabell, där man senare också hakar på deklarationsinformation, minnesbehov och liknande. Att detta är olämpligt och leder till komplikationer senare inses ur följande enkla exempel.

```
begin
    integer SUM;
    begin
        boolean SUM;
        SUM:=true;
    end;
    SUM:=1;
end
```

Här ser vi att det finns två helt olika symboler med namnet SUM. Detta inser vi tack vara att vi ser strukturen och förstår innebörden av hierarkiska block, men detta vet ju en lexikalisk analysator inget om. Naturligtvis skulle vi kunna lära denna vad som menas med deklarationer och blockstruktur, men detta skulle ju innebära att hela ideen med en enkel lexikalisk fas skulle gå förlorad. Den lexikaliska fasen bör således inse att alla förekomster av strängen SUM skall behandlas lika medan syntaxanalysen bör veta att det finns två helt olika symboler båda med namnet SUM. Lexikalisk strängigenkänning och parserns symbolhantering har inget med varandra att göra. Vi inför således en särskild strängtabell (namntabell) att använda under första fasen i kompilatorn. Ur analyssynpunkt behövs strängtabellen endast av lexikaliska analysatorn, dock för att kunna ge läsliga felmeddelande eller ur avlusningssynpunkt kan strängarna behövas också av andra delar av kompilatorn.

Den lexikaliska analysen är ganska enkel varför beskrivningen här blir kortfattad, i stort sätt görs beskrivningen i form av ett exempel. Man kan naturligtvis också läsa om den lexikaliska analysens teori senare i kapitel 7 Vi säger dock några ord om namntabellen som kan vara värda att tänka på. Denna namntabell kan bli ganska stor och man söker ofta i den, detta medför att man måste vara noga med hantering av och sökning i denna tabell. Normalt bör man använda sig av hashkodning i någon form. Vilken typ av hashing man använder betyder troligen inte så mycket, dock bör man tänka sig för så att inte alla variabler får samma hashkod. Ofta beräknar man hashkoden genom att summera koden för tecken i namnen, man brukar då begränsa sig till några och man bör då tänka på att ofta väljer en programmerare namn som börjar på något som har med modulen att göra. Många variabler i en modul börjar på samma prefix. Separatkompilering av rutiner kräver också identifiering av rutin med hjälp av namnsträngen, även mellan olika moduler. Ur separatkompileringssynpunkt är också tidpunkten när man kompilerade ett program av betydelse, kompilatorn döper då om en del namn så att de slutar på något som har med kompileringstidpunkten att göra, en sk time stamp. Detta gör att många namn kan starta och sluta på samma sätt i en viss programmodul. Detta bör man naturligtvis tänka på vid konstruktion av hashkod.

Den lexikaliska fasen kan implementeras dels med hjälp av verktyg t ex Lex som brukar finnas i varje Unix system, dels i form av en enkel automat. Då man använder ett verktyg slipper man att själv programmera de tråkiga detaljerna, man ger endast en beskrivning i form av en grammatik. I stället tvingas man in i den metod verktyget kräver och man har liten frihet själv. Att själv implementera en automat är så pass enkelt att vinsten med verktyg här är ganska begränsad.

Exempel

Jag ger här ett litet exempel där vi först skall inrikta oss på den lexikaliska analysen. Vi skall använda oss av ett enkelt språk (Enkel) för att närmre förklara olika delar av kompilatorn. Språkets syntax beskrivs av en enkel grammatik i sk utvidgad

BNF (Bachus Naur Form).Då denna grammatik är så enkel utgår jag ifrån att läsaren

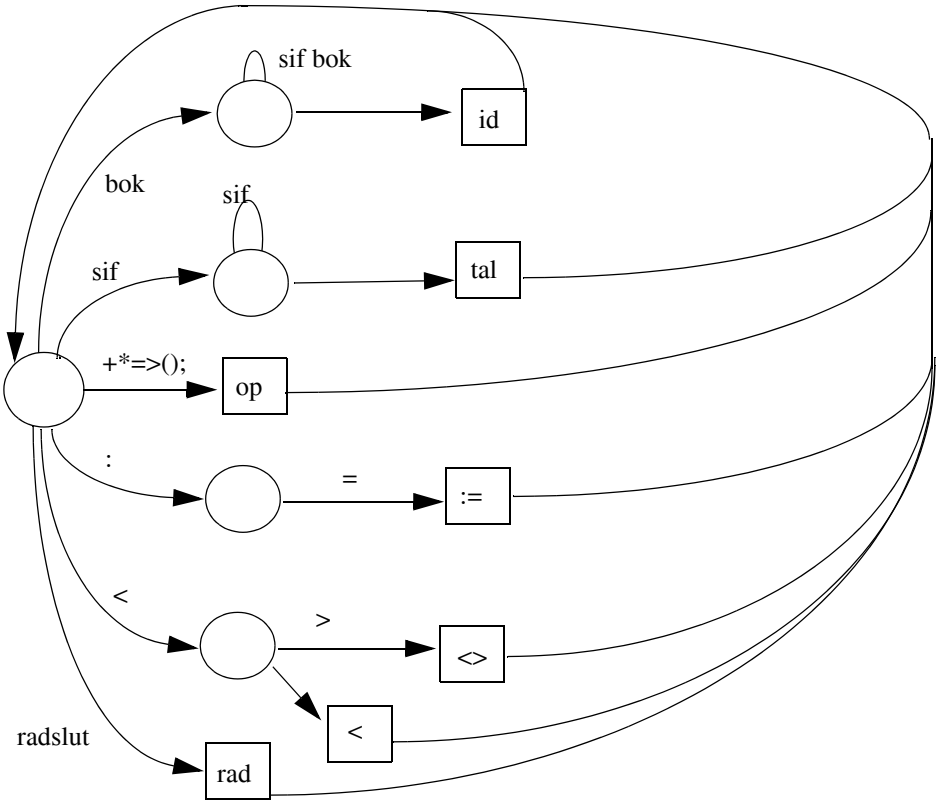
program	Æ	block
block	Æ	**begin** dekl$^+$ sats$^+$ **end**
dekl	Æ	**integer** id ;
sats	Æ	tillordning | villkorssats | block
tillordning	Æ	id := aritm ;
villkorssats	Æ	**if** villkor **then** sats **else** sats ;
villkor	Æ	aritm relop aritm
aritm	Æ	aritm + term | term
term	Æ	term * faktor | faktor
faktor	Æ	(aritm) | id | tal
relop	Æ	= | < | > | <>
id	Æ	bok [bok | sif]*
tal	Æ	sif$^+$
bok	Æ	**a** | **b** | ..... | **z**
sif	Æ	**0** | **1** | .... | **9**

förstår vad som menas utan att här ta med en formell beskrivning. En riktig definition av grammatik och beskrivning av sk produktioner kommer i kompendiets teoridel, kapitel 7. Det som ovan ges i fetstil är terminala symboler och strängar av terminala symboler. Språket Enkel är naturligtvis för dåligt för att användas för programmering, t ex finns inte subtraktion och division, men då detta inte ur kompilatorsynpunkt ger något nytt annat än volym finns det inte med.

Ur lexikalisk synpunkt är inte hela grammatiken ovan av intresse, vi skiljer ut den del som beskriver lexem och lämnar övriga detaljer till följande faser av kompileringen. Dessutom har blank och radslut viss betydelse, de bygger dock inte upp några lexem. Om språket tillåter att man skriver kommentarer så brukar inte heller dessa ge upphov till lexem, utan de ignoreras i den fortsatta analysen. Vissa språk, t ex java, kan knyta viss dockukumentation till kommentarer, detta brukar man dock behandla med hjälp av särskilda program, inte i kompilatorn. För kompilatorn är det bara kommentarer.

Hur man analyserar strängar för att ta ut lexem beskriver jag informellt i form av en graf. Senare under teoridelen ser vi att detta är en ändlig automat och grammatiken för lexem är given i form av ett regulärt uttryck. I grafen använder jag runda noder som mellantillstånd och kvadratiska som tillstånd där jag har funnit ett lexem, som då lämpligen ges ut som resultat (token). Bågarna är normalt märkta med ett antal terminala symboler, detta betyder att om motsvarande finns i insträngen går man över till det tillstånd bågen pekar på, samt går man fram ett steg i strängen. Bågar utan märkning medför att man går till nästa tillstånd utan att bry sig om insträngen, annat än att den inte matchar något av de andra alternativen från noden där man befinner sig. Finner man ett blanktecken eller radslut i insträngen ser man det som ett annat tecken som man dock går förbi då man följer en omärkt båge. Ett radslut ger inget speciellt lexem, men man bör på något sätt generera ett slags pseudotoken för att kunna hålla ordning på radnummer för bl a felutskrifter. Detta behandlas närmre senare, nu kan vi ignorera

detta. Ett token är inte endast en text som är utplockad från källkoden, utan en



struktur med ytterligare information. Våra token innehåller en typ, en kod samt den definierande strängen. Om det förekommer andra icke blanka tecken än de i grafen ovan bör man reagera på något sätt. Ett sätt är att fullständigt ignorera dem, ett bättre sätt är att generera ett feltoken, samt fortsätta med nästa lexem.

Vi låter en identifierare representeras av en namnkod, ett heltal, för att slippa dra med oss namnsträngen i fortsättningen. Alla reserverade ord har fasta namnkoder, i vårt exempel 1-6, användardefinierade namn får sedan koder från 7 och uppåt. Att de reserverade orden har fasta koder är naturligtvis väsentligt, de används ju av den syntaktiska analysen senare, de är ju i någon mening inbyggda i kompilatorn. För att samma namnsträng alltid skall representeras av samma kod behövs en namntabell så att vi känner igen “gamla” identifierare. Ibland är man intresserad av att känna igen tal så att samma tal endast representeras en gång. Detta är kanske av än större intresse då det gäller textkonstanter. Om man endast lagrar ett konstant tal eller text en gång spar man utrymme, detta är dock oftast av mindre betydelse. Om man skall hålla reda på detta måste man också ha en sk literaltabell, ofta struntar man i detta och representerar en mängd likadana konstanter i minnet under exekvering. Nedanstående figur anger hur man i detta enkla exempel skulle kunna

representera ett lexem eller token. Radnummertoken produceras direkt av den lexika-

Typ	Kod	Sträng
id	namnkod, identifierar token	namnet (som sträng)
tal	värdet (om möjligt)	talet i textrepresentation
op	operatorkod	textrepresentation tex :=
err	felkod	feltext
rad	radnummer	"Line#"
spec	information	informationstext
.....	.....	.....

Kod
Typ
Sträng

Ett token

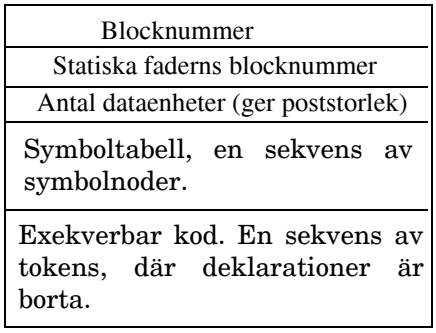
liska analysatorn, och är ur grammatiksynpunkt osynliga. Dessa tokens produceras just då man läser nästa rad från källkodsfilen, detta innebär att de ibland kan placeras lite märkligt i koden. Beroende på språk och språkkonstruktioner, samt på programmerarens sätt att strukturera sin kod i rader kan det vara svårt att hantera informationen om ny rad på ett bra sätt vid den fortsatta kompileringen. Vissa fel upptäcks redan vid lexanalys och produceras som feltokens därifrån, ytterligare feltokens kan sedan produceras vid deklarations och blockanalysen. Token speckod i tabellen ovan förekommer inte som utmatning från den lexikaliska analysen, men samma tokendefinition används senare i kompilatorn, och där kan de förekomma. Detta utseende på ett token kan man bygga på och på så sätt klara av mer komplicerade språk än i exemplet här. Det principiella utseendet duger dock även i mer komplicerade sammanhang.

Deklarations och strukturanalys.

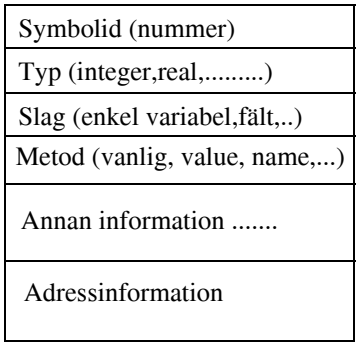
Andra fasen i kompilatorn kan vara en analys av deklarationer och blockstruktur. I språk med fullständig blockhieraki är detta till stor hjälp i kommande faser av kompileringen. Vi förutsätter här att det finns ett blockbegrepp i språket, vilket är sannt för de flesta språk, även för ett så enkelt språk som c. Indata till denna fas är de lexikaliska enheter, tokens, som fås från den lexikaliska analysfasen. Hela första fasen behöver inte göras innan fas två startar, utan man kan anropa lexikaliska analysatorn då man behöver nästa token. Resultatet från andra fasen är då deklarationer och tokensekvenser enligt en ny grammatik som sedan används av nästa fas i kompileringen. Resultatet ges i form av en sekvens av block, där inre block är utbrutna. Blocken får ett unikt nummer (0,1,...), i den ordning deras begin påträffas statiskt i källkoden. Varje block innehåller en beskrivning av blocket, en sekvens av dess deklarationer samt dess exekverbara kod i form av en sekvens av tokens kopierade över från den lexikaliska analysen. I denna sekvens kan också ingå extra tokens genererade av blockanalysen, ett exempel på detta är att ett utbrutet vanligt block ersätts med ett anrop av blocket i form av en speciellt genererad token. Block 0 är här det yttersta block som användaren har skrivit. Ofta anses detta ha en omgivning som inte skrivs av programmeraren, utan det kan vara definierat direkt i språket att man ska betrakta programmet som om det var omgivet av ett speciellt "ditfuskat" block. Detta är dock inget som man bryr sig om vid block och deklarationsanalysen, däremot i den senare analysen tar man hänsyn till detta, man kan då ha blocken -1,...

För administration av senare faser måste vi hålla reda på resultatet som en sekvens av block. Vi skapar då en blocktabell med noder som beskriver ett block och ger den infor-

mation som den senare fasen behöver vid översättningen av ett block, i den miljö där blocket finns. En blocknod har lämpligen följande utseende. Ett block identifieras av



sitt blocknummer, statisk fader är nummret på det textmässigt omgivande blocket i källkoden. Antal dataenheter anger det antal variabler, parametrar och resultatutrymme som krävs i blockets aktiveringspost. Symboltabell är en sekvens av deklarerade symboler, där varje symbol beskrivs av en nod enligt nedan. Kodsekvens är en sekvens av tokens som beskriver den exekverbara koden av blocket. Observera att här finns också feltokens och radnummer, dessutom enligt en något modifierad grammatik. En nod i blockets deklarationstabell innehåller den information som behövs om en deklarerad symbol. Detta brukar kallas symboltabellen. I ett blockindelat språk måste man skilja på en namntabell och en symboltabell. Samma namn kan förekomma på många ställen i ett program, men vilken symbol namnet står för beror på i vilket block den förekommer. En deklarations räckvidd, dvs vilken symbol namnet står för, beror på blockstrukturen enligt givna regler. Tidigare sades det att namntabellen borde hashkodas, hur gör man med symboltabellen? Problemet är att symboltabellen är uppdelad i block med ganska få symboler, och man måste söka blockvis enligt de räckviddsregler som finns i språket. Samtidigt görs många sökningar i denna tabell varför snabb sökning är av betydelse, traditionell hashkodning lönar sig dock oftast inte i språk med hierarkisk blockstruktur. Dessa regler kan i avancerade språk som t ex Simula vara ganska komplicerade. Därför innehåller den lexikaliska analysen en namntabell medan resultatet från andra fasen innehåller en blockindelad symboltabell. Förutom identifikation av symbolen innehåller en symbolnod information om typ, slag, relativadress mm. Innehållet i en symbolnod beror naturligtvis på hur komplicerat språket är. Nedan följer ett exempel, så mycket behöver naturligtvis inte finnas med i vårt exempel. Typ kräver väl knappast någon



närmre förklaring. Slag däremot är inte nödvändigtvis lika självklart, Att det finns enkla variabler och fält är normalt och enkelt. Vidare kan man tänka sig att det är

en funktion som beskrivs, men man kan också tänka sig funktionsvärde. Vad är då det? I vissa språk kan man ge en funktion värde utan att man för den skull lämnar funktionen, man måste då ha en plats att lagra detta värde på. Ett sätt är att reservera plats för en internt deklarerad variabel med samma namn som funktionen men med slaget funktionsvärde, i aktiveringsposten. Denna variabel kan användas på normalt sätt, men endast när man befinner sig till vänster i en tillordningssats, annars är den osynlig. Detta stämmer väl med hur tex funktioner definieras i Simula. Metod är kanske inte lika självklar, men för parametrar till en funktion så behöver man också tala om hur dessa överförs. Man kan tänka sig att överföra värdet, man gör så i java, medan andra tillåter också att man överför adressen, tex var-parametrar i pascal. Även andra sätt finns. Normalt kan relativadressen för en variabel bestämmas redan under deklarationsanalysen, men i mer komplicerade språk händer det att man endast kan bestämma adressen som relativa relativadresser. I det senare fallet kan man först under senare faser av kompileringen slutgiltigt bestämma variabelns position inom AR. Relativadress kan uttryckas på olika sätt, det enklaste är att helt enkelt räkna efter vilka variabler man behöver plats till och beroende på hur stora dessa är ge den verkliga relativadressen i aktiveringsposten. Detta kräver dock att man under kompileringen vet vilka storlekar de olika typerna har. Vet man vilken maskin man kör på vet man förstås också detta. Vill man skriva en portabel kompilator är det inte säkert att kompilatorn känner till detta, men man kan kanske uttrycka storleken i någon gemensam enhet och bestämma hur många bytes den är först under den slutliga kodgenereringen. Man behöver då inte känna till mer än de olika typernas inbördes storlekar. I en fullständigt portabel kompilator vet man dock inte ens detta, det enda man då kan göra är att ge de olika variablerna ett index eller sekvensnummer, samt får man hålla reda på vilka typer det är för att sedan under objektkodsgenereringen räkna ut adresserna. I komplicerade språk och med maskiner som kräver korrekt “alignment” i minnet är denna beräkning ganska besvärlig.

