1 De Von Neumann machine

1.0 Inleiding

In dit hoofdstuk behandelen we de (klassieke) Von Neumann machine en laten we zien hoe eenvoudige programma's door deze machine kunnen worden uitgevoerd. Het belangrijkste kenmerk van de Von Neumann machine is dat het programma en de te bewerken gegevens in één en hetzelfde geheugen worden opgeslagen. Of een waarde in het geheugen een instructie uit het programma of een getal voorstelt is aan die waarde niet te zien: de interpretatie van de waarden in het geheugen wordt bepaald door de manier waarop deze waarden worden verwerkt.

Een ander belangrijk kenmerk van de Von Neumann machine is haar *eenvoud*. Deze eenvoud is het gevolg van de stand van de elektrotechniek in de tijd -- ca. 1945 -- waarin de machine werd ontwikkeld: het bouwen van een betrouwbaar werkende machine was in die tijd een verre van triviale aangelegenheid. Ondanks haar eenvoud is de machine *universeel*: elk algoritme kan, mits het geheugen van de machine groot genoeg is, in instructies van de machine worden gecodeerd. Dit heeft echter wel wat voeten in de aarde en voor de nu gangbare vormen van gebruik -- zoals recursie, datastructuren, gelijktijdige uitvoering van meer programma's -- is de Von Neumann machine in haar oorspronkelijke vorm dan ook onbruikbaar. Vanwege haar eenvoud is de machine echter zeer geschikt om de principes van machinecodeprogrammering te illustreren, en om de behoefte aan verfijndere mechanismen duidelijk te maken.

1.1 De opbouw van de machine

De machine bestaat uit twee componenten, namelijk:

-- het geheugen (Engels: store, Amerikaans: memory), en:
-- de processor (ook wel: CPU of engine).

Daarnaast dienen er uiteraard voorzieningen te zijn voor het invoeren van programma's en gegevens in de machine -- *input* -- en voor het naar buiten uitvoeren van de resultaten van de berekeningen -- *output* -- . Deze laten wij hier verder buiten beschouwing.

Het geheugen is een eindige collectie *cellen* (ook wel *woorden* genoemd). Iedere cel kan een waarde aannemen, waarbij de verzameling van mogelijke waarden eindig is. We nemen echter aan dat het aantal mogelijke waarden dat een cel kan aannemen *voldoende groot* is --waarover later meer-- . Vrijwel zonder uitzondering worden de cellen gerealiseerd als rijtjes *bits*, waarbij iedere bit 1 uit 2 waarden kan aannemen; een rijtje bits ter lengte k kan dan 2^k mogelijke waarden aannemen. Het getal k , dat voor alle cellen hetzelfde is, heet de *woordlengte* van het geheugen. De bitrijtjes kunnen worden geïnterpreteerd als gehele getallen --bijvoorbeeld volgens de *2's complement* representatie-- of als instructies -- volgens het *instruction format* van de processor-- .

Alle cellen hebben dezelfde eigenschappen. In het bijzonder zijn alle cellen even goed en even snel toegankelijk en hebben alle cellen dezelfde opslagcapaciteit: het geheugen is *homogeen*; men spreekt ook wel van een *random access* geheugen.

Om de cellen van elkaar te kunnen onderscheiden zijn ze opeenvolgend genummerd, vanaf 0 . Deze nummers worden *adressen* genoemd: een adres is een natuurlijk getal waarmee een geheugencel correspondeert. Door het aanbrengen van een nummering zijn de cellen tevens geordend: het geheugen is *lineair*. Wanneer dit geen verwarring geeft spreken we van *het adres van X* als we *het adres van de geheugencel die X bevat* bedoelen.

Het geheugen kan aldus worden beschouwd als een *array* van cellen. In het vervolg stellen we het geheugen voor door een variabele S van het type:

S(i:0⩽i<N) : **array of** cell ;

Hierin geeft N het aantal cellen weer waaruit het geheugen bestaat.

De processor voert *sequentieel* --dat is: één voor één-- instructies uit. Deze instructies staan in opeenvolgende cellen in het geheugen. Om de uit te voeren instructie te identificeren bevat de processor een *register* PI (programmaindex) waarvan de waarde een adres is. Dit register voldoet aan de volgende machine-invariant:

PI = "het adres van de volgende uit te voeren instructie" .

De werking van de processor kan nu eenvoudig worden gedefinieerd met behulp van het volgende programmaatje:

```
do true → |[ var IR: cell;
               IR, PI := S•PI, PI + 1
           ; "execute the instruction in IR"
          ]|
od .
```

Hierin is IR (instruction register) een hulpregister van de processor voor het tijdelijk opslaan van de uit te voeren instructie. De actie "execute the instruction in IR" veroorzaakt een toestandsverandering van het geheugen, of de processor, of beide, kortom van de gehele machine.

De gewenste toestandsverandering is als volgt in de instructies gecodeerd. Iedere instructie bestaat uit twee delen, een *operator* en een *adres*. De operator specificeert de gewenste toestandsverandering, waarbij het adres de daarbij betrokken geheugencel aangeeft. Ten behoeve van berekeningen beschikt de processor verder over een register A (accumulator) van hetzelfde type als de geheugencellen. De toestand van de machine bestaat dus uit het drietal $\langle PI, A, S \rangle$. (Merk op dat het instruction register niet tot de toestand van de machine behoort.) De volgende tabel bevat voor iedere instructie de bijbehorende toestandsverandering van de machine. Hierin staat a steeds voor het adres uit de instructie, terwijl de operatoren zijn aangegeven met namen -- load, store, etc. -- .

instructie:		**effect:**
load	a	A := S•a
store	a	S•a := A
add	a	A := A + S•a
subt	a	A := A − S•a
jump	a	PI := a
zjump	a	**if** A = 0 → PI := a [] A ≠ 0 → skip **fi**
njump	a	**if** A < 0 → PI := a [] A ≥ 0 → skip **fi**

De instructies jump, zjump en njump vormen de zogenaamde *spronginstructies* of *sequencing instructions*. Deze instructies beïnvloeden alleen de waarde van de programmaindex PI , en niet de rest van de toestand van de machine. Formeel betekent dit dat iedere assertie over de toestand van de machine waarin PI niet voorkomt een invariant is van de spronginstructies. De asserties in een guarded-commandsprogramma voldoen hieraan, want in de guarded-commandsnotatie komt het begrip PI niet voor. Met behulp van deze instructies kan de volgorde waarin de instructies worden uitgevoerd worden veranderd en zelfs --m.b.v. zjump en njump -- afhankelijk worden gemaakt van eerder uitgerekende waarden.

1.2 Het gebruik van machinecode

Aan de hand van een eenvoudig voorbeeld laten we zien hoe programma's in de guarded-commandsnotatie kunnen worden gecodeerd voor de machine. Als voorbeeld nemen we het bekende programma voor het berekenen van de grootste gemene deler van twee getallen --waarin X en Y positieve constanten zijn-- :

```
  x,y := X,Y  { invariant P:  0<x ∧ 0<y ∧ ggd·X·Y = ggd·x·y  }
; do x>y → x := x-y
   [] y>x → y := y-x
  od  { P ∧ x=y }
; z := x  .
```

In het programma komen constanten X,Y en variabelen x,y,z voor. Deze kunnen worden *gerepresenteerd* in het geheugen van de machine door voor iedere constante en variabele een cel te reserveren waarin de constante resp. de waarde van de variabele wordt opgeslagen. De relatie tussen constanten en variabelen enerzijds en geheugencellen anderzijds heet de *geheugentoewijzing* of *-allocatie*. Formeel kan de geheugenallocatie worden vastgelegd in een zogenaamde *representatieinvariant*. Bijvoorbeeld:

$$X = S{\cdot}0 \ \wedge \ Y = S{\cdot}17 \ \wedge \ x = S{\cdot}103 \ \wedge \ y = S{\cdot}104 \ \wedge \ z = S{\cdot}47 \ .$$

De in dit voorbeeld gekozen toewijzing is nogal willekeurig --daar komen we nog op terug-- ; belangrijk is wel dat verschillende variabelen door verschillende cellen worden gerepresenteerd.

De machine beschikt niet over instructies om constanten te "produceren"; daarom nemen we aan dat de constanten X en Y vóórdat de machine wordt

gestart op de juiste plaatsen in het geheugen zijn gezet, tezamen met de instructies van het programma. De assignment x := X moet nu, bij de gekozen allocatie, worden geïmplementeerd als S·103 := S·0 ; omdat er geen instructie is met dit effect herschrijven we dit tot A := S·0 ; S·103 := A , wat kan worden gecodeerd als load 0 store 103 .

Voor de implementatie van de repetitie gebruiken we de volgende *ontvouwingseigenschap*:

DO: **do** B → S **od** is equivalent met

if ¬B → skip [] B → S ; DO **fi** .

De repetitie DO kan dus worden vervangen door de door ontvouwing verkregen selectiestatement. Hierin komt DO nog steeds voor; omdat deze equivalent is met de gehele constructie mag deze DO worden gecodeerd als jump a , waarin a het adres van de eerste instructie van de selectie is. Hierbij maken we gebruik van de eigenschap dat spronginstructies de toestand van de berekening niet beïnvloeden.

Uitvoering van een selectie vereist het uitvoeren van één van de guarded commands waaruit de selectie bestaat, en wel een waarvan de guard de waarde true heeft. Een mogelijke implementatie hiervan is eerst de guards te evalueren totdat er één de waarde true oplevert, en vervolgens de bijbehorende statement uit te voeren. Het onderscheiden van alle gevallen en het selecteren van (de code van) de juiste statement kan eveneens met spronginstructies worden gecodeerd.

Boolean waarden kunnen, op allerlei manieren, door getallen worden voorgesteld. Een veel voorkomende representatie is 0 voor false en 1 voor true , maar deze keuze is betrekkelijk willekeurig. Ieder tweetal *verschillende* waarden voldoet. De keuze 1 voor false en 0 voor true , bijvoorbeeld, voldoet even goed; het gebruik van 0 en -1 komt ook voor. Bij gebruik van 0 en 1 kan de zjump instructie worden gebruikt om de selectie van de uit te voeren statement te coderen; verder kunnen extra instructies voor de boolean bewerkingen -- and, or, not, etc. -- eenvoudig worden gerealiseerd.

Het programmaatje voor de ggd kan nu als volgt worden gecodeerd. Hierbij is de hierboven aangegeven geheugenallocatie voor de constanten en variabelen gebruikt en is aangenomen dat de instructies in het geheugen worden ondergebracht in de cellen vanaf adres 200. Door middel van een passende annotatie kan de correctheid van de machinecode eenvoudig worden geverifieerd. Merk op dat het omzetten van een programma in machinecode als een gewone *programmatransformatie* kan worden beschouwd. Alleen het gebruik van de spronginstructies vergt speciale aandacht.

```
adres: 200      { true }
                  load   0
       201      { A=X }
                  store  103
       202      { x=X }
                  load   17
       203      { x=X ∧ A=y }
                  store  104
                { x=X ∧ y=y , dus: }
       204      { invariant P:  0<x ∧ 0<y ∧ ggd·X·y = ggd·x·y }
                  load   103
       205      { P ∧ A=x }
                  subt   104
       206      { P ∧ A=x-y }
                  zjump  214
       207      { P ∧ A=x-y ∧ A≠0 , dus: x≠y }
                  njump  210
       208      { P ∧ A=x-y ∧ A≠0 ∧ ¬(A<0) , dus: x>y }
                  store  103
       209      { P }
                  jump   204
       210      { P ∧ A=x-y ∧ A<0 , dus: y>x }
                  load   104
       211      { P ∧ A=y ∧ y>x }
                  subt   103
       212      { P ∧ A=y-x ∧ y>x }
                  store  104
       213      { P }
                  jump   204
       214      { P ∧ x=y , dus: x=ggd·X·y }
                  load   103
       215      { A=x ∧ x=ggd·X·y }
                  store  47
                { z=ggd·X·y }
```

Het gebruik van de spronginstructies brengt extra bewijsverplichtingen met zich mee. De instructie op adres 204 , bijvoorbeeld, heeft als preconditie P ; dit is alleen correct als *alle* spronginstructies jump 204 eveneens P als preconditie

hebben. Ga na dat dit in bovenstaande code inderdaad geldt; ga verder na dat $A \neq 0$ in de postconditie van zjump en dat $\neg(A < 0)$ in de postconditie van njump mag voorkomen.

1.3 Assembleertaal

Het met de hand coderen van een programma is niet moeilijk, maar wel bewerkelijk en het brengt nogal wat administratie met zich mee: pas nadat het gehele programma is gecodeerd, is bekend op welke plaatsen in het geheugen welke instructies komen te staan; pas daarna kunnen de in de spronginstructies voorkomende adressen worden ingevuld. Evenzo kunnen de adressen in de andere instructies pas worden ingevuld nadat de geheugenallocatie is gekozen.

Verder is het hele proces erg gevoelig voor (schrijf)fouten. De gevolgen van, bijvoorbeeld, het schrijven van zjump 213 in plaats van zjump 214 -- in ons voorbeeld -- zijn desastreus. (Ga dit zelf na.)

Een (historisch) eerste poging om althans een gedeelte van het administratieve werk te mechaniseren is de introductie van *assembleertaal*. Dit is gewone machinecode, waarin echter namen kunnen worden gebruikt om adressen aan te geven. Met behulp van een *assembleerprogramma* -- Engels: *assembler* -- kan de machine zelf worden gebruikt om deze namen door de bijbehorende adressen te vervangen. Om het gebruik van namen in spronginstructies op eenvoudige manier mogelijk te maken kunnen namen in de kantlijn links van de instructies worden geplaatst. Zulke namen heten *labels*; een label representeert het adres van de instructie waar hij vóór staat.

In assembleertaal zou ons voorbeeldprogramma als volgt kunnen worden gecodeerd. Hierin is equ een *pseudoinstructie* -- een instructie t.b.v. het assembleerproces -- die aangeeft dat de ervoor staande naam het erachter staande adres representeert; aldus kan de gekozen geheugenallocatie aan het begin van de code worden samengevat. De pseudoinstructie start geeft aan dat de instructies zullen worden ondergebracht in de geheugencellen vanaf adres 200. In onderstaande code is de annotatie weggelaten; deze is precies gelijk aan die in de vorige versie:

```
X       equ     0
y       equ     17
x       equ     103
y       equ     104
z       equ     47
        start   200
```

```
        load    X
        store   x
        load    y
        store   y
do      load    x
        subt    y
        zjump   od
        njump   if1
if0     store   x
        jump    do
if1     load    y
        subt    x
        jump    do
od      load    x
        store   z
```

1.4 Compilers en Operating Systems

Het transformeren van een programma in equivalente machinecode -- dit proces wordt ook wel *vertalen* genoemd -- kan geheel worden gemechaniseerd. Hiervoor wordt een *compiler* gebruikt. Omdat de betekenis van een programma is gedefinieerd aan de hand van de syntactische opbouw van dat programma, bevat iedere compiler een component die deze syntactische opbouw uit de programmatekst reconstrueert. Deze component wordt *parser* genoemd en het reconstructieproces zelf *parsing*. Met behulp van de aldus verkregen informatie zorgt een tweede component van de compiler, de *codegenerator*, voor het samenstellen van de machinecode.

