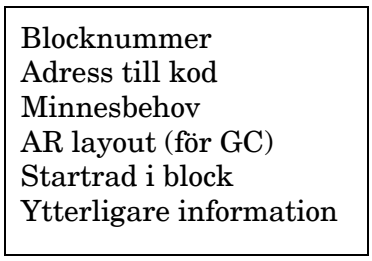
Kapitel 4

# *Exekvering & RTS*

## Funktion av RTS

### Exekvering av enkla block

Vi betraktar blocket som den grundläggande exekveringsenheten. Ett block kan vara en procedur, ett objekt eller ett samanhållet avsnitt i programmet. I många språk karakteriseras det av källkoden *begin dekl+; satser* end*, dvs lokalt deklarerade variabler och normalt ett avsnitt exekverbar kod. Till varje block hör statisk information i form av en template och ett stycke kod som utgör blockets exekverbara del. Varje exekvering av ett block medför att det skapas en instans av blocket. Det är naturligt att tänka sig att rekursiv exekvering av en procedur medför att flera instanser av proceduren måste existera samtidigt. Detta gäller naturligtvis också vid objekt. Även av ett vanligt block kan det finnas flera instanser samtidigt, det kan ju tex vara ett inre block i en procedur. Oavsett hur många instanser som finns av ett block finns det bara en template med statisk information. Lika självklart är det att AR finns i flera upplagor, en per instans. Hur är det då med koden? Vi antar att vi inte använder oss av självmodifierande kod, utan att även koden är statisk. Det gäller då att även koden endast behöver finnas i en upplaga oavsett hur många instanser av blocket som finns. Däremot kan man i de olika instanserna naturligtvis befinna sig på olika ställen i koden, varje instans har således sin egen programräknare, eller som det är i det enkla fallet, återhoppsadress. Denna är då naturligtvis dynamisk information som behöver lagras i AR. Vi ger nu en bild av template och AR för enkla block. Först beskriver vi informationen i template. Blocknummer



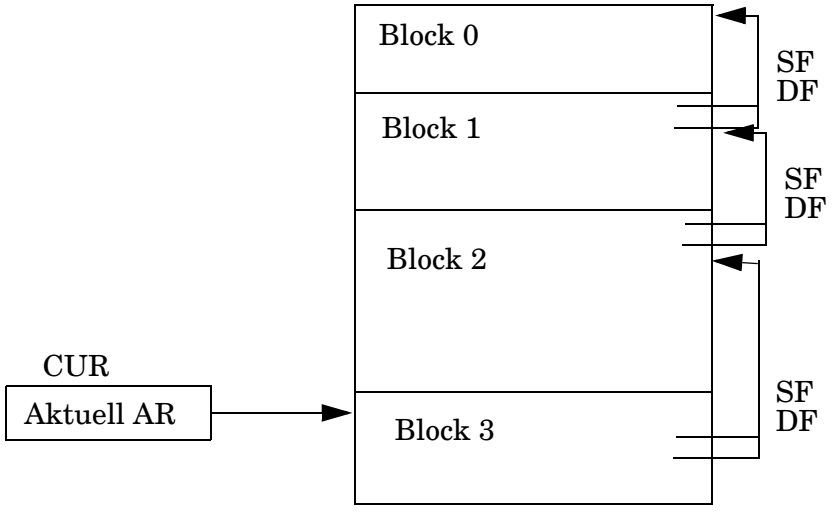

Template


Aktiveringspost (AR)

används för att identifiera blocket och har ingen större betydelse. Adress till kod är dit man skall hoppa när blockets exekvering skall inledas. I vissa språk kan mera komplicerade block ha flera startadresser, mera om detta kommer senare. Uppgiften om minnesbehov behöver RTS för att kunna allokera AR. Layout för AR behöver systemet i de

fall man måste veta vad som verkligen finns i AR, tex behöver en lumpsamlare (GC) information om vad som är pekare. Radnummer används normalt bara av en felrutin för att kunna tal om i vilket block något gått fel. Ytterligare information behövs knappast i det enkla fallet.

Vi övergår nu till aktiveringsposten. Statisk fader behövs för att vi ska kunna nå variabler som finns i omgivande block. Den dynamiska fadern är den blockinstans där instansen skapades, detta behövs för att vi skall veta vilken omgivning vi skall gå tillbaka till då vi har exekverat färdigt i instansen. För vanliga block är SF och DF lika, men för procedurer kan ju det ställe där den deklarerats skilja sig från det ställe där den anropats. I en kedja av rekursiva anrop av en procedur har ju alla instanser samma statiska fader, medan den dynamiska hela tiden är den närmast föregående instansen av proceduren. Pekaren på template behövs speciellt i samband med objekt, bla för att hålla ordning på typer och AR layout vid lumpsamling. Gammal PAR pekare behövs för parameteröverföring vid mer komplicerade anrop av funktioner, beskrivs närmre senare. Återhoppsadressen är den adress i koden dit vi ska återvända då exekveringen av instansen är slut. Lokal PC beskrivs inte nu, hör samman med korutiner och parallell exekvering. Data utrymmet är inte mycket att säga om, mer än möjligen att vi gjort plats för parametrar till en procedur och till det eventuella resultatet vid funktioner. Varför gör vi detta? Ser man på C-kod upptäcker man att parametrar i första hand överförs i register och om dessa inte räcker lägger man dem i den anropande instansens stackram. Resultatet överförs i register. I mera komplicerade språk är det svårt att klara sig med en så enkel parameteröverföring, därför beskrivs här en mera generell lösning som kommer att fungera också i komplicerade fall. Slutligen avslutar vi AR med plats för att lagra undan temporära variabler som vi inte har register nog åt, eller lagrar undan register i vid vissa rutinanrop. Vid enkla fall behövs inte detta, vi kan ha en stack i rts för detta. Vid korutiner och parallellitet är situationen annorlunda. När vi allokerar utrymme i aktiveringsposten för variabler antar vi att alla variabler av en viss typ är lika stora. Hur gör vi då med fält vars storlek inte nödvändigtvis är kända vid kompileringen? Dessa kan man tex allokera minne för i speciella datautrymmen och i AR lägger vi endast ut en pekare på detta utrymme. Även andra sätt att allokera fält är tänkbara.