Kodsekvensen är en sekvens av tokens där de flesta bara kopieras från den lexikaliska analysen, men där vissa tokens produceras av block och deklarationsanalysen. Redan den lexikaliska analysen producerar tokens som representerar radnummer, dessa har ingen betydelse för programmet och ingår inte i språket. De kan användas för att knyta påträffade fel till en viss rad, dessutom kan man generera kod som under programmets exekvering håller reda på radnummer. Detta för att man vid exekveringsfel skall kunna knyta detta till ett visst ställe i programmet. Kodsekvensen följer nu en något modifierad grammatik, det finns ju endast ett block. Dessutom har deklarationerna brutis ut och lagts i en symboltabell. Kvar från deklarationerna är endast några radnummer samt feltokens producerade under deklarationsanalysen. Observera dock att i vissa språk kan en deklaration ge upphov till exekverbar kod. Ett exempel är definition av fält med dynamiska gränser. Detta fanns redan i Algol, och finns kvar i Simula. Följande lilla exempel kan vara lärorikt. Detta program har endast deklarationer och saknar helt

```
begin
    integer procedure P;
    begin
        ! kod som definierar P;
    end;
    integer array A(1:P);

end
```

egentligt exekverbara satser. Dock kommer koden i P att utföras, och inget hindrar

denna från både in och utmatning! Detta betyder att deklarationer kan vara exekverbara, fältgränser kan behöva beräknas under körning. Denna beräkning görs då blocket de finns i ska exekveras, detta sker redan i början då minne ska allokeras. Detta medför att vissa kodtokens härrörande från deklarationerna finns med i kodsekvensen. Vidare har grammatiken för satser ändrats så att ett block inte längre finns som sats, utan detta är ersatt med konstruktionen call block som är ett specialtoken. Detta är således ett token som finns i kodsekvensen i stället för det block som fanns i ursprungskoden.

Om man har ett språk med stark typning, dvs man måste hela tiden kontrollera att variabler och parametrar är av rätt typ och man samtidigt har möjlighet till separat kompilering av procedurer har man extra problem. Här måste man då känna till alla utåt synliga deklarationer i en separat kompilerad procedur. Detta betyder att denna kompilering av proceduren måste ha skett tidigare, samt att dess synliga deklarationer måste finnas tillgängliga, tex läsbara från en särskilt producerad deklarationsfil. Under deklarationsanalysen kan då dessa externa deklarationer läsas in och sedan behandlas som vanligt. Detta underlättar starkt implementationen av separatkompilering.

Exempel (prog1).

Efter den lexikaliska analysen består nu vårt program av en sekvens av tokens, dels tokens från källkoden, dels genererade feltokens beroende på eventuella fel påträffade vid den lexikaliska analysen. I denna fas är vi nu endast intresserade av blockstruktur och deklarationer, allt annat lämnar vi orört. Vi använder oss därför av en helt annan och förenklad grammatik i denna fas.

program→ block
block→**begin** dekl$^+$ [token | block]$^+$ **end**
dekl→**integer id ;**
token→**tal** | **op** | **err** | **id**

Terminala symbolen **id** står här för en godtycklig identifierare, dock inte någon av **begin**, **end** och **integer**. Orden i fetstil är terminala symboler, dock skiljer vi inte på olika identifierare och tal, alla är samma terminala symbol vad avser dess grammatiska betydelse, däremot har de till sig knutet attribut med olika värden, som anger vilken identifierare det verkligen är.

Denna fas går ut på att försöka finna *begin*, samt att analysera detta som ett block tills vi finner motsvarande *end*. Vid analysen av blocket letar vi efter deklarationer så länge vi finner sådana, dessa känns lätt igen då de börjar på nyckelordet integer. Funna deklarationer läggs upp i en symboltabell för blocket. Därefter läser vi resten av blockets tokens, som normalt bara läggs ut i blockets koddel utan förändring eller vidare analys, med följande undantag. Finner vi ett *begin* lägger vi inte ut detta i blockets kod, utan istället lägger vi ut ett nygenererat specialtoken, callblock, samt anropar vi rekursivt analysatorn för ett block. Detta betyder att vi ser på tokens för att se om det är *begin* eller *end*, alla övriga åsikter om ett tokens lämplighet eller inte överlåter vi till en senare fas.

I vårt exempel är deklarationerna särskilt enkla och deras analys är okomplicerad. I ett normalt programspråk kan man också deklarera procedurer. Detta betyder att en deklaration också kan innehålla ett block, som kan innehålla nya deklarationer som kan innehålla block. Detta gör fasen mer komplicerad, men huvudprincipen från vårt exempel gäller fortfarande. En lämplig övning för läsaren är att tänka igenom detta och försöka tänka igenom alla komplikationer som kan inträffa i något känt språk.

Vi ska nu se på ett enkelt exempel enligt grammatiken ovan och körd genom ett första pass som gör lexikalisk analys samt en block och deklarationsanalys. Följande lilla enkla program, somvi ska kalla prog1 i fortsättningen, tar vi som exempel.Vi ska se vad

```
begin
  integer I;
  integer K;
  I:=5;
  K:=I+1;
  begin
     integer K;
     K:=4;
     I:=I+K;
  end;
  K:=I*K;
end
```

som händer när vi har översatt detta till en sekvens av block, med symboltabeller (deklarationer) samt den exekverbara koden (kod) som en sekvens av tokens. Det presenterade resultatet har producerats av en enkel kompilator gjord som “övning” på kompilatorkonstruktion. Här har vi tagit ut passets resultat i en textrepresentation för

att få ett läsbart exempel. Vi presenterar resultatet i textform med lämpliga rubriker. Vi ser att kodens token följer det format som tidigare beskrivits, dvs

```
###PROGRAM###
   2
##BLOCK##
   0  -1   2   2  26
#DEKLARATIONER#
  19   1   0   0   0   0 **    0
  20   1   0   0   0   0 **    1
#KOD#
   3   7   Line#
   4   7   Line#
  19   1   I
   6   5   :=
   5   2   5
   1   5   ;
   5   7   Line#
  20   1   K
   6   5   :=
  19   1   I
  13   5   +
   1   2   1
   1   5   ;
   6   7   Line#
   7   7   Line#
   1  10   Callblock
   1   5   ;
  11   7   Line#
  20   1   K
   6   5   :=
  19   1   I
  15   5   *
  20   1   K
   1   5   ;
  12   7   Line#
   2   1   end
##BLOCKSLUT##
```

```
##BLOCK##
   1   0   1   1  14
#DEKLARATIONER#
  20   1   0   0   0   0 **    0
#KOD#
   8   7   Line#
  20   1   K
   6   5   :=
   4   2   4
   1   5   ;
   9   7   Line#
  19   1   I
   6   5   :=
  19   1   I
  13   5   +
  20   1   K
   1   5   ;
  10   7   Line#
   2   1   end
##BLOCKSLUT##
###PROGRAMSLUT###
```

```
Symbolid=19 (I)
Typ=1 integer
Slag=0 enkel
Metod=0 vanlig
Info= 0 0 **
Adress=0
```

```
Symbolid=20 (K)
Typ=1 integer
Slag=0 enkel
Metod=0 vanlig
Info= 0 0 **
Adress=1
```

```
Symbolid=20 (K)
Typ=1 integer
Slag=0 enkel
Metod=0 vanlig
Info= 0 0 **
Adress=0
```

kod,typ,sträng. Även deklarationsnoden följer tidigare beskrivning, där dock vissa fält egentligen inte behövs enligt vår enkla grammatik, men vi ska utöka den något senare och då behövs även denna information. Observera att vi har två deklarationsnoder för id=20, som är K. Dessa ligger dock i var sitt block och är inte samma symbol. I detta fall är de väldigt lika, endast adressen skiljer sig åt.

Vi tänker oss nu utöka grammatiken så att vi kan deklarera funktioner och fält. Vi tänker oss först en enkel fältdeklaration med endast ett index och med konstanta indexgränser: *integer array A(2:5),B(1:10);* . Dessa två fält ger följande enkla deklarationsnoder efter pass1. Dessa deklarationer kräver ingen exekverbar kod vid allokering. Vi kan komplicera fältdeklarationen ytterligare till att omfatta två index, och tillåta variabla indexgränser, som i följande: *integer array A(2:5,I:I+4);* . Här kommer vi att förutom deklarationsnoder också få exekverbar kod, kod som exekveras när

man allokerar minne för fältet då blocket skapas. Vi har utökat deklarationsnoderna och

```
Deklarationsnoderna för första enkla fältdeklaration.
23   1   1   1   0   0   4 **   1 2 5 A

Symbolid=23 (A)
Typ=1 integer
Slag=1 fält
Info1=1 antal index
Info2,3= 0 0 ignoreras här
Adress=4 plats i AR
Namn= ** ignoreras här
Indexgränser: index 1 låg=2, hög=5
```

ändrat dem lite för att kunna hantera mer komplicerade saker. Denna nod är således lite annorlunda än den vi beskrev tidigare, exemplet kommer från en kompilator som klarar lite mer. Vi ska nu se på den lite svårare definitionen med variabla gränser. Lägg

```
Deklarationsnoden för fältet ser ut så här:
20   1   1   2   0   0   1 **   2 * *   1 2 5 A

Symbolid=20 (A)
Typ=1 integer
Slag=1 fält
Info1=2 antal index
Info2,3= 0 0 ignoreras här
Adress=1 plats i AR
Namn= ** ignoreras här
Indexgränser: index 2 låg=* variabel, hög=* variabel
                    index 1 låg=2, hög=5

Här finns också indexgränser med som tokens i koden.
  0  11   Array limit
 19   1   I
  0  11   Array limit
 19   1   I
 13   5   +
  4   2   4
```

märke till att vi nu genererar tokens i koden som gör att vi kan exekvera beräkningen av fältets gränser.

Syntaktisk och semantisk analys.

I denna fas tar vi hand om den exekverbara koden, gör en syntaxanalys och tar hand om den semantiska betydelsen. Inmatningen till denna fas är ett program som nu är en sekvens av block. Ett block innehåller dels en symboltabell för blocket, dels en sekvens av tokens som utgör blockets kod. Denna fas översätter ett block i taget, i någon lämplig ordning. Vad som är lämplig ordning kan naturligtvis bero på språket. Numrerar man blocken i den ordning deras *begin* förekommer rent lexikaliskt är denna ordning för de

flesta av våra vanliga språk också lämplig för översättning i denna fas. Kodsekvensen har en grammatik som endast innehåller ett block, man kan se blocket som en sekvens av satser. Dessa satser har samma syntax som i den ursprungliga grammatiken, med ett par undantag. Ett inkapslat block är utbrutet och ersatt av ett blockanrop, som kan betraktas som ett anrop av en anonym procedur, som naturligtvis inte har några parametrar. Förutom normala satser innehåller nu också koden eventuella fel, uttryckt som feltokens. Fel som upptäckts i tidigare faser läggs ut som ett felmeddelande i tokensekvensen, och detta låter man ingå i språket som den nya grammatiken beskriver. På så sätt betraktas detta inte som fel längre, utan något som kontrollerat finns med och som har en semantisk mening. Betydelsen av ett sådant upptäckt fel skall under exekveringen vara att man anropar en RTS rutin som talar om detta och som eventuellt förhindrar fortsatt exekvering. Ett block kan vara ett "vanligt" block, en procedure eller ett objekt. Alla dessa är ur kompilerings och exekveringssynpunkt väldigt lika.

I denna fas kan man dels göra en separat syntaktisk analys som producerar ett abstrakt syntaxträd följt av en semantisk tolkning och kodgenerering från dett träd.. Fördelen med att göra så är att man kan låta en optimeringsfas gå in och göra optimeringar på syntaxträdet. En annan metod är att man direkt i programmet tar hand om kodgenereringen i den syntaktiska analysen. Detta är oftast enklare att implementera, men man får då hantera en eventuell optimering på annat sätt. Man kan oftast bygga in en optimering direkt i de rutiner som genererar kod. Problemet är att man då på enkelt sätt endast kommer åt att göra en lokal optimering. Vill man bygga in en mera global analys för att kunna använda vid optimering så är det betydligt svårare än om man har ett träd att manipulera, som man sedan genererar kod från.

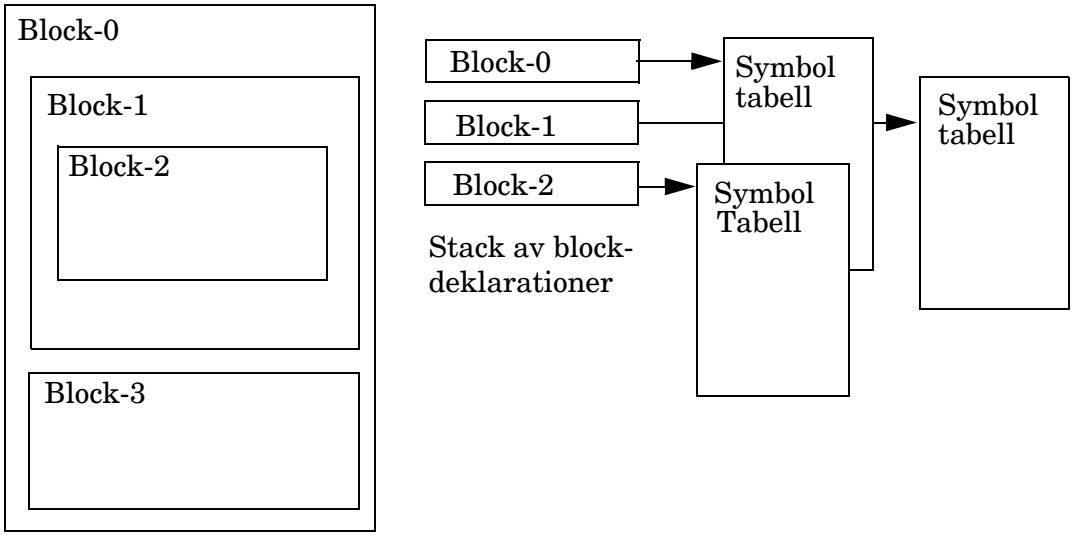
Denna fas kan konstrueras genom att man använder olika former av verktyg, tex YACC som finns i de flesta Unix system. Man kan också använda mer avancerade former av verktyg som bygger på attributgrammatik, se vidare i teoridelen.

Deklarationer och sökning.