Zoals het vertalen van het programma in machinecode volledig kan worden gemechaniseerd, zijn er nog enige aspecten die voor automatisering in aanmerking komen, zoals het verzorgen van input en output en het verzorgen van de geheugenallocatie. In principe kunnen deze taken in de compiler worden ondergebracht. De manier waarop input en output plaatsvinden hangt echter sterk af van de specifieke configuratie van de machine en van de manier waarop deze wordt gebruikt. Evenzo hangt de manier waarop de geheugenallocatie het beste kan plaatsvinden af van de manier waarop de machine wordt gebruikt. Om redenen van eenvoud en flexibiliteit worden input en output en *geheugenbeheer* dan ook niet door de compiler verzorgd maar door een speciaal programma, het *operating system* van de machine. Dit programma is voortdurend in (een deel van) het geheugen van de machine aanwezig.

1.5 Diversen << nog te schrijven >>

>>> wellicht moet dit een apart hoofdstuk, aan het eind, worden <<<

Hierin komen aan de orde:
-- de relatie tussen woordlengte en geheugengrootte; 16-bits machines.
-- bits, bytes, words, double words, ··· , en het alignmentprobleem.
-- meer-adres machines.

1.6 Opgaven

0. Construeer geannoteerde machinecode voor het volgende programma, dat eveneens geschikt is voor de berekening van ggd•X•Y :

```
  x , y := X , -y
; do x + y > 0 → x := x + y
   [] x + y < 0 → y := x + y
  od
; z := x  .
```

1. Kies een representatie voor de boolean waarden en ontwerp stukjes code voor de statements $a := b \wedge c$, $a := b \vee c$, $a := \neg b$, $a := x = y$ en $a := x > y$.

2 Segmentadressering

2.0 Self-modifying code

In de instructies van de in het vorige hoofdstuk behandelde machine komen geheugenadressen als constanten voor. Ten behoeve van de implementatie van, onder andere, *arrays* is het echter nodig dat het adres van een geheugencel kan worden berekend. In zijn eenvoudigste vorm komt dit hierop neer: kunnen we een stukje code maken dat de assignment A := S•A realiseert?

Omdat de adressen in de code constant zijn is de enige manier om dit te doen de code tijdens programmauitvoering te laten veranderen. Men spreekt dan van *self-modifying code*. Hierbij gebruiken we de eigenschap dat zowel getallen als instructies door bitpatronen in het geheugen worden voorgesteld. Het is dus mogelijk het bitpatroon dat een instructie voorstelt als een getal op te vatten. We geven dit aan door de betreffende instructie tussen haakjes ⟦ en ⟧ te plaatsen: voor iedere instructie x is ⟦ x ⟧ ("getal x") dan het overeenkomstige getal. De representatie door bitpatronen heeft nu de eigenschap --dit is bij vrijwel alle machines zo-- dat voor iedere operator **op** en voor ieder adres a geldt:

$$⟦\ \mathbf{op}\ a\ ⟧ = ⟦\ \mathbf{op}\ 0\ ⟧ + a\ .$$

Met behulp hiervan kan het gewenste stukje code worden geconstrueerd. Hierin is M het adres van een geheugencel met daarin een load 0 instructie; de initiële waarde van S•L is irrelevant --ga zelf na waarom-- :

```
            { A = a ∧ S•M = ⟦ load 0 ⟧ }
              add    M
            { A = ⟦ load a ⟧ }
              store  L
L           { S•L = ⟦ load a ⟧ }
              ......
L+1         { A = S•a }


M             load   0  .
```

Het gebruik van self-modifying code is om verschillende redenen ongewenst. Ten eerste is het programma --de tekst van het uit te voeren algoritme-- juist dat wat gedurende een berekening constant blijft; het gebruik van self-modifying code vertroebelt dit: wat is nu, op enig moment tijdens de berekening, "het" programma? Ten tweede is, omdat de programmacode tijdens uitvoering niet constant is, *code sharing* --het tegelijkertijd meer dan eens in uitvoering nemen van dezelfde code-- onmogelijk. Ten derde kan zulke code niet in zogenaamd *read-only memory* worden ondergebracht.

Om het gebruik van self-modifying code onnodig te maken dient te machine te worden voorzien van een uitgebreider mechanisme om het geheugen te adresseren.

2.1 Code sharing

Het tegelijkertijd meer dan eens uitvoeren van dezelfde code komt voor in machines met mogelijkheden voor *multiprocessing*, maar ook bij het uitvoeren van *recursieve* procedures. Het gegeven dat hetzelfde programma meer dan eens in uitvoering kan zijn vereist dat we onderscheid moeten maken tussen de programma*tekst* en de programma*uitvoering*; van de laatste kunnen er nu immers meer dan één per programmatekst zijn. Ten behoeve van dit onderscheid hanteren we daarom de volgende interpretatie:

- programma: de tekst van het programma --ook: de code-- .
- proces : de uitvoering van een programma --ook: een berekening-- .

De in een programma voorkomende variabelen representeren de toestand van het proces dat ontstaat bij uitvoering van dat programma. Het proces speelt zich af in een *toestandsruimte* --Engels: state space-- , die alle mogelijke toestanden van het proces bevat. Verschillende processen spelen zich in principe in verschillende toestandsruimten af, ongeacht of die processen uitvoeringen van hetzelfde programma zijn of niet. Daarom is er ook per proces, en niet per programma, een inbedding in het geheugen nodig van de variabelen die de toestandsruimte van het proces vormen. We hebben dus behoefte aan geheugentoewijzing per proces, en niet per programma. Om hetzelfde programma in verschillende processen te kunnen gebruiken moet de code van dat programma onafhankelijk zijn van de feitelijke geheugentoewijzing. De in het vorige hoofdstuk behandelde code is daarentegen juist sterk afhankelijk van de geheugentoewijzing: door het in de code voorkomen van adressen is de code *allocatiegebonden*.

Om programma's allocatieonafhankelijk te kunnen coderen dient te machine

eveneens van een flexibeler adresseringsmechanisme te worden voorzien.

2.2 Relocating loaders

Zelfs wanneer de mogelijkheid van code sharing niet van belang is is een flexibele vorm van geheugenallocatie gewenst. Wanneer hetzelfde programma vaker dan eens, maar niet tegelijkertijd, wordt uitgevoerd kan de configuratie van de machine van uitvoering tot uitvoering variëren, bijvoorbeeld omdat hetzelfde programma op verschillende exemplaren van hetzelfde machine*type* wordt gebruikt. In zulke gevallen is het wenselijk dat de geheugenallocatie pas wordt gekozen op het moment waarop het programma in uitvoering wordt genomen.

Een --zelfs tegenwoordig nog-- veelgebruikte manier om dit te bewerkstelligen is het gebruik van een *relocating loader*. De loader is het, altijd in de machine aanwezige, programmaatje dat dient voor het vanaf een extern geheugenmedium --bijvoorbeeld: ponsband, floppy disk-- in het geheugen plaatsen van het uit te voeren programma. Een relocating loader is een loader die tijdens het "laden" van het programma de in de code voorkomende adressen op systematische manier aanpast aan de te gebruiken geheugenallocatie.

In multiprocessing systemen is het zo mogelijk hetzelfde programma tegelijkertijd meer dan eens in uitvoering te nemen, mits men voor lief neemt dat er dan voor ieder proces een, door de relocating loader aangepaste, copie van het programma in het geheugen wordt geplaatst. Bij grote en veelgebruikte programma's --zoals compilers-- leidt dit tot een weinig efficiënte benutting van de meestal schaarse geheugenruimte. Het maken van een aan de allocatie aangepaste copie van de code kost niet alleen extra geheugenruimte maar ook rekentijd. Voor de implementatie van recursie is deze techniek dan ook onacceptabel inefficiënt.

De behoefte aan relocating loaders kan eveneens worden vermeden door de code allocatieonafhankelijk te maken.

2.3 Segmenten en indexregisters

Een *segment* is een rijtje opeenvolgende geheugencellen. Met $S(a : p \leqslant a < q)$ geven we het segment aan dat de cellen $S{\cdot}a$ bevat, voor alle a met $p \leqslant a < q$. Het adres van de cel met het laagste nummer, p, noemen we tevens het adres van het segment. De *lengte* van dit segment is $q-p$. In het vervolg karakteriseren we segmenten door hun adres en hun lengte. Het segment met adres p en lengte l is dus $S(a : p \leqslant a < p+l)$.

De elementen van een segment kunnen worden geïdentificeerd door hun adressen maar ook door hun *relatieve posities* ten opzichte van het begin van dat segment: element $S{\cdot}a$, $p \leqslant a < p+l$, identificeren we door middel van $a-p$, waarvoor dan geldt $0 \leqslant a-p < l$. Aldus worden de elementen van het segment geïdentificeerd op een manier die onafhankelijk is van de plaats van dat segment in het geheugen.

Een segment ter lengte l bevat l elementen die we identificeren door consecutieve nummering $i : 0 \leqslant i < l$; als het segment adres p heeft dan is element i van dat segment $S{\cdot}(p+i)$. Het adres van ieder element van een segment wordt nu bepaald door een paar (p,i) , waarin p het adres van het segment is en i de relatieve positie van het element binnen dat segment.

Een *indexregister* is een register waarvan de waarde een adres is. Door de processor van indexregisters te voorzien en het instructierepertoire passend uit te breiden wordt het mogelijk programma's op een allocatieonafhankelijke manier te coderen. Indexregisters worden ook wel *adresregisters* genoemd, terwijl registers zoals de accumulator ter onderscheid wel *dataregisters* worden genoemd. Registers die als adres- en als dataregister kunnen worden gebruikt heten wel *general-purpose* registers.

2.4 Een uitgebreide instructieset

Instructies zoals load a hebben impliciet betrekking op de accumulator A . Wanneer de machine over meer registers beschikt zal in de instructie moeten worden aangegeven op welk register de instructie betrekking heeft. Verder bestaat de uitbreiding van de machine hierin dat het in iedere instructie voorkomende adres wordt vervangen door een meer algemene *operand* om een geheugencel te identificeren.

Voor het definiëren van de instructies gebruiken we BNF-notatie:

instruction	→	jumpinstruction | registerinstruction
jumpinstruction	→	jumpoperator address
registerinstruction	→	operator register operand
jumpoperator	→	'jump' | 'zjump' | 'njump'
operator	→	'load' | 'store' | 'add' | 'subt' | ··· .

Als operand laten we nu niet alleen adressen toe maar ook constanten, registers en paren bestaande uit een indexregister en een constante of een register:

```
operand   → constant | address | register | indexpair
constant  → '#' number
address   → number
register  → 'A' | 'B' | 'C' | ···
indexpair → register ',' index
index     → number | register  .
```

De registers A, B, C, ··· beschouwen we als general-purpose registers. Ze kunnen zowel als dataregisters en als adresregisters worden gebruikt. Het effect van de registerinstructies kan als volgt worden gedefinieerd; hierin geeft R een register en ⟦ x ⟧ de *betekenis* van de operand x aan:

instructie:	**effect:**
load R x	R := ⟦ x ⟧
store R x	⟦ x ⟧ := R
add R x	R := R + ⟦ x ⟧
subt R x	R := R − ⟦ x ⟧

De verschillende mogelijke operanden hebben de volgende betekenis:

operand:	**betekenis:**
#a	a
a	S•a
R	R
[R, a]	S•(R+a)
[R, R']	S•(R+R')

Het effect van, bijvoorbeeld, load A #3 is nu A:=3 ; De instructie store A #3 is uiteraard niet zinvol: aan een constante kun je geen waarde toekennen. Daarom beschouwen we store R #a als een ongeldige combinatie; alle andere combinaties zijn wel zinvol. Met load R R' kan de inhoud van een register in een ander register worden gecopieerd: het effect van load R R' is R:=R' ; met store R' R wordt hetzelfde bewerkstelligd.

Het gebruik van indexparen heet *geïndexeerde addressering*. Met behulp hiervan kunnen de elementen van segmenten worden geadresseerd. Hiertoe zorgen we ervoor dat het adres van het segment zich in een indexregister bevindt.

voorbeeld: Het te adresseren segment heeft adres p en lengte l . Als we aannemen dat B = p dan:

```
{ B=p ∧ 0⩽i<l }
  load  A  [B,i]
{ A=S·(p+i) }
```

en:

```
{ B=p ∧ A=i ∧ 0⩽i<l }
  load  A  [B,A]
{ A=S·(p+i) }
```

□

In dit voorbeeld wordt A als dataregister en B als adresregister gebruikt. Merk op dat het adres p van het segment niet meer in de code voorkomt: de code is nu onafhankelijk van de plaats van het segment in het geheugen.

2.5 Geïndexeerde en relatieve spronginstructies

In spronginstructies laten we in plaats van adressen ook indexparen [R,a] toe, met als effect:

jump [R,a] betekent: PI := R + a .

Met behulp hiervan kunnen spronginstructies op een allocatieonafhankelijke manier worden gecodeerd.

Een andere manier om adressen uit spronginstructies te verwijderen is het gebruik van zogenaamde *relatieve* spronginstructies. Hiermee wordt bedoeld: relatief ten opzichte van het programma-indexregister PI . Hierbij wordt PI als indexregister gebruikt:

jump [PI,a] betekent: PI := PI + a .

Hierin is a een constante die zowel positief -- een *voorwaartse* sprong -- als negatief -- een *achterwaartse* sprong -- kan zijn.

2.6 Een toepassing

Een eenvoudige manier om een programma allocatieonafhankelijk te maken is de volgende. De code van het programma wordt ondergebracht in één segment, dat we het *codesegment* noemen. De programmeur gaat ervan uit dat het adres van het codesegment in register C staat; bij het coderen gebruikt hij uitsluitend geïndexeerde of relatieve spronginstructies. De variabelen van het programma worden ook in één segment ondergebracht, dat we het *datasegment* noemen. De programmeur gaat ervan uit dat het adres van het datasegment in register D staat. Hij kiest de manier waarop de variabelen van het programma in het datasegment worden ondergebracht. Voor het adresseren hiervan gebruikt hij uitsluitend de operandvormen [D,i] en, eventueel, [D,R] . In het programma komen geen instructies voor die de registers C en D wijzigen: zij fungeren als constanten en representeren de geheugenallocatie van het proces.