Ovanstående ger en bild av minneshanteringen vid anrop av normala kapslade block. Här är dynamisk och statisk pekare hela tiden lika. Vi skall nu se hur man

kan adressera data i de olika blocken med hjälp av ovanstående struktur. Vi antar att aktuell AR pekaren ligger i registret CUR, vidare att vi har ett slaskregister LNK samt att vi skall ladda registret R1 med med innehållet i ord x i block 3, block 2, etc. Vi antar att det finns enkla maskininstruktioner LD för att göra en dataöverföring till ett register. Vi ser nedan att adressering i aktuellt block går snabbt, men att för varje nivå utåt kostar det en extra instruktion per nivå utåt. Man kan göra en viss optimering med yttersta blocket, vars läge är fixt och dessutom känt redan vid kompileringen. För att slippa den dyra adresseringen av block uppåt i kjedjan kan man försöka ha fasta pekare på dessa aktiveringsposter. Det vanliga sättet att klara av detta problemet är att man har en vektor med pekare på de olika blockens AR. Vi använder oss av en sk display vector. Denna vektor har hela tiden en pekare på statiskt block på varje nivå. Adressering sker då på samma sätt vad gäller aktuellt block och yttersta block, medan mellanliggande block adresseras med två instruktioner oavsett på vilken nivå den ligger. Vi ser adresseringen i figuren nedan. Vi ser då att metoden med display klarar aktuellt och yttersta block direkt och på samma sätt som med metoden med SF-länkar i AR. En nivå upp kostar två instruktioner oavsett metod. Alla andra nivåer kostar två instruktioner med display vector och mer än två instruktioner i den andra metoden. Vi ser att vektorn aldrig är sämre och ibland bättre, valet borde därför varit enkelt.Vi måste naturligtvis

```
I block 3 (dvs aktuellt)
ld      R1,x,CUR

I block 2 (ett upp)
ld      LNK,sf,CUR
ld      R1,x,LNK

I block 1 (två upp)
ld      LNK,sf,CUR
ld      LNK,sf,LNK
ld      R1,x,LNK

I block 0 (tre upp)
ld      LNK,sf,CUR
ld      LNK,sf,LNK
ld      LNK,sf,LNK
ld      R1,x,LNK

I block 0 (överst, spechantering)
ld      R1,x,GLOB
```

```
I block 3 med didplay
ld      R1,x,CUR

I block 0, överst med specialhantering
ld      R1,x,GLOB

I block n (=2,1,0)med display
ld      LNK,n,DISP
ld      R1,x,LNK
```

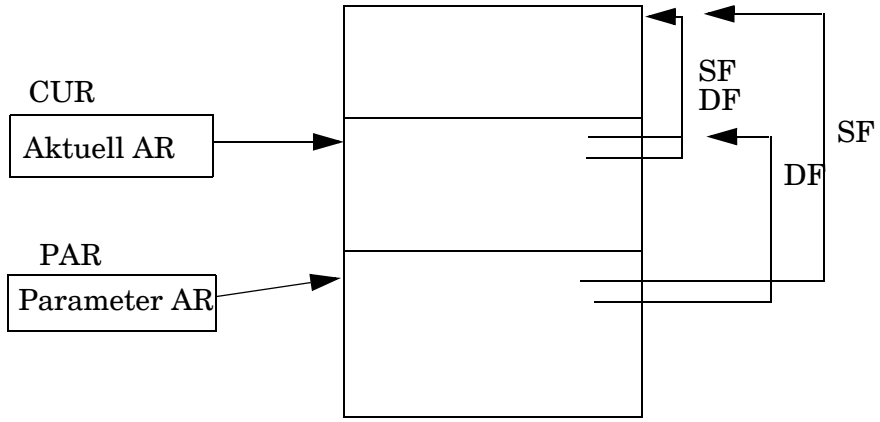
också räkna med att när ett nytt block lägger sitt AR i minnet så måste SF-länkarna uppdateras och även display vektorn. Kan det hända att någon ingång i displayvektorn behöver peka på två olika block vid samma tillfälle (dvs växelvis)? Ja, ty ingången i displayvektorn bestäms av på vilken nivå i blockhierarkin som den statiskt definierats på. Man kan naturligtvis definiera många procedurer i ett block och dessa hamnar alla på samma nivå. Dessa kan sedan dynamiskt anropas på olika nivåer, och vi kan behöva ha en ingång i displayvektorn i princip fungerande som en stack! Denna uppdatering av displayvektorn tar naturligtvis också tid, och nu är valet av metod inte längre lika självklart. Det visar sig att många adresseringar på mellannivåerna fortfarande gör metoden med display fördelaktig. Länge nöjde man sig med detta och slentrianmässigt använde sig av display vilket också rekomenderas i de flesta textböcker i kompilatorteknik. En undersökning av exekveringen av ett antal olika normalt skrivna program visade att nästan alla referenser i ett program skedde till aktuellt block, direkt omgivande block samt till det yttersta blocket. Övriga referenser var ytterst få, någon

procent! Detta betydde att alla referenser praktiskt taget tog lika lång tid med de båda metoderna, och extra tid åtgick för uppdatering av displayvektorn och metoden med länkade SF pekare var betydligt effektivare! Dessutom har erfarenheten från ett antal kompilatorkurser visat att SF-pekarmetoden är mycket enklare att förstå och klara av på ett korrekt sätt.

