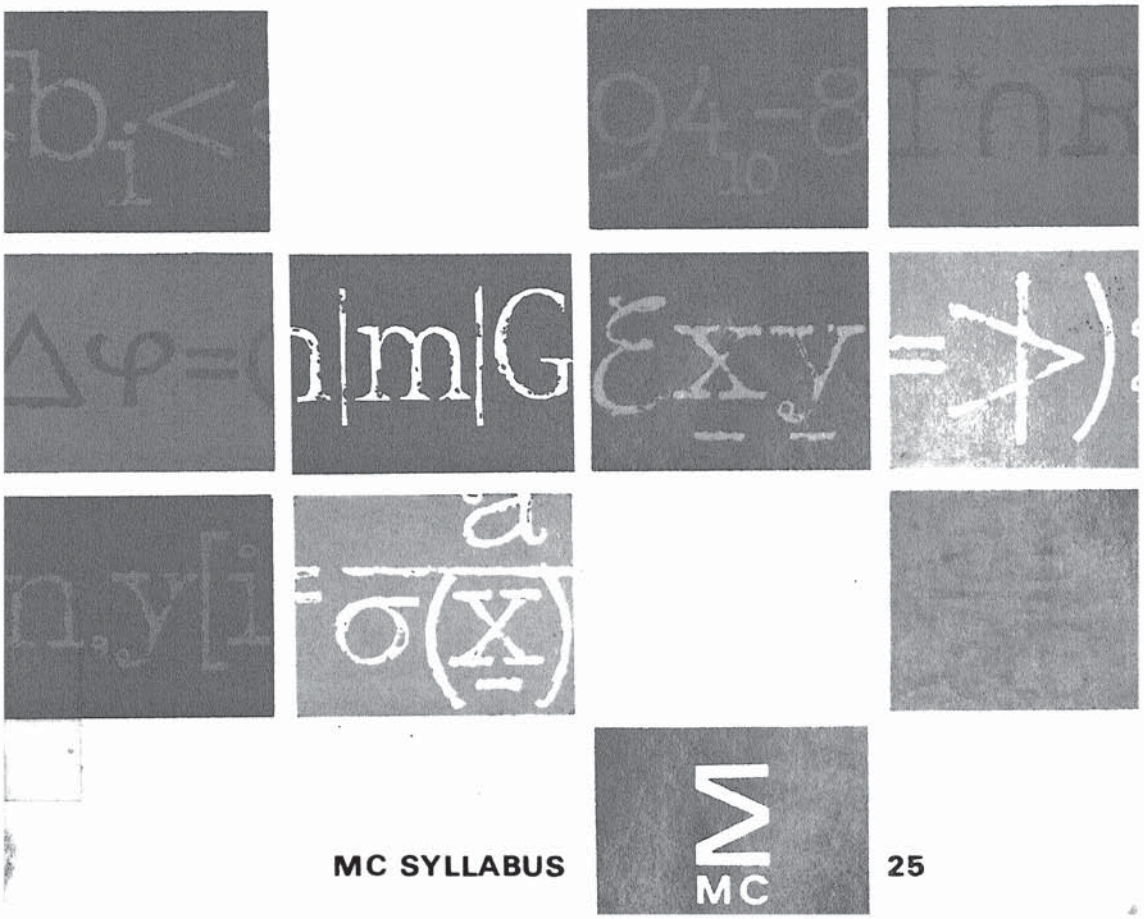
COLLOQUIUM STRUCTUUR VAN PROGRAMMEERTALEN



MC SYLLABUS 25

MC SYLLABUS 25

L. AMMERAAL
H.J. BOOM
D. GRUNE
P.J.W. TEN HAGEN
P. KLINT
F.E.J. KRUSEMAN ARETZ
L. MEERTENS
R.P. VAN DE RIET
W.P. DE ROEVER
A. TANENBAUM

COLLOQUIUM STRUCTUUR VAN PROGRAMMEERTALEN

IA

MATHEMATISCH CENTRUM AMSTERDAM 1976

AMS(MOS) subject classification scheme (1970): 68-00, 68A05, 68A30

ACM-Computing Reviews-categories: 4.20, 4.22, 4.29, 4.12, 8.2, 6.0

ISBN 90 6196 116 5

INHOUD

Inhoud v

Voorwoord IX

I. FLOW-OF-CONTROL . . . . . .
VERGELIJKING VAN BESTAANDE PROGRAMMEERTALEN door D. GRUNE

1. Inleiding 1
2. De talen 2
3. Vormen van flow-of-control 4
4. Samenvatting. 20
Literatuur 20

II. GRAFISCHE PROGRAMMEERTALEN door P.J.W. ten HAGEN

1. Inleiding 23
2. Tekenmachines 24
3. Datastructuren voor tekeningen. 29
4. Grafische taal in ALGOL 68 31
5. Interactief tekenen 39
Literatuur 40

III. DATASTRUCTUREN: VERGELIJKING VAN BESTAANDE
PROGRAMMEERTALEN door L. AMMERAAL

1. Data en geheugenorganisatie 41
2. Over het vergelijken van talen 42
3. Datatypes in enige conventionele talen 44
4. Iets over enkele nieuwere talen 50
5. Over modes in ALGOL 68 51
Literatuur 57

IV. DE HEEL EENVOUDIGE TAAL HET door L. MEERTENS

0. Inleiding 59
1. Primitiva van programmeertalen 61
2. Semantiek van de overige speciale-symbolen 72
3. Hors d'oeuvre: zichzelf reproducerende programma's 73
4. Kanttekeningen 74
Literatuur 77

V. VERGELIJKING VAN ENKELE PROGRAMMEERTALEN VOOR EEN BASISCURSUS INFORMATICA door R.P. VAN DE RIET

0. Samenvatting 79
1. Inleiding 79
2. Het probleem van de torens van Hanoi 80
3. Conclusie 90
Referenties 91

VI. SOME CRITERIA FOR ELEMENTARY PROGRAMMING LANGUAGES by R.P. VAN DE RIET

0. Abstract 93
1. Introduction 93
2. General considerations 94
3. Detailed criteria 98
4. Existing Systems 102
5. BASIS, an interactive system 107
6. BASIS and the original design criteria 115
Acknowledgments 116
References 116

VII. CORRECTHEIDSBEWIJZEN VOOR ZOEK- EN MARKERINGS-ALGORITMEN VAN DYADISCHE DATA-STRUCTUREN door W.P. DE ROEVER

1. Een correctheidsbewijs van een zoekalgoritme voor dyadische bomen zonder hulpstack of hulpbits 121
2. Een correctheidsbewijs van een markeringsalgoritme voor gerichte dyadische grafen zonder hulpstack 124
Literatuur 131

VIII. PORTABILITEIT door P. KLINT

0. Inleiding 133
1. Hardware-oplossingen 134
2. Software-oplossingen 136
3. Besluit 151
Literatuur 151

IX. OBJECT CODE GENERATION by H.J. BOOM

0. Introduction 155
1. Input to the code generator 155
2. Stack and nonstack machines 158
3. Portability, and the UNCOL problem 161
4. Code generation for ALGOL 68 163
5. Directive text 165
6. Example: complex arithmetic 166
7. Tame and wild data 170
8. Register and storage allocation 170
9. Storage addressing 174
10. Operating system compatibility 175
References 175

X. NAAR EEN FOUTENWERENDE PROGRAMMEERTAAL door L. MEERTENS

0. Inleiding 177
1. De methode van de inductie-asserties 177
2. Programma en betekenis 181
Literatuur 193

XI. PROGRAMMING LANGUAGES AND HARDWARE by A. TANENBAUM

0. Introduction 197
1. Ancient history 197
2. Some influences of hardware on the design of programming languages 198
3. The hardware software interface 203
4. The MU5. 208
5. Burroughs B1700 215
References 218

XII. CONCURRENT PASCAL, EEN TAAL VAN P. BRINCH HANSEN door F.E.J. KRUSEMAN ARETZ

0. Introductie 219
1. System types en system components 219
2. Shared Data 221
3. Access rights 224
4. Een spooling system 225

5. Classes 228
6. Besluit 235
Literatuur 236

VOORWOORD

In deze Syllabus zijn de voordrachten gebundeld, gehouden in het academisch jaar 1974/1975 in het door de Afdeling Informatica van het Mathematisch Centrum georganiseerde Colloquium Structuur van Programmeertalen.

Gaarne betuig ik mijn dank aan de sprekers in het Colloquium, in het bijzonder de niet aan de Afdeling verbonden gastsprekers, en aan L.J.M. Geurts voor zijn aandeel in de organisatie van het Colloquium en de verzorging van de Syllabus.

J.W. de Bakker

MATHEMATICAL CENTRE SYLLABUS 25, 1976, I, 1-21

FLOW-OF-CONTROL
VERGELIJKING VAN BESTAANDE PROGRAMMEERTALEN

D. GRUNE

1. INLEIDING

Voordat ik inga op allerlei min of meer geavanceerde vormen van flow-of-control, wil ik de volgende naïeve vraag proberen te beantwoorden: "Waarvoor hebben we flow-of-control nodig? Waarom schrijven we niet gewoon achter elkaar neer wat de computer voor ons moet doen?" Het antwoord, dat ook dát een vorm van flow-of-control is, zal alleen de zeer filosofisch denkenden onder U aanspreken.

Het dwingendste antwoord is, dat we vaak bij het schrijven van het programma niet weten wat de opvolger van een bepaalde opdracht is en dat we daarvoor bij de (invoer)-gegevens te rade moeten gaan (a). Verder gebruiken we een computer om groepen opdrachten herhaald uit te voeren; wanneer het niet bij voorbaat bekend is hoeveel keer de groep uitgevoerd moet worden, kunnen we hem niet uitschrijven (b) en zelfs als dat wel bekend is kunnen we terugschrikken voor de gevolgen in termen van werk- en geheugenruimte (c). Sinds enige jaren is hier een vierde antwoord bijgekomen: structurering. We zouden de uit te voeren instructies wel degelijk ter plaatse op kunnen schrijven, maar dat willen we niet want we vinden dat ze niet op hetzelfde abstractie-niveau staan als de omringende tekst: we willen ter plaatse alleen een op dát niveau passende verwijzing naar de (ondergeschikte) instructies opnemen, en dat impliceert flow-of-control (d) (GEURTS [1]).

Gezien het practische karakter van overwegingen a, b en c is het niet verwonderlijk dat vrijwel alle talen (zie echter 2.2) hierbij passende constructies hebben. Wat overweging (d) betreft, in vele talen kunnen we (soms met moeite en zelfbeheersing) wel ondergeschikte tekst elders plaatsen, maar we worden dan prompt voor onze goede bedoeling gestraft met betreurenswaardige objectcode. Dit is een indicatie van de practische "state of the art in programming language design".

Bovenstaande lijst is niet volledig. Zo danken b.v. co-routines (zie 3.4.2) hun bestaan niet duidelijk en direct aan één van de overwegingen (a) t/m (d). De wensen die leiden tot het gebruik van co-routines kan ik moeilijk onder woorden brengen; het opvallendste punt is de zeer nauwe samenhang tussen informatie-overdracht en flow-of-control.

2. DE TALEN

2.1. *Machine-code*

Hieronder versta ik de set van bit-patronen samen met de assembler die het werken met deze bit-patronen voor de mens acceptabel moet maken. De zichtbare flow-of-control in machine-code is meestal (behalve bij micro-programmeerbare machines) zeer primitief, maar, doordat stukken code en addressen in deze code gemakkelijk als data gehanteerd kunnen worden, kunnen alle in 3. te noemen methoden van flow-of-control in machine-code bedreven worden (uiteraard).