Vid den syntaktiska analysen och i ännu högre grad vid den semantiska analysen behöver man till ett identifierartoken ha tillgång till dess symbolnodsinformation. Hantering av symboltabeller är alltså av stor betydelse. Symboltabellerna är i tidigare fas byggda blockvis, när man skall söka i dessa under den syntaktiska och semantiska fasen måste man söka i flera blocks tabeller i den ordning som definieras av språket. Man måste således bygga en struktur av symboltabeller. I denna fas går vi igenom och kompilerar blockvis. När kompileringen av ett block påbörjas bygger vi den deklarationsstruktur som gäller för det blocket. Antag att vi kompilerar block i lexikalisk ordning, dvs i den ordning deras start (*begin*) förekommer i programtexten. I ett språk med blockhierarki och där räckviddsregler säger att man söker en deklaration från aktuellt block och utåt i omgivningen bygger man lämpligen en stack av symboltabeller. I ett enkelt, hierarkiskt språk är detta ett rent statiskt beroende mellan blocken och "deklarationsstacken" kan byggas direkt i resultatet från deklarations och blockanalysen. Ofta har man ett mer komplicerat beroende där detta beroende ändrar sig under kompileringen, dvs den statiska informationen är i viss mening dynamisk. Dock är det naturligtvis inte dynamiskt i den mening att beroenden ändras under exekvering (run time). De som känner till Simula kan tänka

på *inspect* som har just denna beskrivna effekt. När behöver man söka i symboltabellen? Varje gång man i analysen påträffar en token behöver man information om vad det är, i syntaxen räcker det ofta att veta om det är en identifierare, ett tal eller en operator och eventuellt vilken operator, denna information lagrar man normalt direkt i ett token och ingen sökning behöver göras. Ser man en sats av typen X:=Y+2 vet man inte att det är en sådan sats bara för att man ser X:et, man måste titta framåt(look ahead) efter := innan man vet att det är en tillordning. Utvidgar man då satsen till ett fältelement-stilldelning X(.....):=Y+2, där indexuttrycket ..... kan vara godtyckligt långt är det svårt att titta framåt. I ett sådant fall räcker det att veta vad typ X har, och detta kan man se i symboltabellen. Här ersätter vi en besvärlig look ahead med en tabellsökning. I den semantiska analysen och i kodgenereringen behöver man slå upp tabellinformationen i princip för varje identifierare som inte är ett språkdefinierat ord. Som regel slår man upp informationen flera gånger för samma förekomst av identifieraren. Praktiska försök har visat att man söker mycket mer än man tror. Även om man noggrannt tänker efter när det behövs en sökning, så kommer man troligen att underskatta sökbehovet. För att begränsa sökningen kan man komma ihåg sökresultatet ett litet tag, tex under översättningen av en sats eller liknande.

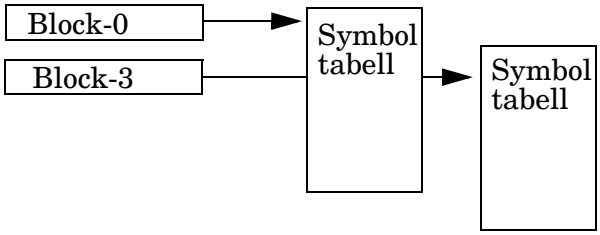
När vi översätter ett block i syntaktiska och semantiska analysen behöver man ha rätt deklarationsomgivning tillgänglig. Om vi först behandlar det enklaste fallet med hierarkiskt kapslade block, men utan att ta med de komplikationer som finns när man tar med objekt och liknande i språket kan vi klara detta med följande enkla modell. Vi översätter blocken i den ordning deras *begin* påträffas i källkoden.



Hierarkisk blockstruktur

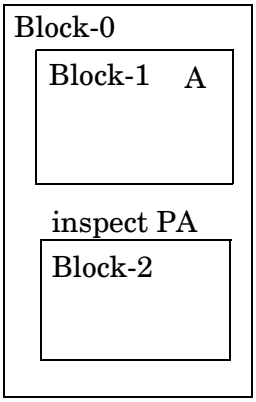
Ovan visar vi den deklarationsstruktur vi har när vi översätter block 2. Sökreglerna säger då att vi först söker genom deklarationerna i symboltabellen för block 2, sedan om vi inte finner deklarationen där söker vi genom block 1 för att eventuellt avsluta med block 0. När vi sedan skall översätta block 3 skall vi inte längre kunna nå deklarationerna i blocken 1 och 2, men naturligtvis de i block 0. Vi vet att den omgivning vi ska se är den statiska, lexikaliska, dvs det block där block 3 är definierad. Det närmast omgivande blocket, i det här fallet block 0, kallar vi blockets statiska fader. Vi söker upp det blocket i stacken och lägger in det nya ovanför detta. Ovannämnda stack behöver

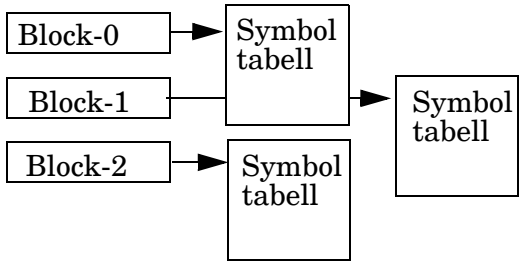
naturligtvis inte fysiskt representeras som en stack, motsvarande struktur kan ju ändå vara tillgänglig. I detta enkla fall är denna deklarationssökordning redan bestämd statiskt av koden. Stacken kan då naturligtvis byggas från början direkt av deklarations och blockanalysen. Följande figur ger en bild av deklarationsstrukturen under översättningen av block 3.



I ett språk som tex Simula, men även Ada och liknande, räcker inte denna enkla sökstruktur utan mer komplicerade sökregler finns. Stacken i exemplet ovan speglade helt den statiska strukturen i källkoden. I Simula kan man lägga in ett block i denna stack med hjälp av *inspect*, detta är då ett block som är definierat med nyckelordet class. I figuren ser vi ett Simula exempel med *inspect*. När vi kompilerar block två

```
begin
   class A
   begin
      ! block 1;
   end;
   ref(A) PA;
   inspect PA do
   begin
      ! block 2;
   end;
end
```





Vid kompilering av block 2.

finns vi inte statiskt inne i block 1 och vi skulle normalt inte ha sett deklarationer i block 1. Nu inspekterar vi dock ett objekt som är en instans av block 1, varför vi skall se deklarationerna i block 1. Observera också att vi mycket väl kunde ha gjort denna inspectsats inne i klassen, dvs block 1. Detta hade medfört att vi haft två block 1 på stacken samtidigt! Detta exempel går att komplicera ytterligare, men för att förstå vad som då händer måste man ha en mycket djup kunskap om Simula och objektorientering. Dessutom är det knappast lämpligt att ta upp för nybörjare i kompilatorteknik, varför vi stoppar här.

Hur hanterar vi nu fältdeklarationer, dessa måste ju ha minnesutrymme i blocket när blocket exekveras. Detta behandlas mer under kapitlet om RTS, men vi måste veta något om detta redan här. Om man allokerar minne i själva aktiveringsposten får man svårigheter då man inför variabla fält, men med fält av konstant, av kompilatorn känd storlek, skulle det kunnat gå. Vi väljer dock det mer flexibla sättet att i aktiveringsposten endast reservera plats för adressen till fältet som allokeras någon annanstans i minnet. Detta måste dock göras då blockinstansen skapas under exekveringen, och fältets adress måste läggas på rätt plats i AR. Detta gör vi genom att generera kod för detta i början av blocket för varje deklarerat fält. Detta betyder att vi måste kunna söka igenom ett blocks deklarationer för att finna eventuella fält-

deklarationer. Vi genererar då mellankod för att föra över indexgränser, en instruktion för varje index. När alla överföringar av indexgränser är genererade så genereras en "allokeringsinstruktion" som vid exekvering ska allokera minne samt lägga adressen till detta i "fältvariabelns" plats i AR. Vi kan tänka oss följande mellankod, närmre detaljer kommer i kapitlet om RTS. Om vi begränsar oss till konstanta indexgränser kommer

```
ARRLIM indexnummer,undre_gräns,övre_gräns
.....
ARRALL refvariabel
```

undre och övre gräns parametrarna alltid att vara konstanter (literaler), hur utvidgningen till variabla gränser kan göras kan man då enkelt tänka sig. Vi lägger således märke till att även dynamiska gränser beräknade under körning, även givna av komplicerade uttryck går att enkelt implementera. Har man flera index så får man använda ARRLIM för varje index. ARRALL har en parameter, nämligen fältvariabeln i AR, detta för att det är hit fältadressen ska överföras.

Parsing (Satslösning)

Sekvensen av tokens skall nu analyseras och avsikten är att vi skall förstå vad strukturen innebär och vilken kod vi skall generera. Ett sätt är att man genererar ett sk härledningsträd som beskrivs närmre i teoridelen. För att skriva den kod som genererar sådana träd kan man använda verktyg. Dessa verktyg kan läsa en grammatik i någon form och från denna generera de rutiner som behövs. Detta behandlas närmre i teoridelen och här beskriver vi mera hur kompilatorn verkligen fungerar för att ge förståelse för detta. Två metoder är troligen dominerande när det gäller handskrivna kompilatorer, dessa är recursive descent och operator precedens. Ofta använder man sig av en kombination av dessa. Denna kombination skulle vi kunna kalla naturmetoden. Skälet till detta är att om man utan direkta kunskaper i kompilatorkonstruktion skulle försöka konstruera en sådan kom man troligen att försöka använda en metod som gjorde ungefär recursive descent på den vanliga programstrukturen och operator precedens på uttryck. Man kan konstruera kompilatorer helt med den ena eller med den andra av dessa metoder. För att kunna använda recursive descent måste man utgå från en grammatik av viss typ, den får tex inte vara vad man kallar vänsterrekursiv, kompilatorn kan då råka ut för att gå i loop. En grammatik är vänsterrekursiv då den innehåller produktioner av typen $E \rightarrow E + T$ eller då motsvarande kan uppkomma genom en kombination av produktioner. Även för operatorprecedens finns restriktioner, som man dock inte har några problem med då det gäller tex aritmetiska uttryck. Närmre beskrivning av metoden kan man få i z eller i teoridelen.

Vi ser nu att efter deklarations och blockanalys så har vi programmet i form av en strukturerad symboltabell och tokensekvenser. Den egentliga koden är uppdelad i block, varför vi nu ska översätta en sekvens av block. Dessa block översätts självständigt och vi har reducerat översättningen till att översätta ett block. Visserligen är blockets kod representerat av en sekvens av tokens, men vi ska höja blicken och se det som en sekvens av satser. Vi ska således översätta satser tills blocket är slut. Det centrala vad gäller parsing och kodöversättning är då att kompilera en sats. Förutom att ett block är en sekvens av satser så kan det i en sats ingå andra satser, ett typiskt exempel är villkorssatsen som ju innehåller två satser. Att översätta en sats är således en rekursiv

aktivitet. Vi koncentrerar oss nu på att översätta en sats, varvid vi i den övergripande strukturen använder recursive descent, medan uttryck översätts med användande av operatorprecedens.

Recursive descent. I teoridelen kan man läsa hur man gör en predictive parser, här ger jag en beskrivning av hur man konstruerar en recursive decscent parser för hand, med hjälp av förnuft och eftertanke. Man utgår ifrån grammatiken och för varje metasymbol konstruerar man en rutin som på något sätt följer högerleden i produktioner med metasymbolen som vänstersymbol. Antag att vi har produktionen *sats* → *whilesats* | *ifsats* | ..... Vi koncentrerar oss först på satser som börjar på while respektive if, då dessa ju är lätta att känna igen utan att vi behöver se framåt i strömmen av tokens. Procedurerna beskrivs med hjälp av figurer och ett hemmagjort språk, som alla bör kunna förstå. Här ser vi direkt samma struktur som i gramma-

procedure SATS

```
T:=GetToken;
case T=
WhileToken:     WHILESATS;
IfToken:        IFSATS;
.............
```

tiken, sedan går vi vidare med procedurer för *while* och *if*. Låt oss se på *if* först och vilken kod som bör genereras. Här ser man att först måste man lägga ut kod för att

```
Beräkna villkor --> T1
Om T1 falskt så hopp till L1
Kod för sats 1
Hoppa till L2
Definiera L1 här
Kod för sats 2
Definiera L2 här
```

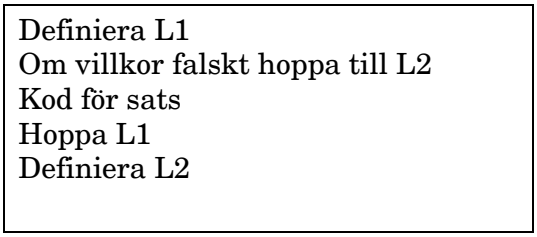
"Kod" för satsen:
if villkor **then** sats1 **else** sats2

procedure IFSATS

```
L1:=GenLabel;
L2:=GenLabel;
T:=JumpFalse(L1);
if T!= ThenToken then ERROR;
SATS;
GenCode(Hopp till L2);
GenCode(Definiera L1);
T:=GetToken;
if T!=ElseToken then ERROR;
SATS;
GenCode(Definiera L2);
```

beräkna villkoret, varefter man hoppar ett stycke längre fram i koden om detta blir falskt. Sedan ligger koden för sats 1, denna kod genereras naturligtvis med ett rekursivt anrop av SATS. Lägg märke till att vi måste generera lägen som L1, L2,..., här måste vi också tänka på att de rekursiva anropen av sats naturligtvis kan ge upphov till nya genererade lägen. Det är inget förbud på att satserna inne i ifsatsen är nya if- eller whilesatser. Även här kan vi följa strukturen i grammatiken, men vi har dekorerat den med diverse detaljer som beräkning av nya lägen och kodgenerering. En while-sats innehåller ungefär samma som en if-sats, framför allt behandlar man

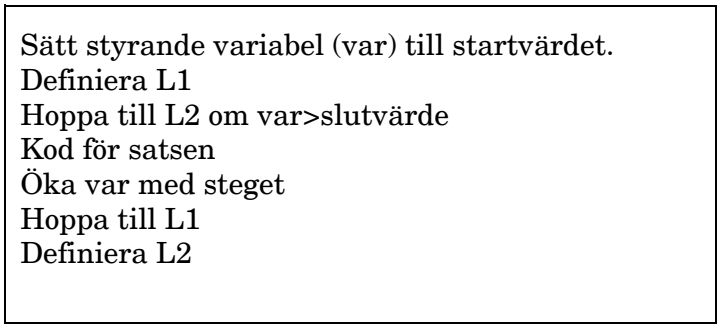
vilkoret på samma sätt. Vi kan här se på den genererade koden (principiellt). Lägg



Kod för satsen
while villkor **do** sats

märke till ordningen mellan test och sats, det är således möjligt att utföra satsen noll gånger.

Om nu if och while ur kompilerinssynpunkt är ganska enkla så är repetitionssatser som for-satsen betydligt besvärligare. En del språk har repetition med en styrande variabel som börjar på noll och går med steget 1 till ett fixt slutvärde. En sådan sats är förhållandevis enkel, genom att tillåta andra steg (>0) blir det inte särskilt mycket svårare. En sådan repetitionssats skulle kunna implementeras med följande kod. Vi kan nu öka



Kodskiss för satsen
for I:=start **step** steg **until** slut **do** sats

på svårigheten med att tillåta ett negativt steg, man går då ned mot slutvärdet, dvs man måste vända testet. Vi förutsätter nu att steg och slut är konstanta värden, dvs literaler, vilket gör det hela ganska enkelt och vi kan generera den enkla koden enligt ovan. Vi skulle nu kunna öka ut definitionen genom att tillåta steg och slut att vara variabler. Tecknet på steget avgör från vilket håll man närmar sig slutet, och om steget är en variabel så vet man inte detta förrän vid körning. Detta löser man genom att beräkna följande: (var-slut)*steg, genom denna multplikation med steget (egentligen "tecknet") så kan man alltid göra testen på samma håll. Steget används då dels vid testen dels vid uppdateringen av variabeln. Vi gör nu nästa svårighetsökning och låter steg och slut vara uttryck. Eftersom ingående variabler kan ändra värde vid exekveringen av satsen så måste man beräkna slut och steg för varje repetition. Skall man beräkna steget en eller två gånger per repetition? Har man i uttrycket för steget anrop av en funktion med bieffekt så är detta av avgörande betydelse. Vi skulle nu kunna tänka oss följande sats: *for I:=K+F(X) step K*G(X,2) until H(K) do begin I:=X+1; K:=H(I) end* Jag antar att ni anar svårigheten vid kompilering och vid exekvering av en sådan sats. I språket Simula är sådana repetitionssatser fullt tillåtna. I språkdefinitionen som är svensk standard finns följande definition av hur ett sk. step-until element i en for-sats ska implemente-

ras. Detta är beskrivet i Simulakod (utan användning av for). Vi ser således att

Elementet A1 step A2 until A3 ska betyda följande (S är satsen)

```
C:=A1;
DELTA:=A2;
while DELTA*(C-A3)<=0 do
begin
  S;
  DELTA:=A2;
  C:=C+DELTA;
end;
```

Observera att A1, A2 och A3 är uttryck som beräknas varje gång de finns med.

steget beräknas en gång innan vi går in i loopen, samt därefter i slutet av varje varv, strax innan den styrande variabeln uppdateras. Som en god övning kan ni tänka igenom vilken kod som man lämpligen genererar för en generell for-sats. För att underlätta det något kan man generera satskoden och stegberäkningen som subrutiner.