Vi har nu sett på minnesstrukturen och på hur man adresserar data i ett AR, vi har ännu inte sett hur man allokerar minne och når den kod man skall exekvera. Vi skall inte skilja på att anropa en procedur, skapa ett objekt eller att gå in i ett block, allt kan betraktas som ett proceduranrop. I fallet med ett vanligt block är det som att anropa en procedur utan namn och utan parametrar. I ett lågnivåspråk som C kan man göra detta på ett väldigt enkelt sätt. Här lägger man parametrar i register samt hoppar till procedurens kod. I starten av proceduren skapar man sitt AR genom att ändra stackpekaren så man får ett lagom stort AR. I ett högnivåspråk fungerar inte detta, bl a kan man inte skapa objekt på stacken eftersom de skall ligga kvar efter skapandet. I vårt första enkla exempel skulle det dock gå, men vi ger redan nu en generell metod. Det första som behövs är att vi allokerar en aktiveringspost i minnet, i vårt enkla fall på en stack. Sedan överför vi parametrarna, ifall det är en procedur med sådana, varefter vi kan starta exekveringen av blockets instans. När vi kommer ur blocket skall vi ta hand om resultatet och avallokera AR. Vi kommer att anropa rutiner i RTS för att göra allokering, exekvering och avallokering.

## Exekvering av procedurer.

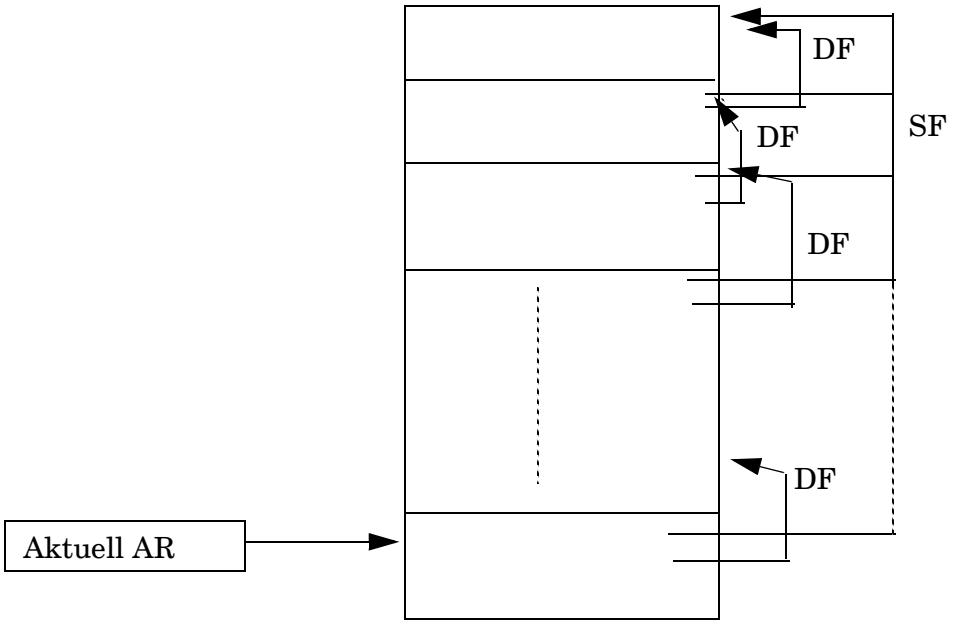
Vi fortsätter nu med att se hur vi skall kunna exekvera en procedur, med parametrar. I slutet av förra avsnittet antyddes hur detta skulle gå till. Vi måste starta med att allokera utrymme för procedurens AR, annars har vi ingenstan att lägga eventuella parametrar. Innan vi överför en parameter måste den dock beräknas, detta måste vi göra i den omgivning där proceduranropet gjordes, den nyallokerade aktiveringsposten får vi ännu inte se, det är fortfarande den omgivande som är aktuell. För att kunna överföra parametrarna måste vi dock ha en pekare så att vi vet var det nya AR finns. Detta betyder att vi anropar en RTS-rutin med en pekare på template som parameter, som allokerar AR, som inte ändrar CUR men som sätter en speciell parameter pekare (PAR) på det nyallokerade AR. Template innehåller den information som behövs för allokering och initiering av nya AR.Vi kommer nu tillbaks till det




anropande blocket och beräknar parametrar som överförs med hjälp av PAR pekaren. När alla parametrar är överförda anropar vi en RTS-rutin för exekvering av

det nya blocket. Denna rutin sätter återhoppsadress, låter CUR peka på nya AR samt hoppar till kodadressen som man finner i template. Template finner man via pekare i AR. Sedan exekverar vi i nya blocket tills det är tid att återvända. Även detta görs med hjälp av en RTS-rutin. I enkla fall kan den avallokera AR, lägga resultatet i ett register samt hoppa tillbaks med hjälp av återhoppsadressen i det nyss avallokerade AR. En mer generell metod är att man inte avallokerar AR och låter ett eventuellt resultat ligga kvar i AR, ställer om CUR till den dynamiska omgivningen (DF), samt behåller en pekare på gamla AR och återvänder. Det anropande blocket som man just återvänt till kan då i lugn och ro överföra resultatet, det är tex lätt att implementera procedurer som ger flera resultat. Därefter anropas en RTS-rutin som avallokerar det gamla AR.

Ett proceduranrop kan naturligtvis komma inifrån exekveringen av en procedur, tex samma procedur. Sådana rekursiva anrop ger egentligen inte upphov till några större problem, så länge som vi allokerar ett eget AR på det sätt vi angivit. Man skall naturligtvis observera att all rekursion inte är av det enkla slag som i den normala implementationen av fakultet. Man kan ha indirekt rekursion, dvs en procedur anropar en annan som så småningom anropar den första. Vi ger också en bild av en enkel rekursiv anropssekvens. Det nya i denna bild är att alla AR utom det översta har samma sta-




tiska fader, de är ju instanser av samma block, proceduren. Däremot har de naturligtvis olika dynamiska fäder, det är den instans där anropet skedde. Vi ser här att vi inte har en lång dyr adresseringskedja via SF pekare, vilket naturligtvis är väsentligt för den här metoden.