In machine-code is een vorm van flow-of-control mogelijk die vanwege (ondanks?) zijn uiterst dubieus karakter in het volgende niet behandeld zal worden: het overschrijven van instructies met andere instructies, waardoor een gegeven groep instructies plotseling een andere betekenis krijgt. Het is duidelijk dat hierbij de relatie tussen programma-tekst en semantiek nihil geworden is. Deze techniek hoort thuis in handen van de professionele catch-as-catch-can programmeur, die er onder omstandigheden inderdaad uiterst efficiente code mee kan produceren, wat in real-time systemen doorslaggevend kan zijn.

2.2. *Job Control Languages (JCL's)* [2,3]

Deze dienen om het globale verloop van een computer-run te sturen. Zij zijn in zekere zin geen programmeertalen omdat men er zelfs met de beste wil geen Turing-machine op kan simuleren, en ze komen het dichtst bij ons "ideaal" van sequentieel neerschrijven wat er moet gebeuren.

2.3. *Fortran* [4]

Een veel-gebruikte, goed-gesupporte taal van het één-regel-is-één-opdracht type. Is begonnen als omzetter van rekenkundige uitdrukkingen naar machine-code. De flow-of-control is die van vrij slimme machine-code.

2.4. *Algol 60* [5]

Een nestende programmeertaal met een, zeker voor 1960, uitmuntende flow-of-control. Enkele facetten (springen uit een procedure of via een parameter) maken het run-time system moeizamer dan nodig was geweest.

2.5. *PL/I* [6,7]

Een op Algol 60 gelijkende taal die duidelijk de sporen toont van het feit dat hij door een commissie ontworpen is (BY NAME-assignment, LIKE-attribute, etc.). De weglatingen uit het Algol-60-repertoire zijn ongelukkig (RECURSIVE-attribute), de toevoegingen eveneens (SIGNAL-statement). Praktisch gezien is de taal onbruikbaar door de onbegrijpelijke en onhanteerbare automatische conversie tussen data-typen.

2.6. *Simula 67* [8,9]

Algol 60 uitgebreid met uitmuntende data-handling en een, daarin ingebedde,(vrij primitieve) vorm van co-routines.

2.7. *ALEPH* [10]

Een niet op Algol of Fortran gelijkende taal, gebaseerd op de gedachte, dat, aangezien nu eenmaal alle programmeertalen de gebruiker dwingen zich in bochten te wringen, het beter is, hem te dwingen zich in goede bochten te wringen; de gevolgen hiervan zijn gunstig. De flow-of-control is van eenvoudige deugdelijkheid, slechts gebaseerd op selectie en procedure-aanroep.

2.8. *Algol 68* [11]

Een nestende programmeertaal met een rijk arsenaal aan mogelijkheden, zowel wat flow-of-control als wat data-handling betreft. De taal lijdt wat flow-of-control betreft aan dezelfde euvels als Algol 60, maar de problemen zijn wel meer expliciet geworden. Een aantal, in andere talen als ad-hoc constructies aanwezige, vormen van flow-of-control (b.v. co-routines, computed goto) zijn in Algol 68 bijzondere gevallen van algemenere constructies.

3. VORMEN VAN FLOW-OF-CONTROL

3.1. *Tekstuele opvolging (sequentialiteit)*

De opdracht B volgt A op omdat in de tekst B direct volgt op de opvolgings-operator na A. Deze opvolgings-operator wordt in b.v. Fortran en de meeste machine-codes verdoezeld als einde-kaart.

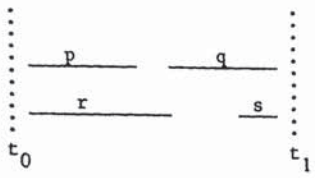
Vrijwel alle talen bezitten deze vorm van flow-of-control. Uitzonderingen zijn sommige antieke machine-codes waarbij elke opdracht expliciet een verwijzing naar een opvolger in zich borg (programma's op draaiende trommels).

3.2. *Collateraliteit*

In sommige talen (Algol 68) kunnen twee opdrachten A en B collateraal worden uitgevoerd, d.w.z. A en B worden tegelijkertijd gestart, a.h.w. op twee onafhankelijke processoren; de vraag naar de precieze volgorde van de uitvoering van de onderdelen van A en B is per definitie zinloos: ze worden langs twee verschillende, paralelle tijds-assen uitgevoerd.
De uitvoering van het stukje Algol 68:

```
((p;q), (r;s))
```

waarin de komma de collateraliteits-operator is en de punt-komma de opvolgings-operator, kunnen we b.v. als volgt in beeld brengen



waarbij we langs beide tijds-assen de schaal mogen vervormen.

Wanneer gewerkt wordt met één processor en wanneer we stellen dat p, q, r en s elk ondeelbare handelingen zijn, dan laat

```
((p,q), (r,s))
```

alle permutaties van p, q, r en s toe als mogelijke uitvoerings-volgordes, de vorm

```
((p;q), (r;s))
```

laat de zes volgordes

```
pqrs        rpqs
prqs        rpsq
prsq        rspq
```

toe, terwijl

```
((p;q); (r;s))
```

alleen pqrs toelaat.
Door gebruik van collateraliteit wordt de volgorde van uitvoeren op subtiele en soms moeilijk te overziene wijze vrij gelaten. De lezer die denkt dat dat wel meevalt, wordt aangeraden de mogelijke resultaten van de Algol 68 statement

```
(int n := 0;   print ((n := n + 1, n := n + 1, n := n + 1)))
```

te overdenken.

Collateraliteit is eigenlijk een merkwaardige vorm van flow-of-control: de nadruk ligt meer op het bewust niet willen specificeren van een volgorde, omdat deze voor het probleem niet ter zake zou doen. De reden van gebruik van collateraliteit ligt dus in de wens tot structurering. (Er wordt ook wel als voordeel aangegeven dat door het niet eisen van een bepaalde volgorde de compiler meer vrijheid krijgt voor optimalisering. Deze vrijheid is compiler-technisch echter moeilijk uit te buiten).

Als voorbeeld moge dienen een stukje Algol 68 dat de grootste waarde die een proc (int) int p op het interval [1:n] aflevert, opzoekt, met bijbehorende parameter par. We nemen aan dat we willen voorkomen de procedure p vaker dan n maal aan te roepen.

```
int par := 0, max := min int;
for i to n
do if int pi = p(i); pi > max then
      (par := i, max := pi ) fi
od
```

De nadruk ligt hier op het feit dat voor onze algorithme de volgorde van de assignments tussen then en fi volkomen onbelangrijk is, sterker nog, dat we ze samen het liefst als één handeling zouden willen zien.

Aan het gebruik van collateraliteit kleven bezwaren. Het effect is nog weinig onderzocht en voor de niet zeer geroutineerde programmeur is het vaak moeilijk te overzien wat het gevolg is van de vrijheden die hij geeft (zie print-statement boven). Dit bezwaar zou ondervangen kunnen worden door te eisen dat, wanneer één van de collaterale componenten de waarde van een globale variabele X wijzigt, deze variabele op geen enkele wijze door de andere componenten gebruikt mag worden. Dit is echter in de bestaande programmeertalen (met uitzondering van ALEPH, dat echter geen collateraliteit kent) niet redelijkerwijze met een compiler test of zelfs maar een run-time test na te gaan. Een betere aanpak is, te eisen dat de collaterale componenten geen van allen globale effecten hebben; het enige wat dan van belang is, is het resultaat dat ze afleveren. Bovenstaand voorbeeld krijgt dan de vorm:

```
struct (int par, max) s := (0, min int);
for i to n
do if int pi = p(i); pi > max of s then
      s := (i, pi) fi
od
```

hetgeen ook beter uitdrukt dat we de assignment als één geheel willen zien.

Algol 68 is de enige taal waarin collateraliteit expliciet aanwezig is; in vele andere talen is de volgorde van uitwerking van parameters zo weinig nauwkeurig gedefinieerd dat we practisch van collateraliteit kunnen spreken. Zoals hierboven aangetoond laat de vorm van collateraliteit in Algol 68 de programmeur te veel de gelegenheid om brokken te maken.

3.3. *Sprong zonder gegarandeerde terugkeer*

3.3.1. *De gewone sprong*

De gewone sprong-opdracht wijst door middel van een label zijn opvolger direct aan. Deze vorm van flow-of-control is in alle talen aanwezig; in Algol 60, PL/I, Simula en Algol 68 is de plaatsing van de label beperkt door de blokstructuur, in ALEPH zijn alleen die sprong-opdrachten toegestaan die equivalent zijn met een rechts-recursieve procedure-aanroep.

Op de slechte diensten die de gewone goto de programmeur bewijst, wordt elders (VAN DE RIET [12], WULF [13], WEGNER [14]) uitgebreid ingegaan. Wanneer een aantal andere vormen van flow-of-control beschikbaar is, kunnen sprongen vermeden worden. Een terugwaartse sprong duidt in een redelijk programma meestal op herhaling en kan dan vervangen worden door een expliciet herhalingsmechanisme (Fortran, Algol 60, Simula 67, Algol 68) of door een rechts-recursieve aanroep (ALEPH). Een voorwaartse sprong duidt meestal op het overslaan van een aantal opdrachten op grond van een nu of eerder genomen beslissing. Zo'n sprong dient vermeden te worden door het gebruik van een geschikt selectie-mechanisme (if-statement, case-statement). Wanneer de sprong niet tot een van deze twee categorieën is terug te brengen, zal het programma geherstructureerd moeten worden.