Vi går nu tillbaka till punkterna i produktionen för sats, vi antar att det finns proceduranrop också i vårt språk. *sats* → ..... | *tillordning* |*procanrop* Det problem vi har här är att vi inte direkt får ett nyckelord i tokenströmmen som avgör vilket alternativ det är. Ser vi P .... eller till och med P (..... så kan vi inte direkt se om det är början av en tillordningssats eller om det är ett proceduranrop. Traditionellt löser man detta genom att titta framåt, vi letar efter := eller om satsen tar slut innan. Här kan vi lätt se att om vi har otur så får vi leta länge, i princip godtyckligt länge. Tänk om vi kunde skiljt på olika token P och sett om det är en procedur, en variabel eller en fältvariabel (array), då hade vi ju också vetat vad sats det måste vara. Jo, men genom att vi har infört en särskild block och deklarationsanalysfas är det ju precis så det är. Vi har ju en symboltabell där vi kan söka upp denna information, vi kan således anse att deklarationsinformationen finns med i token P. Det är då lätt att fylla i punkterna i proceduren för sats tidigare, detta är ju nu inget principiellt nytt. Vi kan sedan genom att betrakta := som en tilldelningsoperator låta operatorpresedence ta hand om hela tillordningssatsen och vi behöver inte skilja på tillordning och proceduranrop i denna fas.

På sammat sätt kan vi nu översätta resten av grammatiken till procedurer i en parser fas. Detta är egentligen en ganska lätt uppgift, möjligen med två undantag. Det första är att om man vill ha optimal kod måste vi antagligen krångla till det en hel del, vi har ju inte räknat fram något härledningsträd som vi kan manipulera, vi kan naturligtvis fortfarande optimera i den genererade mellankoden. Det andra problemet är att vi endast har konstruerat våra procedurer för att hantera grammatiskt korrekt kod, vad händer om vi får in en felaktig ström av tokens. Detta är ett stort problem, som dessutom är väsentligt att lösa om inte användarna av kompilatorn skall bli mycket sura och irriterade. Målet med felhantering kan man formulera ganska kort. Man skall finna alla fel som finns i det aktuella programmet och inga andra, dvs inga följdfel. Detta är naturligtvis inte möjligt, men man kan försöka närma sig detta ideal. Ett förslag som på fullt allvar framförs i litteraturen är att

formulera en grammatik som innehåller felproduktioner. Man får på så sätt inga syntaktiskt felaktiga program, däremot genereras naturligtvis fel för de strukturer som egentligen inte ingår i språket. Problemet är att man knappast kan skriva en sådan grammatik. Man kan möjligen lägga till produktioner som fångar de vanligaste felen, men dessa är också de lättaste att klara av oavsett hur man gör. Den normala felhanteringsmetoden är att när man funnit ett fel så försöker man leta framåt i tokenströmmen tills man hittar något som gör att man kommer i takt igen och kan fortsätta översättningen. Se även särskilt kapitel om felhantering. Svårigheten är att finna ett sådant token utan att man hoppat över för mycket av koden och därmed missat en del av kontrollen.

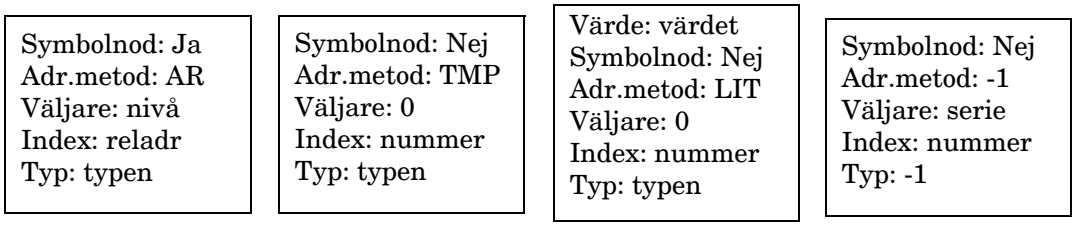
Operatorprecedens. Med hjälp av recursive descent kan man klara hela språket, men det känns lite onaturligt att analysera aritmetiska och booleska uttryck med denna metod. Här använder man sig ofta av operator precedens, som känns naturligare. Man kan här använda sig av metoder att generera en parser direkt från grammatiken med formella metoder. För att klara detta måste man dessutom formulera grammatiken för uttryck på en viss form. Enklare är att klara av detta för hand med hjälp av följande alternativa metod, som vid närmre eftertanke verkligen är en operator precedens metod. Vi startar med ett exempel, som vi utvecklar mer sedan. Vi kan börja med att först se, utan tanke på kompilering, hur man genom att från höger till vänster och läsa en enhet i taget, beräknar värdet av ett uttryck, 2+3*2+1. Vi använder oss av två stackar för att komma ihåg sådant som vi inte kan hantera direkt. Först läser vi talet 2, som vi inte ännu vet vad vi ska göra med, vi lägger det på en operandstack. Därefter läser vi operatorn +, men vi har ännu inga tal att addera, bara ett, varför vi lägger vårt + på en operatorstack. Sedan läser vi talet 3, vi har nu visserligen två tal att addera, men vi vet inte om vi ska göra detta, det kan ju komma en operator med högre prioritet senare. Vi lägger därför också 3 på operandstacken. Nu läser vi operatorn *, vi ser på toppen av operatorstacken och jämför prioritet och finner att eftersom * har högre prioritet än + så vet vi ännu inte vad vi ska göra. Vi lägger även * på operatorstacken. Vi läser nu 2 som efter liknande resonemang som tidigare hamnar på operandstacken. Nästa enhet som läses är + och den har lägre prioritet än * på stacken, varför vi nu utför operatorn * och tar bort den från stacken. Då vi utför * tar vi två operander från toppen av operandstacken, som bör vara 3 och 2, resultatet 6 läggs sedan överst på operandstacken. Nu har vi fortfarande ett + i handen och ett överst på stacken, då dessa har samma prioritet utför vi åter operatorn på stacktoppen, dvs +. Detta görs nu på operanderna 2 och 6, varefter resultatet 8 läggs på stacken. Vi har fortfarande ett + i handen, men en tom operatorstack, varför + läggs på stacken och vi läser nästa som är talet 1. När vi läser ett tal vet vi aldrig vad vi ska göra med det, varför vi lägger det på stacken. Nästa gång vi läser ser vi att det är slut, detta kan ses som att vi får en operator i handen som medför att vi alltid utför operatorn på stacken tills stacken är tom, då vi är färdiga med resultatet på toppen av operandstacken, i detta fall talet 9. Detta var en lång krånglig beskrivning för något så trivialt som att beräkna ett enkelt uttryck med +, * och heltal. Vi som människor använder oss dock av att vi på något sätt kan få en bild av hela uttrycket på en gång det kan inte ett program, dessutom har vi en något annan situation i kompilatorn, vi ska inte beräkna enkla uttryck, vi ska generera kod som beräknar uttryck, och inte bara enkla sådana. Den nyss beskrivna metoden kan modifieras och utvidgas till att klara av även detta.

Den kod vi ska generera beräknar uttryck ungefär som nu beskrivits, och vi kan efterlikna detta i kompileringsfasen. Vi använder oss av två stackar på samma sätt, det som läggs i operandstacken är då inte längre tal eller operander under exekvering av det genererade programmet. Det vi räknar med här som operander är i stället beskrivnin-

gar av operander, och operandstacken är en stack av operandbeskrivningar. Operatorerna under kompileringen är också beskrivningar av de operatorer som skall finnas under exekveringen av den genererade koden. Under kompileringen utför man en operatorbeskrivning med operandbeskrivningar som operander och resultatet är en ny operandbeskrivning. Som bieffekt genereras då också mellankod som skall göra motsvarande operationer i den exekverbara objektkoden.

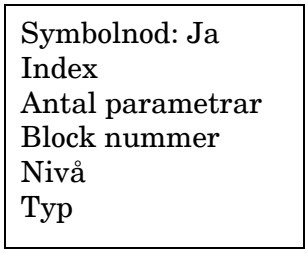
Vi återvänder till vårt enkla exempel men byter nu ut talen mot variabler: a+b*c+d. Vi kan nu upprepa hela beräkningsgången, och endast byta ut tex talet 2 mot variabeln a och vi lägger beskrivningen av a på operandstacken. Att utföra tex * är nu att dels generera kod för att multiplicera de operander som beskrivs med de två beskrivningarna överst på operandstacken. Vi får naturligtvis också bestämma oss för var resultatet skall finnas, tex i den temporära variabeln nummer 1. Vi gör då en operandbeskrivning av denna att lägga som resultat överst på operandstacken.

Vad skall då operandbeskrivningen innehålla? Det som behövs för att generera mellankod, såsom typ, adress. Detta betyder att i stort samma information som finns i en nod i symboltabellen. För varje variabel som operand i högnivåspråket måste vi söka upp den i symboltabellen. Detta för att vi ska få de uppgifter om variabeln som vi behöver för vår kodgenerering. En viktig del av denna information är variabelns minnesadress under körning. Redan i symbolnoden finns dess relativa placering i aktiveringsposten för dess block. Det som dock inte finns i symbolnoden, och inte kan finnas där är adressen till aktiveringsposten själv. Denna adress är först möjlig att bestämma under körning. Vad man kan veta är hur aktiveringsposten ligger relativt det aktuella (innersta) blocket just då man refererar variabeln. Det man kan ange är hur många statiska nivåer upp från där vi är, det block som variabeln finns i ligger. Vid enkel sökning som beskrivits tidigare betyder det att man söker upp variabeln i aktuellt blocks symboltabell, finns den där är relativa nivån 0, finns den inte där noterar vi en “miss” och söker vidare i symboltabellen för statiskt omgivande block. Detta fortsätter vi med tills vi finner variabeln, och vi noterar noggrannt antalet “missar”, den relativa nivån blir just antalet missar. Denna enkla sökning gäller i enkla språk med hierarkisk struktur, men utan objekt och andra komplikationer. Normalt har vi då ett litet antal sökregler, oftast en, i svårare språk kan vi dock behöva betydligt fler regler, kanske ett tjugotal eller fler. Som operander behövs inte endast variabler utan det tillkommer dock deklarationer av temporära variabler som ju inte finns i den normala symboltabellen och som adresseras på annat sätt. Liknande är då beskrivningar av literaler som operander. Ibland finns en referens

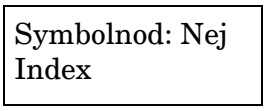


till symboltabellen med i operandbeskrivningen, ibland finns ingen sådan. Nivå betyder hur många gånger jag måste gå upp i den statiska kedjan innan variabelns eller funktionens definition bev funnen. Typen anger heltal, flyttal,....., nummer är interna nummer för att identifiera enheten. Vad gäller temporära variabler fungerar

dessa som en stack och nummreras 0,1,2,.. upptill så många som behövs. Reladr anger platsen i en aktiveringspost. Även operatorer beskrivs av en nod. Alla operatorer har ett



Användarfunktion

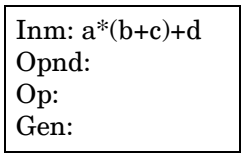


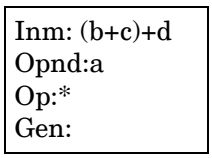
Operator (tex +)

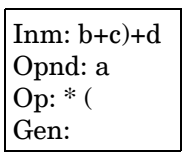
index som används som ingång i en precedensmatris. En användarfunktion har ganska mycket uppgifter, hur skulle man annars kunna generera ett funktionsanrop, medan en inbyggd operator i stort endast har sin egen typ och nästan ingen övrig information. Notera att även en användarfunktion har en nivåskillnad, som beräknas på samma sätt som för variabler. Denna nivåskillnad behövs för att RTS ska kunna "hänga in" funktionens aktiveringspost i rätt statiska omgivning, den relativa nivån används som parameter till RTS-rutinen för allokering. Detta ser vi längre fram i mellankodsexemplet för funktionsanrop (PALL). Naturligtvis behövs det en hel del andra nodtyper än vad som beskrivs här, vilka naturligtvis även beror på vilket högnivåspråk man kompilerar.

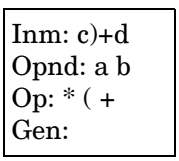
Idén att använda två stackar underlättar en hel del av hanteringen, men tyvärr innebär det en särkoppling av operandsekvens från operatorsekvens. Detta medför att a + b och a b + kommer att uppfattas på samma sätt, dvs fullständigt felaktiga sekvenser kan bli "legala". Detta måste naturligtvis förhindras, en enkel metod för att lösa problemet är att funktionen "ge mig nästa enhet" får bli lite mer intelligent, dvs får testa vad som får följa olika typer av enheter. Detta är ganska enkelt att definiera med hjälp av en "följematris", dvs ett schema som beskriver vad som kan följa efter olika enheter. Med hjälp av detta kan man filtrera bort felaktiga sekvenser redan vid inmatningen.

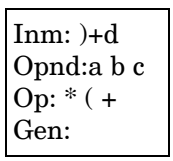
Vi går nu över till svårare uttryck. Antag vi har a*(b+c)+d, parenteser används då för att förhindra de normala prioriteterna att bestämma beräkningen. En vänsterparentes kan då betraktas som en operator, som i princip fungerar som en tillfällig stackbotten och en högerparentes som ett tillfälligt uttrycksslut. När man ser" (" på stacktoppen och har ")" i handen så utför man "(" genom att helt enkelt ta bort den från stacken och kasta bort vad man har i handen och läsa nästa. Komplikationen är således inte särskilt stor. Vi ser nu stegvis vad som händer i detta uttryck. I en del fall är stegen triviala, tex när

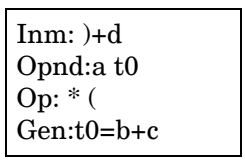


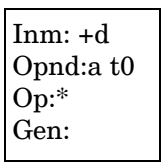


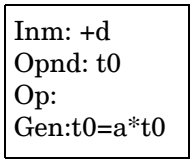


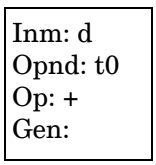


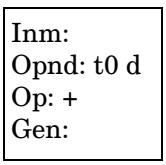


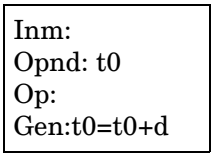












operander läggs på stacken, då har jag slagit samman två steg i beskrivningen ovan. Vi antar nu att vi inför funktioner i uttryck, a+f(b,c)+d. Här kan vi då betrakta f som en

operator, som till skillnad från de systemdefinierade såsom + nu är användardefinierade. På operatorstacken lade vi operatorbeskrivningar och vi brydde oss inte så mycket om hur dessa såg ut, var det ett + så räckte detta, dess funktion var ju känd av kompilatorn redan när kompilatorn skrevs. De användardefinierade är ju lite annorlunda, men symbolnoden beskriver f och det är således ungefär den informationen som duger som operatorbeskrivning. Hur skall nu prioriteten mellan + på stacken och f i handen vara? Jo sådan att f hamnar på stacken, vilket gäller oavsett operator på stacktoppen. När sedan f ligger i toppen av stacken och vi har + i handen så är det f som skall utföras. Mellan f och nästa operator +, ligger ju parameteruttrycket (b,c), hur klarar vi detta? Enklast är att betrakta parenteserna på samma sätt som tidigare och hantera uttrycken som parametrar med lite trick. Detta går rätt bra men om man komplicerar det ytterligare får man problem, särskilt om man får fel antal parametrar eller nått annat svårt. Vi tar nu och betraktar själva anropet f(b,c) för sig själv. Först ser vi vad som bör göras under exekveringen. Detta beskrivs naturligtvis mera i detalj under kapitlet om RTS, men ungefär följande verkar lämpligt: Allokera minne för f, beräkna och överför parametrarna, exekvera funktionen, ta hand om resultatet. Ett effektivt sätt att hantera detta är följande: Vi lägger f på operandstacken som vi sagt tidigare samt genererar kod för allokeringen. Detta med att generera kod vid stackningen av en operator är nytt. När vi nu får en vänsterparentes och har en funktion i stacktoppen stackar vi inte parentesen utan vi stackar i stället en “funktionsparentes”. Denna “superparentes” har ytterligare egenskaper än en vanlig parentes, den fungerar visserligen som stackbotten men den innehåller också en parameterräknare som håller ordning på vilken parameter vi håller på att överföra och kan då också utföra felkontroll. När vi ser ett kommatecken fungerar detta som ett uttrycksslut och när man kommer ner till funktionsparentesen förorsakar kommat en parameteröverföring. Den avslutande högerparentesen fungerar som vanligt ända tills man träffar på funktionsparentesen i stacktoppen, då överför man sista parametern och i övrigt som vid vanligt “parentesmöte”. När sedan funktionen utförs så genererar man funktionsanrop och hantering av resultat. Vad händer om vi komplicerar ytterligare med kapslade funktionsanrop? Tex a+f(b,g(c,d),a)+b, detta får läsaren som hemläxa. Som tips kan sägas att med ovanstående metod så händer inte särskilt mycket, dvs komplikationen är synnerligen måttlig. Arbetar man enligt den första enkla skissen så ökar problemen med ökad komplexitet.