Detta sätt att exekvera procedurer kan tyckas vara väl komplicerat och ineffektivt. Fördelen är att det klarar av komplicerade språk och komplicerade procedurer. Nu är det så att även om ett språk tillåter komplicerad definition och användning av procedurer så är ändå många anrop av betydligt enklare typ, skulle man kanske kunna särbehandla dessa. En vanlig typ av procedur är att man vill göra en enkel beräkning, men

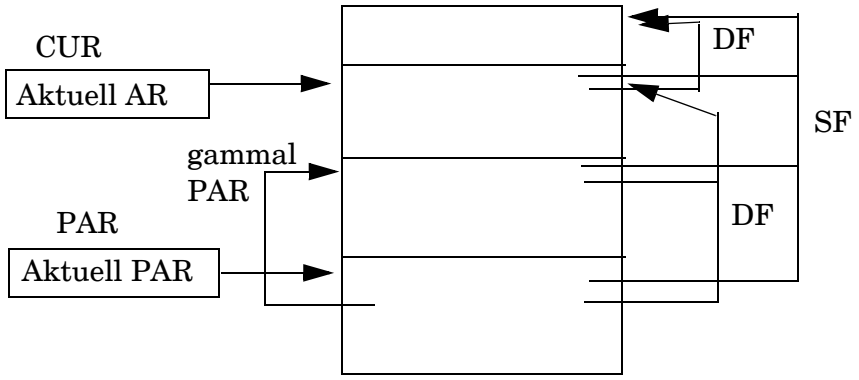
man gör den kanske på många ställen, eller man vill med ett proceduranrop klart säga vad som görs, dvs man vill ha lättläslig kod.

```
integer procedure Sqr(X); integer x;
begin
    Sqr:=x*x;
end;
```

Här har vi en procedur som inte anropar någon annan procedur, som har ett fåtal enkla värdeanropade parametrar, inget lokalt databehov och inte har någon annan typ av komplikation. Denna ger dessutom ganska lite kod att exekvera. Här skulle man kunna strunta i att det är en procedur och direkt lägga in koden för vad proceduren skall göra, i stället för att lägga ut ett proceduranrop. Detta kallas ”inline code” och vissa språk tillåter programmeraren att specificera att detta ska ske. I så här enkla fall skulle kompilatorns front end själv kunna avgöra detta. Om koden som proceduren skall utföra är betydligt längre kan dock inline code vara alltför utrymmesineffektivt. Om komplexiteten i övrigt är som ovan kan man istället tänka sig att man lägger ut ett enklare anrop. Man behöver tex inte skapa något eget AR, vi har ju inga lokala variabler, man kommer tillbaks från rutinen utan att ha behövt anropa andra rutiner. Man skulle således kunna överföra parametrar i register och göra ett enkelt anrop av C-typ, och få tillbaks resultatet i ett register. Problemet är att få kompilatorn att känna igen när man kan göra så eller inte. I en del enkla fall som för Sqr ovan är det kanske inte så svårt, men i en del fall kan det vara det. Detta är egentligen en form av optimering.

## Mer komplicerad parameteröverföring.

I förra avsnittet talade vi om parameteröverföring via särskild parameterpekare (PAR), men inte om de komplikationer som kan ske vid beräkning av parametrar. En komplikation är att beräkningen kan leda till nya proceduranrop, som skall ha sin parameteröverföring med hjälp av PAR. Den nyanropade proceduren kan vara samma procedur som vi först överför parametrar till, eller någon annan. Observera att detta inte är ett rekursivt anrop, vi har ju ännu inte startat den första procedurens exekvering. Observera vidare att det inte är fråga om att överföra någon proce-




RTS struktur vid P(P(a...)....)

dur som parameter, endast att använda en procedur vid beräkning av parametrar. Komplikationen vi får är att vi måste använda PAR till att innehålla flera pekare samtidigt, PAR blir en stack. Detta kan lösas genom att alltid spara det gamla PAR

värdet i ett AR och på så sätt få en länkad stack av parameterpekare, där stacktoppen ligger i PAR. Vi hade plats för gammal PAR i AR:s administrativa del. Vi ser här att de olika ej aktiva AR, som används för parameteröverföring har samma dynamiska fader, i vårt exempel hade de också samma statiska fader, detta gäller inte alltid, är det olika procedurer kan ju dessa vara definierade i olika omgivande block. Denna hantering av parameteröverföring kan synas väl komplicerad särskilt jämfört med den C kompilatorer lägger ut. Men å andra sidan är den generell och håller för mer komplicerade miljöer. Längre fram ska vi se på objekt och kvasiparallella system, som komplicerar vår värld ytterligare.