Bij het in uitvoering nemen van het programma moeten er nu in het geheugen twee, uiteraard disjuncte, segmenten van de juiste grootte worden gereserveerd. De loader plaatst de code van het programma in het codesegment. De adressen van deze segmenten worden in de registers C en D geplaatst. Het administreren van vrije en in gebruik zijnde geheugenruimte, het reserveren van ruimte voor segmenten, het laden van programma's en het initialiseren van de registers C en D zijn taken van het *operating system* van de machine.

Code sharing is nu eenvoudig mogelijk: wanneer twee processen uitvoeringen van hetzelfde programma zijn, dan kunnen zij hetzelfde codesegment gebruiken maar hebben zij verschillende datasegmenten.

voorbeeld: Het programma voor de ggd kan nu als volgt worden gecodeerd. Hierin geeft de pseudoinstructie data 3 aan dat voor de uitvoering van het programma een datasegment ter lengte 3 nodig is. Dit is in feite een instructie aan de loader. Door het gebruik van de pseudoinstructie start 0 geven de labels in het programma nu de relatieve posities van de betreffende instructies ten opzichte van het begin van het codesegment aan:

```
        data    3
x       equ     0   { variabele x op plaats 0 in het datasegment }
y       equ     1   { variabele y op plaats 1 in het datasegment }
z       equ     2   { variabele z op plaats 2 in het datasegment }
        start   0
        load  A #X
        store A [D,x]
```

```
        load  A  #y
        store A  [D,y]
do      load  A  [D,x]
        subt  A  [D,y]
        zjump    [C,od]
        njump    [C,if1]
if0     store A  [D,x]
        jump     [C,do]
if1     load  A  [D,y]
        subt  A  [D,x]
        store A  [D,y]
        jump     [C,do]
od      load  A  [D,x]
        store A  [D,z]
```

2.7 Opgaven

0. Wat is het effect van jump [PI,0] ?
   Wat is het effect van jump [PI,-1] ?
   Welke instructie bewerkstelligt A := S•A ?

1. In een programma komen twee integer variabelen x en y voor en twee variabelen s en t van het type **array** [0..99] **of** integer . Geef aan hoe deze variabelen in één datasegment kunnen worden ondergebracht en geef geannoteerde stukjes (allocatieonafhankelijke) code voor de volgende assignments:
   x := s[y] , s[x] := s[x] + y , s[x] := t[y] en s := t .

2. Gegeven is een collectie van N segmenten. De adressen van deze segmenten zijn zelf opgeslagen in een segment ter lengte N ; we nemen aan dat het adres van dit laatste segment zich in register C bevindt.
   a) Construeer een geannoteerd stukje code met als postconditie:
      A = "element j van segment i" , voor constanten i en j , $0 \leqslant i < N \wedge 0 \leqslant j$.
   b) Aannemende dat de N segmenten code bevatten, construeer een stukje code dat een "sprong naar element j van segment i" bewerkstelligt, voor constanten i en j , $0 \leqslant i < N \wedge 0 \leqslant j$.

3 Twee ontwerpregels

3.0 Inleiding

De manier waarop we in het vorige hoofdstuk de code van een programma onafhankelijk van de geheugenallocatie hebben gemaakt vertoont een patroon dat vaker voorkomt. Dit patroon kunnen we vastleggen in twee ontwerpregels. Deze regels zijn voor het eerst expliciet geformuleerd door E.W. Dijkstra [0]. Hoewel we ze hier vooral toepassen op het probleem van de geheugenadressering zijn de regels van algemeen belang bij het ontwerpen van een flexibele en toch efficiënte wijze van gegevensopslag --bijvoorbeeld: databases-- . Ter illustratie citeren we uit [0]:

[···] *I was pleasantly surprised when observing how clear the guidance was that came from the combination of the two design principles --avoid the frequent processing of open nomenclatures and avoid the duplication of volatile information-- . The principles themselves are not very surprising, but their combination is surprisingly effective.*

3.1 Avoid duplication of volatile information

In het klassieke model neemt de programmeur de allocatiebeslissingen. Het resultaat hiervan is dat iedere programmavariabele wordt geïdentifieerd door zijn adres; deze adressen worden vervolgens vrijelijk in de programmacode gebruikt. Aldus wordt informatie over de genomen allocatiebeslissing op grote schaal in de code *gedupliceerd*. Hetzelfde geldt voor de keuze van de plaats van de code in het geheugen: door het voorkomen van adressen in sprongінstructies wordt de informatie over de gekozen plaats in de code zelf gedupliceerd.

Het herzien van een allocatiebeslissing vereist bij deze werkwijze dat alle voorkomens van de relevante adressen worden gelocaliseerd en gewijzigd. Wanneer dit maar vaak genoeg voorkomt geeft het hiermee gepaard gaande werk aanleiding tot een onaardvaardbaar efficiëntieverlies. Als de geheugenallocatie regelmatig wordt veranderd dient de informatie die deze allocatie weergeeft als *vluchtig* te worden beschouwd. De moraal van het verhaal is dat, omwille van de efficiëntie, het (ongebreideld) dupliceren van vluchtige informatie moet worden vermeden. Aldus kunnen we de volgende ontwerpregel formuleren:

Avoid duplication of volatile information

Deze regel kan op twee manieren worden nageleefd: door informatie niet te dupliceren en door informatie niet vluchtig te laten zijn.

3.2 Avoid frequent processing of open nomenclatures

Het dupliceren van vluchtige informatie kan worden vermeden door aan die informatie een constante --niet-vluchtige-- *naam* te geven, de informatie zelf op slechts één plaats op te slaan en vervolgens slechts via die constante naam naar de informatie te verwijzen. Omdat de naam constant is kan deze naam vrijelijk worden gedupliceerd. In het in §2.6 behandelde voorbeeld is het adres van het datasegment vluchtig; de enige plaats waar dit adres wordt opgeslagen is register D . De naam van dit register fungeert nu als de constante identificatie van het datasegment. De programmavariabelen werden hierbij geïdentificeerd door hun relatieve posities ten opzichte van het begin van het datasegment. Deze posities vormen een (eveneens constante) naamgeving van de variabelen; zij mogen dan ook vrijelijk in de code worden gedupliceerd.

Deze werkwijze vereist wel een --niet al te inefficiënt-- mechanisme waarmee een naam kan worden afgebeeld op (het adres van) de erbij behorende informatie. Hierbij spelen *gesloten* naamgevingen een belangrijke rol. Een gesloten naamgeving is een interval $i : m \leqslant i < n$, voor $m, n : m \leqslant n$, van de gehele getallen. Meestal geldt hierbij $m = 0$, maar noodzakelijk is dat niet: $i : -37 \leqslant i < 13$ is een gesloten naamgeving met 50 elementen. Voor het contrast noemen we iedere verzameling namen die niet een gesloten naamgeving vormt een *open* naamgeving.

Gesloten naamgevingen hebben aantrekkelijke eigenschappen:

- de verzameling kan eenvoudig worden gekarakteriseerd (en dus ook gerepresenteerd), namelijk door de getallen m en n .
- er is een eenvoudige geldigheidstest op namen mogelijk: "i zit in de verzameling" is equivalent met $m \leqslant i \wedge i < n$.
- de namen kunnen compact worden gerepresenteerd: de getallen $i : 0 \leqslant i < n$ kunnen in ongeveer $^2\log \cdot n$ bits worden gecodeerd.
- de namen kunnen efficiënt op de erbij behorende waarden worden afgebeeld, namelijk door middel van geïndexeerde adressering --iets abstracter: met behulp van een array-- .

Met name de afbeelding van namen op waarden is bij open naamgevingen veel lastiger op een efficiënte manier te realiseren; dit vereist technieken zoals *searching* en *hashing*. De moraal van dit verhaal is dan ook de volgende ontwerpregel:

Avoid frequent processing of open nomenclatures

voorbeeld: De namen van alle mensen in een stad vormen een open naamgeving, terwijl hun telefoonnummers een (min of meer) gesloten naamgeving vormen. Vandaar dat de PTT nummers gebruikt! De afbeelding van namen van mensen op telefoonnummers wordt, via het telefoonboek, aan de gebruikers van het telefoonnetwerk overgelaten. Hoewel het tegenwoordig mogelijk is deze afbeelding te automatiseren was dat in de tijd waarin de automatische telefooncentrales hun intrede deden zeker niet zo.

□

Ter illustratie bekijken we het voorbeeld uit § 2.6 nogmaals. Wanneer het geheugen van de machine bijvoorbeeld 10000 cellen groot is dan zijn voor de binaire representatie van ieder adres tenminste 14 bits nodig. Voor de codering van de nummers van de 3 programmavariabelen volstaan 2 bits. Instructies waarin geïndexeerde adressering wordt gebruikt kunnen in de regel in minder bits worden gecodeerd dan instructies waarin adressen voorkomen: de bij geïndexeerde adressering voorkomende relatieve posities zijn *vaak* -- maar niet altijd -- kleine getallen; door in de instructiecode een compacte versie van de indexpairs -- zie § 2.4 -- op te nemen kunnen programma's compacter worden gecodeerd. Dit levert niet alleen ruimte- maar ook snelheidswinst op: voor het ophalen van de instructies heeft de processor nu minder geheugenaccessen nodig.

De oplossing in § 2.6 paart eenvoud en efficiëntie aan *flexibiliteit*: het is nu zelfs mogelijk de programmauitvoering tijdelijk te onderbreken, tijdens deze onderbreking het data- of codesegment in het geheugen te verplaatsen, en vervolgens na aanpassing van uitsluitend het D respectievelijk C register het proces te hervatten. Dit heet *dynamic relocation*.

referentie

[0] E.W. Dijkstra
On not duplicating volatile information
EWD 719, Nuenen, 1979.

4 De inbedding van variabelen in het geheugen

4.0 Inleiding

In dit hoofdstuk beschouwen we een klasse eenvoudige programma's, namelijk programma's met alleen *globale* variabelen, zonder binnenblokken of procedures. We geven een systematische behandeling van hoe de variabelen van zulke programma's in het geheugen kunnen worden gerepresenteerd. Hierbij onderscheiden we 4 *categorieën* variabelen, afhankelijk van de hoeveelheid geheugencellen die nodig is om (de waarde van) iedere variabele te representeren:

i: Voor iedere variabele is 1 cel voldoende. In deze categorie vallen variabelen van eenvoudige typen, zoals integer, boolean, character, real, en enumeratietypen.

ii: Voor iedere variabele is een constant en van tevoren bekend aantal cellen voldoende. In deze categorie vallen bijvoorbeeld records en Pascal arrays.

iii: Voor sommige variabelen is het aantal benodigde cellen gedurende de berekening weliswaar constant, maar dit aantal is niet van tevoren bekend (bijvoorbeeld omdat dit aantal per uitvoering van het programma anders kan zijn). In deze categorie vallen bijvoorbeeld Algol-60 arrays en arrays in guarded commands programma's.

iv: Het voor een variabele benodigde aantal cellen varieert zelfs tijdens de uitvoering van het programma. In deze categorie vallen "dynamische datastructuren" zoals *stapels*, *lijsten* en *bomen*.

Categorie iv laten we in deze cursus buiten beschouwing. Merk op dat de implementatie van datastructuren uit deze categorie in veel programmeertalen, zoals Pascal, simpelweg aan de programmeur wordt overgelaten.

De hier gegeven categorieën zijn geordend: iedere categorie omvat alle eraan voorafgaande en is in deze zin dus algemener. Iedere voor een categorie behandelde oplossing kan derhalve ook voor alle eenvoudigere categorieën worden gebruikt. We behandelen uitsluitend allocatieonafhankelijke oplossingen. Bij iedere oplossing moeten we dan ook aangeven hoe de variabelen in de programmacode worden benoemd -- *naamgeving* --, hoe de variabelen in het geheugen worden gerepresenteerd -- *inbedding* -- en hoe de namen tijdens programmauitvoering op de overeenkomstige adressen worden afgebeeld -- *adressering* --.

Iedere behandelde oplossing kan met de instructieset uit §2.4 worden

geïmplementeerd, maar we werken dat niet steeds uit. Wel laten we met voorbeelden zien dat, met name voor de wat ingewikkeldere gevallen, een met de gekozen adressering overeenkomende uitbreiding van het instructierepertoire de efficiëntie van de adressering kan verhogen.

4.1 Categorie i

Noem het aantal in het programma gedeclareerde variabelen N. Door deze variabelen, op welke manier dan ook, oplopend vanaf 0 te nummeren verkrijgen we de gesloten naamgeving $i : 0 \leqslant i < N$. Voor iedere variabele is 1 geheugencel nodig; voor alle variabelen zijn dus N cellen toereikend. Hiervoor gebruiken we een segment ter lengte N en gebruiken hiervan de cel op (relatieve) positie i voor variabele i. Met het adres van dit segment in register D is het adres van variabele i dan $D+i$. Voor de adressering kan geïndexeerde adressering worden gebruikt; bijvoorbeeld: load A [D,i] kent de waarde van variabele i aan register A toe. Het in § 2.6 besproken voorbeeld is een toepassing van deze werkwijze.

4.2 Categorie ii

Noem het aantal in het programma gedeclareerde variabelen N. Door deze variabelen, op welke manier dan ook, oplopend vanaf 0 te nummeren verkrijgen we de gesloten naamgeving $i : 0 \leqslant i < N$. Laat a_i het aantal voor variabele i benodigde geheugencellen zijn; voor alle variabelen te zamen zijn dus M cellen toereikend, waarbij:

$$M = (\Sigma\, i : 0 \leqslant i < N : a_i) \quad .$$

Hiervoor gebruiken we een segment ter lengte M, waarin de variabelen als volgt consecutief worden opgeslagen: variabele i beslaat de posities $j : b_i \leqslant j < b_i + a_i$, met:

$$b_0 = 0$$
$$b_{i+1} = b_i + a_i \quad , 0 \leqslant i < N-1 \quad .$$

Uit deze recursieve definitie van b volgt uiteraard dat:

$$b_i = (\Sigma\, j : 0 \leqslant j < i : a_j) \quad , 0 \leqslant i < N \quad .$$

De eenvoudigste adressering wordt nu verkregen als we variabele i in de programmacode niet identificeren door zijn nummer i maar direct door zijn relatieve positie b_i ; het adres van variabele i is dan immers $D+b_i$, met D het register dat het adres van het segment bevat, zodat weer eenvoudig geïndexeerde adressering kan worden gebruikt.

voorbeeld: Laat x en y integer variabelen en s een variabele van het type **array** [0..99] **of** integer zijn. Stel dat we de volgende nummering gebruiken:

x → 0 , dan $a_0=1 \quad \wedge b_0=0$
s → 1 , dan $a_1=100 \wedge b_1=1$
y → 2 , dan $a_2=1 \quad \wedge b_2=101$.