Man brukar beräkna en prioritetsmatris på så sätt att man har en matris med stackoperatorn som första index och handoperatorn som andra index och där ingångarna inte egentligen ger prioriteten utan en aktion som skall utföras. De vanligaste aktionerna är stacka handoperatorn, utför stacktoppsoperatorn eller fel. Även andra aktioner kommer att finnas, tex för hantering av parenteser, stackbottnar och uttrycksslut. Arbetet med att skriva en parser med hjälp av operatorprecedens är naturligtvis att konstruera denna matris och förstås att definiera vad en operator skall göra då den utförs. Som regel är detta förhållandevis enkelt då det gäller aritmetiska och logiska uttryck, men svårare om vi skall se den normala satsstrukturen. Hur fungerar while, if then och else som operatorer, det går att definiera men man brukar inte använda operatorprescedens för dessa. Hur ser då en sådan matris ut? Här ges ett exempel från ett “halvstort” språk, det språk vars grammatik finns som uppgiftsförslag i tillägg z. Först har vi index för ingångarna i matrisen, index 0 är reserverat för operander, men sådana är inte aktuella i just denna matris. Vi defini-

erar följande. De flesta operatorerna här är väl ganska självklara, men funktionspar-

1: tilldelningsoperatorn, :=
2: additionsoperatorer, + -
3: multiplikationsoperatorer, * /
4: jämförelseoperatorer, = <> <= < >= >
5: användarfunktioner
6: fält (behandlas som operator)
7: parameterseparator, ,
8: vänsterparentes, (används som “stackbotten”
9: högerparentes,)
10: funktionsparentes, (efter funktion
11: tom stack respektive slut på uttrycket.

entes är egentligen en vänsterparentes som har upptäckts precis efter en användarfunktion, man lägger då en funktionsparentes i stället för vänsterparentesen i stacken. En sådan “superparentes” känner till att man ska överföra parametrar och kan också hålla räkning på parametrar. Vissa operatorer, tex en högerparentes kommer aldrig att läggas på stacken, den finns bara “i handen” från inmatningen. Vi har här i detta exempel ett begränsat antal operatorer, vi saknar tex alla logiska operatorer, men även en del andra. Att lägga till dessa ändrar bara storleken på matrisen, men adderar ingen komplexitet till den. Det är först då vi utökar språket med operatorer för objekt och andra mera komplicerade strukturer som det händer något väsentligt. Dessa komplikationer ska vi bortse ifrån nu och ta upp i ett eget senare avsnitt. Vi övergår nu till att ge ett förslag till aktionssmatris. Principen med operatorprecedens är ju att man läser en

	1	2	3	4	5	6	7	8	9	10	11
1	E	S	S	S	F	S	U	S	E	E	U
2	E	U	S	U	F	S	U	S	U	E	U
3	E	U	U	U	F	S	U	S	U	E	U
4	E	S	S	E	F	S	U	S	U	E	U
5	U	U	U	U	?	?	U	C	U	E	U
6	U	U	U	U	?	?	U	S	U	E	U
7	E	E	E	E	E	E	E	E	E	E	E
8	S	S	S	S	F	S	E	S	P	E	E
9	E	E	E	E	E	E	E	E	E	E	E
10	S	S	S	S	F	S	T	S	L	E	E
11	S	S	S	S	F	S	?	S	E	E	A

E - Det är fel, hantera detta.
S - Nya operatorn till stacken, läs nästa
U - Utför operatorn på stacktoppen (gör pop)
A - Acceptera, översättningen klar.
P - Ta bort stacktopp (pop), läs nästa
F - Nya op. stackas, dess prolog utförs, läs nästa
C - Nya op. är (, stacka funk.parentes, läs nästa
T - Överför en parameter, läs nästa
L - Överför sista parameter, läs nästa
? - Ännu ej implementerat, behandlas som fel.

enhet från inmatningen, är det en operand så lägger man den på operandstacken, är det en operator man läst (“har i handen”) så går man in i matrisen med stacktoppsoperatorn som första och den i handen som andra index. I matrisen får man då vilken handling man ska utföra. Det normala är att man lägger den nya överst på stacken och läser nästa, eller utför man den operator som ligger överst, tar bort den och har kvar vad man har i handen och går åter in i matrisen. Fel är en om inte önskvärd så dock ganska normal handling, dessutom görs naturligtvis som regel en “accept” vid varje uttrycks översättning. De övriga är lite mer speciella, vilka sådana man behöver och hur de ska se ut är beroende på det språk man översätter. De ovan beskrivna specialhandlingarna kan väl betraktas som relativt normala.

Vi har ytterligare en sak man kan ha i uttryck, nämligen att i stället för enkla variabler och literaler har vi element i fält. I uttrycket b+a(c,d)+e antar vi att a är ett tvådimensionellt fält. Detta kan vi lösa genom att överföra det på funktionsmetoden ovan. Vi kan betrakta fältet a som en funktion som har indeces som parametrar och vars värde är motsvarande matriselement. Adressering av matriser och hur man gör vid “matrisanrop” kommer senare. Det är visserligen sannt att man kan betrakta det som en funktion, men man gör anropen lite annorlunda. Vi har beskrivit användarfunktioner, men vi har ofta också särskilda inbyggda funktioner, systemfunktioner som anropas på ett något annorlunda sätt. Dock själva hanteringen på operatorprecedensnivå är i allt väsentligt samma som för användarfunktioner.

Andra typer av uttryck som innehåller jämförelseoperatorer (<,>,=,...) eller logiska operatorer (and, or,...) ger inte upphov till någon annan typ av lösning. En vanlig operator som har ett litet annat mönster är tillordningsoperatorn (:=,:-,..). I de flesta språk har den tex inget värde, men det finns språk där den också har värde. Detta är dock inget problem. En annan egenskap är att högeroperanden är som för andra operatorer, men vänsteroperanden skall särbehandlas. Det är inte värdet som är operand, utan det är själva adressen till värdet som skall användas. Detta betyder naturligtvis att 2:=b och a+3:=b knappast kan vara tillåtna uttryck, vänsteroperanden har ingen lämplig adress, och naturligtvis skulle betydelsen i övrigt som regel också vara vansinnig. Dock kan naturligtvis a(b,c):=d vara ett tillåtet uttryck, om nämligen a är ett fält.

Operatorprecedens är ett enkelt och trevligt sätt för konstruktion av parsers för uttryck, men en ofta anförd nackdel är att det skall vara svårt med felanalysen. Som regel går det att klara detta, möjligen på ett sätt så man upptäcker ett fel lite senare i uttrycket än där det egentligen är. Felhanteringens detaljer lämnar vi till ett senare avsnitt. Man kan konstatera att felhantering alltid är svårt och knappast svårare vid operatorprecedens än med andra metoder.

Kontaktytor recursive descent och operatorprecedens. Gör man blandningen mellan dessa två metoder så betyder det att man i rec. desc. ska anropa operatorprecedens. Låt oss se på exemplet med if-satsen. Här översatte vi först ett villkor, med operatorprecedens, och fortsatte därefter med att generera kod från recursive descent rutinen. Detta betyder att vi måste överföra information från den ena delen till den andra, information som man nog skulle vilja hålla intern inom det objekt som sköter operatorprecedens. Detta kan man göra om man modifierar sina anrop något. Se på föl-

jande jämförelse. Här låter man all kod genereras av VILLKOR, inklusive hoppet till L1.

```
T1:=VILLKOR;
L1:=GenLabel;
GenCode(Hopp om T1 falskt till L1);
T:=GetToken;
```

```
L1:=GenLabel;
T:=VILLKOR(L1);
```

Om fler alternativ än hopp om falskt kan tänkas i språket så behöver man kanske sända med ytterligare information som parametrar. Andra exempel där lämplig "brottyta" mellan de två metoderna kan underlätta är t ex hur man skiljer på tillordning, och proceduranrop. Låter man alla sådana satser (dvs både x:=....; x(...):=...; och p(...);) gå direkt till operatorprecedens och att man betraktar := som en operator, så finner man ganska lätt, automatiskt, vad som ska göras. Tidigare har sagts att man behöver "Look ahead" för att veta vad som ska göras. Detta kunde i stället hanteras genom att vi hade en symboltabell att hämta tillräcklig information från. Här har vi nu ytterligare ett sätt att trolla bort svårigheten på, som visserligen även den förutsätter att vi kan söka i symboltabeller.

Exempel.

Vi går vidare på vårt exempel med språket Enkel. Vi analyserar nu ett block i taget och blockets koddel har nu en modifierad grammatik. De terminala symbolerna är nu tokens från tidigare faser, t ex *then* är nu ett odelbart token, som inte består av ett antal tecken längre, och som även innehåller information av olika slag. I grammatiken betraktar vi också identifierare som en terminal symbol och vi tar inte syntaktisk hänsyn till att det finns olika identifierare. Ett identifierartoken innehåller naturligtvis ytterligare information, t ex vilken identifierare det är, och detta tar vi naturligtvis semantisk hänsyn till. Terminala symbolen feltoken behandlas på liknande sätt, den innehåller naturligtvis information om vilket fel det gäller. Fel har genererats av deklarations och strukturanalysen. Denna fas påträffar inga fel i koden, som ju inte närmre analyseras, men kan däremot påträffa fel i deklarationerna. Ingående i idtoken kan vi också anse deklarationsinformationen vara, även om vi tekniskt hittar den genom att söka i symboltabeller.

block→ **feltoken*** sats^{+} **end**
sats→tillordning | villkorssats | **callblock | feltoken**
tillordning→idtoken **:=** aritm **;**
villkorssats→**if** villkor **then** sats **else** sats **;**
villkor→aritm relop aritm
aritm→aritm **+** term | term
term→term * faktor | faktor
faktor→**(** aritm **)** | **idtoken** | **taltoken**
relop→**=** | **<** | **>** | **<>**

Mellankod och mellankodsgenerering

Kompilatorns front-end är normalt sådan att allt beroende av maskinens arkitektur har tagits bort och allt som beror på högnivåspråket har tagits hand om här. Skall man skriva en portabel kompilator, dvs en som implementerar ett visst högnivåspråk på ett

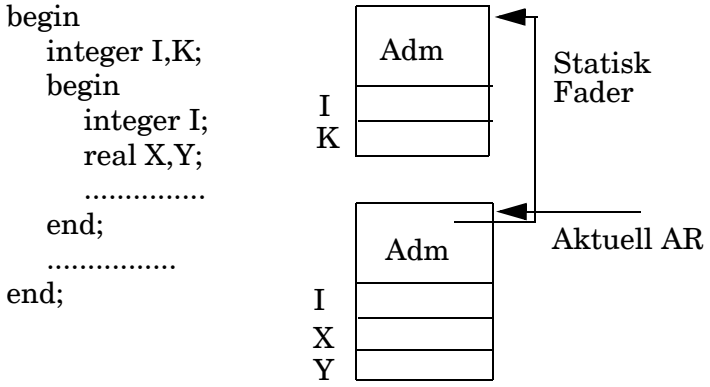
flertal olika datorarkitekturer, så kan man använda samma front-end oberoende av maskin. Allt som sedan beror på maskinarkitekturen tar man hand om i back-end. Man måste då ha ett klart definierat gränssnitt mellan front-end och back-end. Man representerar resultatet av front-end i form av en mellankod. Ofta presenterar vi denna i form av en läsbar text, men intern representerar vi denna naturligtvis på ett helt annat sätt. Avsikten är att vi naturligtvis inte ska behöva göra en ny analys, utan att vi direkt på ett enkelt sätt skall kunna generera den slutliga objektkoden. Mellankoden kan uttrycka resultatet dels direkt i trädform, men också som en sekvens av instruktioner som liknar ett maskinkodsprogram. Vi ska nu bortse från den interna representationen och i stället använda motsvarande textrepresentation när vi i fortsättningen talar om mellankoden.

Krav på mellankod.

Först ska vi försöka ange de krav man har på en mellankod. Ett krav är naturligtvis att man skall kunna representera källkodens innehåll på ett korrekt och helst också effektivt sätt. Under översättningens gång kastar man ofta bort information som finns i källkoden, men som man inte längre anser sig ha någon nytta av. Viss information är otvivelaktigt onödig i fortsättningen, tex ingående kommentarer, men troligen också annat såsom variablers faktiska namn och en del annat. Vilken information som behövs kan vara beroende av maskinarkitektur. Således bör mellankoden kunna representera all den information som någon arkitektur kan ha användning av. Ett annat krav är att man vill att front-end, som ju är gemensam för alla maskiner, skall göra så mycket som möjligt av översättningsarbetet och på så sätt underlätta konstruktionen av en back-end. Gör man mycket av översättningen kommer man att förlora mycket information, som man skulle behövt för att fullt kunna utnyttja aktuell maskin. Det finns således en konflikt mellan att lägga mellankoden på en hög nivå nära källkoden och på en låg nivå nära en maskinarkitektur. Vad är då lösningen? Enkelt uttryckt, låt mellankoden vara som en maskinkod som skulle kunna fungera på alla aktuella arkitekturer. Man lägger den således så nära maskinnivån som möjligt utan att man kommer för nära någon särskild maskin. Ett av problemen med detta är naturligtvis att då man konstruerar en front-end och bestämmer sig för ett mellankodsformat så känner man inte till alla de maskiner som kommer att bli aktuella. En del av dessa är troligen ännu inte konstruerade eller ens planerade!

Mellankoden är naturligtvis också beroende av hur man tänker sig exekveringen med Run-time system och främst minneshantering. Både RTS och back end tas upp i kapitel 4, där det närmre utseendet av template och aktiveringspost (AR). Här ges endast en kort allmän beskrivning. Varje block måste under exekvering ha ett min-

nesutrymme tillgängligt. I figuren ser vi två kapslade block av enkel typ och de motsvar-



ande aktiveringsposterna då vi exekverar i det inre blocket. Vi ser att man kan nå data (variabler I,X,Y) i det inre blocket, samt att vi kan nå variabler i det yttre via pekaren till statisk fader (SF). Om vi tänker oss ett block som representerar en procedur så innehåller den statisk information som alltid är lika, oberoende varifrån och hur många gånger proceduren anropas. Denna statiska information lägger man i en template. Varje gång en procedur anropas skapas en instans av blocket. Då en procedur kan anropas rekursivt kan det samtidigt finnas flera instanser av samma block. Varje instans måste naturligtvis ha ett eget minnesutrymme för de lokala variabler som finns i blocket. Vid ett givet tillfälle har således blocket en template och ett antal AR allokerade i minnet. Detta betyder att AR måste allokeras dynamiskt i ett minnesutrymme. AR innehåller dels den dynamiska information som behövs för att administrera blockets instans, dels utrymme för data (variabler, parametrar o dyl). Mellankoden måste ha ett sätt att adressera data i en aktiveringspost. Lokala variabler ligger i den aktuella aktiveringsposten, men man skall också kunna adressera variabler som ligger i blockets statiska omgivning, dvs i dess aktiveringsposter. Varje aktiveringspost bör då ha en pekare till den AR som representerar dess statiska faderblock. Innehåll och hantering av template och AR ges under RTS-beskrivningen.

För att kunna adressera data i en AR måste vi dels veta var AR ligger i minnet, vilket är dynamiskt och man håller en pekare under körning som pekar på aktuell AR (CUR), dels måste man känna den relativa positionen inom AR, vilket kan beräknas av kompilatorn. För att vi skall kunna beräkna dessa relativadresser måste kompilatorn känna till de olika datatypernas utrymmesbehov, vidare måste man känna till om maskinen kan adressera data av olika storlek på godtyckliga byteadresser eller vilka regler (alignment) som gäller. Detta kan naturligtvis inte front-end veta, så länge vi kräver att denna skall vara arkitekturoberoende. Man kan då tänka sig två metoder att adressera data från mellankoden. Första sättet är att man endast numrerar variablerna(0,1,2,...), man måste då också på något sätt i mellankoden skicka typinformation till back-end. Kompilatorn klarar fortfarande att beräkna positionen inom AR, men verkliga relativadresser kan först beräknas i back-end och med ganska besvärliga regler. Fördelen med detta är att vi fortfarande har möjlighet att med full effektivitet klara adresseringen och samtidigt hålla front-end maskinoberoende. Nackdelen är naturligtvis komplexiteten hos back-end, särskilt gäller detta när objekt introduceras med full flexibilitet. Den andra metoden är att man säger att man visserligen inte vet hur stora ett heltal, flyttal eller adresser är, men vi kan ange dem i ett gemensamt mått. Vi vet tex att ett heltal och en adress är lika långa, att ett långt flyttal är dubbelt så långt, dvs vi känner

inbördes storleksförhållande. Vi kan då fortfarande ge dataadresser i mellankoden i ett arkitekturoberoende format utan att beräkningen i back-end blir särskilt besvärlig. Problemet med alignment och inflexibiliteten mellan data av olika typer kan sätta ned effektiviteten i den genererade objektkoden något. Om vi inte är speciellt intresserade av väldigt flexibel och effektiv kod använder vi den andra metoden.