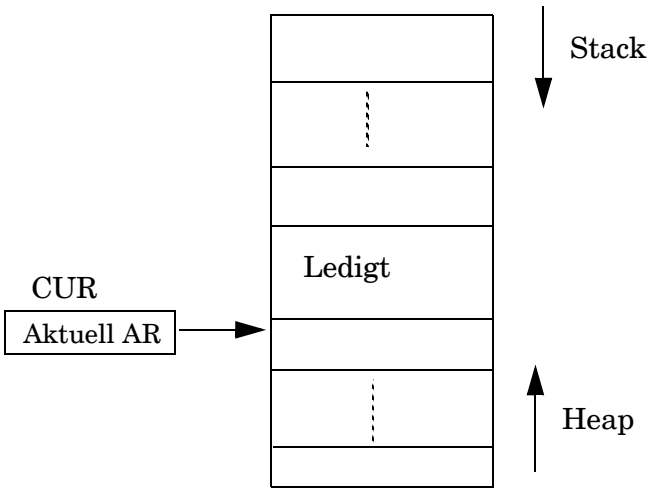
Vi har nu beskrivit hur man överför parametrar till ett AR, och i vilken miljö man beräknar dem. En annan fråga som är viktig att besvara är: Vad är det som överförs till AR? Man kan tänka sig ett antal olika metoder, man brukar tala om tre sådana. Det kanske enklaste är värdeöverföring, som betyder att man överför parameterns värde till det nya AR, och den ursprungliga variabeln kommer man inte åt från det nya blocket. Om man i proceduren ändrar parametervärdet får detta effekt endast lokalt i proceduren. En annan metod är referensöverföring, där man överför adressen till parametern. När man då använder parametern så är det den ursprungliga som avses, och eventuella ändringar av värdet är inte längre endast lokalt, utan sker i den ursprungliga variabeln. Det tredje sättet är nog mera ovanligt och härstammar från Algol, det är namnöverföring. Det man då överför är inte värdet, och inte ens adressen till värdet, utan en regel om hur man kommer åt värdet, dvs det som överförs är adressen till ett stycke kod som beräknar värdet. Denna kod brukar kallas *thunk*. Hur hänför sig nu dessa metoder till våra kända språk. Jag jämför Pascal (=Modula) och Simula. I Pascal använder man sig av värdeanrop och *var* anrop, det senare är ett referensanrop. Simula har alla tre typerna av parameteröverföring, men alla är inte tillgängliga för alla typer. Normala enkla variabler överförs med värdeanrop, men kan definieras som namnanropade. Motsvarigheten till Pascals *var* finns inte då enkla typer inte kan referensanropas, men namnanrop kan ge liknande effekt. Objekt överförs lämpligen som referensanrop, dvs man överför inte hela strukturen, utan en pekare till strukturen överförs. För närmre beskrivning av parameteröverföring i olika språk hänvisar vi till respektive språk, här intresserar vi oss främst vad vi ska göra kodmässigt i de olika fallen.

För värdeanropade parametrar är det enkelt, vi definierar helt enkelt en plats i AR för parametern och vi överför den aktuella parameterns värde dit. Sedan adresserar vi den internt i proceduren som vilken lokal variabel som helst. Vid referensanrop reserverar vi plats för den aktuella parameterns adress i AR och överför adressen, inte värdet där. När vi sedan skall adressera värdet i proceduren måste detta då göras indirekt, dvs vi får först ta fram adressen från dess plats i AR och sedan använda denna som adress till värdet. Vad händer nu om man som aktuell parameter vid ett anrop har en konstant, literal, tex talet 2. Vid värdeanrop är det enkelt, vi överför helt enkelt värdet 2. Vid referensanrop skall vi överföra adressen till en plats där värdet 2 ligger. Någon sådan plats finns ju inte om vi inte vid kodgenereringen lägger ut värdet två någonstans i minnet. Detta måste således göras, vi kan inte utnyttja att talet 2 skulle få plats direkt i instruktionen som sk immediate konstant. Ofta används ju referensanrop, tex var i Pascal för att föra ut ett resultat från en procedur, man gör p:=....., där p är en var parameter. I sådana fall är det naturligtvis meningslöst, och förbjudet, att ge 2 som aktuell parameter. Namnanropade parametrar är naturligvis besvärligare att ta hand om. I proceduren måste man således anropa ett stycke kod som beräknar det värde man vill ha, adressen till denna kod har överförts som parameter. Vid det aktuella anropet måste då detta

kodavsnitt ha genererats av kompilatorn. Även om man kan säga att det inte är god programmeringssed att använda namnanrop, så kan man ju inte som kompilatorkonstruktör av moraliska skäl låta bli att implementera detta. Har språket det måste man ta med det, men det används troligen inte så ofta så implementationens effektivitet är knappast av avgörande betydelse.

## Skapande av objekt.

Vi har sett aktiveringsposter som skapats för att ge minnesutrymme för procedurer och vanliga block (anonyma procedurer), när ett sådant block lämnas är inte längre detta utrymme åtkomligt för programmet. Vad är då ett objekt? En här viktig egenskap är att objektet är en datastruktur som är åtkomlig så länge något i övrigt åtkomligt pekar på det. Detta betyder att minnesutrymmet måste ligga kvar, dvs aktiveringsposten är åtkomlig även efter att den skapats och lämnats. I vissa språk innehåller ett objekt även kod som utförs då objektet skapas, så är det i Simula. Detta betyder att skapandet av ett objekt görs på samma sätt som ett proceduranrop, normalt kan man även här ha parametrar. Skillnaden är att då man lämnar detta proceduranrop kan man inte ta bort AR utan man lämnar i stället en pekare på det som resultat. Hur påverkar nu detta minnesallokeringen? Jo tidigare AR kunde allokeras på en stack och tas bort då man lämnade proceduren (blocket), och den procedur man lämnar ligger alltid överst på stacken. Vi har alltså en enkel stackallokering av den typ C använder sig av. Objekt kan naturligtvis inte allokeras på detta sätt, de lever ju kvar under en obestämd tid. För att klara detta allokerar man objekt på en sk heap, som helt enkelt är ett minnesutrymme för objekt där man inte bryr sig om i vilken ordning de ligger, inget ligger nödvändigtvis överst. Förväxla inte detta begrepp heap med vad ni eventuellt har hört om i algoritmteori, det är inte riktigt samma begrepp. Vi ser här att aktuell AR mycket väl kan vara ett objekt i heapen.