3.3.2. *De sprong met berekend adres*

Deze vorm die o.a. in Fortran als "computed goto", in Algol 60 en Simula 67 als switch en in Algol 68 als [] proc void beschikbaar is, wordt voornamelijk gebruikt om, om efficientie-redenen, snel op grond van de waarde van een integer variabele de volgende opdracht aan te wijzen. Alle aldus aangewezen opdrachtgroepen eindigen dan op een sprong naar één en dezelfde label. Zo'n constructie kan in Algol 68 vervangen worden door de veel overzichtelijker case-clause; in ALEPH is de enige uitdrukkingsmogelijkheid hiervoor de "classification". In Algol 68 kan b.v.

```
([] proc void switch = (ℓ1,ℓ2,ℓ3);
 switch [i];
ℓ1: action one; goto skip;
ℓ2: action two; goto skip;
ℓ3: action three; skip : skip
)
```

veel beter geschreven worden als

```
case i in
     action one,
     action two,
     action three
esac        .
```

3.3.3. *Aanroep van een procedure waaruit gesprongen wordt*

Deze woeste vorm van flow-of-control is beschikbaar in o.a. Algol 60, Simula 67 en Algol 68. Bij uitvoering van zo'n sprong wordt en passant een stuk van de run-time stack weggegooid. Zeker wanneer de label een parameter is van een recursieve procedure, vereist het de nodige studie de semantiek van zo'n sprong te doorgronden, nog afgezien van het feit dat het het run-time systeem nodeloos ingewikkeld maakt. De sceptische lezer mag proberen de werking van het volgende Algol 60 programma te analyseren en zijn resultaten vergelijken met die van een computer-run.

```
begin procedure p (i,label); value i, label;
          integer i; label label;
     begin if i = 0 then goto label else
          p(i-1,if i : 2 * 2 = i then here else label);
     here: print (i);
          if i : 4 * 4 = i then goto label
     end p;
     p (10,end); end:
end
```

Het advies dat ik geneigd ben te geven voor het gebruik van deze vorm van flow-of-control is "Niet doen". Zo eenvoudig ligt het echter niet. Wanneer we in een goed top-down gestructureerd programma (b.v. een compiler) heel ergens in de diepte, in een sub-sub-sub-routine, tot de ontdekking komen dat we niet verder kunnen (onverwacht eind invoergegevens, gegevens waarop de gevraagde operatie niet uitgevoerd kan worden, enz.), dan willen we "er uit", terug naar een veel hoger niveau, d.w.z. we willen inderdaad een stuk van de stack weggooien. In een in dit opzicht goed georganiseerde taal (b.v. ALEPH) is de enige manier om dit te bewerkstelligen het uitpro-

grammeren van deze vluchtweg terug dwars door alle tussenliggende lagen heen. Mathematisch en filosofisch gezien is dit inderdaad de enig juiste manier: door op de tussenliggende niveaus geen rekening te houden met voor lagere niveaus onoplosbare moeilijkheden, hebben we het probleem ontoelaatbaar vereenvoudigd. Om niet helemaal duidelijke redenen strijdt deze aanpak echter tegen ieders esthetische gevoelens: hij is totaal onverkoopbaar. Om dit probleem (dat weinig voorkomt, maar als het voorkomt, zeer hardnekkig is) op te lossen, moeten we óf onze esthetische gevoelens veranderen, óf er achter zien te komen waarom het tegen onze gevoelens strijdt (het feit dat tussenliggende lagen principieel niets anders kunnen doen dan de informatie doorgeven?), óf de pil vergulden. Er is hierover weinig gepubliceerd (LOMET [15]) en de (onbevredigende) voorstellen die ik gehoord heb, richten zich alle op het laatste.

3.4. *Sprong met gegarandeerde terugkeer*

3.4.1. *De procedure-aanroep*

De procedure-aanroep, die in alle talen (zelfs sommige JCL's) beschikbaar is, bergt tijdens de sprong een verwijzing naar de opvolger van de procedure-aanroep ergens op, zodat na beëindiging van de procedure de verwerking op dat punt voortgezet kan worden. Bij een nette procedure is dat de enige manier waarop hij beëindigd kan worden (anders valt hij onder paragraaf 3.3.3.). Wanneer dit opbergen op een stack geschiedt, kan de procedure recursief aangeroepen worden. (Algol 60, Simula 67, ALEPH, Algol 68).
De procedure-aanroep is ons beste hulpmiddel voor ruimtebesparing en structurering. Hij geeft weinig aanleiding tot moeilijkheden en dient de programmeur al sinds het begin van het programmeren.

3.4.2. *De co-routine-aanroep*

Bij een co-routine-aanroep wordt de terugkeerverwijzing niet, zoals bij de recursieve procedure-aanroep, op een stack bewaard, of, zoals bij de niet-recursieve procedure-aanroep, bij de aangeroepen procedure, maar bij de aanroepende procedure. Wanneer de aanroepende procedure, die dan dus stopt, later opnieuw gestart wordt, vervolgt hij zijn werk op het door de terugkeerverwijzing aangewezen punt. Dit maakt het mogelijk, van twee verschillende procedures om beurten stukjes uit te voeren. De co-routine-aan-

roep is in expliciete vorm alleen in Simula 67 beschikbaar (DAHL [8]).

De practijk heeft geleerd dat de co-routine-aanroep niet uitgelegd, maar alleen gedemonstreerd kan worden. Vandaar het volgende voorbeeld: We hebben een proces A dat characters leest en deze ook weer uitvoert, behalve dat waar in de invoerstroom "aa" voorkomt "b" uitgevoerd wordt; verder hebben we een soortgelijk proces B dat "bb" in "c" omzet. We willen nu deze twee processen laten samenwerken door de uitvoer van A als invoer aan B aan te bieden. We zouden nu B tot hoofdprogramma kunnen benoemen en hem A als subroutine laten aanroepen. Wanneer we echter ruw de structuur van het programma A beschouwen, dan komen daarin verspreid vier uitvoer-opdrachten voor; om van A een sub-routine te maken, zouden deze allemaal naar het eind verhuisd moeten worden (het character moet nu immers afgeleverd worden) wat een forse ingreep in de structuur van A is. We laten A dus liever hoofdprogramma naast B (A als hoofdprogramma en B als uitvoer-routine geeft natuurlijk dezelfde moeilijkheden) en dat kan door tussen beide een co-routine-communicatie te leggen.

Het volgende in Algol 68 neergeschreven machine-code programma geeft weer hoe dat functioneert (zie ook KRIEG [16]):

```
begin proc void label A := skip, label B := ℓ6;
      proc void co call A = (proc void ℓ) void : (label B := ℓ; label A),
                co call B = (proc void ℓ) void : (label A := ℓ; label B);
      char ch, ch1;

      proc A: read (ch); if ch = "a" then goto a found fi;
              co call B (ℓ2); ℓ2 : goto proc A;
      a found: read (ch); if ch = "a" then goto generate b fi;
               ch1 := ch; ch := "a"; co call B (ℓ3); ℓ3 :
               ch := ch1; co call B (ℓ4); ℓ4 : goto proc A;
      generate b: ch := "b"; co call B (ℓ5); ℓ5 : goto proc A;

      proc B: co call A (ℓ6); ℓ6 : if ch = "b" then goto b found fi;
              print (ch); goto proc B;
      b found: co call A (ℓ7); ℓ7 : if ch = "b" then goto generate c fi;
               print ("b"); print (ch); goto proc B;
      generate c: print ("c"); goto proc B
      end
```

Hierin worden de proc void's label A en label B gebruikt om de positie in proc A resp. proc B vast te houden. Een co-call A (gedaan vanuit proc B) slaat de meegekregen positie in label B op en (her)start proc A op label A.

Tegen dit programma zijn de nodige bezwaren in te brengen. Ten eerste hebben we de programma's A en B moeten villen om alle er in voorkomende labels toegankelijk te maken voor het co-routine-mechanisme. Ten tweede zijn we gedwongen de labels ℓ2 t/m ℓ7 in te voeren, die er bepaald niet ten gerieve van de programmeur staan. Ten derde komt op geen enkele wijze tot uiting dat waar het eigenlijk bij de co-routine-aanvraag om gaat, de informatie-overdracht via de variabele ch is: telkens wanneer proc A weer een waarde van ch klaar heeft volgt een co-call B en telkens wanneer proc B een waarde van ch nodig heeft volgt een co-call A. Ten vierde is er in deze schrijftechniek niets dat verhindert om in proc A een co-call A te doen (tot groot onheil). En ten vijfde is de initialisatie van label A en label B weinig doorzichtig.

Het is duidelijk dat we op grond van het bovenstaande niet kunnen zeggen dat we in Algol 68 co-routines kunnen schrijven.
Opmerking: als we het bovenstaande programma op de traditionele co-routine-manier hadden geschreven, d.w.z. als we label A en label B hadden laten samenvallen en de co-routine-aanroep hadden uitgevoerd als verwisseling van adressen (een optimalisatie die alleen voor twee co-routines werkt, maar die voor drie uiterst verwarrend is) en de gelabelde sprongen hadden weggewerkt, dan zou het programma baanbrekend op het gebied van onleesbaarheid zijn geworden.

Zoals gezegd zijn in Simula 67 co-routines expliciet beschikbaar. In Simula 67 ziet het programma er als volgt uit:

```
begin integer ch;

      class double a to b;
      begin detach;
            while true do
            begin read (ch);
                  if ch = char a then
                  begin read (ch);
                        if ch = char a then
                        begin ch := char b; resume (proc B) end
                        else begin integer ch1; ch1 := ch;
                                    ch := char a; resume (proc B);
                                    ch := ch1; resume (proc B)
                              end
                  end else resume (proc B)
            end infinite loop
      end double a to b;

      class double b to c;
      begin detach;
            while true do
            begin resume (proc A);
                  if ch = char b then
                  begin resume (proc A);
                        if ch = char b then print (char c) else
                        begin print (char b); print (ch) end
                  end else print (ch)
            end infinite loop
      end double b to c;

      ref (double a to b) proc A;
      ref (double b to c) proc B;

      proc A :- new double a to b;
      proc B :- new double b to c;
      call (proc A)
end
```

De class ... begin ... end-constructie is de definitie van een klasse van processen zoals tussen begin en end. De statements die beginnen met ref declareren objecten van de aangeduide klasse en de twee statements met de :- operator creëren elk een proces van de genoemde klasse en kennen dat toe aan proc A, resp. proc B. Met de statement call (proc A) wordt proc A gestart. De co-routine-aanroep is resume (proces). De aanroep detach geeft het einde van de initialisering in elk proces aan (welke plaats vindt tijdens de creatie).

Dit is een aanzienlijke verbetering ten opzichte van het eerste geval: de processen behouden hun eigen ongeschonden vorm, we hoeven geen labels toe te voegen en de initialisatie is min of meer duidelijk. Echter, nog afgezien van het door de auteurs van Simula 67 [8] al aangegeven bezwaar, namelijk dat het eigenlijk niet nodig zou moeten zijn, voor de losse co-routine-toepassing het hele zware class-mechanisme van stal te halen, zijn de overige bezwaren gebleven en is er nog een bezwaar bij gekomen. Nog steeds wordt de informatie-overdracht via ch niet de plaats gegeven die hem toekomt en nog steeds kunnen we in proc A een resume (proc A) doen (run-time check?). Het bijgekomen bezwaar blijkt niet uit het voorbeeld maar wel uit het voorbeeld dat de auteurs op blz. 188-189 van [8] geven. Wanneer we namelijk drie processen A, B en C op bovenstaande wijze achter elkaar schakelen, dan roept A B aan om informatie kwijt te raken, door middel van resume (B), terwijl C B aanroept om informatie te verkrijgen, door middel van dezelfde opdracht resume (B). En aangezien uit de class-declaraties nog helemaal niet volgt, hoe de processen gebruikt worden, is het effect van de opdracht resume (B) statisch volledig duister.

In het licht van al deze moeilijkheden is het merkwaardig dat in Algol 68 het beoogde effect eenvoudig en inzichtelijk te bereiken is met de in andere opzichten rudimentaire middelen par en sema, zonder expliciet van co-routines gebruik te maken. Het programma luidt dan:

```
begin struct (sema write, ref char ch, sema read) interface =
                (level 1, loc char   , level 0);

    proc co write = (char ch) void:
    (down write of interface; ch of interface := ch; up read of interface);

    proc co read = (ref char ch) void:
    (down read of interface; ch := ch of interface; up write of interface);

    par begin # proc A: #
        do char ch; read (ch);
           if ch = "a"
           then read (ch);
                if ch = "a" then  co write ("b")
                else co write ("a");  co write (ch) fi
           else co write (ch)
           fi
        od,
        # proc B: #
        do char ch; co read (ch);
           if ch = "b"
           then co read (ch);
                if ch = "b" then write ("c")
                else write ("b"); write (ch) fi
           else write (ch)
           fi
        od
    end
end
```

In de eerste twee regels wordt een interface gedefinieerd, bestaande uit een ref char ch, met daarvoor een slagboom sema write en erachter een slagboom sema read. In het begin is de write-slagboom open, de read-slagboom dicht en de ref char bevat een niet ter zaken doende waarde.