Mellankoden behöver också kunna adressera data på annat sätt än inom aktiveringsposter. Det mesta ligger visserligen i något AR, men man behöver också kunna adressera literaler, dvs konstanter som finns i källkoden. När kompilatorn lägger ut kod för att beräkna uttryck behöver man ofta utrymme för mellanresultat, detta måste man också kunna adressera. Detta bukar kallas temporära variabler, de finns inte i källkoden utan genereras av kompilatorn själv. Ofta försöker man i back-end lägga sådana temporära variabler i register, varför man ofta kallar dessa just “register”. Vi skall försöka att kalla dem temporära variabler och att se deras allokering till eventuella register som en implementationsdetalj i back-end. Finns det då annan data att adressera? Egentligen inte om man bortser från att vissa mellankodsinstruktioner behöver adressera templates och aktiveringsposter i stället för data inom sådana.

De aktiveringsposter som finns i aktuellt block och dess omgivande block säges ligga i en aktiv kedja av blockinstanser, medan objekt som är en blockinstans som ej längre är aktiv inte finns i denna kedja, men som har en pekare på sig från något annat AR. Objekt adresseras genom att man har adressen till AR i en temporär variabel, i övrigt sker adresseringen som för en normal aktuell aktiveringspost.

Exempel (prog1).

Vi tittar på den mellankod som vi genererat från programmet prog1 som vi hade som exempel vid deklaration och strukturanalysen tidigare. Även om mellankoden normalt

LABEL	S0	Start av yttre fiktivt block
PALL	T0,0	Som allokerar yttre block,
PEX		anropar det
DALL		och avallokerar då vi återvänt
RETURN		återvänd från fiktivt block (Slut!)
START	S0	Talar om var vi ska starta.
LABEL	B0	Här börjar kod för yttre block (#0)
LITERAL	id0,int,5	Definierar literalen 5, ett heltal.
MOVE	LIT,0,id0,int AR,0,0,int	Flytta 5 till I, i aktuell AR.
LITERAL	id1,int,1	Definiera literalen 1.
ADD	AR,0,0,int[1)] LIT,0,id1,int TMP,0,1,int	I+1 till tempvariabel (t1) [1)]
MOVE	TMP,0,1,int AR,0,1,int	Lagra t1:s värde i var K, i AR.
PALL	T1,0	Allokera inre block (#1),
PEX		anropa det och
DALL		avallokera då vi återvänt.
MUL	AR,0,0,int AR,0,1,int TMP,0,1,int	Beräkna I*K till temp (t1)
MOVE	TMP,0,1,int AR,0,1,int	överför t1 till K
RETURN		Återvänd från blocket (#0)
LABEL	T0	Template för block #0
GENWORD	2	Minnesbehov för 2 variabler (I,K)
GENADR	B0	Adress till koden.
LABEL	B1	Här börjar kod för inre block (#1)
LITERAL	id2,int,4	Definiera literalen 4.
MOVE	LIT,0,id2,int AR,0,0,int	Flytta 4 till K, i AR.
ADD	AR,1,0,int[2)] AR,0,0,int TMP,0,1,int	Beräkna I+K, värde till temp (t1)[2)]
MOVE	TMP,0,1,int AR,1,0,int	Flytta värdet (t1) till var I, i AR.
RETURN		Återvänd från block #1.
LABEL	T1	Template för block #1
GENWORD	1	Minnesbehov för en variabel (K)
GENADR	B1	Adress till koden.
END		Fysiskt slut på programmkoden

[1)]Lägg märke till att I i det yttre blocket har adress AR,0,0

[2)]Här adresseras I med AR,1,0, dvs det är ju I i det yttre blocket som adresseras från det inre blocket. I finns således en nivå upp. K är här den lokala variabeln som definieras i inre blocket, dvs adress AR,0,0. Jämför med adressen till K i yttre blocket, således inte samma relativadress.

är en sekvens av heltal skrivs den här på ett lite mer läsbart sätt. Operationer har fått namn, även vissa parametrar har fått namn och lägen anges inte med två tal 2,1 anges som T1. Exekveringen av nya block görs genom att man först måste allokera minne till blockets variabler (PALL), sedan exekverar man blockets kod (PEX). När man så småningom återvänder (RETURN) så skall blockets minne avallokeras (DALL). Alla dessa saker görs med hjälp av rutiner i RTS. De är något mer komplicerade än man kan tro, bla måste man hantera minnet samt bygga upp de strukturer med AR som är nödvändiga för adressering av variabler i omgivande block. Ovanstående uppdelning av PALL-PEX klarar också proceduranrop, för vanliga block kunde det förenklats till ett

enklare anrop av en annan RTS-rutin. Fördelen med detta är att samma teknik kan användas för mer komplicerade anrop än bara nya enkla block, dvs även för procedurer och objekt.

Hantering av funktionsanrop.

Användardefinierade. Dels anropar vi subrutiner, dvs rena procedurer som ej ger något värde, dels anropar vi funktioner som returnerar ett värde. I språk som java finns inga procedurer och funktioner, men metoder som ger eller inte ger ett värde är naturligtvis samma sak. Skillnaden mellan funktion och subrutin är inte särskilt stor, ser vi på hur en funktion anropas så kan man enkelt göra motsvarande för subrutiner. Vid funktionsanrop måste vi också hantera att de sker inne i en beräkning av ett uttryck. Man kan dela upp ett funktionsanrop i flera delar. Först måste vi se till att funktionen har ett minnesutrymme, en aktiveringspost, att lägga lokala variabler, parametrar och resultat i. Man kan naturligtvis överföra parametrar i "register" eller i temporära variabler, ett mera flexibelt sätt är att överföra dem till funktionens aktiveringspost (AR). Detta betyder att man då måste allokera en sådan innan man överför parametrarna. Vi har tidigare i operatorprecedensbeskrivningen påpekat att så fort man stackar funktionen som en operator så utför man också en prolog. Denna prolog kan generera ett anrop till RTS för att allokera AR. Man måste dock inse att denna nyallokerade aktiveringspost inte är den aktuella (CUR), som syns vid beräkning av parametrarna. Vi är kvar i tidigare omgivning, beräknar parametrar en efter en och överför dem till aktiveringsposten för funktionen (se RTS). När alla parametrar är beräknade och överförda så ska funktionen anropas. Även detta görs via RTS, som nu gör funktionens AR till det aktuella (sätter CUR), samt hoppar till funktionens kod. När vi kommer tillbaks från funktionen så måste vi ta hand om resultatet, varefter vi avallokerar funktionens minnesutrymme (AR). Vi kan ge en försmak på den genererade mellankoden ("pseudomellankod"). Vi antar vi anropar en funktion vars definition utgörs av block n,

```
PALL    Tn, m
        Överför parametrar
SAVE    p
PEX
        Överför resultatet
PDALL
```

alla rutiner och funktioner definieras ju som ett block. PALL är en mellankodsoperation som betyder ett anrop till en RTS-rutin för att allokera minne för ett block. Som parametrar har man först adressen till blockets template, dvs dess statiska information som ger de uppgifter RTS behöver för allokeringen. Som andra parameter sänder man adressen till statisk fader. Med "adress" menas här hur långt upp i kedjan av statiska omgivningar som funktionen är definierad. Detta behöver RTS veta för att kunna bestämma vilken AR som representerar funktionens statiska omgivning. Observera att statiskt omgivande block är en dynamisk information! Det kan ju samtidigt finnas flera aktiveringsposter aktiva för ett visst block (procedur), statisk fader kan ju endast vara ett av dessa, denna information är ju dynamisk. Nu beräknas parametrarna i en omgivning där inte den nyallokerade AR syns. Parametrarna överförs till detta AR, som således syns "lite", dvs för själva överföringen. Instruktionen SAVE förklaras senare i denna beskrivning. PEX är en instruktion som är ett anrop av en RTS-rutin för att hantera själva "hoppet" till funktionens kod. När man kommer tillbaks har man naturligtvis inte bara exekverat RTS-rutinen, utan också funktionens kod. Vi har då kvar funktionens AR, men synligt endast för att överföra resultat. Vi ser här att ville man ha en funktion som samtidigt gav flera resultat så är det närmast trivialt att

implementera detta när man gör så här. När resultatet överförts så anropar man RTS för att ta bort funktionens minne (AR).

Vi har nu en komplikation, anropet kan ju komma mitt i beräkningen av ett uttryck, där vi kan ha ett antal temporära variabler i användning. Funktionen själv använder troligen också ett antal temporära variabler vid sin exekvering. Om vi bara ser på temporära som en stack som finns vid exekveringen skulle vi ju inte behöva bry oss om detta, vi jobbar ju bara ovanför på stacken och har tagit bort detta tills vi återvänder. Nu är dock problemet att vi försöker använda register i maskinen för att effektivisera användandet av temporära variabler. Denna registerhantering i funktionens kod vet ju inget om varifrån den anropas, vilket ju kan vara från flera helt olika ställen. Man måste således tala om för mellankoden att se till att de temporära variablerna verkligen hanteras som stack vid anropet. Instruktionen SAVE kan göra detta.

En komplikation som kan uppkomma i språk som kan hantera någon form av parallellitet eller kvasiparallellitet är att man kan avbryta beräkningen mitt i en funktion i ett uttryck, för att fortsätta att beräkna i ett annat uttryck. Sedan kan man växla mellan dessa. I språk för realtid byter man verkligen omgivning med separata stackar, medan man i språk som Simula, med just kvasiparallella system byggda kring objekt hanterar man detta genom att spara undan stackar separat i aktiveringar av objekt. Detta är inte något som man bör ta allvarligt på som nybörjare i kompilatorteknik.

När man beräknar parametrar att överföra till en funktion kan detta ju medföra att man måste beräkna ett uttryck, där det finns ett nytt funktionsanrop, med parametrar. Detta kan naturligtvis fortsätta till godtyckligt djup. De nya funktioner man anropar kan vara samma, eller någon annan, vilket saknar betydelse. Observera att detta inte är något rekursivt anrop. Mellankodsmönstret blir som följer. Vi ser här att de nya anro-

```
PALL    Tn, m
        Överför parametrar
        PALL    Tn, m
                Överför parametrar
                .............
        SAVE    q
        PEX
                Överför resultat....
        PDALL
SAVE    p
PEX
        Överför resultatet
PDALL
```

pen av funktioner sker enligt “stackprincipen”, dvs vi återkommer alltid på ett snyggt sätt och nya anrop ligger naturligtvis helt inne i ett annat. Med den visade mellankoden och med de metoder som beskrivs i kapitlet om RTS så löser sig problemen utan att man egentligen upptäcker dem.

Exempel. Här ges ett enkelt exempel med anrop av en funktion, Vi antar att språket utökats med funktioner. Vi ser här det lilla enkla programmet och ska se närmre på den genererade mellankoden. Vi använder samma pseudokodsformat som tidigare.

```
begin                                                  Här finns ett enkelt program som definierar en
      integer N;                                       funktion, och som använder denna. Observera
      integer function Nfak(I); integer I;             att funktionen är definierad rekursivt.
      begin
          if I=0 then Nfak:=1
          else Nfak:=I*Nfak(I-1);
      end;
      N:=5;
      N:=Nfak(N);
end
LABEL       S0                                         Start och inledning av programmet är samma
PALL        T0,0                                       som i det tidigare exemplet.
PEX
DALL
RETURN
START       S0                                         Här startar yttre blocket.
LABEL       B0
LITERAL     id0,int,5
MOVE        LIT,0,id0,int AR,0,0,int
PALL        T1,0                                       Allokera minne för Nfak(T1), def på samma nivå
MOVE        AR,0,0,int AR,-1,1,int                     Överför parameter, -1 för nyallokerad AR
SAVE        0                                          Till registerhanteraren spara reg (0 st)
PEX                                                    Anropa Nfak
RESTORE     0                                          Återställ register
MOVE        AR,-1,0,int TMP,0,1,int                    Tag fram resultatet, -1 anger "gammalt" AR
MOVE        TMP,0,1,int AR,0,0,int                     Lägg värdet i N
DALL                                                   Avallokera detta nu.
RETURN
LABEL       T0
GENWORD     1
GENADR      B0

LABEL       B1                                         Här är koden för Nfak
LITERAL     id1,int,0                                  Detta block fungerar på normalt sätt.
JNE         AR,0,1,int LIT,0,id1,int L0
LITERAL     id2,int,1
MOVE        LIT,0,id2,int AR,0,0,int
JUMP        L1
LABEL       L0
PALL        T1,1                                       Anropa Nfak(T1), här definierat en nivå upp
LITERAL     id3,int,1
SUB         AR,0,1,int LIT,0,id3,int TMP,0,1,int
MOVE        TMP,0,1,int AR,-1,1,int
SAVE        0
PEX
RESTORE     0
MOVE        AR,-1,0,int TMP,0,1,int
DALL
MUL         AR,0,1,int TMP,0,1,int TMP,0,1,int
MOVE        TMP,0,1,int AR,0,0,int
LABEL       L1
RETURN
LABEL       T1
GENWORD     2
GENADR      B1
END                                                    Slut.
```

Systemdefinierade. Ett högnivåspråk brukar tillhandahålla ett antal standardfunktioner eller standardprocedurer. Dessa är fördefinierade och är naturligtvis oftast implementerade direkt i RTS eller ett liknande bibliotek, i form av maskinkod. De är ofta skrivna direkt i assemblyspråk eller i C. De har ett enklare anropsprotokoll än de användardefinierade funktionerna, minneshantering, rekursion och liknande kan ju hanteras på annat sätt. Det normala är att man använder ett C-protokoll vid anrop. Detta betyder att man lägger ut en anorlunda mellankodssekvens än för de användardefinierade. Man överför parametrar till "riktiga" register och gör ett enkelt subrutinanrop (SPARC gör ett call). Systemdefinierade funktioner kan ha en form av "inlödda deklarationsnoder" som motsvarar de användardefinierades symbolnoder. Dessa kan antingen sökas efter på ett speciellt sätt, eller lägger man in dem i en deklarationsstruktur hörande till ett fiktivt omgivande block (-1). Eftersom namnen på standardrutiner normalt inte är reserverade ord i högnivåspråket måste man göra en normal sökning, som avslutas med sökningen i den speciella omgivningen. Ofta låter man parametrarnas typer samt i förekommande fall resultatets typ vara mer komplicerat än vad vi är vana vid för användarfunktioner. Med komplicerad typ menar jag inte komplexa objekt, utan att man kan ha olika typer vid olika tillfällen. Vi kan se på en enkel standardrutin *write(p)*, där p kan vara av typ integer, real eller text. Egentligen är det då olika typer av write som anropas, beroende på den aktuella parameterns typ. Man kan betrakta högnivåspråkets write som en lista av funktioner, där den enskilde funktionen bestäms av parameterns typ. Även överföringen av parameter bestämms av parameterns aktuella typ. Vi ger ett litet enkelt exempel på den mellankod som kan genereras för anrop av systemdefinierade rutiner. Lägg märke till att namnen på de i högnivåspråket inbyggda

```
begin                                   Det enkla programmet.
      integer I;
      I:=readint;
      write(I);
end
EXTREF      S1,iread                    Definiera namn på externa rutiner
EXTREF      S2,iwrite
LABEL       S0                          Programstart
PALL        T0,0
PEX
DALL
RETURN
START       S0

LABEL       B0                          Här startar yttre blockets kod.
JUMPR       S1                          Vi anropar S1, dvs iread
MOVE        REG,0,8,int TMP,0,1,int     Denna ger ett värde i speciellt register
MOVE        TMP,0,1,int AR,0,0,int      Kunde naturligtvis optimerats!
MOVE        AR,0,0,int REG,0,8,int      Överför parameter i speciellt register
JUMPR       S2                          Anropa rutinen S2, dvs iwrite
RETURN
LABEL       T0
GENWORD     1
GENADR      B0
END                                     slut
```

rutinerna inte automatiskt knyts till motsvarande namn i mellankod och RTS. Den knytning som behövs görs av kompilatorn.

Hantering av temporära variabler.