Ovanstående bild symboliserar skapandet av ett objekt. När objektet är klart används naturligtvis adressen till objektet för att representera dess värde. Om man som i Simula kan exekvera objekt parallellt eller kvasiparallellt måste man inse att man under exekvering i ett objekt tillfälligt kan förflytta exekveringen till annat objekt. Detta är i viss mån förödande för den enkla stackmodellen. Därför måste alla block som allokeras på något sätt inifrån ett objekt allokeras på heapen, vilket kan bli lite ineffektivt. Ofta har man ju en rad av blockaktiveringar från tex en rekursiv funktion anropad inifrån ett objekt. Denna arbetar ju normalt enligt stackprincipen

och dess aktiveringar borde kunna avallokeras på enkelt sätt. Detta kan man också göra om man använder "toppen" av heapen som en stack, så länge det går. Detta misslyckas eventuellt om man startar en parallell aktivitet, eller allokerar nya objekt som ligger kvar. Då får man avallokera denna "misslyckade stack" med hjälp av lumpsamlaren.

Administrationen av en stack där man kan avallokera poster från stacktoppen är betydligt effektivare än en heap där elementen ligger i en ordning så att de som kan tas bort ligger spridda bland de som måste vara kvar. För att effektivt kunna återanvända dessa hål i heapen bör man dels finna hålen, dels kunna kompaktera minnet så att hålen samlas till ett större samanhängande utrymme. För att klara detta brukar man implementera en så kallad lumpsamlare (Garbage Collector) i RTS, denna kan vara relativt tidskrävande. En teknik är att man tar det tillgängliga minnesutrymmet och lägger en stack i ena änden och en heap i andra, samt låter dessa växa i riktning mot varandra. När de går ihop aktiverar man lumpsamlaren för att få tag på outnyttjat minne i heapen, samt att tränga ihop de använda objekten i botten av heapen. Utformning av lumpsamlare är en vetenskap för sig, som inte tas upp här. En utmärkt genomgång av olika metoder för lumpsamling finner man i n.

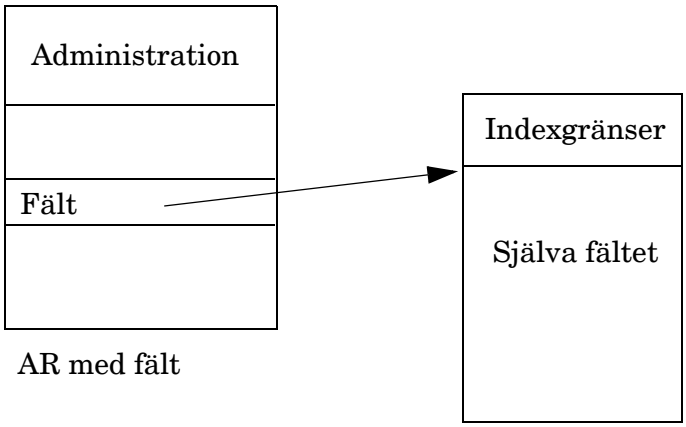
Objekt är inte bara en datastruktur som Pascals record och C:s struct, det är även något levande, som kan exekveras. Till objektet kan höra en kod på samma sätt som det gör det till en procedur. I det enkla fallet kan denna kod betraktas som en initialisering av objektet. I objektorientering använder man objekten för att skapa en modell av den verklighet man behandlar. Här finns då ett antal objekt som på något sätt lever parallellt med varandra och som vid vissa punkter dessutom eventuellt kommunicerar med andra objekt. Detta kan ske antingen verkligt parallellt på datorer med flera processorer, eller sker exekveringen växelvis. I det senare fallet kan det ske programmässigt, (kvasiparallellt) eller påtvingat av yttre händelser. I Simula tex finns kvasiparallellitet, detta implementeras så att under exekveringen av koden i ett objekt anropas RTS (detach), exekveringspunkt i objektet sparas och exekveringen fortsätter i ett annat objekt i enlighet med språkets regler. Detta brukar kallas korutiner i motsats till vanliga rutiner, procedurer. Detta medför ur RTS organisationssynpunkt att stackhantering vid uttrycksberäkningar kompliceras, registerhantering kompliceras och vilande objekt ligger i minnet. Observera att ett sådant exeveringsbyte av objekt kan komma även mitt i beräkningen av ett uttryck, på alla nivåer.

Inom objektorientering talar man om ärvning, hur påverkar nu detta ovan beskrivna strukturer? Med andra ord hur hanteras en instans av ett block med prefix? Vad gäller aktiveringsposten så görs en konkatenering av alla variabler så att endast ett AR skapas som innehåller hela instansens minne. Hur gör man då med den statiska informationen kod och template. Vi vill ju inte dubblera sådan information, tex koden. Därför får varje block i prefixkedjan sin egen koddel. Ett blocks koddel kan på så sätt ingå i många instansers kod. Detta kan man ta hand om i template, där vi kan lägga flera startadresser efter varandra. Då en instans kod skall exekveras så görs det genom att man i tur och ordning via template hoppar till respektive avsnitt. Det kan påpekas att i tex Simula så finns ett begrepp som heter inner, vilket gör att dessutom måste koden i ett block som förverkligar ett objekt (class) delas i två delar, före och efter inner.

## Allokering av andra datastrukturer.

Variabler och plats för parametrar och funktionsresultat allokeras i aktiveringsposter. För enkla typer som heltal och flyttal, men också för referenser till objekt krävs plats av fix storlek och detta är oproblematiskt. Vad gäller själva objekten får dessa ju ett eget AR, varför även detta är utan problem.