De volgende regels definiëren twee routines, co write and co read. De routine co write doet de write-slagboom achter zich dicht (of wacht als hij al dicht is), schrijft het meegekregen character op de ref char en laat de read-slagboom achter zich open. Co read doet het omgekeerde en leegt het

interface.

De constructie par begin A, B end houdt in dat A en B zodanig collateraal uitgevoerd worden dat synchronisatie tussen beide alleen op semaforen berust.

Alle genoemde bezwaren zijn nu op één na verdwenen: de beveiligde informatie-overdracht heeft een centrale plaats en we hebben ondubbelzinnig gedefinieerde aanroepen voor die overdracht. Wel is het nog steeds mogelijk b.v. binnen proc A co read aan te roepen of buiten co read en co write aan het interface te prutsen; dit gebrek aan protectie is een euvel van alle programmeertalen. Hier staat tegenover dat het laatste programma uit structureringsoogpunt veel beter uitdrukt wat we willen dan de twee vorige: op het moment dat A B aanroept en een character overdraagt, is het helemaal niet noodzakelijk dat B er direct mee verder gaat, A wil het alleen maar kwijt. Pas wanneer de buffer volraakt móét B verder. In de coroutine-versie kán dit niet gespecificeerd worden; in de semaforen-versie gebeurt het vanzelf, en wordt op de juiste wijze in het midden gelaten of B of A verder gaat.

3.5. *Selectie*

Bij selectie wordt op grond van een bepaalde beslissingsprocedure uit een aantal mogelijke opdrachten-groepen één bepaalde gekozen. Deze beslissingsprocedure kan voor elke groep totaal anders zijn (toegangssleutels), of alle beslissingsprocedures kunnen gebaseerd zijn op een en dezelfde berekende waarde.

3.5.1. *Selectie door middel van toegangssleutels*

Deze vorm is in Algol 60, Simula 67, ALEPH en Algol 68 beschikbaar. De ALEPH-vorm is:

```
(key 1, actions 1;
 key 2, actions 2;
        actions 3),
```

de Algol 68 vorm is:

```
  if key 1 then actions 1
elif key 2 then actions 2
           else actions 3 fi
```

In ALEPH wordt geëist (en gecheckt) dat de toegangssleutels geen globale effecten hebben als ze falen; een mislukte test kan dus de volgende tests niet beïnvloeden.

3.5.2. *Selectie op grond van een waarde*

Vaak maken alle toegangssleutels gebruik van één en dezelfde waarde:

```
  if c[i] = 1 then actions 1
elif c[i] = 2 then actions 2
elif c[i] = 3 then actions 3
              else actions 4 fi.
```

Het ligt dan voor de hand deze waarde van te voren uit te rekenen en als parameter mee te geven:

```
case c[i] in
       actions 1,
       actions 2,
       actions 3,
out    actions 4
esac.
```

3.5.2.1. *Op grond van een boolean*

Dit is in zekere zin een triviaal geval van 3.5.1. Deze vorm is in Algol 60, Simula 67 (if bool then a else b) en Algol 68 (if bool then a else b fi) beschikbaar.

3.5.2.2. *Op grond van een integer*

Selectie op grond van het teken is beschikbaar in Fortran (IF(I) 100, 200, 300) en op grond van de waarde in het gebied 1 : n in Algol 68 (case i in i is 1, i is 2, i is 3, ... i is n esac), en op grond van het vallen van de waarde in aangeduide gebieden in ALEPH

```
(= i =
 [:-1], negative i;
 [0], zero i;
 [1:255], small i;
          rest positive i)
```

3.5.2.3. *Op grond van een character of een string*

Deze vormen van het flow-of-control die men zich gemakkelijk kan voorstellen, komen in geen enkele taal voor. Wel kan men in ALEPH, op grond van het feit dat de characters op de integer worden afgebeeld, schrijven:

```
(= ch =
 [/,/], comma;
 [/a/:/z/], letter;
 [/0/:/9/], digit).
```

3.5.2.4. *Op grond van het type (mode) van een gegeven*

Deze vorm is in de enige taal waarin men het type van een variabele enigszins kan manipuleren, Algol 68, beschikbaar. De variabele is hier gedeclareerd van de mode union van een aantal modes. Deze union-mode bestaat alleen tijdens compilatie, tijdens de uitvoering heeft de variabele een waarde van één van de in de union toegestane modes. Door een bepaalde vorm van de case-clause kan de opdrachtengroep voor deze mode geselecteerd worden en tegelijkertijd de waarde syntactisch van zijn mode union van andere modes omgezet worden in zijn werkelijke mode. Als uir een variabele van de mode union (int, real) is, dan kan door

```
case uir in
     (int i) : action on integer (i),
     (real r) : action on real (r)
esac
```

tijdens de uitvoering beslist worden of de huidige waarde van uir int of real is; als het int is, wordt de waarde aan int i toegekend en action on integer (i) uitgevoerd, als het real is, wordt de waarde aan real r toegekend en action on real (r) uitgevoerd.

Bij deze vorm van case-clause is de volgorde waarin de te selecteren

opdrachtgroepen gegeven worden, niet van belang; ze worden door een label-achtige constructie geïdentificeerd.

3.5.2.5. *Op grond van andere waarden*

In zijn algemeenheid hoort bij elke voor de taal primitieve soort gegevens een case-clause. Zo zal een taal die sets als primitief datatype heeft, een case- clause voor selectie over het aanwezig zijn in een set moeten hebben. Op die grond zouden we kunnen zeggen dat in Algol 68 een case-clause over de identiteit van variabelen ontbreekt; het nuttig effect van zo'n constructie zou echter gering zijn, b.v. voor het afzoeken van een lineaire lijst:

```
case handle # ref list # in
     (nil): not found
out if item = item of handle then found
    else handle := next of handle fi
esac.
```

3.6. *Herhaling*

In ALEPH worden alle herhalingen uitgedrukt door recursieve (meestal rechts-recursieve) procedure-aanroepen, en als zodanig bestaat er dus geen speciaal herhalingsmechanisme. In de andere talen ligt dit anders, en wel als volgt.

3.6.1. *Herhaling zolang een gegeven conditie vervuld is*

Mogelijkheden hiertoe zijn in Algol 60, Simula 67 en Algol 68 aanwezig in de vorm van een while- statement. De Algol 68-vorm is superieur omdat hierin in het while-gedeelte een zeer wijde klasse van opdrachten (serial clauses) toegestaan is, in tegenstelling tot Algol 60 en Simula 67 waar alleen een expressie toegelaten wordt. Deze mogelijkheid is van belang voor structurering.

Soms wordt bij de ontwikkeling van een taal een until-statement overwogen, die achteraf test of er herhaald moet worden. Voorstellen voor een until-statement komen mijns inziens voort uit de moeilijk uitroeibare mening dat een lege set geen set is.

Merkwaardig veel interessanter wordt de toestand wanneer het probleem van nature om zowel een while- als een until-clause vraagt. Stel we willen een naam in een kaartenbak met namen opzoeken. Dan lijkt de volgende hypothetische constructie wel goed:

```
while er zijn nog namen in de bak
do neem de volgende naam
until het is de naam die we zoeken.
```

Het vervelende is echter dat we achteraf willen weten of de herhaling gestopt is omdat we de naam gevonden hebben (until) of omdat de namen op waren (not while), en dat zonder herberekening. We komen in de verleiding de constructie uit te breiden met

```
ifnotwhile voeg naam toe; lever adres
ifuntil lever adres,
```

maar nu groeit de constructie ons over het hoofd. De enige bevredigende oplossing die ik ken is de rechts-recursieve, in Algol 68

```
proc search = (ref name name) ref name :
if nil (name) then add name
elif name sought = name of name then name
else search (next of name) fi,
```

zoals deze ook in ALEPH gebruikt wordt (zie ook WEGNER [14]).

3.6.2. *Herhaling over een eindige set in gespecificeerde volgorde*

Deze vorm is rudimentair in Fortran aanwezig (DO 20 I,100), waarbij over één equidistante set van integers herhaald kan worden. De ruimste vorm is in Algol 60 en Simula 67 beschikbaar waar er herhaald kan worden over meerdere sets van equidistante integers of reals, naar wens afgewisseld met while-condities. In Algol 68 is alleen herhaling over één equidistante set van integers mogelijk, naar wens gecombineerd met een while-conditie. De Algol 60 statement

```
for i := 1, 9, 6, 8 do something (i)
```

is slechts moeizaam in Algol 68 om te zetten; door de beschikbaarheid van

geïnitialiseerde array's is de constructie echter vaak overbodig.

3.6.3. *Herhaling over een eindige set in ongespecificeerde volgorde*

Deze vorm wordt door geen enkele programmeertaal geleverd. Echter, om een array op nul te zetten doet de volgorde waarin dat gebeurt helemaal niet ter zake. Toch dwingen alle programmeertalen de gebruiker deze volgorde te specificeren:

```
for i to upb array do array [i] := 0 od.
```

4. SAMENVATTING

Wat flow-of-control betreft is Algol 68 superieur aan alle andere talen, ondanks een aantal feilen en tekortkomingen die in het bovenstaande gesignaleerd zijn, en het feit dat er naast een aantal fraaie constructies ook een aantal lelijke constructies mogelijk zijn.

Verder verdient ALEPH vermelding, dat met uiterst eenvoudige middelen een goed gestructureerde flow-of-control bereikt.

LITERATUUR

[1] GEURTS, L., *Cursus Programmeren, Deel 1 en 2*, MC Syllabus 16.1 en 16.2, Mathematisch Centrum, Amsterdam, 1973.

[2] *IBM OS/VS JCL Reference. GC 28 - 0618.*

[3] *Control Data Scope Reference Manual* 60307200.

[4] *Control Data Fortran Extended Reference Manual* 60305600.

[5] BACKUS, J.W., et al., *Revised Report on the Algorithmic Language Algol 60*, Num. Math., 4 (1963), 420-453, Springer-Verlag, 1963.

[6] *IBM System 360 PL/I Language Specification.* GY 33-6003.

[7] MEELHUIJSEN, W.; *PL/1 Gids, deel 1 en 2*, Rekencentrum T.H. Delft, Delft, 1971.

[8] DAHL, O-J., et al., *Structured Programming, p. 175-220*, APIC Studies in Data Processing 8, Academic Press, London, 1972.

[9] *Control Data Simula Reference Manual* 60251900.

[10] GRUNE, D., et al., *ALEPH Manual,* Mathematical Centre Report IW 17/74, Amsterdam, 1974.

[11] VAN WIJNGAARDEN, A., ed., *Revised Report on the Algorithmic Language ALGOL 68,* Technical Report TR 74-3, University of Alberta, Edmonton, Alberta, 1974.

[12] VAN DE RIET, R.P., *Over goto's en programmacorrectheid,* Syllabus van het colloquium Programmacorrectheid, 2, Mathematisch Centrum, Amsterdam, 1973.

[13] WULF, W.A., *A case against the GOTO,* SIGPLAN Notices 7 (11) 63-69, 1972.

[14] WEGNER, E., *Tree-Structured Programs,* Bericht Nr. 73-11, Technische Universität Berlin, Berlin, 1973.

[15] LOMET, D.B., *Control Structures and the Return Statement,* Proceedings of IFIP Congress 1974, p. 403-407, North Holland Publ. Company, Amsterdam, 1974.