Temporära variabler adresseras med hjälp av ett löpande nummer, eventuellt kan man tänka sig flera serier av temporära variabler, man skiljer ibland på temporära för flyttal, adresser och annat (tex heltal). Det finns all anledning att kompilatorn hanterar denna sekvens av temporära variabler som en stack. Detta underlättar för back-end att ge effektiv kod, och det är ett naturligt sätt för kompilatorn att arbeta. Vi antar att vi ska använda temporära variabler för att hålla mellanresultat medan vi beräknar ett uttryck, vidare försöker vi inte optimera koden med att hålla deluttryck tillgängliga mellan olika uttryck. Vi bestämmer oss då för att se de temporära variablerna som en stack av mellanresultat, där vi kan lägga värden av alla typer i samma stack. All allokering av verkligt minnes eller registerutrymme handhas av back-end, här ser vi det endast som att det finns en stack som effektivt kan hantera våra problem. Det är väsentligt att front-end håller sig till vissa regler om hur denna stack hanteras. I operationer där vi använder en inoperand kan denna ligga överst på stacken, aldrig längre ned. I operationer, som add, där vi har två inoperander, kan dessa båda ligga som de två översta elementen på stacken, eller ligger bara den ena och då överst på stacken. Vi antar vidare att så fort en operand på stacken har använts är den inte längre aktuell och kan således tas bort från stacken. Vi ser nu på en operation av följande utseende: *op* $opnd_1$,$opnd_2$,$opnd_{res}$. Vi antar att operander kan läggas i minnet (AR), i temporär variabel (S), eller kan finnas som literal (L). Märk också att vi antagit att stacken, temporär variabel, kan hålla värden av alla typer. Vi är således oberoende av typ. Vi vill då se vilka regler som gäller, och kan åskådliggöra detta med en matris. När vi gör operationen antar vi vidare att vi

$Operand_1$ \ $Operand_2$	AR	S	L
AR	S_{ny}	S_0	S_{ny}
S	S_0	S_r	S_0
L	S_{ny}	S_0	L_{ny}

använt de temporära variablerna t_1,t_2,t_3,...,t_p. Vi ska då se hur denna stack av temporära hanteras i de olika fallen.

S_{ny} Här lägger vi resultatet överst i stacken, Vi behöver således en ny temporär variabel. Stacken blir: t_1,t_2,t_3,...,t_p,t_{p+1}.

S_0 Här tar vi bort en operand från stacken och lägger resultatet på stacken. Således "återanvänder" vi den temporära variabeln. Stacken blir:t_1,t_2,t_3,...,t_p, dvs oförändrad i storlek.

S_r Här har vi båda variablerna i stacktoppen, som vi tar bort, varefter vi lägger resultatet i stacken. Stacken blir:t_1,t_2,t_3,...,t_{p-1}. Här reduceras således stackens storlek med en temporär variabel.

L_{ny} Här har vi två literaler som operander. Detta kan ju kompilatorn själv klara av att beräkna, samt generera en ny literal. Man behöver inte generera någon kod och man behöver inte lagra något mellanresultat vid körning. Ofta

ignorerar man detta och utför operationen på vanligt sätt, då väljer man $L_{ny}=S_{ny}$.

Operatorer som endast har en ingående operand kan följa liknande regler, tex Move flyttar ofta från temporär till något annat, denna temporära kan då naturligtvis tas bort från stacktoppen. När man beräknar ett uttryck och behöver ett antal temporära variabler så kan det naturligtvis ingå funktionsanrop som komplicerar det hela. Följer man dock vad som sagts om funktionsanrop behöver man inte bry sig här, eventuella problem tar man då i back-end. Har man operationer med fler operander kan man naturligtvis hantera detta på motsvarande sätt, vi förutsätter endast att används n operander från stacken så är det de n översta.

Hur stor blir nu stacken av temporära variabler i ett uttryck? Man kan enkelt säga att man får en ny för varje prioritetsnivå av operatorer. Detta betyder att man i de flesta fall har noll, en eller möjligen två temporära samtidigt. Detta verkar ju enkelt, då behöver man ju inte göra så mycket åt att tex register inte räcker till. Oturligt nog är det dock så att man genom att använda parenteser i uttryck så kan man skapa nya prioritetsnivåer. Det betyder att hur många temporära du än har i ett visst uttryck så kan man alltid skriva ett uttryck som behöver en till! Sådana märkliga parentesuttryck lär dock inte uppträda normalt, annat än möjligen för att testa kompilatorer. Man måste således kunna hantera det men man kan ta lätt på effektiviteten. Se tex på uttrycket (x+x)((x+x)((x+x)((x+x)(x+x)))), fortsättningen är inte svår att se.

Naturligtvis behöver vi regler för vilken typ ett resultat ska ha vid en viss operation. Denna typ är oftast beroende av de ingående operandernas typer, samt ibland även av operationen. Vidare är detta också beroende av hur man definierat semantiken hos det språk vi översätter.

Literaler.

Literaler är konstanter som finns direkt i källkoden och som ofta kan läggas direkt i maskinkoden. Då de kan vara av lite olika typ och storlek finns de inte direkt i instruktionen, utan i en kompilatortabell där de identifieras med ett nummer (index, identitet). Som vi sett ovan behöver inte alla literaler finnas i källkoden, kompilatorn kan själv generera nya. Som vi såg ovan så kan naturligtvis 3+2 ge upphov till den nya literalen 5. Denna literaltabell måste naturligtvis också representeras på något sätt i mellankoden.

Mellankod.

Vi har nu kommit fram till att försöka definiera själva mellankoden. Detta gör vi dock inte här utan hänvisar till kapitlet om back-end. Dessutom finns ett exempel på “mellanspråk” i tillägg A . De flesta högnivåspråk tillåter att man anropar rutiner skrivna i ett främmande språk. Då måste kompilatorns back-end lägga ut anrop och parameteröverföringar på ett sätt som bestäms av det andra språket. Kompilatorn har ju ingen möjlighet att ha kontroll av vad som ett annat språks översättare gör. Detta betyder att man får införa instruktioner för detta i mellanspråket, dessa instruktioner behöver ofta ta med sig mycket mer information än vad som annars behövs, detta för att kunna lägga ut kod som ju egentligen dikteras av ett helt annat och för front-end helt okänt system.

Mer avancerad kompilering.

Vi har hittils främst beskrivet ett okomplicerat hierarkiskt språk, utan objekt och parallell exekvering. Vidare har vi antagit att hela programmet kompileras som en enhet. Genom att tillföra objekt, parallellitet och separatkompilering kommer vi att komplicera kompilatorn ganska mycket. Separat kompilerade moduler gör det inte särskilt mer komplicerat, medan objekt betyder lite mer. Grunden kan vi visserligen använda i samtliga fall, men vi får fler komplicerade operatorer, fler sökregler för variabler, jobbigare hantering av temporära variabler och andra komplikationer. Denna skrift behandlar främst det vanliga enkla språket och att som nybörjare göra en kompilator för detta, men här ska ändå ges en kort introduktion vad gäller besvärligare konstruktioner.

Separat kompilering.

När program blir stora är det svårt att hantera som en enda sammanhängande textmassa att översätta på en gång. Första problemet är att behöva hantera all programtext på en gång, tex. vid editering, utskrifter och liknande. Detta brukar man lösa genom att tillåta en konstruktion *include*, vilket betyder att man rent textmässigt inkluderar en text från en annan fil på angiven plats. I java finns inte denna möjlighet, men i de flesta andra språk brukar det finnas. Observera att detta inte har något med separat kompilering att göra, all text kompileras fortfarande sammanhängande vid ett tillfälle. Det är visserligen till stor hjälp för att praktiskt hantera textmässigt stora program, men för en kompilatortillverkare är det naturligtvis fullständigt ointressant. Det påverkar inte kompilatorn och kompileringen på något sätt. Det löser inte heller problemet med att man måste kompilera om allt även vid en mindre och begränsad ändring av programmet. För att lösa detta behöver man kunna kompilera mindre programmoduler separat, utan att behöva bry sig om övriga moduler av programmet. Vad ska vi då mena med programmodul? Kan det vara ett kodavsnitt vilket som helst, eller behöver man mera restriktioner för att kunna hantera det. Normalt låter man en programmodul utgöra en klass, som i java, eller tillåter man att en modul utgör en procedur. Vi kan här begränsa oss till att en modul är ett namngivet block. Detta betyder just klass eller procedur eller något som genom kompilator och exekvering hänger samman som en enhet. Hur infogar man nu en separat kompilerad modul i ett program, eller annan modul? I java görs det genom att man importerar klassen på översta nivån i en annan modul, och man har inget av hierarkisk struktur, medan man i Simula kan infoga den på vilken plats som helst i blockstrukturen. Vi ska här hantera det på det senare betydligt mera kraftfulla sättet. Detta betyder att man på en plats där en deklaration är tillåten kan placera en fullständig deklaration eller importera en extern separat kompilerad enhet. Vad finns det då för krav för att man ska kunna kompilera separata moduler? Vi utgår från att vi har ett starkt typat språk. Vi kan anta att den separata modulen är en klass, dvs. representerar en typ, varför vi måste känna till den och vad som

publikt kan nås i den då man kompilerar modulen som använder denna klass. Vi ser på följande exempel:

```
class A(I); integer I;
begin
  integer K;
  integer function GetK; GetK:=K;
  ........
end;
```

Separat modul för A

```
begin
  integer J;
  external class A;
  ..........
end;
```

Program som använder A

Då vi kompilerar huvudprogrammet gör vi en externdeklaration av A, vilket betyder att A definierats i en annan modul som kompileras separat. Här måste kompilatorn kunna ta reda på de egenskaper hos A som man kan nå. Detta betyder att modulen som definierar A måste ha kompilerats före huvudprogrammet, samt att dess egenskaper ska finnas sparade. Detta löser vi enklast genom att då man kompilerar en modul så produceras två resultatfiler, dels objektkodsfilen, dels en deklarationsfil. Deklarationsfilen innehåller de delar av symboltabellen som behövs utifrån. När huvudmodulen översätter external class A; så måste dels klassen A in i symboltabellen, med sina egenskaper, dels måste de metoddeklarationer som kan nås också in i symboltabellen. Skillnaden mot om klassen hade funnits med sin definierande text här är att istället för att analysera texten för att få information till symboltabellen så läses denna in från den deklarationsfil som producerats då modulen med klassen A kompilerades. Som vi ser här så är separatkompilering förhållandevis enkelt. Naturligtvis finns det krav på att en modul måste vara kompilerad före det att den används (genom en external deklaration). Detta betyder också att cirkulära beroende mellan moduler inte fungerar. Vad är det då man ska spara i deklarationsfilen? Det är all den information som kompilatorn behöver för att använda klassen och dess nåbara delar, däremot sådant som är internt inte behöver finnas med. En deklarationsnod för klassen A måste finnas, som infogas i huvudprogrammets deklarationstabell. Även ett block för klassen A måste finnas med, som innehåller en symboltabell, men naturligtvis ingen kod. Denna symboltabell innehåller de deklarationer som man kan nå utifrån. För enkla variabler är det därmed klart, men för metoder är det lite annorlunda, dess parametrar måste ju vara kända. En metod medför således att ytterligare ett block, inre i klassen A, måste finnas med. Detta block innehåller en symboltabell, som endast behöver beskriva parametrar och ett eventuellt resultat, lokala variabler behöver inte vara med i denna tabell. Någon kod för blocket behövs inte. Vi har nu sett vad den importerande modulen behöver, hur gör man vid kompileringen av den importerade modulen. Modulen kompileras på precis samma sätt som då man inte har separatkompilering, skillnaden är att det inte finns något omgivande fiktivt block och att man inte genererar någon uppstartskod för program. Man genererar endast kod för konstruktor, och blockets normala kod. Dessutom måste man generera vissa externdefinitioner så att man kan anropa koden utifrån. De inre blocken kompileras på samma sätt som utan separatkompilering. Den återstående skillnaden är då att man producerar en deklarationsfil. Denna görs genom att man direkt överför informationen från de normala symboltabellerna till deklarationsfilen och därvid filtrerar bort sådant som inte ska vara med. I vilket format skrivs nu deklarationsfilen? Detta är inte väsentligt, annat än att man naturligtvis måste vara överens mellan kompileringarna. Ett enkelt sätt är att göra det i form av en textfil där symbolnoderna beskrivs i form av en sekvens av heltal och där så behövs en “namnsträng”. Fördelen är att det är lätt att göra, distribuera och hantera, nackdelen kan vara att tex-

thantering är lite långsam. Då det ofta inte är några stora mängder text är effektiviteten oftast här av underordnad betydelse. Är det någon skillnad, som inte nämnts, mellan att separat kompilera en klass eller procedur och att kompilera den i sitt sammanhang i ett program. Nej knappast, däremot för en användare så betyder det att vid separatkompilering så känner modulen inte till sin blivande omgivning och kan inte referera något i den, medan då den kompileras i sin omgivning så når man ju denna på normalt sätt. Detta är inte ett egentligt problem vid kompilering, utan ett problem för användaren, programmeraren. Man kan dock tänka sig olika sätt att för kompilatorn beskriva en sådan "kommande omgivning" och därmed kunna nå sin omgivning även vid separatkompilering. Detta är dock ett nytt principiellt problem i både språkkonstruktion och kompilatorteknik som vi inte ska ta upp här.

Vad görs då vid kompileringen av den modul som importerar deklarationen. Jo här läser man under deklarationsanalysen in deklarationsfilen och placerar den externt definierade symbolen i aktuell symboltabell, samt tar man in övriga block som behöver tas in. Deklarationsfilen innehåller vad som behövs, dock med ett problem. Namn på symboler har ju fått en i sin modul intern identifikation, och samma namn har ju troligen i aktuell modul fått en annan identifikation. Vid sökning efter en metod kommer man då inte att finna den, om man inte ändrar identifikationen i de importerade deklarationerna. Eftersom namnet som sträng också finns med kan man enkelt klara av en sådan översättning. Även för blocknummer gäller att motsvarande interna nummer som måste översättas.

Enkla objekt.

I ett språk med objekt så definieras dessa tex genom att man definierar (deklarerar) en klass. Man skapar sedan objekt av den klassen. Vi kan se på följande exempel. En klass

```
begin
      class A(X); integer X;
      begin
         integer Y;
         Y:=5;
      end;
      ref(A) Pa;
      Pa:=new A(2);
end
LABEL     S0                          Programstart
PALL      T0,0
PEX
DALL
RETURN
START     S0
LABEL     B0                          Start av yttre block
OALL      T1,0                        Allokera AR för objekt
LITERAL   id0,int,2
MOVE      LIT,0,id0,int AR,-1,0,int   Överför "parameter"
SAVE      0                           Motsvarar proceduranrop
OCREATE                               men PEX ersätts av detta
RESTORE   0
MOVE      REG,0,3,adr TMP,0,1,adr     Överför resultat (register 3)
OEXIT                                 Avallokeras ej, men "lämnas"
MOVE      TMP,0,1,adr AR,0,0,adr      Lagras som vanligt värde
RETURN                                Slut på yttre block
LABEL     T0
GENWORD   1
GENADR    B0
LABEL     B1                          Här börjar "konstruktorn"
LITERAL   id1,int,5                   Denna kod är samma
MOVE      LIT,0,id1,int AR,0,1,int    som för andra blocktyper
RETURN
LABEL     T1                          Template för objektet
GENWORD   2                           ingen skillnad för
GENADR    B1                          så enkelt objekt
END
```

definieras på i princip samma sätt som en användarfunktion, den anropas då man gör new. Anropet ser ut på ungefär samma sätt som ett proceduranrop, dvs det allokeras en aktiveringspost, man överför parametrar och man exekverar inledande kod. I språkexemplet ovan som liknar Simula exekveras den koden som finns i klassen, medan java och C++ inte kan ha kod här, utan har koden i en särskild konstruktor. Skriver man som ovan så ser det verkligen ut som en procedur. Skillnaden är att värdet av anropet är objektet, som helt enkelt representeras av sin aktiveringspost. Resultatet blir således adressen till denna, dessutom måste naturligtvis denna aktiveringspost allokeras på ett sådant sätt att den finns kvar så länge det finns någon referens till den. Hanteringen av variabler som har objekt som värden sker på samma sätt som hanteringen av andra variabler, om de innehåller ett heltal eller ett objekt, dvs en adress till objektet, gör ingen

egentlig skillnad. När man ska använda objekten och nå dess attribut, dvs använda punknotering, betyder detta att man inför en ny operator (.). Denna tänker man först hantera som vilken operator som helst, men vid närmre eftertanke inser man att den är något mer komplicerad än så. Dels är själva utförandet av operatorn punkt något mer komplicerad än tex +, dels måste man söka efter attributen på annat sätt än vanligt. Ser vi på uttrycket P.X, så måste vi söka upp P i symboltabellen, men vi måste också söka upp X i symboltabellen. Normalt är en sådan sökning oberoende av hur variabeln förekommer i uttrycket, men detta gäller naturligtvis inte när en variabel föregås av punkt. Vi måste nu söka efter X i det block som representerar typen av variabeln P. Vi har infört ytterligare en sökregel, som naturligtvis inte är särskilt mycket svårare än den tidigare.