Het segment heeft lengte $a_0+a_1+a_2$, i.e.: 102 . Element k van het array s bevindt zich dan op positie b_1+k van het segment. De assignment x := s[y] kan nu als volgt worden gecodeerd:

```
b0      equ     0     { variabele x op plaats b0 in het datasegment }
b1      equ     1     { variabele s op plaats b1 in het datasegment }
b2      equ     101   { variabele y op plaats b2 in het datasegment }
        load  A  #b1
      { A = b1 }
        add   A  [D,b2]
      { A = b1 + y }
        load  A  [D,A]
      { A = s[y] }
        store A  [D,b0]
      { x = s[y] }
```

De berekening van het adres van s[y] kan worden verkort wanneer hiervoor een passende instructie ter beschikking staat:

```
        load  A  [D,b2]
      { A = y }
        load  A  [D,b1,A]
      { A = S·(D+b1+y) , dus: A = s[y] }
```

□

4.3 Categorie iii

Net als bij de behandeling van categorie ii introduceren we de gesloten naamgeving $i : 0 \leqslant i < N$ voor de variabelen en a_i voor de hoeveelheid voor de opslag van variabele i benodigde cellen. De variabelen worden weer consecutief in een segment geplaatst, zodat variabele i de posities $j : b_i \leqslant j < b_i+a_i$ beslaat. Het verschil met categorie ii is dat de waarden van a_i en dus ook b_i, voor alle i, pas bekend zijn op het moment dat het programma in uitvoering wordt genomen. We kunnen b_i dus niet meer gebruiken om variabele i in de code te representeren. Er zit nu niets anders op dan de variabelen te identificeren door hun nummers en deze nummers met behulp van een tabel $T(i : 0 \leqslant i < N)$ op de segmentposities af te beelden: $T \cdot i = b_i$. Deze tabel moet worden geïnitialiseerd met behulp van een stukje code, ter berekening van de b_i, dat aan het eigenlijke programma voorafgaat. De tabel kunnen we in hetzelfde segment onderbrengen als de variabelen, bijvoorbeeld in de eerste N cellen van dat segment. Dit geeft:

$$b_0 = N$$
$$b_{i+1} = b_i + a_i \quad , 0 \leqslant i < N-1 \quad .$$

I.e.:

$$b_i = N + (\Sigma j : 0 \leqslant j < i : a_j) \quad , 0 \leqslant i < N \quad .$$

Met D weer voor het adres van het segment geldt dan:

$$T \cdot i = S \cdot (D+i) \quad .$$

Het adres van variabele i is dan $D + T \cdot i$, i.e.: $D + S \cdot (D+i)$. Om het adres van variabele i te berekenen is nu een extra geheugenacces nodig, namelijk om $T \cdot i$ "op te halen". Dit wordt *indirecte adressering* genoemd.

voorbeeld: Wanneer integer variabele x nummer i heeft gekregen dan kan zijn waarde als volgt in register A worden geplaatst:

```
  load  A  [D,i]
{ A = T·i , i.e.: A = b_i }
  load  A  [D,A]
{ A = x }
```

Om dit in één instructie te kunnen coderen hebben we instructies nodig waarin indirecte en geïndexeerde adressering kunnen worden gecombineerd. Een

instructie die A := S•(D + S•(D+i)) bewerkstelligt zou er zo uit kunnen zien:

load A [D,[D,i]]

□

opgave: Werk het voorbeeld uit de vorige paragraaf uit voor deze categorie.

□

4.4 Variaties en mengvormen

In plaats van de relatieve posities van de variabelen in het segment kunnen in tabel T ook direct de adressen van de variabelen worden opgeslagen; dan krijgen we $T•i = D + b_i$. Nu wordt wel informatie over de positie van het segment gedupliceerd. Bij verplaatsing, tijdens (een onderbreking van) het proces, van het segment moet nu niet alleen D maar ook de hele tabel T worden aangepast: de informatie in T is dan vluchtig. (Dit hoeft geen bezwaar te zijn.) De code uit het vorige voorbeeld ziet er dan als volgt uit:

(0) load A [D,i]
→→→ { $A = T•i$, i.e.: $A = D+b_i$ }
load A [A,0]
{ A = x } ,

of als:

(1) load A [[D,i]] .

waarschuwing: Men zou kunnen denken dat de coderingen (0) en (1) volledig equivalent zijn en dat het dus niet nodig is de processor met gecompliceerde instructies zoals in (1) uit te rusten. Wanneer echter de mogelijkheid bestaat dat de programmauitvoering bij →→→ wordt onderbroken -- bij machines met *interrupts* is deze mogelijkheid reëel -- , dan bevat register A op zo'n moment het adres van een locatie in het datasegment. Als nu relocatie van dit segment mogelijk is dan is de informatie in A eveneens vluchtig! Dit illustreert dat de correcte implementatie van dynamic relocation beslist geen sinecure is. De hier geschetste complicatie kan geheel worden vermeden door gebruik van de oplossing uit de vorige paragraaf.

Uiteraard maakt de processor bij het uitvoeren van een instructie zoals load A [[D,i]] ook een (interne) copie van het adres van de variabele, maar de uitvoering van zo'n instructie kan in de regel niet worden onderbroken. Tijdens de uitvoering van deze instructie hoeft de (tijdelijke) copie van dat

adres dan ook niet als vluchtig te worden beschouwd. We geven dit wel weer door te zeggen dat een enkele instructie als een *ondeelbare actie* -- of: *atomic action* -- kan worden beschouwd, maar twee opeenvolgende instructies niet.

□

Het gebruik van de tabel T maakt het ook mogelijk alle variabelen niet bij elkaar in één segment op te slaan maar iedere variabele in een eigen segment. Element T•i is dan het adres van het segment waarin variabele i is opgeslagen. In totaal zijn zo N+1 segmenten nodig. Dit kan aantrekkelijk zijn wanneer het beheren van een grote collectie kleine segmenten efficiënter is dan het beheren van een kleine collectie grote segmenten.

De snelheid van programmauitvoering wordt sterk bepaald door het aantal geheugenaccessen. (De verbinding tussen processor en geheugen wordt niet voor niets de *Von Neumann bottleneck* genoemd!) Uitsparen van geheugenaccessen komt de snelheid ten goede. In dit licht is het gebruik van indirecte adressering, door middel van tabel T , voor *alle* variabelen wat veel van het goede. De tabel is alleen nodig voor de variabelen uit categorie iii . In de praktijk bevatten programma's variabelen uit alle 3 categorieën. Omwille van de efficiëntie is dan ook een mengvorm van de verschillende oplossingen aan te bevelen.

voorbeeld: We combineren de oplossingen van categorieën i en iii . We gebruiken nog steeds de tabel T , maar variabelen die in 1 cel passen slaan we in de tabel op. Dat wil zeggen, als variabele i in 1 cel past dan is zijn adres D+i en is T•i dus de waarde van de variabele; ten behoeve van de berekening van de posities van de andere variabelen nemen we voor deze variabele $a_i = 0$.

Als speciaal geval hiervan kunnen de, zeg, K categorie i variabelen in de eerste K cellen van het segment worden geplaatst; de N-K relatieve posities van de categorie iii variabelen worden daar achter geplaatst. Deze scheiding maakt controle op de categorie mogelijk: de nummers kleiner dan K representeren de categorie i variabelen.

□

opgave: Werk een combinatie uit van de categorieën i , ii en iii .

□

5 Blokstructuur

5.0 Inleiding

In dit hoofdstuk behandelen we de implementatie van programma's waarin *binnenblokken* met *lokale* variabelen voorkomen. De syntactische structuur van zulke programma's kan als volgt worden gedefinieerd:

```
program     → block
block       → '|[' { declaration } statement ']|'
declaration → 'var' names ':' type ';'
statement   → ··· | statement ';' statement | block .
```

Hierin staat de syntactische categorie *names* voor rijen variabelennamen en geeft *type* de klasse van mogelijke datatypen aan.

In de behandeling van de implementatie van zulke programma's maken we onderscheid tussen de (globale) blokstructuur van het programma en, per blok, de (lokale) interne structuur van dat blok in termen van de in dat blok gedeclareerde variabelen. Bij de bespreking van de blokstructuur van het programma is de interne structuur van de blokken nauwelijks van belang. Voor wat betreft zijn interne structuur kan ieder blok als een afzonderlijk programma worden beschouwd; voor de implementatie hiervan kunnen de technieken uit het vorige hoofdstuk worden gebruikt. In dit hoofdstuk ligt de nadruk dan ook op de globale blokstructuur.

Als lopend voorbeeld gebruiken we steeds het volgende programma; hierin staan S0, S1, S2 en S3 voor niet nader gespecificeerde statements:

```
|[ var a,b,c : int;
   S0
 ; |[ var d,e : int; S1 ]|
 ; |[ var p,q,r : int; S2 ]|
 ; S3
]|  .
```

5.1 Een naïeve benadering

We kunnen blokstructuur als *syntactic sugar* beschouwen die eenvoudig kan

worden geëlimineerd door alle declaraties van variabelen naar het buitenste programmablok te verplaatsen. Hierdoor worden alle lokale variabelen globaal gemaakt. Eventueel optredende naamsconflicten --in verschillende blokken gedeclareerde variabelen mogen dezelfde naam hebben-- kunnen door systematische naamsveranderingen worden opgelost. Aldus wordt een programma met uitsluitend globale variabelen verkregen dat op de in hoofdstuk 4 behandelde manier kan worden geïmplementeerd.

Voor programma's uit categorie iii geeft deze werkwijze problemen, omdat de geheugenbehoefte van lokale variabelen kan afhangen van de waarden van globale variabelen. Voor programma's uit categorie ii is deze werkwijze correct maar inefficiënt: de waarden van de variabelen die in een blok zijn gedeclareerd zijn alleen dan relevant wanneer er de statements uit dat blok wordt uitgevoerd. Bij de hier geschetste oplossing wordt onnodig kwistig met geheugenruimte omgesprongen. Voor de variabelen uit ons voorbeeldprogramma zouden zo, bijvoorbeeld, 8 geheugenplaatsen nodig zijn --voor a,b,c,d,e,p,q,r-- terwijl we zullen zien dat 6 toereikend is.

5.2 De structuur van de toestandsruimte

De uitvoering van een programma speelt zich af in een *toestandsruimte* --*state space*-- : op elk moment tijdens de berekening kunnen we spreken over de *toestand* van de berekening. De verzameling van alle denkbare toestanden is de toestandsruimte. Deze wordt opgespannen door de programmavariabelen: de waarden die de variabelen op enig moment hebben vormen de toestand op dat moment.

Tijdens de uitvoering van een programma zonder binnenblokken bestaat de toestandsruimte uit alle programmavariabelen. Wanneer het programma wel binnenblokken bevat dan varieert de toestandsruimte gedurende de berekening. Bij het begin van de uitvoering van een (binnen)blok wordt de toestandsruimte uitgebreid met de lokale variabelen van dat blok. Deze uitbreiding is alleen relevant voor de in dat blok voorkomende statements: na voltooiing van de uitvoering van het blok wordt de uitbreiding weer ongedaan gemaakt. Het blok is dus de *eenheid* van toestandsruimteverandering. Voor elke statement bestaat de toestandsruimte waarin die statement wordt uitgevoerd uit (alle variabelen van) al die blokken die de statement tekstueel omvatten; de toestandsruimte correspondeert derhalve met een collectie *geneste* blokken.

voorbeeld: In het volgende tabelletje staat voor iedere statement uit het voorbeeldprogrammaatje aangegeven uit welke variabelen de toestandsruimte voor die

statement bestaat:

S0	a,b,c
S1	a,b,c,d,e
S2	a,b,c,p,q,r
S3	a,b,c

□

5.3 Naamgeving

Op elk moment tijdens de programmauitvoering hoeven slechts (de variabelen van) die blokken in het geheugen te worden gerepresenteerd die deel uitmaken van de toestandsruimte. We voeren daarom een naamgeving in die voor iedere collectie geneste blokken een gesloten naamgeving oplevert. We nummeren de blokken volgens het principe van *het laagste vrije nummer*; dat wil zeggen, het allerbuitenste blok krijgt nummer 0 en het nummer van ieder binnenblok is 1 plus het nummer van het tekstueel omvattende blok. Voor ons voorbeeldprogramma levert dit:

```
|[0 var a,b,c : int;
   S0
 ; |[1 var d,e : int; S1 ]|
 ; |[1 var p,q,r : int; S2 ]|
 ; S3
]|  .
```

De toestandsruimte van, bijvoorbeeld, S1 omvat nu blokken met nummers 0 en 1, maar dit geldt ook voor S2 : voor alle blokken van het programma te zamen levert deze nummering geen correcte naamgeving op, maar dat is ook niet nodig.

5.4 Inbedding en adressering

Ter administratie van de toestandsruimte voeren we een *hulpvariabele* cbh ("current block height") in die het aantal blokken waaruit de toestandsruimte bestaat representeert: voor iedere statement is cbh het aantal blokken dat deze statement tekstueel omvat. Deze variabele wordt als volgt gemanipuleerd:

initieel : cbh = 0
bij *blockentry* : cbh := cbh + 1
bij *blockexit* : cbh := cbh − 1 .

Hierbij is het programma zelf ook een blok; blockentry en blockexit geven het begin en einde van de uitvoering van een blok aan.

De blokken die de toestandsruimte vormen zijn nu genummerd $k : 0 \leqslant k < cbh$. Door nu per blok de in dat blok gedeclareerde lokale variabelen te voorzien van een *lokale naamgeving* kan iedere variabele uit de toestandsruimte eenduidig worden geïdentificeerd door middel van een paar (k,i) , waarbij k het nummer is van het blok waarin die variabele is gedeclareerd en i de lokale naam van die variabele. Voor het programma uit het voorbeeld levert dit de volgende naamgeving op:

a → (0,0) , b → (0,1) , c → (0,2) ,
d → (1,0) , e → (1,1) ,
p → (1,0) , q → (1,1) , r → (1,2) .

Sommige variabelen, zoals d en p , worden door hetzelfde paar gerepresenteerd; dat hindert niet, want zulke variabelen maken nooit *tegelijkertijd* deel uit van de toestandsruimte (zie § 5.2).

Per blok in de toestandsruimte gebruiken we nu een geheugensegment voor de inbedding van de lokale variabelen van dat blok. (In geval van de in § 4.4 aangegeven variatie met meer segmenten verstaan we hier onder "het segment" het segment dat de tabel T bevat.) Dientengevolge wordt er bij iedere blockentry een nieuw segment in gebruik genomen en komt er bij iedere blockexit een segment vrij. Het *beheer* van deze geheugensegmenten is een taak van het operating system van de machine.