**Fältallokering och adressering.** Vad gäller fält (array) är det värre, dessa kan antingen vara av fix, av kompilatorn känd, storlek och skulle således kunna allokeras inom AR, men också av dynamisk, först under körning, bestämd storlek. Denna storlek är normalt känd då aktiveringsposten allokeras, varför även här skulle fälten kunna allokeras i AR. Problemet är då att instanser av ett och samma block skulle kunna ha olika storlek, vilket är besvärligt ur hanteringssynpunkt, bla för lumpsamling. I vissa språk kan man dessutom ha ännu mer dynamiska fält, vars storlek inte ens är känd under allokeringen av AR och kanske till och med vara av dynamiskt föränderlig storlek (open array). Vi tänker oss följande allokering av fält. Här allokerar vi



AR med fält

utrymme för själva fältet utanför AR, och i aktiveringsposten representeras fältet endast av adressen till det verkliga utrymmet. Ofta är det möjligt att allokera själva fältet på stacken just över AR. Vid fixa indexgränser är naturligtvis denna allokering ganska enkel och man skulle kunna klarat den på ett kanske enklare sätt, är indexgränserna variabla, men fixerade vid blockets allokering, är det lite mer komplicerat. Ovanstående metod duger dock fortfarande. Nästa komplikation är att ha dynamiskt föränderliga indexgränser, man kan tex utöka fältets storlek vid behov. Detta är dock ett betydligt större problem som oftast leder till att man får flytta fältet under körning. Effektiviteten blir lätt ganska låg. Detta senare fall bortser vi i fortsättnin-

gen ifrån. Här ges ett enkelt exempel med mellankod. Vi ger ett programexempel, samt

```
begin
        integer I;
        integer array A(1:5);
        A(I):=A(3)+2;
end
```

```
Label     1,0
Literal   0,Integer,1
Literal   1,Integer,5
Arrlim    1,Lit,0,0,Integer,Lit,0,1,Integer
Arrall    Ar,0,1,Pointer
Arrind    Ar,0,0,Integer
Arradr    Ar,0,1,pointer,Tmp,0,1,Pointer
Literal   2,Integer,3
Arrind    Lit,0,2,Integer
Arradr    Ar,0,1,Pointer,Tmp,0,2,Pointer
Literal   3,Integer,2
Add       Arr,2,0,Integer,Lit,0,3,Integer,Tmp,0,2,Integer
Move      Tmp,0,2,Integer,Arr,1,0,Integer,
```

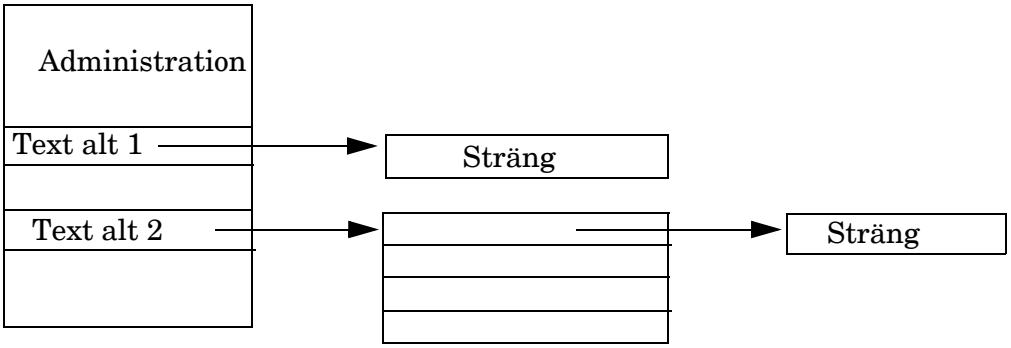
vid sidan av den relevanta delen av mellankoden. Mellankoden är egentligen endast en följd av heltal, men den ges här i en pseudovariant där en del heltal ersatts med minnesstödjande namn. Först ges nu en beskrivning av mellankoden för just fälthantering, sedan kommer en kommentar instruktion för instruktion av koden ovan.

Vi har ett antal mellankodsoperatorer för att hantera allokering och adressering av fält. ARRLIM och ARRALL används för allokering av fält. Detta görs när man börjar exekvera ett block. Antag att blocket har ett fält deklarerat, vi genererar då följande:
LABEL Bn
ARRLIM indexnummer,undre_gräns,övre_gräns
ARRALL refvariabel
ARRLIM används för att överföra undre och övre gräns till RTS. Indexnummer ovan anger för vilket index gränserna gäller. Vi är således förberedda på flerdimensionella fält. Gränserna anges som vanliga operander vilket gör att vi med detta klarar dynamiska gränser. Följande är således fullt möjligt: *integer array A(1:N+2,N-3:F(N,N+3))*. ARRALL används för att allokera minne för det fält som definieras med ARRLIM operationer. Resultatet av ARRALL är en referens till fältet, som lagras i refvariabel. Refvariabel anger alltid en plats i aktuell aktiveringspost. Varje fält som deklareras i blocket ger således upphov till dessa satser i början av blockets kod, de är exekverbara.
Sedan ska vi lämpligen kunna adressera enskilda element i ett fält. Till hjälp har vi då mellankodsoperatorerna ARRIND och ARRADR
ARRIND indexoperand
Här är indexoperanden en normal adress till ett index (Literal, temp eller variabel), som till RTS överför värdet på ett index.
ARRADR refvariabel,elementref
Här anger refvariabeln den pekare på fältet som beräknades av ARRALL, därefter beräknas adressen till det element som överförda index anger. Detta resultat läggs i operanden elementref som normalt är en temporär variabel. Vid adressberäkningen kan också RTS göra en kontroll av indexgränser. Detta betyder att vi har beräknat en adress till elementet, och att denna adress ligger i en temporär variabel. Vi använder sedan denna och det nya adresseringssättet ARR, som finns beskrivet i kapitlet om back-end.