[16] KRIEG, B., *A Class of Recursive Coroutines,* Proceedings of IFIP Congress 1974, p. 408-412, North Holland Publ. Company, Amsterdam, 1974.

MATHEMATICAL CENTRE SYLLABUS 25, 1976, II, 23-40

GRAFISCHE PROGRAMMEERTALEN

P.J.W. TEN HAGEN

1. INLEIDING

Ondanks het bestaan van zgn. *general purpose programmeertalen*, bestaat er een indrukwekkend aantal talen die speciaal ontworpen zijn voor het vastleggen van informatie in de vorm van een tekening. Kennelijk zijn algemene programmeertalen niet algemeen genoeg om het manipuleren van tekeningen adekwaat te beschrijven.

Grafische programmeertalen zijn een onderdeel van een zgn. interactief grafisch systeem. Dit is een bedrijfssysteem dat het mogelijk maakt het resultaat van een berekening in de vorm van een tekening aan de gebruiker te presenteren en waarbij de gebruiker bovendien tijdens uitvoering van zijn programma kan ingrijpen om nieuwe gegevens, eventueel in de vorm van een tekening, in te voeren.

Dat een grafisch systeem essentieel interactief moet zijn heeft twee belangrijke redenen:

1. Het prepareren van de juiste invoer gebeurt, door deze als tekening te beoordelen en daarop zo nodig aan het beeldscherm wijzigingen aan te brengen (picture editing), voordat ze aan het programma wordt overgedragen.
2. De tekenmachine waarmee een in te voeren tekening moet worden gemaakt werkt interactief. De basisfuncties waarmee getekend wordt, maken gebruik van de wisselwerking (terugkoppeling) tussen gebruiker en beeldscherm (b.v. lichtpen, functietoetsencombinaties + refreshmode).

Een grafisch systeem moet behalve tot het genereren en inlezen van een tekening ook in staat zijn een gegeven tekening te transformeren (b.v. bij 3-dim. toepassingen). Daarvoor is het bij de huidige stand van zaken nodig

de tekening te representeren als datastructuur, immers het transformeren van instructies is onmogelijk of wordt sterk afgeraden.

In de datastructuur moet ook informatie kunnen worden opgeslagen die met de tekening geassocieerd is (b.v. de electronische eigenschappen van een getekend circuit). Er moet dus ruimte gemaakt kunnen worden voor verwijzingen tussen de tekening en de bijbehorende interpretatie.

Resumerend kunnen we zeggen dat aan grafische talen vier belangrijke aspecten te onderscheiden zijn:
- besturing van tekenmachines;
- control-statements voor interactie;
- datastructuren voor tekeningen;
- associaties met andere data.

In het volgende zullen we aan de eerste drie aspecten aandacht schenken. De opbouw van associaties met andere data is eigenlijk geen specifiek probleem voor grafische talen. De keuze van de manier waarop deze associaties in de datastructuur worden opgenomen vindt meestal plaats bij het implementeren van de toepassingen.

De bespreking vindt plaats aan de hand van een model voor een grafisch systeem. Hoewel in de literatuur tal van afwijkende modellen worden aangetroffen, geeft een overzicht aan de hand van één model in ieder geval de gelegenheid de aspecten zowel afzonderlijk als in hun onderlinge samenhang te bespreken. Daarnaast blijft bij interessante details vergelijking met andere oplossingen mogelijk. Dit model is opgesteld door een werkgroep op het MC bestaande uit P. Klint, H. Noot, T. Hagen en P. ten Hagen, en het wordt gebruikt om de problemen die bij de opzet van een grafisch systeem opduiken te kunnen omschrijven.

2. TEKENMACHINES

Iedere tekenmachine wordt bestuurd d.m.v. een aantal commando's uit een vast repertoire. Een tekening voor zo'n machine is een programma bestaande uit een rij commando's uit dat repertoire, aangevuld met de nodige numerieke gegevens (b.v. coördinaten).

Een tekening die door een grafisch programma geproduceerd wordt, moet

op de een of andere manier geconverteerd worden naar een programma voor de tekenmachine. Dit kan op triviale manier gebeuren door het commando-repertoire van de tekenmachine op te nemen in de grafische taal als primitieve acties. Een eenvoudiger oplossing wordt bereikt, door de tekening te genereren voor een virtuele machine. Voor iedere tekenmachine is er dan tevens een correspondentie met de virtuele machine gedefinieerd.

Een grafisch systeem dat ingebed is in een general purpose operating system, zal mogelijk meerdere grafische talen bevatten, b.v. in het geval dat deze zijn gedefinieerd en geïmplementeerd als extensies van de (hogere) programmeertalen die onder het operating system gebruikt kunnen worden. Voor elk van deze talen moet dan een afbeelding bestaan naar de virtuele machine. Op deze wijze ontstaat de situatie zoals in fig.1 is geschetst.

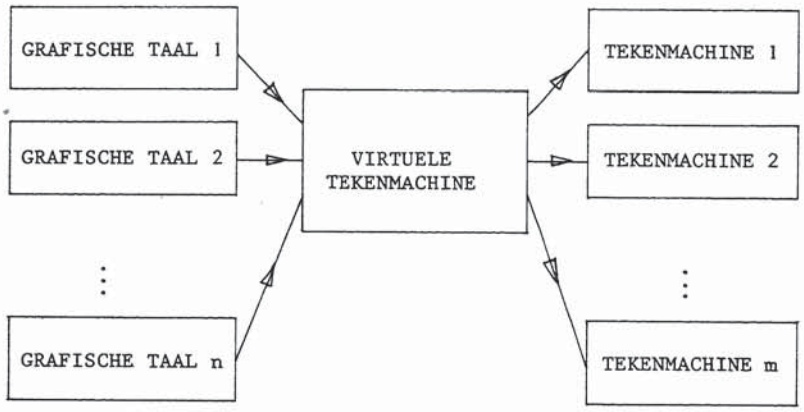


fig. 1

Het voordeel van de invoering van een virtuele machine is tweeledig:
- het minimaliseert het aantal conversies n.l. één per tekenmachine en één per grafische taal;
- het minimaliseert het aantal primitieve acties in de grafische taal.

We zullen nu dat deel van de virtuele machine gaan behandelen dat zorgt voor het genereren van lijntekeningen. Daarmee hebben we dan meteen de grootste klasse tekenmachines binnen ons bereik.

Wat niet in het model is opgenomen, is de apparatuur waarmee zgn. gray-scale-tekeningen worden gemaakt. We zullen deze apparatuur verder buiten beschouwing laten. De opzet van een grafische taal kan voldoende worden toegelicht aan de hand van het model voor lijntekenmachines.

Het instructie-repertoire van de virtuele machine is opgebouwd uit de volgende groepen instructies:

```
- acties:          initialise          ; Neem een leeg vel papier en zet
                                       ; de pen in de oorsprong 0.
                   move [to] (x,y)     ; Positioneer de pen op de plaats(x,y).
                                       ; De vector (x,y) te nemen vanaf 0
                                       ; (= absoluut) of vanaf de huidige penpo-
                                          sitie (to ontbreekt = relatief).
                   line [to] (x,y)     ; Teken een lijn naar (x,y)
                                       ; (relatief of absoluut).
                   point [to](x,y)     ; Teken een punt.
                   text (string)       ; Schrijf de tekst "string" beginnend bij
                                       ; de penpositie.
- voorvoegsels:    dot                 ; Voorvoegsels vóór een line-instructie
                   dash                ; die de lijnsoort bepalen.
                   red                 ;
                   green               ;
                   detect              ; maak de tekening "zichtbaar" resp.
                   undetect            ; "onzichtbaar" voor de licht-(gevoelige)
                                       ;  pen.
- control statements:
                   call (label)        ;  subroutine sprong.
                   return              ; exit subroutine.
```

jump(label)	; goto, alleen gebruikt in bijzondere situaties.
"identifier"	; label, voor een instructie, die tevens subroutine entry is.

- transformaties:

at (x,y)	; translatie over (x,y)
scale (x,y)	; vergroting (en spiegeling!)
rot (a)	; rotatie (over a radialen).
trans (A)	; algemene lineaire transformatie met matrix A.

Met behulp van de subroutines kunnen subtekeningen worden gedefinieerd. De subroutines mogen genest voorkomen, maar vanwege het ontbreken van conditionele statements niet recursief.

Transformaties kunnen worden toegevoegd aan acties b.v. *line (x,y) at (p,q)*, of aan subroutine-call's b.v. *call (pict 1) rot (b) scale (sx,sy)*. Een transformatie wordt gedefinieerd als een 3 * 3-matrixvermenigvuldiging met de vector (x,y,1), waarbij (x,y) het te transformeren coördinaten-paar is. Zo kunnen alle toegestane transformaties op uniforme wijze worden onthouden:

(x,y) at (p,q):
$$(x,y,1).\begin{pmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ p & q & 1\end{pmatrix} = \begin{pmatrix}x+p\\ y+p\\ 1\end{pmatrix}$$

(x,y) rot (a) :
$$(x,y,1).\begin{pmatrix}\cos a & -\sin a & 0\\ \sin a & \cos a & 0\\ 0 & 0 & 1\end{pmatrix}$$

(x,g) scale (sx,sy):
$$(x,y,1).\begin{pmatrix}sx & 0 & 0\\ 0 & sy & 0\\ 0 & 0 & 1\end{pmatrix}.$$

Merk op dat de matrix-vermenigvuldiging niet commutatief is.

Voorbeeld van een Virtuele-Machine-programma:

```
Square: line (1,0); line (1,1); line (0,1); line (0,0);
        return;
        call square;
        call square at (.5,.5) scale (2.1); move to (0,0);
        call square scale (2,1) at (.5,.5);
        stop
```

Dit programma tekent 3 rechthoeken (fig. 2):

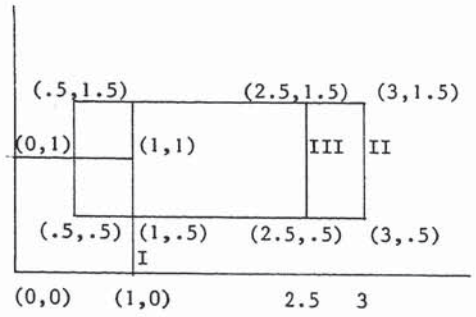


fig. 2

Bij rechthoek II wordt de x-translatie mee vergroot. Alle rechthoeken zijn gepositioneerd t.o.v. de oorsprong. Daarom wordt na rechthoek II de *move to* (0,0) ingelast.

Een transformatie toegepast op een actie verandert alleen de actie. Een transformatie toegepast op een subroutine wordt toegepast op alle elementen van die subroutine. Dat betekent met name bij geneste subroutines, dat er een zgn. transformatie-stapel moet worden bijgehouden, opdat bij terugkeer uit een subroutine, de oorspronkelijke transformatie ingesteld kan worden.