Laat $D{\cdot}k$ het adres zijn van het segment dat hoort bij blok k , $0 \leqslant k < cbh$. Voor blok k is er nu een functie $rp{\cdot}k$ nodig die de lokale namen van variabelen uit dat blok afbeeldt op de relatieve posities van die variabelen in het segment. We krijgen dan:

het adres van variabele (k,i) is $D{\cdot}k + rp{\cdot}k{\cdot}j$.

Voor de in het vorige hoofdstuk behandelde ontwerpen kan rp als volgt worden gekozen:

categorie i : rp•k•i = i
categorie ii : rp•k•i = $b_{k,i}$, met $b_{k,i}$ de voor blok k geldende b_i
categorie iii : rp•k•i = S•(D•k + i) .

5.5 Administratie van de segmentadressen

De segmentadressen D•k , $0 \leqslant k < cbh$, kunnen op allerlei manieren worden geadministreerd. Allereerst observeren we dat de waarde van cbh begrensd is. Er bestaat een --voor ieder programma mogelijk andere-- constante M zodat $cbh \leqslant M$ invariant is. M hangt uitsluitend van de programmatekst af: M is het maximale aantal geneste blokken. In ons voorbeeldprogramma geldt $M = 2$ en alleen bij heel ingewikkelde programma's zal M groter dan, zeg, 16 zijn.

De segmentadressen kunnen in een tabel d(k : $0 \leqslant k < M$) --het *display*-- worden opgeslagen, met, uiteraard, $d•k = D•k$, $0 \leqslant k < cbh$. Deze tabel kan in een segment ter lengte M worden ondergebracht, waarvan het adres in een adresregister wordt gehouden; elke variabele in de toestandsruimte kan nu met een combinatie van geïndexeerde en indirecte adressering worden geselecteerd.

voorbeeld: Met C het register dat het adres van d bevat geldt nu voor een categorie i programma:

het adres van variabele (k,i) is S•(C+k) + i .

Het in register A plaatsen van de waarde van variabele (k,i) kan dan als volgt worden gecodeerd --hiervoor zijn dus 2 geheugenaccessen nodig-- :

```
load  B  [C,k]
load  A  [B,i]
```

□

Dat zelfs het adresseren van een eenvoudige integervariabele 2 geheugenaccessen kost kan bezwaarlijk zijn. Dit kan worden ondervangen door de segmentadressen in adresregisters te houden; hiervoor zijn dan ten hoogste M registers nodig en we hebben gezien dat M in het algemeen geen grote waarden aanneemt. Noemen we deze registers C_k , $0 \leqslant k < M$, met $C_k = D•k$, dan kan een integer variabele (k,i) nu worden "opgehaald" met de instructie load A [C_k,i] .

Bij gebrek aan voldoende veel adresregisters kan men de tabel d in het geheugen opslaan, maar de meest frequent geraadpleegde elementen in registers

houden. Hiervoor komen vooral D•0 en D•(cbh-1) in aanmerking, als men aanneemt dat in programma's accessen aan globale variabelen en aan de meest lokale variabelen het vaakste voorkomen.

waarschuwing: Een dergelijke aanname is niet zonder gevaar! De implementatie moet neutraal zijn ten aanzien van de gewenste stijl van programmeren, of deze ondersteunen, maar niet ongunstig beïnvloeden. Wanneer de programmeur weet dat in zijn machine sommige variabelen efficiënter worden geadresseerd dan kan hij zich door deze kennis laten beïnvloeden --namelijk door er gebruik van te maken-- . Aldus wordt de aanname een *self fulfilling prophesy* zonder dat zij daadwerkelijk een programmeerstijl weerspiegelt. Bezien in dit licht verdienen zo *homogeen* mogelijke implementaties de voorkeur. De hierboven gegeven suggestie alleen D•0 en D•(cbh-1) in registers te houden is heel wel verdedigbaar: het kost niet veel, de efficiëntiewinst is mooi meegenomen en de erdoor veroorzaakte inhomogeniteit is niet zeer groot --daarvoor is het efficiëntieverschil weer niet groot genoeg-- : een competente programmeur laat zich door deze kennis niet misleiden.

□

opgave: Wanneer het adres van d in register C0 wordt gehouden en D•(cbh-1) in register C1 , dan betekent dit dat C1 = D•(cbh-1) invariant is. Ga na wat er bij blockentry en -exit moet gebeuren om dit invariant te houden en codeer deze operaties in instructies.

□

Een andere manier om de segmentadressen te administreren is de volgende. In register C houden we D•(cbh-1) bij --in de veronderstelling dat dit het meest geraadpleegde segmentadres is-- . In ieder segment reserveren we de cel op positie 0 voor het adres van het segment dat correspondeert met het omvattende blok; dat wil zeggen:

$$C = D\bullet(cbh-1)$$
$$S\bullet(D\bullet i) = D\bullet(i-1) \quad , \quad 0 < i < cbh \quad .$$

De segmenten vormen op deze manier een *lijst*; de waarden in de cellen op positie 0 worden wel *static links* genoemd. Het adres van variabele (k,i) kan nu als volgt worden berekend:

```
  B,h := C , cbh-1  { invariant: k ⩽ h < cbh ∧ B = D•h }
; do k ≠ h → { 0 ⩽ k < h < cbh ∧ B = D•h , dus: 0 < h ∧ S•B = D•(h-1) }
               B,h := S•B , h-1
; od
{ B = D•k , dus: het adres van (k,i) is  B + rp•k•i  }
```

Merk op dat de waarde van cbh op elk punt in de programmatekst bekend is. Het is dan ook niet nodig cbh in een variabele bij te houden. Verder is het gebruik van een repetitie niet echt nodig: het aantal slagen dat de repetitie vergt is cbh − 1 − k ; de gerepeteerde statement kan dus cbh − 1 − k keer worden opgeschreven.

voorbeeld: Het ophalen van de integer variabele (3,i) , wanneer cbh = 7 kan aldus als volgt worden gecodeerd:

```
load  B  C       { B := C }
load  B  [B,0]   { B := S•B }
load  B  [B,0]   { B := S•B }
load  B  [B,0]   { B := S•B }
load  A  [B,i]
```

De eerste twee instructies kunnen worden gecombineerd tot load B [C,0] .

□

Bij deze werkwijze zijn alleen al voor de berekening van het segmentadres van een variabele in blok k dus cbh − 1 − k geheugenaccessen nodig, tegen 1 wanneer de segmentadressen in een tabel in het geheugen staan. Het gebruik van het display -- in registers of in het geheugen, eventueel aangevuld met een register voor D•(cbh-1) -- verdient dan ook de voorkeur boven het gebruik van static links.

5.6 Een speciaal geval

Bij blockexit krimpt de toestandsruimte doordat de het *laatst toegevoegde* variabelen uit de toestandsruimte worden verwijderd: de toestandsruimte groeit en slinkt volgens het principe Last-In-First-Out (LIFO). Een datastructuur die zo groeit en slinkt wordt een *stapel* -- of: *stack* -- genoemd. Omdat de adressen van de hierbij gebruikte segmenten worden geadministreerd, kunnen deze segmenten in principe overal -- i.e.: onderling niet gerelateerd -- in het geheugen staan.

Vanwege het LIFO-karakter kunnen de segmenten echter ook aaneengesloten worden geplaatst zonder dat dit aanleiding geeft tot bewerkelijke reorganisaties van de segmentenstructuur.

Voor programma's uit categorie ii is bij consecutieve stapeling van de segmenten nog een interessante vereenvoudiging mogelijk. Bij zulke programma's is immers de grootte van ieder segment a priori bekend; daarmee kunnen ook, voor iedere statement in het programma, de segmentadressen van de bij die statement behorende toestandsruimte van te voren te worden berekend. Aldus kan de gehele toestandsruimte in één groot segment worden ondergebracht; de lengte van dit segment moet dan tenminste de maximale afmeting van de toestandsruimte zijn. Iedere variabele uit het programma kan nu weer door zijn relatieve positie in dit segment worden geïdentificeerd: deze relatieve posities zijn constant en van te voren berekenbaar.

voorbeeld: Ons voorbeeldprogramma zit in categorie i en dus ook in ii. Passen we bovenstaande vereenvoudiging hierop toe dan is een segment van 6 cellen toereikend en kunnen de programmavariabelen als volgt in dat segment worden ondergebracht:

$$a \to 0\ ,\ b \to 1\ ,\ c \to 2\ ,\ d \to 3\ ,\ e \to 4\ ,\ p \to 3\ ,\ q \to 4\ ,\ r \to 5$$

□

5.7 Nabeschouwing

De introductie van de variabele cbh , van de naamgeving der blokken en van de rij D der segmentadressen maakt het mogelijk een hele klasse implementaties systematisch te bestuderen. Dat deze grootheden in sommige varianten van de implementatie niet meer voorkomen is irrelevant. Daar fungeren zij als hulpgrootheden ten behoeve van de documentatie van het ontwerp en zijn correctheid (zie bijvoorbeeld de adresberekening in het geval van de static links).

6 Het evalueren van expressies

6.0 De waarde van een expressie

We definiëren een eenvoudige klasse integerexpressies, als volgt:

exp → con | var | '(' exp op exp ')'
con → ⟨n⟩ , voor integer n
var → name .

Hierin staat ⟨n⟩ voor de decimale representatie van het getal n . Nota bene: er is verschil tussen de *cijferrij* "423" en het *getal* 423 ; in de regel kunnen we dit onderscheid verwaarlozen, maar wanneer we het over de implementatie van expressies hebben juist *niet*! Aldus staat ⟨n⟩ voor de cijferrij die het getal n voorstelt.

Verder staat var voor de variabelennamen en staat op voor de binaire operatoren, zoals + , − , ∗ , et cetera. Om het onderscheid tussen de *symbolen* en de ermee bedoelde rekenkundige *functies* zichtbaar te maken geven we het symbool hier aan met ⊕ en de functie met + . Hierbij gebruiken we ⊕ resp. + als representanten van alle binaire operatoren.

Iedere expressie heeft een waarde; de waarde van expressie E geven we aan met ⟦ E ⟧ . De waarde van een expressie hangt hierbij af van de waarden van de in die expressie voorkomende variabelen. Deze waarden zijn in het geheugen opgeslagen en met addr·x geven we het adres van variabele x aan; dat addr·x pas tijdens programmauitvoering kan worden berekend doet er hier niet toe. Zo krijgen we:

⟦ ⟨n⟩ ⟧ = n
⟦ x ⟧ = S·(addr·x)
⟦ E ⊕ F ⟧ = ⟦ E ⟧ + ⟦ F ⟧ .

6.1 Het evalueren van een expressie

Expressies komen voor in assignment statements en in guards. De assignment x := E kan als volgt worden gecodeerd:

```
  code·E
{ A=[[ E ]] }
  store A  addr·x
```

Hierbij dient code·E een stukje code te zijn dat A=[[E]] bewerkstelligt. Hetzelfde stukje code kan worden gebruikt voor expressies die als guards worden gebruikt; de guarded commands kunnen dan met behulp van conditionele spronginstructies worden geïmplementeerd.

Uit deze specificatie en de definitie van [[E]] volgt onmiddellijk hoe code·E moet worden gekozen voor de integer constanten en de variabelen:

```
code·⟨n⟩ :    load  A  #n

code·x :      load  A  addr·x
```

Voor samengestelde expressies hebben we [[E ⊕ F]] = [[E]] + [[F]] ; derhalve kunnen we code·(E ⊕ F) uitdrukken in code·E en code·F . Omdat zowel code·E als code·F een waarde aan register A toekennen moet de als eerste uitgerekende waarde in een hulpvariabele h worden opgeslagen totdat ook de code voor de tweede deelexpressie is uitgevoerd. Dit kan formeel worden weergegeven door middel van een binnenblok waarin h als lokale variabele wordt gedeclareerd. De volgorde waarin E en F worden uitgerekend is irrelevant; de hieronder gekozen volgorde past goed bij het gebruik van de statement A := A ⊕ h , die kan worden gecodeerd als add A addr·h . (Nota bene: optelling is symmetrisch, maar dit geldt niet voor alle binaire operatoren.) In gewone programmanotatie kan het evalueren van E ⊕ F in termen van het evalueren van E en F als volgt worden genoteerd, waarbij we register A als een gewone (globale) variabele opvatten:

```
|[ var h : int;
    A := F   { A=F }
  ; h := A   { h=F }
  ; A := E   { A=E ∧ h=F , dus  E⊕F = A⊕h }
  ; A := A⊕h
  { A  = E⊕F }
]|   .
```

Vertaling van dit programma in machinecode levert dan voor code·(E ⊕ F) :

```
code•(E ⊕ F) :    code•F
                  store A   addr•h
                  code•E
                  add   A   addr•h
```

Hierin is de implementatie van de blokstructuur -- i.e.: allocatie van een geheugenplaats voor variabele h -- achterwege gelaten. Merk echter op dat in code•E en code•F eveneens zulke hulpvariabelen voor kunnen komen. Zoals een collectie geneste binnenblokken groeit en slinkt als een stapel -- zie § 5.6 -- , kunnen alle voor de evaluatie van een expressie benodigde hulpvariabelen worden gestapeld.

6.2 Stapelmachines

Meestal wordt voor het evalueren van expressies niet gebruik gemaakt van het binnenblokmechanisme maar van een speciaal hiervoor gecreëerde *rekenstapel*. Het rekenregister A vormt hierbij het bovenste element van deze stapel; code•E bewerkstelligt dan het op de stapel plaatsen van ⟦ E ⟧ , terwijl bij een rekenkundige bewerking de laatst toegevoegde twee stapelelementen worden vervangen door het resultaat van de bewerking.

De stapel kan worden gerepresenteerd in het geheugen en worden bespeeld met behulp van een indexregister. Dit werkt vooral efficiënt als de benodigde *stapeloperaties* ook in het instructierepertoire voorkomen. De verwerkingssnelheid kan nog worden vergroot door (een gedeelte van) de stapel in registers te houden, waardoor geheugenaccessen kunnen worden uitgespaard. Dit vereist voorzieningen in de hardware van de processor, of in de compiler -- register allocation -- . Omdat de meeste in programma's voorkomende expressies eenvoudig zijn levert het gebruik van zelfs een klein aantal registers een aanzienlijke besparing op geheugenaccessen op.

7 Procedures

7.0 Procedureaanroep en -terugkeer

In dit hoofdstuk bestuderen we de implementatie van procedures in hun eenvoudigste vorm, dat wil zeggen zonder lokale variabelen en parameters. In de volgende hoofdstukken komen procedures met lokale variabelen en parameters aan de orde.