Vi ser nu på exemplet ovan. Först definieras literalerna 1 och 5 som vi tänker använda som gränser. Med Arrlim sänder vi dessa gränser till RTS, för första (och här enda) index. Därefter görs Arrlim, som motsvarar en RTS-rutin som allokerar minne samt lägger adressen till detta område i aktiveringspostens utrymme för A (typen är pointer). Vi ska nu adressera fältelementet A(I), och sänder index (=I) till RTS. Därefter görs Arradr som

anropar RTS-rutin som kontrollerar indexgränser och beräknar adressen till angivet element. Arradr:s första parametrar är adressen till fältet och den andra är en resultatoperand, som här säger att adressen till elementet läggs i temporära variabeln nr 1. Därefter görs på motsvarande sätt adresseringen av elementet A(3). Då detta är ett fixt index kunde vi klarat av mer under kompileringen än vi gjort här. Sedan adresserar vi elementen i en Add-instruktion med Arr-metoden, som har den temporära variabeln som innehåller adressen till fältelementet som andra parameter.

**Texter.** Andra strukturer som allokeras på särskilt sätt är texter. I vissa enklare språk så är en text eller sträng just inget annat än en följd av tecken (characters). I andra språk är en text snarare ett objekt som innehåller dels positionspekare och liknande information, dels en sträng som representerar själva texten. Ett textobjekt kan också innehålla systemdefinierade procedurer. Ett textobjekt allokeras inte på samma sätt som ett vanligt objekt, utan snarare inifrån andra instanser under exekveringen. Själva textsträngen kan ha en storlek som är fullständigt dynamisk och föränderlig under exekvering. Fördelen med metoden som beskrivs här är att ett fält och en text alltid tar lika stor plats i aktiveringsposten, och det föränderliga utrymmet allokeras utanför i ett eget utrymme. Detta möjliggör en generell implementation av dynamiska storheter av olika typer. Nedan ser vi två alternativa sätt att allokera texter, det ena




är att man endast betraktar texten som en sträng, medan andera metoden ser texten som något mer än en sträng, som positionsräknare, mm. Ett ytterligare alternativ är att lägga textboxen direkt i AR, den har ju fix storlek. Hur man hanterar texter är också beroende av hur högnivåspråket hanterar texter.

**Undantag.** För att klara undantagshantering (“exceptions”) behöver man ytterligare lite struktur i template och aktiveringspost. I template tillkommer en adress till en undantagshanterare, som är den kod man hamnar i då man “fångar” (catch) ett undantag. Denna kod är således till för att hantera undantaget då det inträffat. Då undantaget inträffar skapar man ett objekt av typ “undantag” som man “kastar”, dvs. som undantagshanteraren får som parameter. Var finns då adressen till detta objekt? Man reserverar plats i aktiveringspostens huvud till en “variabel” som kan hålla ett sådant objekt när det är aktuellt. Det som behövs för undantagshanteringen är: en adress till en rutin i template, kod för rutinen, plats för ett undantagsobjekt (referens) i aktiveringsposten, samt lite kod i RTS för att administrera hanteringen. Enkla undantag är inte så svåra att implementera, medan återgång till rätt nivå och process i ett mer komplicerat system med icke-sekventiell exekvering

kan bli ganska komplicerad. Det RTS som det finns delar av i tillägg E innehåller en enkel undantagshantering.

**Separat kompilering.** Att kompilera delar av ett program separat är inte särskilt komplicerat, men det kräver lite extra hantering av symboltabeller, som ska vara kända mellan de separat kompilerade modulerna. Detta har inte så mycket med RTS och exekveringa att göra, men vissa saker måste man tänka på. Det behövs knappast något iRTS för detta, däremot behöver man kunna hantera externa adresser i den genererade objektkoden. Detta betyder att man behöver mellankodsinstruktioner som EXTDEF och EXTREF, vidare behöver man möjlighet att anropa vissa koddelar som subrutiner. Anropar man inte subrutiner annars? Jo men dels är det rutiner som ingår i RTS och som således länkas in fixt från ett bibliotek, dels är det högnivårutiner som anropas genom rutiner iRTS, som redan beskrivits tidigare i detta kapitel. Här behöver man generera ett stycke kod, som just hoppar till och återvänder från, man behöver mellankodsinstruktioner "PUSHJ och POPJ". Man kan också behöva komplettera templates med lite fler adresser än vad vi haft tidigare. Innan låg endast kodblocksadresser här, men vid lite mer komplicerade strukturer kan man behöva adresser här även till andra kodavsnitt (konstruktorer, undantagshanterare, "länkkod" till externa konstruktorer, mm).

**Dynamiskt inlänkad kod.** Normalt översätter man hela programmet inklusive alla separat kompilerade moduler innan man tänker köra sitt program, vidare är alla biblioteksrutiner översatta och inkluderade i bibliotek. Man länkar sedan samman sitt program från dessa redan klara objektkodsmoduler till ett körbart program. När detta sedan exekveras är det försent att översätta ytterligare kod. Speciellt när det gäller utvecklingsfasen av ett program är det av intresse att köra sitt program, stoppa det i en avlusningsfas och göra kompletteringar av koden. Detta går inte att göra med traditionella metoder. Det finns inkrementella översättare som kan klara av att översätta programdelar i den takt de behövs, samt att också ändra kod under exekvering och återöversätta den vid behov. I en sådan miljö använder man inte traditionell länkning, annat än av de grundläggande RTS-rutinerna som hanterar allt detta. Den "egna koden" dvs. den som kommer från högnivåprogrammet fogas in av RTS och länkning används inte. Ska man foga in redan översatta rutiner, kanske skrivna i c eller assemblyspråk, så måste man i RTS inkludera någon form av länkare. Detta betyder således att det finns delar av RTS som förstår sig på objektkod i relokerbar form och som kan överföra denna i absolut kod och infoga i den exekverbara koden, dvs. hantera adresser och liknande. Detta är fullt möjligt och en sådan lösning har utvecklats inom ett samnordiskt forskningsprojekt, Mjölner, och testats. För ytterligar information om detta hänvisas till särskilda skrifter i ämnet.