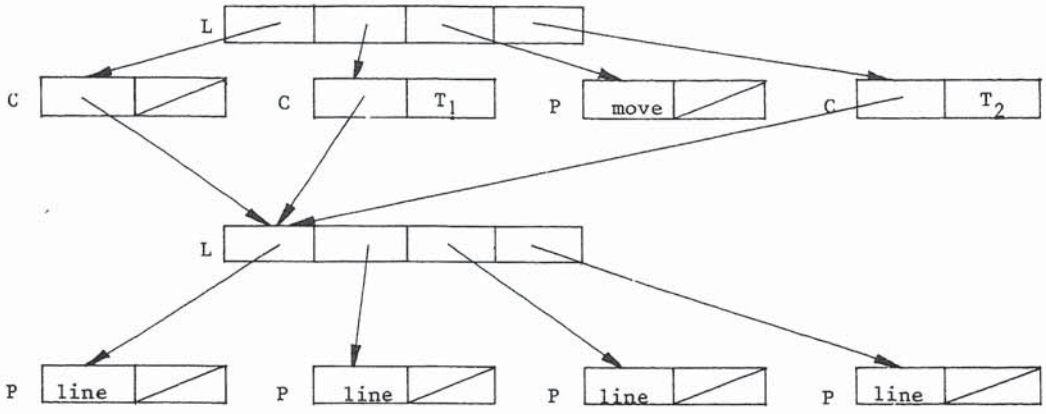
Concluderend kunnen we zeggen dat een tekening voor de virtuele machine een boomstructuur heeft. Er zijn 3 typen knopen in de boom:

L - knopen, bestaande uit een lijst verwijzingen naar knopen van het type P of C.

P - knopen, bestaande uit een primitieve actie en een transformatie (eindknoop).

C - knopen, bestaande uit verwijzing naar een L-knoop en een transformatie.

Het bovenstaande programmaatje heeft b.v. de volgende boomstructuur:



N.B.:  staat voor de eenheidstransformatie

fig. 3

We zullen nu trachten deze eenvoudige notie van een tekenmachine in te bedden in een grafische programmeertaal.

3. DATASTRUCTUREN VOOR TEKENINGEN

Er bestaan voor de opzet van een grafische taal twee principieel verschillende methoden. De eerste methode hanteert het begrip *display procedure* (zie NEWMAN [1]). Hierbij wordt een tekening uitsluitend beschreven met behulp van een stel procedures eventueel voorzien van parameters.

De meest elementaire acties binnen de procedures zijn de acties en transformaties uit onze virtuele machine. Iedere actie wordt, na transformatie , rechtstreeks omgezet in instructies voor de desbetreffende tekenmachine, d.m.v. uitvoerroutines. Dit heeft konsekwenties voor de interactie Voor zover die dan nog mogelijk is. Een dergelijke opzet is alleen mogelijk m.b.v. programmeertalen waarbij variabelen van het type statement en/of procedure voorkomen zoals in EULER (zie WIRTH & WEBER [2]). Een procedure kan dan gewijzigd worden door aan een statement-variabele een nieuw statement te assigneren. We zullen op de konsekwenties voor de interactie later

terugkomen.

Bij de tweede methode wordt er gesproken over het zgn. *data-structuren-probleem* voor grafische talen. Dit vindt zijn oorzaak in het feit, dat met een tekening de volgende manipulaties mogelijk moeten zijn:
De gebruiker wijst een element aan uit de tekening, m.b.v. lichtpen of door de positie op te geven, en specificeert vervolgens wat er met dit element moet gebeuren: wijzigingen, vervangen, uitwissen etc.

Het programma moet in het algemeen hierop een tweeledige reactie geven:

1. zoek de kleinste subboom die dat element bevat en voer de gevraagde actie uit;
2. herstel de onderlinge relaties tussen de delen van de tekening voor zover die door deze actie verstoord zijn.

De eerste van de twee reacties moet vanwege de on-line-faciliteiten snel uitgevoerd worden. De tweede reactie kan en mag aanleiding geven tot een langdurig rekenproces, dat mogelijk zelfs niet slaagt.

Snel localiseren van een element in de boom is alleen mogelijk wanneer er bij iedere knoop informatie opgeslagen is, die vertelt of verder zoeken in de subboom kans op succes heeft.

Voor het herstellen van de onderlinge relaties moet over de boom een extra netwerk van pointers gelegd zijn (waarin cykels mogen voorkomen), om gerelateerde elementen te kunnen vinden. Het is duidelijk dat voor de meeste programmeertalen in een dergelijke datastructuur niet kan worden voorzien.

In de literatuur kunnen we tal van voorstellen voor datastructuren aantreffen waarmee dit probleem kan worden opgelost. Voor een overzicht hierover verwijzen we naar WILLIAMS [3].

Deze voorstellen zijn ruwweg in te delen in twee groepen:

1. Maak een algemene datastructuur voor associatieve gegevens en beschouw tekeningen als een van de soorten informatie, die daarmee kunnen worden opgeborgen;
2. maak een datastructuur uitsluitend voor tekeningen en ontwerp per toepassing een meer uitgebreide datastructuur.

De laatste methode schuift de moeilijkheden af naar de gebruiker. Bij de eerste methode zien we meestal een datastructuur die relatief duur is, omdat voor het meeste tekenwerk slechts zeer beperkt gebruik wordt gemaakt van alle voorzieningen.

We zullen nu een voorstel voor een grafische taal behandelen, waarin de virtuele machine terug te vinden is als een stel eenvoudige handelingen, en waarbij bovendien aandacht is geschonken aan beide bovengenoemde problemen.

4. GRAFISCHE TAAL IN ALGOL 68

4.1. *De primitieve laag*

De grafische taal is het tweede belangrijke element uit het model. Van tevoren zijn aan de taal een aantal eisen opgelegd, die samenhangen met het te verwachten gebruik van het grafisch systeem enerzijds en met een algemene filosofie over de opbouw van een taal anderzijds.
Een programmeertaal dient eenvoudig gereedschap te bevatten voor gebruikers, die slechts een project van bescheiden omvang willen realiseren. Dit geldt in het bijzonder voor tekenprogramma's. De eerste eis luidt daarom: de grafische taal moet een complete en tegelijk eenvoudige subset bevatten voor het kleine tekenwerk. Compleet wil zeggen dat de virtuele machine volledig bestuurbaar moet zijn. Eenvoudig wil zeggen dat het aantal elementen uit de subtaal klein moet zijn, gemakkelijk te leren en te hanteren, en bovendien geen extra overhead moet geven in geheugenruimte of accestijden in verband met mogelijk ingewikkelder toepassingen.

De subtaal zullen we de *primitieve laag* noemen en hieronder gedeeltelijk beschrijven.
De primitieve laag bestaat uit een aantal primitieve modes (types) en een aantal operatoren, die gedefinieerd zijn voor deze primitieve modes. Met behulp van de operatoren worden nieuwe primitieve elementen van de bestaande modes opgebouwd. M.a.w. geen enkele combinatie van primitieve modes en primitieve operatoren voert ons buiten de primitieve laag.

Primitieve modes:

```
mode prim  = union (point, vector, line);
mode point = struct (real x,y);
mode vector = struct (real u,v);
mode line  = struct (point bp, vector ep);
```

zowel *point* als *vector* bevatten een coordinaten-paar.

Evenwel met het paar (x,y) wordt absoluut gerekend (= t.o.v. de oorsprong) en met het paar (u,v) relatief t.o.v. de huidige penpositie. Een toepassing hiervan zien we bij de definitie van *line:* Het beginpunt *bp* is t.o.v. de oorsprong gegeven, het eindpunt *ep* is t.o.v. het beginpunt gegeven. Dit sluit aan bij de veel gebruikte vector-voorstelling voor een lijn: $l = \vec{bp} + \lambda.\vec{ep}$. We laten voorlopig in het midden op welke wijze een lijn of vector getekend moet worden. D.w.z. we hebben op dit moment nog de vrijheid om te beslissen of met *line l = (bp,ep)* bedoeld wordt het lijnstuk tussen *bp* en *ep* dan wel de gehele lijn etc. In de primitieve laag bevinden zich ook de modes *real*, *int*, *char* en *bool*.
De meest ingewikkelde constructie uit de primitieve laag is van de mode [ ] *prim*, [ ] *real*, [ ] *int* of *string* (*d.w.z.* [ ] *char*).

Primitieve operatoren:

- operaties met scalairen:

```
op * = (real s, vector w) vector: (s*u of w, s*v of w);
```

lengte van een *vector:*

```
op mod = (vector w) real:
         (real u=u of w, v=v of w, sqrt(u*u + v*v));
```

inproduct van twee vectoren:

```
op inp = (vector w1, w2) real:
         (u of w1 * u of w2 + v of w1 * v of w2);
```

de determinant van de vierkante matrix gevormd door twee vectorkolommen:

```
op det = (vector w1, w2) real:
         (u of w1 * v of w2 - v of w1 * u of w2);
```

de normering van een vector op lengte 1:

```
op norm = (vector w) vector:
          (real n:= mod w ;(u of w/n, v of w/n));
```

monadische - voor vectoren:

```
op - = (vector w) vector: (-u of w, -v of w);
```

- operaties tussen primitieven.

positionering: rt (relative to):

```
op rt = (point p, vector w) point:
        (x of p + u of w, y of p + v of w);
op rt = (vector w1, w2) vector:
        (u of w1 + u of w2, v of w1 + v of w2);
op rt = (line l, vector w) line:
        (bp of l rt w, ep of l);
```

verschilvector tussen twee punten:

```
op dv = (point p,q) vector:
        (x of p - x of q, y of p - y of q);
```

merk op dat de conversie *point* → *vector* gedefinieerd kan worden als: *vector v : = p dv(0,0);* de conversie *vector* → *point* als: *p: =(point(0,0)) rt v;*

- operaties voor geometrische constructies:

verbindingslijn tussen twee punten:

```
op connect = (point p,q) line: (p,q dv p);
```

snijpunt van twee lijnen *cp* (cutting point):

hierbij doet zich het pathologische geval voor van evenwijdige of samenvallende lijnen. In dat geval wordt het unieke punt *omega* afgeleverd. Dit punt valt buiten het tekenvlak van de virtuele tekenmachine. Tekenen van het punt *omega* leidt tot een foutmelding. De gebruiker kan evenwel van ieder punt testen of het gelijk is aan *omega*.

```
op cp = (line l1,l2) point:
        (vector ep1 = ep of l1, ep2 = ep of l2,
                v12:= bp of l1 dv bp of l2;
         real d:= ep1 det ep2;
         (d=0)|omega|bp of l2 rt
                    ((v12 det perp ep1)/d*ep2)
        );
```

vector loodrecht op een gegeven vector: *perp* (perpendicular); deze operator werd al toegepast bij de definitie van cp.

```
op perp = (vector u) vector: (v of w,-u of w);
```

een loodlijn vanuit een punt (beginpunt) op een lijn (eindpunt): Indien het punt of de lijn ligt, wordt het eindpunt bepaald door een loodvector met lengte gelijk aan *ep of l*.

```
op perp = (point p, line l) line:
          (p,(point q:=l op (p,perp ep of l);
             (p=q|perp ep of l|q dv p)
          );
```

- operaties voor condities:
  test of een punt op een lijn ligt:

```
op on = (point p, line l) bool:
        (p dv bp of l) det ep of l = 0;
```

Ter illustratie volgt hieronder een programaatje geschreven in de subtaal. Het programma leest 3 coördinaten-paren in, en beschouwt ze als de hoekpunten van een driehoek. Van deze driehoek worden vervolgens het hoogtepunt H, zwaartepunt Z en het deelpunt D berekend.

```
(point A,B,C;
 input point (A); input point (B); input point (C);
 line a := B connect C, b := C connect A, c := A connect B;
 point H := (A perp a) op (B perp b);
 point Z := (A connect (B rt .5*ep of a) op
            (B connect (C rt .5*ep of b);
 point D := (A perp((A rt norm ep of b) connect
            (A rt norm ep of c))
           op
            (B perp ((B rt norm ep of a) connect
                    (B rt norm ep of c))
);
```