In zijn eenvoudigste vorm is een procedure een statement --dat kan dus ook een rij statements zijn-- met een naam. Deze *procedurenaam* wordt aan de statement --de *procedurebody*-- gegeven in een *proceduredefinitie*. De meeste programmeertalen hebben een zodanige syntax dat proceduredefinities in programma's kunnen voorkomen op dezelfde plaatsen als variabelendeclaraties. In het blok dat de proceduredefinitie bevat mag de procedurenaam als statement voorkomen: dit wordt een *procedurestatement* genoemd. De betekenis van een procedurestatement is de betekenis van de overeenkomstige procedurebody: de in een procedurestatement gebruikte naam kan worden beschouwd als een afkorting van de erbij behorende procedurebody.

Verreweg de eenvoudigste implementatie wordt verkregen door *substitutie*: iedere procedurestatement wordt hierbij vervangen door een copie van de tekst van de overeenkomstige procedurebody. Deze werkwijze is eenvoudig en correct maar inefficiënt in geheugengebruik. Vooral bij grotere programma's vormen procedures een belangrijk abstractie- en structureringsmiddel; het veelvuldig gebruik van procedurestatements leidt dan tot het veelvuldig dupliceren van stukken code. Het is daarom wenselijk om iedere procedurebody slechts één keer in machinecode in het geheugen op te slaan.

voorbeeld: We bekijken het volgende programma:

```
|[ proc P = |[ S0 ; S1 ]| ;
 ; S2
 ; P
 ; S3
 ; P
 ; S4
]| .
```

Het uitvoeren van dit programma vereist het uitvoeren van de volgende rij statements: S2 ; S0 ; S1 ; S3 ; S0 ; S1 ; S4 . Dit illustreert dat steeds na uitvoering van de procedurebody die statement wordt uitgevoerd die onmiddellijk volgt op de procedurestatement die aanleiding gaf tot het uitvoeren van die body; in dit geval is dat S3 respectievelijk S4 .

□

De procedurebody wordt afzonderlijk in machinecode vertaald en in het geheugen geplaatst. Laat p het adres zijn van de als eerste uit te voeren instructie van de body van procedure P . Hoe moet nu iedere procedurestatement P worden gecodeerd? Het netto effect hiervan moet zijn dat (de code van) de body van P een keer wordt uitgevoerd, waarna de programmauitvoering verder gaat met de code die volgt op de procedurestatement. Deze laatste wens maakt dat de procedurestatement *niet* kan worden gecodeerd als een spronginstructie jump p : deze bewerkstelligt wel dat de body van de procedure wordt uitgevoerd, maar daarna? De instructie jump p komt immers neer op PI := p , wat tot gevolg heeft dat de actuele waarde van PI verloren gaat. Dit is echter precies het adres van de instructie die volgt op de spronginstructie --zie § 1.1-- . Dit adres vormt de zogenaamde *terugkeerinformatie* die tijdens de uitvoering van de procedurebody moet worden behouden; na uitvoering van de body gaat de processor verder met de door de terugkeerinformatie geïdentificeerde code.

Het met behoud van terugkeerinformatie beginnen aan de uitvoering van een procedurebody heet een *procedureaanroep* (of *-call*) terwijl het met gebruikmaking van de terugkeerinformatie verder gaan met de code die volgt op de procedurestatement de *procedureterugkeer* (of *-return*) heet. Voor de realisatie hiervan dient het instructierepertoire van de processor voorzien te zijn van instructies die call en return bewerkstelligen.

7.1 Procedures en invocaties

Zoals we in § 2.1 onderscheid hebben gemaakt tussen programma's en processen dienen we ook onderscheid te maken tussen een procedure en de uitvoeringen daarvan. Deze laatste noemen we *invocaties*. Op elk moment tijdens een proces kunnen er van iedere procedure in ieder geval 0 of 1 invocaties zijn, maar ook meer dan 1 , namelijk in het geval van *recursieve* procedures. Iedere invocatie is dus een procedure-in-uitvoering en met iedere procedure kunnen, op een en hetzelfde tijdstip, verschillende invocaties corresponderen. Bij iedere procedureaanroep ontstaat er een invocatie; bij iedere terugkeer wordt er een

invocatie voltooid en wel van alle op dat moment bestaande invocaties degene die als laatste is ontstaan.

De in een proces optredende calls en returns vormen een rij; de mogelijke rijen kunnen als volgt worden gedefinieerd met behulp van EBNF-notatie:

seq → { **'call'** seq **'return'** } .

Deze regel illustreert dat het patroon van calls en returns *goedhaaks* is -- vergelijk dit met de haakjesstructuur van expressies of van binnenblokken -- ; ieder haakjespaar komt hierbij overeen met een invocatie. Omdat in een procedurebody immers ook procedurestatements voor kunnen komen, kunnen invocaties *genest* zijn.

7.2 De invocatienaamgeving en de rij programmaindices

De gehele berekening bestaat nu uit een geneste collectie invocaties. Om deze te identificeren voeren we een *invocatienaamgeving* in. Deze komt neer op nummering van de invocaties volgens het principe van het laagste vrije nummer. In termen van de call- en returnstructuur kan het nummer van iedere invocatie worden gedefinieerd als het aantal call-return paren dat die invocatie omvat. De invocatienaamgeving kan nu worden gekarakteriseerd als $i : 0 \leqslant i \leqslant cin$ waarin de systeemvariabele cin -- current invocation number -- het nummer van de meest recente invocatie voorstelt. Initieel geldt $cin = 0$ en bij iedere call wordt cin dan met 1 verhoogd terwijl cin bij iedere return met 1 wordt verlaagd.

Iedere invocatie kan worden beschouwd als een afzonderlijke berekening: het is de uitvoering van een procedurebody, waarbij de uitvoering van de code die de procedureaanroep bevat tijdelijk is opgeschort. Voor iedere invocatie is nu een programmaindex nodig die, voor die invocatie, de volgende uit te voeren instructie identificeert. In plaats van 1 programmaindexregister PI hebben we nu dus een hele rij $pi(i:0 \leqslant i \leqslant cin)$ van programmaindices nodig. Procedureaanroep en -terugkeer kunnen nu worden geïmplementeerd door de processor met de volgende twee instructies uit te rusten:

instructie:	**effect:**
call p	cin := cin + 1 ; pi•cin := p
return	cin := cin - 1

Deze instructies worden als volgt gebruikt. Iedere procedurestatement P wordt gecodeerd als call p , waarbij p het adres van de als eerste uit te voeren instructie van de body van P is. Verder wordt de code van iedere procedurebody afgesloten door return .

pi•cin is nu de programmaindex van de meest recente invocatie. Dit is de *actieve* invocatie, in die zin dat de voortzetting van alle andere invocaties is opgeschort. In de instructieuitvoeringscyclus, zoals beschreven in § 1.1 , moet daarom PI worden vervangen door pi•cin :

```
do true → |[ var IR: cell;
             IR , pi•cin := S•(pi•cin) , pi•cin + 1
           ; "execute the instruction in IR"
          ]|
od .
```

7.3 Een kleine maar essentiële optimalisering

In het bijzonder bij programma's met recursieve procedures is de lengte van de rij pi in principe onbegrensd. pi moet derhalve in het geheugen worden gerepresenteerd. Wel zien we dat het element pi•cin zeer frequent wordt gebruikt, namelijk bij het uitvoeren van *iedere* instructie. Representatie van pi•cin in het geheugen vereist dan ook 2 extra geheugenaccessen per uitgevoerde instructie. Dit is uiteraard onacceptabel inefficiënt.

Deze inefficiëntie kan worden vermeden door pi•cin in een register te houden en alleen de rest van pi , i.e.: pi(i:0≤i<cin) , in het geheugen onder te brengen. Dit register wordt dan weer *de* programmaindex PI genoemd, terwijl de rij pi(i:0≤i<cin) de *terugkeerinformatie* wordt genoemd. Hiertoe worden de instructies call en return als volgt gedefinieerd:

instructie:	**effect:**
call p	pi•cin := PI ; cin := cin + 1 ; PI := p
return	cin := cin − 1 ; PI := pi•cin

opgave: Toon aan dat dit een correcte implementatie van de in § 7.2 ingevoerde instructies voor call en return is.

□

De rij pi wordt nu gebruikt als een stapel. pi heet dan ook wel de *terugkeeradresstapel*; de elementen van pi worden wel terugkeeradressen genoemd.

7.3 Niet-vluchtige representatie van de terugkeerinformatie

De elementen van pi zijn adressen van instructies in de programmacode. In een omgeving waarin dynamische relocatie van code mogelijk is, is deze informatie vluchtig. Om te vermijden dat bij relocatie van (een deel van) de code alle terugkeeradressen moeten worden gecontroleerd en zonodig worden gewijzigd, verdient het in zulke gevallen de voorkeur de terugkeerinformatie op een allocatieonafhankelijke manier op te slaan. Verder dienen dan de in de call-instructies voorkomende adressen door een allocatieonafhankelijke identificatie van de aangeroepen procedure te worden vervangen.

opgave: Werk dit in detail uit voor het geval waarin de code van iedere procedurebody in een apart -- i.e.: onafhankelijk te alloceren -- segment wordt ondergebracht. Welke voorzieningen in de processor zijn hiervoor nodig?

□

8 Procedures en variabelen

8.0 De structuur van de toestandsruimte

Iedere invocatie kan worden beschouwd als een afzonderlijk proces. De toestandsruimte waarin dit proces zich afspeelt bestaat uit de toestandsruimte waarin de procedureaanroep wordt uitgevoerd, uitgebreid met de lokale variabelen van de aangeroepen procedure. Aldus correspondeert er met iedere invocatie een stukje van de toestandsruimte; zo'n stukje bestaat dan uit de lokale variabelen van de met die invocatie overeenkomende procedure. Omdat er van een procedure tegelijkertijd meer dan één invocaties kunnen zijn kan de toestandsruimte dus verschillende *instances* van de lokale variabelen van een procedure bevatten.

Deze structuur is in wezen dezelfde als de structuur van de toestandsruimte bij programma's met uitsluitend binnenblokken, zoals behandeld in §5.2. Daar bestaat de toestandsruimte uit (de variabelen van) alle blokken die de statement in uitvoering tekstueel omvatten, maar dat zijn precies de in uitvoering zijnde blokken. Bij de behandeling van de blokstructuur hadden we ook onderscheid kunnen maken tussen blokken en hun invocaties, waarbij eveneens met iedere invocatie een stukje van de toestandsruimte overeenkomt. Dan geldt: van ieder blok dat de in uitvoering zijnde statement omvat is er sprake van 1 invocatie, van alle overige in het programma voorkomende blokken is het aantal invocaties 0.

In het vervolg verstaan we onder een *blok* zowel een binnenblok, zoals behandeld in hoofdstuk 5, als een procedurebody. Desgewenst kunnen binnenblokken worden opgevat als de bodies van naamloze procedures die op de (enige) plaats van aanroep zijn gesubstitueerd. Aldus zijn procedures algemener dan binnenblokken.

Het bij een invocatie behorende stukje toestandsruimte geven we aan met de --historisch bepaalde-- naam *activation record*. Een activation record bestaat dus uit (een instance van) de lokale variabelen van een blok --binnenblok of procedurebody--. De in het vorige hoofdstuk al ingevoerde invocatienaamgeving levert een gesloten naamgeving voor de activation records die de toestandsruimte vormen. Wanneer nu ieder activation record in een geheugensegment wordt ondergebracht kan de gehele toestandsruimte worden geadministreerd met behulp van een tabel Inv(i:0≤i≤cin) --*invocation table*-- waarin de adressen van deze segmenten worden opgeslagen:

Inv·i = "het adres van het segment dat het activation record behorend bij invocatie i bevat" .

Iedere instance van een variabele in de toestandsruimte kan nu worden geïdentificeerd door middel van een paar (i,j) , waarin i een invocatienummer is en waarin j de variabele identificeert onder de lokale variabelen van het blok waarin die variabele is gedeclareerd. Deze naamgeving noemen we de *invocatie-naamgeving* (execution nomenclature) der variabelen, ter onderscheid met de nog in te voeren *tekstuele naamgeving* der variabelen. In het (eenvoudige) geval van categorie i variabelen is het adres van variabele (i,j) dan Inv•i+j .

8.1 De context

Nu het kan voorkomen dat er van één en dezelfde procedure meer dan 1 invocaties zijn, en er dus met één en dezelfde naam meer dan 1 instances van een variabele kunnen corresponderen, rijst de vraag op *welke* instance van die variabele operaties op de variabele betrekking hebben. We hebben hierin geen keuzevrijheid: de semantiek van de programmeertaal is zodanig dat we iedere in het programma voorkomende naam van een variabele moeten opvatten als een verwijzing naar de *meest recente* instance van de betreffende variabele. De meest recente instance van een variabele bevindt zich in het activation record van de meest recente invocatie van het blok waarin die variabele is gedeclareerd. De meest recente invocatie van een blok is, van alle invocaties van dat blok, degene met het *hoogste* invocatienummer.

Behalve lokale variabelen kunnen in een procedurebody ook *globale* -- i.e.: buiten deze body gedeclareerde -- variabelen voorkomen. Volgens de regels van de programmeertaal zijn deze variabelen gedeclareerd in een blok dat de procedure-body -- en dus de proceduredefinitie -- tekstueel omvat. Omdat alle aanroepen van zo'n procedure ook door deze blokken worden omvat, levert deze regel een zinvolle interpretatie op voor alle in een procedurebody voorkomende globale variabelen, voor elke aanroep van de procedure: nog steeds kan een procedure-statement worden opgevat als een afkorting van een procedurebody. Ook hier geldt dat, tijdens programmauitvoering, de variabelen in de tekst verwijzen naar de meest recente instances van die variabelen.

Omdat op elk moment tijdens de programmauitvoering iedere variabele in de tekst op ten hoogste één instance van die variabele betrekking kan hebben, is op elk moment tijdens de berekening slechts een gedeelte van de toestandsruimte adresseerbaar. Dit gedeelte noemen we de *context*. Tijdens uitvoering van een statement bestaat er van iedere die statement omvattend blok tenminste één invocatie. De context van een statement bestaat nu uit de activation records van de meest recente invocaties van alle die statement tekstueel omvattende blokken. Ga na

dat we in hoofdstuk 5 zo de toestandsruimte hebben gedefinieerd: in het geval van programma's met uitsluitend binnenblokken is de context de gehele toestandsruimte.

voorbeeld: We beschouwen het volgende programma:

```
|[ var x;
   proc A = |[ var y;
               proc B = |[ x := y ]|;
               proc C = |[ var z; ··· B ··· ]|;
               B
              ; C
             ]|;
   A
]|  .
```

De blokken die in dit programma de body van procedure B omvatten zijn de body van B zelf, de body van procedure A en het blok dat het programma vormt. Alleen de in deze blokken gedeclareerde variabelen, x en y, mogen in de body van B voorkomen. De in de body van C gedeclareerde variabele z kan niet in de body van B voorkomen, ook al kan de body van C wel aanroepen van B bevatten.