Ska vi nu anropa en metod i ett objekt, dvs exekvera en procedur som är definierad i en klass och anropad i ett objekt via punktnotering, så ser det nästan likadant ut som vid anrop av en vanlig procedur, men inte riktigt. Den skillnad som finns är naturligtvis allokeringen. När vi vid ett vanligt proceduranrop allokerar minne med operationen PALL så ger vi denna två parametrar, dels adress till template som beskriver proceduren, dels en “adress” till statisk fader. Denna adress till statisk fader är helt enkelt hur många nivåer upp i den statiska kedjan som vi funnit procedurdeklarationen, dvs en relativ nivåadress. Detta gäller naturligtvis inte när vi funnit proceduren via punktnotering, den statiska fadern är naturligtvis då objektet. Vi byter ut PALL mot en särskild metodallokering MALL vars andra parameter inte är en nivåskillnad utan en normal adress till ett objekt.

Felhantering.

För att göra en kompilator praktiskt användbar måste den kunna hantera felaktiga program på ett vettigt sätt. Då man skriver mindre program och dessutom ofta är en oerfaren programmerare behöver man att kompilatorn reagerar för felet och ger en begriplig felutskrift. Ett sätt som vissa kompilatorsystem gör är att man vid upptäckten av ett fel avbryter kompileringen och ger användaren kontrollen i en editor så att man omedelbart rättar felet. För nybörjaren är detta ofta bekvämt, men den mer avancerade programmeraren vill nog se alla sina fel först innan han går över för att rätta till dem. En helt annan situation har den avancerade programmeraren som arbetar inom ett större projekt tillsammans med flera andra. Här vill man säkert kunna kompilera ett helt program även om det finns fel i det, vissa fel beror kanske på någon annan och jag vill kunna ignorera dem tillsvidare, för att koncentrera mig på mina egna. Detta kan betyda att jag även vill kunna exekvera det felaktiga programmet, eftersom jag vet att det jag ska testa inte gör att den felaktiga koden kommer att exekveras. En kompilator som hittar ett fel och som sedan lägger av kommer man då inte stå ut med särskilt länge. Det krav man då har är att man ska kunna kompilera ett program, om man hittar ett fel skall man fortsätta för att finna alla andra fel som finns, och helst inte så många följdfel till tidigare rapporterade fel. Dessutom skall man producera exekverbar kod, trots att man funnit fel. Naturligtvis kommer koden kring det funna felet att vara felaktig och därmed troligen ganska olämplig att exekvera. Detta problem löser man med att man på ett sådant ställe genererar kod som är ett anrop av RTS för att skriva ut att här finns ett kompileringsfel och att exekveringen avslutas här.

Vi skiljer naturligtvis på två helt olika sorters fel, det är fel som påträffas vid kompileringen, och det är fel som påträffas då programmet exekveras. Vi startar med kompileringsfel. Man delar lämpligen in dessa fel i olika kategorier. En sorts fel är sådana som är ett brott mot språkets regler, men kompilatorn kan gissa vad som troligen menas(warning). I Simula skriver man tex tillordning som := för enkla värden och :- för objekt, skulle man göra fel här kan kompilatorn ändå förstå vad som menas. En annan sak är att en viss konstruktion kan vara korrekt, men på något sätt ovanlig så att man kan misstänka att det egentlgen är fel. Ett exempel från Simula är endkommentarer och glömda semikolon efter end. Går en endkommentar över mer än en rad bör kompilatorn bli misstänksam. I dessa fall skriver man en varning, och genererar kod utan några felavbrott. Vid andra normala fel ger man en felutskrift samt genererar kod med felavbrott som beskrivits ovan(error). Vissa fel kan vara så allvarliga att kompilatorn har svårt att återhämta sig och överhuvudtaget generera någon vettig kod, sådana fel brukar man kalla fatala och man avbryter kompileringen(fatal). Ett annat exempel är att man vid läsning av infilen eller vid produktion av resultatet kommer i konflikt med operativsystemet, tex infilen finns inte eller man har inte rätt att läsa från den(OS-error). I sådana fall bör man skriva ut vad som hänt med filnamn och sådan information, samt naturligtvis avbryta kompileringen. Ibland kan kompilatorn finna väldigt många fel kanske hundratals. I ett sådant fall kommer troligen rättningen av några fel att kraftigt påverka resten av felen och det är knappast meningsfullt att få ut så många fel. Det brukar finns en felgräns där kompilatorn lägger av. En femte sorts kompileringsfel bör inte men kan tyvärr inträffa(internal). Detta är interna fel i kompilatorn, som inte kompilatorn direkt kan knyta till ett fel i källkoden. Detta är egentligen ett fel i kompilatorn själv, som inte ska kunna inträffa, men inga program av någorlunda komplexitet är felfria, detta gäller också kompilatorer. Vad är det då man ska ta hand om här? Se på följande exempel: En intern rutin i kompilatorn skall ha en symbolnod som parameter, och det kan inte inträffa att den får något annat eller att ingen symbolnod sänds som parameter (nil, none,...). Man bör ändå testa att man får en symbolnod och om inte se det som ett internt fel. Vad ska då användaren göra då han får ett internt fel? Han kan ju ta kontakt med kompilatortillverkaren och sända en felrapport dit. Han kan också se närmre på sitt program och se om det finns några fel i programmet. Interna fel kan bero på att kompilatorn är dålig, men också på att källkoden innehåller ett så klurigt fel att kompilatorn inte kunnat finna det på normalt sätt och därmed gått vilse. I väl uttestade kompilatorer bör detta naturligtvis vara något som i stort sätt aldrig inträffar. Jag tror dock att en användare av kompilatorn tycker att det är bättre att få ett internt fel under kompileringen än att kompilatorn gör fel och att man under exekvering av programmet eller redan under kompileringen får ett fel av typen: segmentation violation eller bus error.

Även under exekvering av programmet upptäcks fel som medför att körningen avbryts, med ett lämpligt felmeddelande. Ibland finns en särskild avlusare, debugger, med och det kan då vara lämpligt att vid fel gå över till denna, inte för att exekvera vidare, utan för att se på de datastrukturer och variabelvärden som finns. Ett av de fel som vi bör ta hand om är fel som redan kompilatorn upptäckt. Andra fel är adressering utanför indexgränser, referenser till obefintliga objekt, filer som ej finns eller kan kommas åt, minne som tar slut, fel vid aritmetik (overflow, division med noll, ...) bör hanteras. Särskilt vid ett programs uttestning är det väsentligt att få alla sådana kontroller utförda. En nackdel är att sådana tester tar tid vid exekveringen, därför händer det att man har möjlighet att slå av sådana tester när programmet anses korrekt (=utan kända fel) och går till produktionskörningar. Detta är dock en medveten risk man i så fall tar för att få en snabbare exekvering.

Felhantering vid recursive decent.

Med denna metod är det ganska enkelt att upptäcka ett fel, man vet tex att det skall komma ett *then*, men det gör det inte. Det är då lätt att rapportera felet med ett förståligt felmeddelande, men hur skall man fortsätta. En enkel metod, som faktiskt har använts, är att man fortsätter att leta tills man får vad man vill, i det här fallet ett *then*. Att denna metod är synnerligen dålig bör man upptäcka med endast lite eftertanke och vi lämnar den helt utan ytterligare behandling. En vanlig metod är att man söker framåt tills man finner något som man åter känner igen som något man kan översätta vidare från. Allt fram till detta token ignorerar man. Man letar efter ett sk synkroniseringstoken och där återstartar man översättningen. Svårigheten är att bestämma vad man skall använda för synkroniseringen. Ett extremt val är att använda programslut, det ger visserligen inga onödiga följdfel, men är nog ändå knappast acceptabelt. Det förekommer dock kompilatorer som gör så, dvs de finner högst ett fel per kompilering, eller i en något bättre variant ett fel per pass. Ett annat vanligt synkroniseringstoken är ; eller *end* detta betyder att man försöker starta efter att satsen tagit slut. Detta är inte så bra om felet är ett glömt semikolon, man kommer att ignorera väl mycket. Ur en grammatik kan man få fram vilka token man kan förvänta sig följa på ett visst token, vad kan tex komma efter *then*. Man kan också ta fram vad som kan komma först i en viss härledning i en produktion. Dessa mängder FOLLOW och FIRST har en central betydelse i teorin för parsing, dvs analys av tokensekvenser från ett språk. Mer om detta i teoridelen. De kan också användas för att beräkna lämpliga synkroniseringsmängder. Detta är ganska svårt och man ser ingen väl utvecklad teori i litteraturen som behandlar detta. Man kan försöka att låta de tokens som finns i followmängden utgöra synkroniseringsmängd. Man finner då att man missar väl mycket, man borde också synkronisera på sådant som kan starta något nytt, tex ny sats (har med first att göra). Om man utvidgar synkroniseringsmängden med detta finner man att man ibland återstartar lite väl tidigt och man får för mycket följdfel. Om man vid beräkningen av follow- och firstmängderna inte helt använder sig av FOLLOW och FIRST, utan tar hänsyn till historien dvs var man är i analysen så kan man få en bättre synkroniseringsmängd. Man definierar en form av lokalt follow och first. Betraktar ni de kompilatorer ni blivit utsatta för kanske ni misstänker att detta med felåterhämtning inte är så lätt. När ni så försöker implementera detta själva i en enkel kompilator tror ni säkert att misstanken är riktig. Så är det, detta är svårt, även om man kanske inte behöver göra så dålig felåterhämtning som många mer eller mindre proffesionellt tillverkade kompilatorer gör.

Felhantering vid operatorprecedens.

Denna metod är känd för att felåterhämtningen är särskilt besvärlig. När man får ett felaktigt token är det inte så lätt att omedelbart inse att det måste vara fel, man upptäcker ofta först efter ytterligare några steg i analysen att det inte stämmer. Den enkla metod för operatorprecedens som jag tidigare beskrivit med två stackar, en för operatorer och en för operander, är en alternativ metod som dessutom gör felåterhämtningen ännu besvärligare om man inte gör något åt det. Betrakta följande exempel: a+b*c+d som vi kan betrakta som ett korrekt uttryck och a b c+*+d som vi kan se som ett något felaktigt uttryck. Eftersom vi sorterar operander och operatorer på två olika stackar utan egentlig inbördes ordning så kommer dessa två uttryck att båda accepteras och med samma resultat. Nu brukar man oftast inte göra denna typ

av fel, men följande är ganska vanligt: a b*c+d. Här kommer vi naturligtvis att finna att vi inte har tillräckligt med operatorer, ty vi kommer att sluta med två operander på operandstacken vilket måste vara fel, men vi upptäcker det inte förrän vi översatt hela uttrycket. För att upptäcka fel av den senare typen räcker det att göra en enkel stackkontroll när uttrycket är beräknat, men i det första fallet räcker inte detta, då allt ser rätt ut då vi är färdiga. Här kan vi istället klara oss genom att då vi får ett token från inströmmen kontrolleras att vissa regler följs. Dessa regler är förhållandevis enkla att formulera. Vi kan enkelt notera att i de uttryck vi använt som exempel är det inte tillåtet att få två operander i rad utan mellanliggande operator. Tyvärr gäller inte samma enkla regel för operatorer, se tex a+f(1,...., där ju +, f och (är tre på varandra följande operatorer. Det är dock bara vissa operatorer som får förekomma på detta sätt och man kan relativt lätt formulera en regel även här. Genom att formulera sådana villkor som alltid måste vara uppfyllda, genom att göra stackkontroller och genom att också ta hänsyn till övrig information som finns i de olika token (symbolinformation mm) klarar man även här av en bra felhantering. Så om första meningen i detta avsnittet var avskräckande för felhanteringen, så kan jag ta tillbaks detta nu och påstå att om man väver in lite enkla regler och kontroller i metoden så är felhanteringen inte särskilt besvärlig.

Detta var fel som kan upptäckas direkt i inmatningsströmmen, eller genom att man får stackfel i operand och operatorstackarna. Naturligtvis kan operatorer och operander följa på varandra på ett sätt som visserligen är fel men som man inte enkelt kan se direkt i inmatningsströmmen. Dessa fel ser man oftast i aktionsmatrisen, dvs vi hamnar på en felingång där. Dessa fel är lätta att upptäcka, men naturligtvis oftast lika svåra att återhämta sig från som de flesta andra fel, motsvarar vad man får göra i rekursive descent. Även om vi inte finner några fel med de två metoder som nu beskrivits så betyder inte detta att det inte finns fel. Vi har nu funnit de syntaktiska felen, men vi kan ju ha semantiska fel. Vi kan försöka addera ett heltal med en text, vilket i de flesta språk är förbjudet. Dessa fel kan man ganska lätt finna genom att man testar på typer under "utförande" av operatorer, dvs vid genereringen av mellankod. Då man söker efter operander kan man naturligtvis upptäcka att man inte finner dem, dvs man har använt sig av odefinierade, eller på denna plats ej definierade variabler. Detta fel är oftast lätt att återhämta sig från, man låtsas bara att den är definierad med någon allomfattande lättanvänd typ.

Vi har nu endast behandlat operatorprecedens i samband med aritmetiska och logiska uttryck, vi har inte tagit upp vad som händer om vi också använder denna metod för övriga språkkonstruktioner. Det som händer är ju att man får fler och något mer komplicerade operatorer (tex if, then, while, ...) detta medför att kontroller också blir något svårare. Framförallt blir det fler och kanske även mer komplicerade villkor som man måste formulera, men i princip händer inte så mycket.

En viktig princip vid felhantering är att man så mycket och så fort som möjligt bör få det att bli "rätt". Antag att vi upptäcker en addition mellan operander av olika och oadderbara typer. Generera då kod som anropar en felrutin i rts, och som omöjliggör att man exekverar förbi felet. Låtsas sedan att additionen är tillåten, generera kod för detta, som ju inte kommer att exekveras varför man kan generera vad som helst. Antag sedan att vi får ett resultat av typen "anytype" som ju visserligen tyder på ett fel men som låter oss fortsätta. Vi antar vidare att denna typ duger till allting, varför vi inte kommer att rapportera felet gång på gång. Ett liknande problem är odefinierade variabler. Man rapporterar naturligtvis felet och genererar felkod, sedan definierar man variabeln tillfäl-

ligt med lämplig typ, tex “anytype”, och fortsätter kodgenereringen. De flesta semantiska fel kan man hantera på detta sätt, med rimligt resultat. Syntaxfel är visserligen enkla att upptäcka, men lite svårare att återhämta sig från. Upptäcker vi fel i ett uttryck så försöker man snabbt återhämta sig, eventuellt kan man “rätta” felet på ungefär samma sätt som vid semantiska fel. Vi finner dock att man kan få lite svårare konsekvenser. Vi ser på satsen *if x>a+5 A(x-5):=F(x,a) else x:=4;* , vilket fel innehåller den? Vi som ser satsens mönster inser lätt att det saknas ett *then* före till delning av värde till ett element i A. En kompilator läser dock enheter från vänster till höger och avgör vad det är utan att se långt framåt. Kompilatorn ser det som att det saknas en operator mellan 5 och A, i varje fall om den inte utför en större analys. Återhämtar man sig från detta fel så får man snart följdfelen att := inte får förekomma mitt i uttrycket, samt att else kommer utan att man har en then-del. Vi ser här att en väl fungerande felåterhämtning inte är så enkel. Man kan naturligtvis ha mindre ambitioner och inte försöka komma rätt så fort som möjligt utan hoppa över allt fram till semikolon. Detta leder dock till att man missar eventuella fel i else-delen i exemplet ovan. Detta betyder kanske inte så mycket i satsen ovan, men om else-delen är ett längre stycke kod, tex en sammansatt sats är detta mindre önskvärt. När man som människa ser exemplet ovan brukar det inte vara så omöjligt att finna regler för kompilatorn som skulle ge en bra felhantering av exemplet ovan. Problem med sådana regler är att de är svåra att få generella, dvs om man råkar ut för ett lite annorlunda fel så får inte reglerna leda till att det nya felet hanteras riktigt illa, kanske tom katastrofalt med att kompilatorn spårar ur. Det svåra är inte att finna regler som hanterar vissa fel bra, det svåra är att finna regler som hanterar de flesta fel acceptabelt och vanligt förekommande fel bra.

Ovan har vi tänkt oss att det förekommer ett fel och inte ytterligare ett, i varje fall inte förrän lite “senare”. Om vi har två eller flera fel i varandras omedelbara närhet blir det naturligtvis betydligt svårare att återhämta sig. Här får man väl acceptera att det andra felet inte upptäcks, men man vill helst inte spåra ur helt. Problemet är att då man försöker återhämta sig från det första felet så tolkar man ju fortsättningen för att på så sätt se hur man kommer över det första felet. Men om den struktur man då analyserar innehåller ytterligare fel kommer denna analys med stor sannolikhet att helt misslyckas vilket kan leda till att man hanterar det första felet på ett tokigt sätt.