4.2. *Samengestelde tekeningen*

Samengestelde tekeningen worden gevormd door eenvoudiger subtekeningen aan elkaar te koppelen. De meest triviale koppeling is de zgn. vereniging van een aantal tekeningen. De betekenis hiervan luidt: teken alle subtekeningen uit de verzameling op één papier t.o.v. dezelfde oorsprong. De verenigingsoperatie wordt uitgedrukt met behulp van de [ ] - notatie (row of ...).

```
mode pictset = [ ] pict;
mode pict    = union (prim, subpict);
```

De vereniging van een aantal primitieven (i.e.[ ]*prim*) is een bijzonder geval van de mode *pictset*. Een *subpict* is een samengestelde tekening. Behalve de verwijzing naar de onderdelen bevat een *subpict* een verwijzing naar een verzameling eigenschappen van de *subpict* zelf. Dit is juist het verschil tussen een *subpict* en een *prim*. Van een *prim* worden geen bijzondere eigenschappen bijgehouden (dus geen overhead). De bedoeling is dat deze eigenschappen telkens op de plaats waar ze nodig zijn worden uitgerekend.

```
mode subpict = struct (ref pictset ps,ref box b)
mode box     = struct (point l,u,s,f,bool abs);
```

Alle eigenschappen van samengestelde tekeningen, die door het grafische systeem automatisch worden bijgehouden zijn geplaatst in een doos, de zgn. box. Het box-concept is het onderwerp van de volgende paragraaf.
Merk op dat de drie typen knopen uit de datastructuur-representatie van de virtuele tekenmachine (zie 2-3) zijn terug te vinden in de 3 modes:

P-knoop ~ *prim*
S-knoop ~ *subpict*
L-knoop ~ *pictset*.

4.3. *De box-operator*

Een overgang van de primitieve laag naar de samengestelde laag wordt bewerkstelligd door aan een verzameling primitieven zijn *box* toe te voegen. Deze conversie wordt uitgevoerd door de monadische box-operator (notatie:*bx*).
In omgekeerde richting kan een samengestelde tekening worden ontbonden tot een [ ] *prim*. Dit gebeurt met de omgekeerde box-operator (notatie:*xb*). Hierbij gaat informatie over de subtekening verloren of raakt ongedefinieerd (b.v. de zgn. beammode). Interessant is de combinatie *bx(xb p)* waarbij *p* een subtekening is. Het effect hiervan is dat de boomstructuur van *p* wordt gereduceerd tot één L-knoop, nl. [ ] *prim*, die weer van een box wordt voorzien. De *bx*- en *xp*- operatoren zijn de enige verbindingsdraden tussen de primitieve en de samengestelde laag.

We zullen, bij wijze van voorbeeld, een box behandelen waarin een gedeelte van de eigenschappen vermeld staan die we willen bijhouden.

De naam box is geinspireerd door een van de belangrijkste eigenschappen die een tekening kan hebben, n.l. zijn positie in het vlak van tekening. In het volgende zullen we de uitdrukking box soms gebruiken voor alle eigenschappen, die worden bijgehouden en soms om alleen de gegevens over de positionering aan te duiden. Welke van de twee bedoeld wordt volgt uit de context, of wordt zo nodig expliciet vermeld.

Een *(positionerings-)box* is de kleinste rechthoek die om de tekening past. De zijden van de rechthoek lopen evenwijdig aan één van beide assen. De box bevat altijd de oorsprong O van de tekening (zie fig. 4-1).

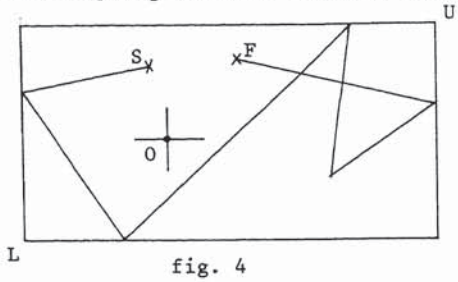


fig. 4

Alle delen van de tekening zijn t.o.v. O gepositioneerd. De vorm van de box is vastgelegd m.b.v. de punten L en U (lower en upper). Verder is S het startpunt van de tekening en F het eindpunt (finish). M.a.w. F valt samen met de penpositie als de tekening af is. Het tekenen moet beginnen met de instructie *move to (S)*.

De *bool abs* uit de box vertelt ons hoe de subtekening in het geheel gepast moet worden:

abs = true : laat O samenvallen met de heersende oorsprong.

abs = false: laat O samenvallen met de penpositie. Dit is een generalisatie van de begrippen absoluut en relatief die we al zijn tegengekomen bij de primitieven *point* en *vector*.

We zullen nu de box-operator definiëren. We maken daarbij gebruik van de operatoren *llc* (lower left corner) en *urc* (upper right corner). Deze primitieve operatoren bepalen de hoekpunten L resp. U van een box om twee punten.

```
op llc = (point p,q) point:
(real px = x of p, py = y of p, qx = x of q, qy = y of q;
((px < py|px|py),(qx < qy|qx|qy))
);
op urc = (point p,q) point:
(real px = x of p, py = y of p, qx = x of q, qy = y of q;
 ((px > py|px|py),(qx > qy|qx|qy))
);
op bx = (pictset ps) subpict:
 (point u: = (0,0), l: = (0,0), s,f;
  for i to upb ps do
  case ps [i] in
  (point point):    if i = 1 then s := point fi; f := point;
                    l := f llc l; u := f urc u,
  (vector vector): if i = 1 then s := (0,0) fi; f := f rt vector;
                    l := f llc l; u := f urc u,
  (line line):      point p := bp of line; if i = 1 then s := p fi;
                    f := p rt ep of line; l := p llc f llc l;
                    u := p urc f urc u,
  (subpict spict): ref box bb = b of spict;
                    if bb = ref box (nil) then
                    ps [i] := spict := bx (ps of spict) fi;
                    if i = 1 then s := s of bb; f := f of bb;
                      u := u of bb; l := l of bb
                    elif abs of bb then f := f of bb;
                      l := l llc l of bb; u := u urc u of bb
                    else vector v := f dv (0,0); f := f of bb rt v;
                      l := l llc (l of bb rt v);
                      u := u urc (u of bb rt v)
                    fi
  esac
  od; (pictset (ps), heap box ((l,u,s,f,true)))
  );
```

De box-operator-definitie impliceert een definitie van de *vereniging* van een aantal tekeningen ,

$$T = \bigcup_{i=1}^{n} T_i \text{ als volgt:}$$

zij B de box van T, bevattende O,S,F,L,U en abs.
zij B_i de box van T_i bevattende O_i, S_i, F_i, L_i, U_i en abs_i (i=1,...,n).
Dan geldt: $S = S_1$;
Verder geldt: $O_1 = 0$, O_{i+1} = *if* abs_{i+1} *then* 0 *else* F_i (i=1,...,n-1).
Uit deze gegevens kunnen L_{i+1}, U_{i+1} en F_{i+1} berekend worden (i=1,...,n-1).
Verder geldt: $F = F_n$, $L = L_n$ en $U = U_n$.
In de algoritme voor de *box*-operator geldt voor de j - de slag van de *for* - statement dat *s, f, en u* de *box* van $\bigcup_{i=1} T_i$ bepalen (j=1,...,n).
abs wordt geïnitialiseerd op *true*.

4.4. *Toepassingen van de box*

Het meest in het oog springende voordeel van een box is het gemak waarmee nieuwe koppelings-operatoren ingevoerd kunnen worden. Daarbij kan de onderlinge positie van de deeltekeningen aan de hand van hun box-gegevens vastgelegd worden. Dit gebeurt door:

1. de *bool* abs uit de deelboxen te specificeren en
2. vast te leggen hoe de omvattende box moet worden samengesteld uit de deelboxen.

Een tweede toepassing is de mogelijkheid het ontstaan van circulaire datastructuren tijdens koppelings-acties te detecteren en eventueel te verhinderen, b.v. in

ps [i] of subpict a: = subpicta.

Vergelijking van de boxen links en rechts vlak voor de assignment kan vertellen of er kans op circulariteit bestaat. In dat geval moet er gezocht worden naar een sub-box die precies past. Op deze wijze kan op redelijk efficiënte wijze worden vastgesteld of er een copie van een referentie gemaakt moet worden, die de cirkel doorbreekt.

In het algemeen kunnen we zeggen dat de informatie uit de box gebruikt wordt om zoekprocessen in de subtekening overbodig te maken. Beide bovengenoemde toepassingen zijn daar een voorbeeld van. In de volgende paragraaf zullen we een derde toepassing van het box-concept tegenkomen, die op hetzelfde principe berust, bij het realiseren van interactie-mogelijkheden.

5. INTERACTIEF TEKENEN

Om interactief te werken met tekeningen moeten we op z'n minst tekeningen kunnen invoeren. In ons model kunnen we het invoeren van tekeningen beschrijven met behulp van de virtuele tekenmachine. Een grafisch programma dat een tekening inleest, ontvangt een programma voor de virtuele machine.

Door dit uitgangspunt wordt op programma-niveau de invoer-uitvoer-situatie symmetrisch. Wat we daarbij nodig hebben is een afbeelding van de virtuele machine naar de grafische taal. Volledige symmetrie wil dus zeggen, dat de afbeelding van grafische taal naar virtuele tekenmachine omkeerbaar is. Het is nu ook duidelijk wat er moet gebeuren met de ruwe invoer, die geproduceerd wordt door de tekenmachines uit het systeem. Iedere tekenmachine wordt op het laagste niveau bestuurd door een programma, dat de invoer vertaalt naar instructies voor de virtuele machine.

In het algemeen kan men over invoer-verwerkende programma's opmerken, dat naarmate de invoer omvangrijker en ingewikkelder wordt, een strengere controle op de correctheid van de invoer nodig is. Het is duidelijk dat aan de omvang en de complexiteit van tekeningen moeilijk van te voren grenzen gesteld kunnen worden. Dit geldt zeker voor een general purpose systeem. Veel beter is het aan de programmeur hulpmiddelen in de taal zelf te verschaffen waarmee hij de klasse van toegelaten invoer-gegevens kan afbakenen.

Dit principe is in het grafische systeem toegepast. De programmeur kan een subtekening genereren c.q. uit de bibliotheek selecteren en tot *symbool* verklaren. Een *symbool*-declaratie heeft tot gevolg, dat deze subtekening wordt bekendgemaakt aan de tekenmachine. Een analoge definitie is mogelijk voor koppelings-operatoren (zgn. *connector*). Een tekenmachine mag vervolgens bij het produceren van tekeningen uitsluitend gebruik maken van symbol en *connector*. De elementen uit de primitieve laag zijn per definitie *symbol* of *connector*.

```
mode symbol = struct (ref subpict spchar, ref string pname);
mode connector = struct (ref proc opchar, ref string oname);
```

Een dergelijke opzet veronderstelt dat een tekenmachine gekoppeld wordt via een lokaal grafisch systeem voor:

1. opslag van *symbol*'s en *connector*'s
2. editor om niet toegelaten tekeningen te verbeteren.

De tekening komt bij het programma aan als expressies van de vorm:

S c S.