Tijdens de vanuit de body van C geïnitieerde invocatie van B bestaat de state space uit x, y en z, terwijl de context alleen x en y bevat.

□

8.2 Tekstuele naamgeving en adressering

De uitvoering van een statement heeft betrekking op het gedeelte van de toestandsruimte dat de context heet. Het is dan ook voldoende in de code van het programma een naamgeving te hebben voor (de instances van) de variabelen in de context. Deze naamgeving noemen we, als in hoofdstuk 5, de *tekstuele naamgeving*. Hiertoe nummeren we de blokken --nu: binnenblokken en procedurebodies-- weer als volgt: het nummer van een blok is het aantal blokken die het blok tekstueel omvatten. De tekstuele naam van een variabele is dan het paar (i,j), met i het (tekstuele) nummer van het blok dat de declaratie van de variabele bevat en j het (lokale) nummer van die variabele binnen dat blok. Deze naamgeving is dezelfde als de in hoofdstuk 5 geïntroduceerde, waarbij procedurebodies ook als blokken worden beschouwd.

De adressering van de variabelen in de context kan nu als volgt worden gerealiseerd. Met behulp van een tabel C(i:0⩽i⩽cbh) --de *contextdescriptor*-- wordt de tekstuele naamgeving van de blokken afgebeeld op de invocatienaamgeving:

C•i = "het maximale nummer van enige invocatie van blok i" .

Voor categorie i variabelen geldt dan:

het adres van variabele (i,j) is: Inv•(C•i) + j .

Het blijkt dat C alleen wordt gebruikt in deze adresberekening; we kunnen even goed een tabel D(i:0⩽i<cbh) --wederom het *display* genaamd-- invoeren, met:
D•i = Inv•(C•i) .

Tabel D beeldt dus de tekstuele bloknummers af op de adressen van de segmenten die de activation records uit de context bevatten. Zie verder §5.5 voor de mogelijke inbeddingen van D . Ga na dat de in §5.6 behandelde vereenvoudiging nu niet kan worden toegepast.

Door de introductie van het display D is het adres van variabele (i,j) nu D•i + j ; aldus is de adressering net zo eenvoudig als bij programma's met louter binnenblokken en zonder procedures. Wanneer D in registers is gerepresenteerd kan iedere variabele uit de context met slechts 1 geheugenacces worden benaderd. In de volgende paragraaf zullen we echter zien dat de informatie in D wordt gedupliceerd. De elementen van D zijn adressen; wanneer deze als vluchtig moeten worden beschouwd dan verdient het gebruik van de --constante-- invocatienaamgeving en de tabellen C en Inv de voorkeur.

8.3 Contextveranderingen

Zowel de toestandsruimte als de context veranderen bij iedere call en return. Hier behandelen we de administratie van deze veranderingen voor het geval dat de adressering via C en Inv plaatsvindt. Zoals al in §5.5 is aangegeven bestaat er een, louter van de programmatekst afhankelijke, constante M zodat cbh ⩽ M invariant is. In plaats van voor i : 0 ⩽ i < cbh definiëren we C•i nu voor alle i met 0 ⩽ i < M , als volgt:

C•i = "het maximale nummer van enige invocatie van enig blok met tekstueel nummer i" .

Ga na dat dit voor $i < cbh$ juist is. Voor $cbh \leqslant i$ hoeven er geen invocaties van enig blok met tekstueel nummer i te bestaan; in dat geval is de waarde van $C{\cdot}i$ irrelevant.

Bij procedureaanroep -- en blockentry -- ontstaat een nieuwe invocatie met nummer $cin + 1$; dit is het grootste is van alle in gebruik zijnde invocatienummers. Wanneer h het tekstuele nummer is van het aldus in uitvoering genomen blok dan dient $C{\cdot}h$ dus de waarde $cin + 1$ te krijgen. De waarde van $C{\cdot}h$ vlak voor de call zal tevens de waarde van $C{\cdot}h$ na beëindiging van deze invocatie dienen te zijn; deze waarde moet dan ook worden "bewaard". Bij procedureterugkeer -- en blockexit -- eindigt de als laatste gecreëerde invocatie. Zowel de toestandsruimte als de context onmiddellijk na return zijn hetzelfde als vlak voor de bijbehorende call. Bij return dient dus de situatie van vlak voor de call te worden hersteld. In het bijzonder moet, met h het tekstuele nummer van het blok waarvan de invocatie eindigt, $C{\cdot}h$ weer gelijk worden gemaakt aan de "bewaarde" waarde van $C{\cdot}h$.

De bij aanroep en terugkeer optredende contextveranderingen kunnen als volgt worden gecodeerd; hierbij staat h voor het tekstuele nummer van de procedurebody waarop de invocatie betrekking heeft:

```
call   :   "save C·h"
         ; cin := cin + 1
         ; Inv·cin := "het adres van een segment t.b.v. het
                        activation record voor deze call"
         ; C·h := cin

return :   "geef het segment met adres Inv·cin vrij"
         ; cin := cin - 1
         ; "restore C·h"
```

Hierin vereist de toekenning van een adres aan $Inv{\cdot}cin$ het reserveren van een geheugensegment van de "juiste" grootte -- allocatie -- ; bij de overeenkomstige return wordt dit segment weer vrijgegeven -- deallocatie -- . Zoals voor het opslaan van terugkeeradressen kan ook voor het opslaan van de te bewaren $C{\cdot}h$ waarden gebruik worden gemaakt van een *stapel*.

opgave: Werk een combinatie uit van het hier gegeven ontwerp, met daarin C en Inv vervangen door het display D , en de in hoofdstuk 7 behandelde administratie van de terugkeerinformatie; werk ook de voor het bewaren van $C{\cdot}h$ benodigde stapeloperaties in detail uit.

□

9 Procedures en parameters

9.0 Soorten parameters

Wanneer we het uitvoeren van een procedurebody als een apart proces opvatten dan kunnen we parameters beschouwen als een middel om gegevens over te dragen tussen het aanroepende en het aangeroepen proces. Dit kan worden vergeleken met input en output. Hiertoe onderscheiden wij 3 soorten parameters, die we *value-*, *result-* en *value-result*parameters noemen. Voor de definitie hiervan gebruiken we de volgende (prototype) proceduredefinitie:

```
proc P(?x; !y; z) = |[ S ]|   .
```

Hierin geeft het vraagteken aan dat x een valueparameter is, geeft het uitroepteken aan dat y een resultparameter is en geeft het ontbreken van een symbool aan dat z een value-resultparameter is. Parameters hebben, zoals variabelen, *types*, maar deze types zijn voor de discussie hier irrelevant; daarom zijn de typeaanduidingen van de parameters hier weggelaten.

Aanroepen van P zijn van de vorm P(E,b,c) , waarin E een expressie is en waarin b en c *verschillende* variabelen zijn. De expressie E en de variabelen b en c heten de *argumenten* --ook wel de *actuele parameters*-- van de aanroep. Zo'n aanroep is equivalent met het volgende binnenblok:

```
|[ var x,y,z;
   x,z := E,c
 ; S
 ; b,c := y,z
]|   .
```

In de statement S van de procedurebody kunnen x , y en z als gewone lokale variabelen voorkomen. Valueparameter x is hierbij een lokale variabele die wordt geïnitialiseerd met de waarde van de in de aanroep gespecificeerde expressie E ; dit moet een geldige expressie zijn in de context van de aanroep. Resultparameter y daarentegen moet in de statement S worden geïnitialiseerd. Na voltooiing van S wordt de waarde van y toegekend aan het argument b . De initiële waarde van b is irrelevant: resultparameters kunnen worden gebruikt om variabelen te initialiseren. Value-resultparameters, tenslotte, zijn in feite een combinatie van

value- en resultparameters. Zij kunnen worden gebruikt voor het modificeren van variabelen.

9.1 Valueparameters

De implementatie van valueparameters is eenvoudig. De parameter wordt behandeld als een lokale variabele van de procedurebody. Voordat de procedurebody wordt uitgevoerd, wordt de als argument opgegeven expressie geëvalueerd en wordt de zo verkregen waarde aan de parameter toegekend.

De enige complicatie hierbij is dat de expressie moet worden geëvalueerd in de context van het proces dat de aanroep uitvoert, terwijl de lokale variabelen van de procedurebody pas "bestaan" *na* de contextverandering die met de aanroep gepaard gaat. Daarom wordt meestal de volgende werkwijze toegepast. De code voor de evaluatie van de expressie gaat vooraf aan de call-instructie. Als gevolg van het uitvoeren van deze code ligt de waarde van de expressie op de rekenstapel. Na de contextverandering en na allocatie van ruimte voor de lokale variabelen wordt deze waarde van de rekenstapel verwijderd en aan de parameter toegekend.

9.2 Resultparameters

De implementatie van resultparameters is zo mogelijk nog eenvoudiger dan die van valueparameters. Het verschil bestaat immers slechts in de *richting* van de waardeoverdracht. Bovendien is er bij resultparameters niet sprake van een expressie maar van een enkele variabele. Ook hier is de bij return optredende contextverandering de enige complicatie en ook hier biedt de rekenstapel de benodigde bufferruimte: vlak voor return wordt de waarde van de resultparameter op de stapel gelegd; de code voor het van de rekenstapel halen en aan de argumentvariabele toekennen van die waarde wordt dan in het aanroepende programma opgenomen, direct na de code voor de aanroep zelf.

Hoewel deze oplossing eenvoudig is, is zij tamelijk inefficiënt wanneer het gaat om variabelen waarvoor veel ruimte nodig is, zoals arrays: de tijd die nodig is voor het copiëren van een variabele is evenredig met de hoeveelheid ruimte die die variabele beslaat. Dit is des te schrijnender als we observeren dat, na uitvoering van de assignment b := y , de voor y gereserveerde ruimte wordt vrijgegeven en dat de initiële waarde van b verloren gaat. Waarom gebruiken we niet de al voor b gereserveerde ruimte voor y en laten we aldus y *dezelfde* variabele zijn als b , zij het onder een andere naam? We noemen y en b in zo'n geval

aliases van elkaar. Dit spaart de assignment b := y uit en bovendien de geheugenruimte voor y .

Het antwoord op deze vraag is dat de variabele b ook tevens als globale variabele in de procedurebody S , of in de body van enige in S aangeroepen procedure, kan voorkomen. Als dat zo is dan is de voorgestelde werkwijze fout.

voorbeeld: Wanneer b en y aliases van elkaar zijn, dan kunnen we het volgende merkwaardige stukje programma opschrijven:

```
{ b = 3 }
  y := 5
{ b = 5 }  .
```

Dat b en y aliases zijn betekent dat b=y invariant is: wijzigingen in y zijn dan tevens wijzigingen in b ; dientengevolge geldt de regel voor de assignmentstatement niet meer.

□

Wanneer het toegestaan is in de procedurebody voorkomende globale variabelen als argumenten voor resultparameters te gebruiken dan zit er niets anders op dan de resultparameteroverdracht met behulp van copiëren te implementeren. In het algemeen wordt dit echter niet als een goede programmeergewoonte beschouwd die bovendien gemakkelijk kan worden vermeden. Daarom beperken we ons tot programma's die aan de, zogenaamde, *anti-aliasingvoorwaarde* voldoen:

> De globale variabelen die in de procedurebody, of in de body van enige daarin aangeroepen procedure, voorkomen worden niet als argument voor enige resultparameter van die procedure gebruikt.

Als aan de anti-aliasingvoorwaarde is voldaan mogen de parameter en het argument worden vereenzelvigd. Het enige probleem hierbij is nog hoe het, van invocatie tot invocatie verschillende, argument in de code van de procedurebody wordt geïdentificeerd. Hiervoor kan indirecte adressering worden gebruikt. De resultparameter wordt opgevat als een lokale variabele waarvan de waarde de identiteit -- in een of andere vorm -- van de als argument meegegeven variabele is. Deze identiteit wordt bij de call als waarde overgedragen ter initialisatie van de resultparameter. De resultparameter wordt nu dus als een gewone valueparameter behandeld, zij het dat referenties aan deze parameter in de procedurebody een extra indirectie bij de adressering vereisen.

Een variabele waarvan de waarde de identiteit van een andere variabele is wordt wel een *descriptor* genoemd --vergelijkbaar met een *pointer* in een Pascal programma-- . Als (representatie van de) identiteit van een variabele komen in aanmerking het adres van die variabele en, wanneer adressen te vluchtig zijn, de invocatienaam --zie §8.0-- van die variabele. De tekstuele naam --zie §8.2-- kan hiervoor niet worden gebruikt, want de betekenis hiervan is context-afhankelijk. Het hier beschreven mechanisme, waarbij resultparameters worden geïmplementeerd met behulp van descriptors, wordt wel *call by reference* genoemd. Het Pascal mechanisme met **var**-parameters komt eveneens neer op call by reference. We hebben hier laten zien dat dit een goede implementatie van result-parameters oplevert mits aan de anti-aliasingvoorwaarde is voldaan.

9.3 Value-resultparameters

Value-resultparameters verschillen alleen hierin van resultparameters dat de value-resultparameter *wel* en de resultparameter *niet* bij de procedureaanroep wordt geïnitialiseerd met de waarde van het meegegeven argument. Het in de vorige paragraaf beschreven call by reference mechanisme bewerkstelligt echter vanzelf dat de parameter bij aanroep de waarde van het argument krijgt. Mits de anti-aliasingvoorwaarde ook geldt voor variabelen die als argument voor value-result-parameters worden gebruikt, kunnen value-resultparameters dan ook op precies dezelfde manier worden behandeld als resultparameters.

10 Geheugenbeheer

10.0 Inleiding

In de voorafgaande hoofdstukken hebben we steeds gebruik gemaakt van het begrip geheugensegment. Hierbij hebben we aangenomen dat er een mechanisme is waarmee naar behoefte segmenten kunnen worden gereserveerd en weer worden vrijgegeven. De cruciale regel hierbij is uiteraard dat segmenten *disjunct* moeten zijn, wat betekent dat zij elkaar niet overlappen. (Nota bene: het gaat hier om ruimtereservering; het is heel wel denkbaar dat een segment wordt onderverdeeld in *subsegmenten* die dan per definitie niet disjunct zijn met het segment waar zij deel van zijn.)

Afhankelijk van de eigenschappen van de uit te voeren programma's en van de wijze van gebruik van de machine kan het mechanisme voor het geheugenbeheer in complexiteit variëren: van een enkele machineinstructie tot een procedure in het operating-system waarvan de uitvoering --af en toe-- een aanzienlijke hoeveelheid rekentijd vergt. Dit laatste kan nodig zijn omwille van de gewenste *flexibiliteit* maar we dienen daarbij te beseffen dat alle (reken)tijd en (geheugen)ruimte die wordt gebruikt voor geheugenbeheer --algemener: *systeembeheer*-- niet beschikbaar is voor de eigenlijke programmauitvoering. In een goed ontworpen systeem is deze *overhead* niet excessief groot in verhouding tot wat ermee wordt bereikt.

Merk op dat we *niet* eisen dat de overhead *minimaal* is. Ten eerste is niet duidelijk wat dat zou moeten betekenen, want verkleining van het ruimtebeslag heeft meestal een vergroting van het tijdgebruik tot gevolg, en vice versa. Ten tweede kan het, omwille van de flexibiliteit, heel wel verdedigbaar zijn om een vrij aanzienlijk deel van de capaciteit van het systeem voor systeembeheer te gebruiken. Hierbij speelt verder een rol dat minimalisering van de overhead op één plaats elders inefficiënties kan introduceren --*suboptimalisatie* is geen optimalisatie-- .

We kunnen deze beschouwingen samenvatten in de volgende ontwerpregel:

De hoeveelheid aan administratie bestede tijd en ruimte moet in redelijke verhouding staan tot het tijd- en ruimtegebruik van het geadministreerde.

voorbeeld: Het is redelijk aan te nemen dat een segment ter lengte N gedurende tenminste O(N) tijd in gebruik zal zijn. Alleen al het toekennen van waarden aan elk der elementen van het segment --i.e.: de initialisatie-- kost immers O(N) tijd. Wanneer allocatie van een segment nu, gemiddeld, niet meer dan

O(N) tijd kost dan kan dit nog als redelijk worden beschouwd. Wanneer deze kosten echter veel groter dan lineair in N zijn dan is de overhead van het geheugenbeheer onacceptabel groot.

□

10.1 Segmentallocatie: een eenvoudig geval

Wanneer de collectie voor een proces benodigde segmenten groeit en slinkt zoals een stapel --dit wordt bepaald door de structuur van het programma-- en wanneer er op elk moment maar één proces is, dan is het geheugenbeheer eenvoudig. De benodigde segmenten kunnen dan aaneengesloten in het geheugen worden geplaatst. De zo gecreëerde segmentenrij --vaak eenvoudigweg *de stapel* genoemd-- kan nu op twee manieren groeien: beginnend bij de laagste adressen worden (cellen met) steeds hogere adressen in gebruik genomen, of beginnend bij de hoogste adressen worden steeds lagere adressen in gebruik genomen.

De hele segmentenrij kan worden gerepresenteerd door een enkel adres-register dat het adres van hetzij de eerste vrije cel hetzij de laatst in gebruik genomen cel bevat. Allocatie van een segment ter lengte N vindt dan plaats door de inhoud van dat adresregister met N te verhogen dan wel te verlagen, afhankelijk van de richting waarin de rij groeit.

Het is zelfs mogelijk twee zulke segmentenrijen onder te brengen: de ene groeit dan tegen de andere in. Aldus kan de machine 2 processen die ieder 1 zo'n rij gebruiken of 1 proces dat 2 zulke rijen gebruikt herbergen. Het is daarbij wel gewenst dat de processor over een mechanisme beschikt dat bij segmentallocaties verifieert dat de twee rijen disjunct blijven.

10.2 Segmentallocatie: een algemener geval

We behandelen hier één van de vele mogelijke manieren om een groeiende en slinkende collectie segmenten te beheren. We maken nu geen veronderstellingen over de volgorde waarin gereserveerde segmenten weer worden vrijgegeven. Als gevolg hiervan is het voor de discussie niet relevant of die segmenten tot hetzelfde proces of tot verschillende processen behoren. We behandelen het ontwerp slechts informeel; de formele uitwerking hiervan is een "gewone" programmeeropgave.

We beschouwen het gehele geheugen als één aaneengesloten rij segmenten. Ieder segment in deze rij is ofwel in gebruik bij een (of meer) der processen ofwel bij geen der processen in gebruik. De in gebruik zijnde segmenten noemen

we *bezet* en de niet in gebruik zijnde segmenten noemen we *vrij*. Voor het beheer van de collectie segmenten zal wel enige administratie nodig zijn. Dat werken we hier niet uit; merk echter op dat hiervoor de ruimte in de vrije segmenten gebruikt mag worden.

Twee opeenvolgende vrije segmenten kunnen zonder bezwaar tot één groter segment worden gecombineerd; een vrij segment kan immers altijd, naar behoefte, in kleinere worden gesplitst. Het is dus mogelijk invariant te houden dat geen twee opeenvolgende segmenten vrij zijn. Voor de correctheid van het hier behandelde geheugenbeheer is dit echter niet nodig. Vrijgeven van een bezet segment komt dan neer op het, in de administratie, als vrij markeren van dat segment, eventueel gevolgd door het combineren van dat segment met vrije buursegmenten.

Reservering van een segment ter lengte N vindt als volgt plaats. Uit de vrije segmenten wordt er één geselecteerd ter lengte M, met $N \leqslant M$. We nemen hierbij voorlopig aan dat er zo'n vrij segment is. Het geselecteerde vrije segment wordt gesplitst in een bezet segment ter lengte N en een vrij segment ter lengte $M-N$. In de regel zullen er meer dan één vrije segmenten een lengte tenminste N hebben. Er zijn dan verschillende selectiestrategieën denkbaar, waaronder de volgende, die bekend en eenvoudig zijn:

first fit : Het segment wordt gekozen met behulp van, bijvoorbeeld, Linear Search. Het eerste het beste segment dat groot genoeg is wordt gebruikt.

best fit : Het kleinste segment dat groot genoeg is wordt geselecteerd.

worst fit: Het grootste vrije segment wordt geselecteerd.

De motivatie van first fit is eenvoud. De motivatie van best fit is dat gehoopt wordt zo een hoge *bezettingsgraad* van het geheugen te bereiken. Helaas pakt dit verkeerd uit: als gevolg van deze strategie ontstaan er, met grote waarschijnlijkheid, grote aantallen erg kleine vrije segmenten, te klein om nog bruikbaar te zijn. De motivatie van worst fit is juist dat het splitsen van een groot segment hopelijk een wel bruikbaar restant oplevert.

We bestuderen nu het geval waarin er geen voldoende groot vrij segment is. Onder de *bezettingsgraad* van het geheugen verstaan we de verhouding tussen de som der lengtes der bezette segmenten en de totale geheugengrootte. De bezettingsgraad is dus een getal tussen 0 en 1, waarbij 1 volledige benutting en 0 volledige leegte voorstelt. De slechtste situatie doet zich voor wanneer het geheugen bestaat uit een afwisseling van bezette segmenten ter lengte 1 en vrije

segmenten ter lengte N-1 . De bezettingsgraad is dan 1/N , wat voor enigszins grote N zo gering is dat de "oplossing" die erin bestaat dat de aanvraag niet wordt gehonoreerd omdat het geheugen te vol is --i.e.: voortijdige beëindiging van het aanvragende proces met de melding "machine te klein"-- onacceptabel is.

Op het gehele geheugen wordt nu een bewerking uitgevoerd die *compactie* wordt genoemd: alle bezette segmenten worden zó verplaatst dat zij een aaneengesloten rij, aan het begin van het geheugen, vormen. Het resultaat hiervan is dat alle vrije ruimte nu één groot vrij segment vormt, aan het eind van het geheugen. Wanneer dit ene segment nog steeds kleiner is dan N dan is het geheugen echt te klein voor het proces of --in het geval van multiprocessing-- voor de aanwezige combinatie van processen.

Omdat segmenten nu kunnen worden verplaatst zijn alle geheugenadressen vluchtig. Wanneer in het geheugenbeheer compactie wordt gebruikt is het dan ook gewenst geheugenadressen niet te dupliceren. In eerdere hoofdstukken hebben we aangetoond dat dit kan en dat een zodanige organisatie mogelijk is dat ieder segmentadres op slechts één plaats --behoudens een enkele copie in een bekend register, zoals PI-- wordt geadministreerd. Omdat deze segmentadressen bij compactie moeten worden gewijzigd moeten de locaties van deze adressen uiteraard wel voor de compactieprocedure toegankelijk zijn. Dit kan op twee manieren worden gerealiseerd.

Allereerst kunnen alle bezette segmenten worden geïdentificeerd door ze te nummeren. Deze nummers worden op de adressen van de bijbehorende segmenten afgebeeld met behulp van een tabel. Alle adressering van informatie in de segmenten verloopt via deze tabel. De compactieprocedure hoeft dan alleen deze tabel aan te passen. Zonodig kan in ieder segment op een vaste positie --bijvoorbeeld positie 0-- het segmentnummer worden opgeslagen. Deze werkwijze is eenvoudig maar heeft het bezwaren dat ieder geheugenacces nu een indirectie via de tabel vereist en dat er ruimte voor de tabel moet worden gereserveerd.

Een verfijndere werkwijze bestaat hierin dat in ieder segment op een vaste positie informatie wordt opgeslagen die de geheugenplaats --in de administratie van het proces dat het segment gebruikt-- identificeert waarin het segmentadres is opgeslagen. Om verdere duplicatie van adressen te vermijden moet deze informatie niet een adres zijn maar een positieonafhankelijke codering hiervan. Wanneer het segment in gebruik is voor een activation record, dan kan dit, bijvoorbeeld, het invocatienummer samen met --in het geval van multiprocessing-- de naam van het proces zijn. Via de invocation table --zie §8.0-- van dat proces kan dan het segmentadres worden teruggevonden. Wanneer de invocation table zelf een ook segment in beslag neemt dan wordt het adres hiervan waarschijnlijk in een (vast) adresregister gehouden. In dit geval volstaat een codering die, behalve de naam van

het proces, aangeeft dat het om de invocation table gaat. Deze werkwijze is alleen dan realiseerbaar wanneer voor elke vorm van segmentgebruik door de processen een geschikte identificatie van het segmentadres kan worden ontworpen.

10.3 De 50-procentregel

De operatie "reserveer een segment ter grootte N" zal af en toe aanleiding geven tot een compactieoperatie. Dit kost in de regel een hoeveelheid tijd die evenredig is met de afmeting van het gehele geheugen. Is dat niet in strijd met de in § 10.0 behandelde ontwerpregel, op grond waarvan de reserveringsoperatie niet meer dan O(N) tijd mag kosten? Het antwoord hierop is dat het niet erg is --behoudens in zogenaamde *real-time systemen*-- dat zo'n operatie *incidenteel* duur is, zolang maar het gemiddelde van de kosten van een opeenvolgende reeks operaties aan de redelijkheidseis voldoet. (Het begrip *gemiddelde* wordt hier dus niet in de kanstheoretische betekenis van *verwachting* gebruikt.) Dit wordt wel *geamortizeerde efficiëntie* genoemd. Mits nu het geheugen van de machine voldoende groot is zal de voor een compactieoperatie benodigde tijd ten hoogste evenredig zijn met de som van de lengtes van de sinds de vorige compactieoperatie gereserveerde segmenten.

We analyseren dit als volgt. Met N de maximaal optredende afmeting, in cellen, van de collectie bezette segmenten en M de afmeting, eveneens in cellen, van het geheugen, kunnen we drie interessante gevallen onderscheiden:

- M=N: Dit geval representeert de ideale situatie vanuit het oogpunt van ruimtegebruik. De collectie segmenten wordt ingebed in het kleinst mogelijke geheugen. Het is echter denkbaar dat iedere reservering een compactie vereist, zodat geen enkele bovengrens voor het tijdgebruik van de machine kan worden gegeven.
- M=2*N: Omdat de collectie segmenten altijd ten hoogste N cellen groot is beslaat de hoeveelheid vrije ruimte op enig moment ten minste N cellen. De compactieoperatie, die O(M) , en nu dus ook O(N) tijd kost, verenigt alle vrije ruimte in een segment dat ten minste N groot is. Dit is bovendien pas nodig nadat er voor tenminste N cellen segmenten zijn gereserveerd en weer vrijgegeven. Redelijkerwijs heeft het gebruik van deze ruimte dan zólang geduurd dat er tussen iedere twee compacties tenminste zoveel tijd verloopt als een compactie kost.

- $M = \infty$: Dit geval representeert de ideale situatie vanuit het oogpunt van tijdgebruik. Omdat het geheugen oneindig groot is is compactie niet nodig en wordt de collectie segmenten beheerd met het kleinst mogelijke gebruik van tijd. Er kan echter geen enkele bovengrens voor het ruimtegebruik van de machine worden gegeven.

De gevallen $M = N$ en $M = \infty$ zijn, in zekere zin, *duaal* onder verwisseling van tijd en ruimte. Het tussenliggende geval vormt hierin het best denkbare compromis: met een geheugen dat tweemaal zo groot is als het minimaal benodigde kan het systeem (ongeveer) de helft van de maximaal haalbare snelheid halen. Dit kan ook iets anders worden geformuleerd, in wat we de *50-procentregel* noemen:

> *Wanneer de bezettingsgraad van het geheugen ten hoogste 50% is haalt de machine ten minste 50% van de maximale snelheid.*

De stelling is dat we met een flexibel ontwerp dat aan de 50-procentregel voldoet heel tevreden kunnen zijn: waarschijnlijk kan het niet beter en meer dan een factor 2 valt er niet meer te verdienen.

Laat N de minimale grootte van het geheugen zijn waarin een programma kan worden uitgevoerd; laat T de minimale hoeveelheid tijd zijn die een gegeven processor nodig heeft om het programma uit te voeren. Wanneer de geheugengrootte daadwerkelijk N is dan zal de benodigde tijd ∞ zijn -- in de zin dat we er geen bovengrens voor kunnen aangeven -- , terwijl om het programma daadwerkelijk in T tijd uit te voeren het geheugen juist ∞ groot moet zijn. Wanneer het systeem aan de 50-procentregel voldoet dan kan zo'n programma in een geheugen ter grootte ten minste $2*N$ worden uitgevoerd in ten hoogste $2*T$ tijd, ofwel in T tijd wanneer de processor door een twee keer zo snel exemplaar wordt vervangen. Kortom, om dezelfde snelheid te halen als in het ideale geval volstaat het de geheugencapaciteit en de processorsnelheid twee keer zo groot te laten zijn als wat minimaal nodig is.

De 50-procentregel is een vuistregel. In sommige situaties kan het heel wel verdigbaar zijn dat zelfs een 4 maal zo groot geheugen nodig is om de processor tenminste 1/4 van de snelheid te laten halen; in andere gevallen lukt het wellicht 80% van de capaciteit daadwerkelijk over te houden. Waar het om gaat is dat we, als systeemontwerper, kunnen *garanderen* dat als de bezettingsgraad van het geheugen een gegeven waarde niet te boven gaat, dat dan de bezettingsgraad van de processor niet onder een gegeven waarde komt.

(einde van rh144)