Hierbij is S een primitieve, een *symbol* of een expressie tussen haakjes; c is een *connector* of een primitieve operator.

Een van de belangrijkste conclusies luidt dan ook, dat een grafisch systeem als sub-systeem een formule-manipulatie-systeem moet bevatten.

Merk op dat deze wijze van werken niet mogelijk is bij gebruik van display-procedures (zie sectie 3.) omdat daarbij uitsluitend rijen symbolen kunnen worden aangeboden, d.w.z. er is één type *connector* nl. onze verenigingsoperator.

Een bijzonder geval van tekeningen-invoer vindt plaats tijdens de klassieke vorm van interactie. Daarbij heeft de gebruiker maar één middel ter beschikking nl. een "aanwijsstok" (light-pen) waarmee hij een subtekening kan aanwijzen. Een aanwijsactie resulteert in eerste instantie in een *point*. Vervolgens moet een selector m.b.v. dat punt, uit de tekening op het beeldscherm een subtekening isoleren, bij voorkeur een *symbol*. Het is duidelijk dat de box hier wederom uitstekend dienst kan doen.

LITERATUUR

[1] NEWMAN, W.M., *Display Procedures*, CACM 14 (1971) 651.

[2] WIRTH, N., & H. WEBER, *EULER: A Generalization of Algol, and its Formal Definition: Part I*, CACM 9 (1966) 13-25.

[3] WILLIAMS, R., *A Survey of data structures for Computer Graphics Systems*, Computing Surveys 3 (1971) 1-21.

MATHEMATICAL CENTRE SYLLABUS 25, 1976, III, 41-58

DATASTRUCTUREN: VERGELIJKING VAN BESTAANDE PROGRAMMEERTALEN

L. AMMERAAL

1. DATA EN GEHEUGENORGANISATIE

In een programma worden operaties op interne objecten uitgevoerd die een elementaire of meer samengestelde structuur kunnen hebben. Deze objecten noemt men wel "*data*" of "*datastructuren*" en een aantal eigenschappen van zo'n object vat men samen onder het begrip "*datatype*". Behalve door de behoeften van de gebruiker, zijn de ontwikkelingen op het gebied van datatypes in programmeertalen sterk beïnvloed door de ontwikkelingen in machine-ontwerp. Van oudsher bestaan er de volgende twee verschillende opvattingen over de functie van de computer:

A. een computer is een administratiemachine en dient om stukjes alfanumerieke tekst in te lezen en na enige simpele bewerkingen (waarin bij uitzondering een optelling kan voorkomen) weer als uitvoer af te leveren.
B. een computer is een rekentuig en dient om te manipuleren met getallen.

Voor beide opvattingen bestaat een klasse van machines. Deze twee machinetypen (A en B) hebben een verschillende geheugenorganisatie.

Het geheugen van een A-machine is opgebouwd uit "posities" oftewel "bytes", elk bestaande uit n bits ($n \leq 8$) en elk geschikt om één alfanumeriek karakter op te slaan. Elke "positie" is direct adresseerbaar. Een getal wordt als een rijtje decimale cijfers opgeborgen als een bijzonder geval van een stukje tekst. De lengte van elk stukje tekst en dus ook van een getal is in de regel variabel; een speciaal karakter markeert het eind van elk stukje tekst. Instructies zijn eveneens stukjes tekst van variabele lengte.

Het geheugen van een B-machine is opgebouwd uit "woorden", elk bestaande uit n bits ($12 \leq n \leq 60$).
Gewoonlijk wordt een getal, zuiver binair gecodeerd, in één woord opgeslagen, evenals een instructie.

Het spreekt vanzelf dat, uit het oogpunt van machine-ontwerp bezien, voor een A-machine een "characterstring" en voor een B-machine een "integer" de meest natuurlijke datastructuur is. Een andere belangrijke hardware-faciliteit die met datastructuren in verband kan worden gebracht is het ingebouwde stapelmechanisme van sommige moderne machines. Nog maar al te vaak heeft een machine een uiterst rommelig instructierepertoire waarin deze faciliteit ontbreekt.

2. OVER HET VERGELIJKEN VAN TALEN.

Een objectieve vergelijking van de relatieve kwaliteiten van programmeertalen wordt door een groot aantal factoren bemoeilijkt. Deze zijn voor een deel van psychologische aard. Men is zelf bevooroordeeld doordat men gewoonlijk één taal gebruikt, waaraan men gehecht is, en men verwacht van een ander een dergelijk vooroordeel. Ook wordt een vergelijkende verhandeling ernstig bemoeilijkt door het verschil in terminologie: een begrip uit de ene taal correspondeert min of meer met een begrip uit een andere taal maar is daar onder een andere naam bekend. Deze moeilijkheid treedt vooral op bij een bespreking van datatypes. Zo correspondeert de ALGOL 68-declaratie

```
[1 : 20] char c
```

vrij aardig met de PL/1-statement

```
DECLARE  C  CHAR(20); ,
```

maar in PL/1 noemt men C een "character string" en in ALGOL 68 is de mode van c "reference to row of character". Spreekt men nu losjes over een

"string", dan denkt de ALGOL 68-gebruiker aan een mode die hij als string specificeert, hetgeen flex [] char betekent, de ALGOL 60-gebruiker aan iets als ´ABCDE`en de PL/1-gebruiker aan zowel een "character string" als een "bitstring". Is het ALGOL 68-begrip string bedoeld, dan dient voor de PL/1-gebruiker die dit wil begrijpen de term VARYING niet onvermeld te worden gelaten. Als de bespreking van zo'n eenvoudig begrip als een "string" reeds aanleiding is tot begripsverwarring tussen de gebruikers van de drie talen ALGOL 60, ALGOL 68 en PL/1 die nog tamelijk verwant zijn, dan behoeft het geen verwondering te wekken dat een uitgebreidere discussie over programmeertalen zelden of nooit tot overeenstemming leidt: men begrijpt elkaar eenvoudig niet.

Een andere voortdurende bron van misverstand vormt het subtiele onderscheid tussen een taal en zijn implementatie. Een gebruiker staat soms voor het dilemma dat hij moet kiezen tussen een slechte implementatie van een goede taal en een goede implementatie van een slechte taal. Van fundamenteel standpunt bekeken en voor lange-termijn-beschouwingen heeft genoemd onderscheid inderdaad zin; de meer praktisch ingestelde computergebruiker daarentegen spreekt de slogan "Een taal is even goed als zijn implementatie" (WIRTH [4]) meer aan.

Is het al dubieus of de kwaliteit van een implementatie de taalontwerpers aangerekend mag worden, nog bedenkelijker is het als men het aantal "aanhangers" als maatstaf gaat hanteren. Deze dwaling komt wellicht voort uit een te ver doorgevoerde parallel met "natuurlijke" talen; hier geldt inderdaad dat kennis van het engels nuttiger is dan kennis van het Esperanto. Deze overtrokken analogie met natuurlijke talen verklaart ook het flauwe argument dat steevast bij een bespreking over programmeertalen opduikt: "Het maakt niet uit welke taal je gebruikt; het hangt er maar vanaf wat je gewend bent". Het is daarom nodig zich te realiseren dat een programmeertaal een kunsttaal is die als algorithmisch uitdrukkingsmiddel mede onze technische en intellectuele mogelijkheden bepaalt en daarom met de grootst mogelijke zorg dient te worden ontworpen of uitgekozen.

In verband met deze overweging zal in de volgende beschouwingen een bespreking van enkele frekwent toegepaste maar minder interessante talen als FORTRAN en BASIC achterwege blijven.

3. DATATYPES IN ENIGE CONVENTIONELE TALEN

In het ALGOL 60-rapport [1] vindt men de syntactische definitie

<type> ::= real | integer | Boolean .

In assignment statements kunnen in ALGOL 60 uitsluitend waarden van deze drie types als object worden gebruikt. Een uitbreiding van deze uiterst beperkte verzameling van datatypes verkrijgt men door na te gaan wat er zoal kan worden gedeclareerd. Men vindt:

<declaration> ::= <type declaration> | <array declaration> |
<switch declaration> | < procedure declaration>.

Ter vergelijking met andere talen is het goed bij array's te letten op de volgende punten:

- alle elementen zijn van hetzelfde type, n l. ôf real ôf integer ôf Boolean,
- zowel voor de grenzen als voor de subscripts kunnen willekeurige expressies worden gebruikt,
- de dimensie is een willekeurig natuurlijk getal (dus niet beperkt tot b.v. 3).

In ALGOL 60 is er niet veel reden om de switch en de procedures tot de data types te rekenen. Wel kunnen switches en procedures als actuele parameter in een procedure-statement opgeven, hetgeen eveneens kan met een string en een label. De onmogelijkheid om in ALGOL 60 met een string meer te doen dan dat verraadt de numeriek wiskundige herkomst van deze taal en is een van de oorzaken dat deze taal voor commercieel-administratieve toepassingen vrijwel geen ingang heeft gevonden. ALGOL 60 is, evenals trouwens FORTRAN, een taal die volgens de eerder gebruikte terminologie beter past bij het machinetype B dan bij het machinetype A.

Het tegenovergestelde is het geval met COBOL [5]. In deze taal definieert men in de "data-division" b.v. de record description

```
01  STAMRECORD.
  02  NAAM.
    03  ACHTERNAAM PIC X(20).
    03  VOORNAAM-1 PIC X(10).
    03  RESTVOORLET PIC X(5).
  02  GEBDAT.
    03  JAAR PIC 99.
    03  MAAND PIC 99.
    03  DAG PIC 99.
  02  GESLACHT PIC X.
  02  HUWSTAAT PIC X.
  02  WERKNUMMER PIC 9999.
```

Deze hiërarchische opbouw van een stel samenhangende gegevens is bijzonder overzichtelijk en aantrekkelijk voor eenvoudige administratieve toepassingen. Daarentegen is deze taal zeer onaantrekkelijk voor rekenkundige werkzaamheden; het stukje COBOL-programma

```
MOVE 1 TO A.
MOVE 2 TO B.
SUBTRACT B FROM A GIVING C.
```

met de betekenis

```
a := 1  ;  b := 2  ;  c := a - b
```

illustreert dit reeds.

Om aan de wensen van zowel de "rekenaars" als de "administratievelingen" te voldoen zijn in de IBM 360/370-serie de beide geheugenorganisatievormen A en B verenigd. Men vindt dit idee terug in de programmeertaal PL/1. Declareert men in deze taal

```
DCL C CHAR(25) ;   (of: DCL C PICTURE'(25)X';)
DCL I FIXED BIN(31);
```

en laat men het PL/1 programma, zoals in negenennegentig van de honderd gevallen gebeurt, uitvoeren op een 360/370-computer, dan worden voor C en I resp. de geheugenorganisaties A en B gebruikt. De variabele C beslaat 25 bytes van elk 8 bits; voor I wordt één woord van 32 bits gebruikt; elke byte is adresseerbaar en elk byte-adres dat een 4-voud is kan opgevat worden als het adres van een woord. De sterke verwantschap tussen PL/1 en de IBM 360 komt vooral aan het licht als men let op de begrippen

PACKED DECIMAL (een 360-hardware term) en
FIXED DECIMAL (een PL/1-term) .

De invoering van deze begrippen vloeit voort uit de overweging dat de opbouw van een byte uit 8 bits wel mooi lijkt als men denkt aan een vrij uitgebreide karakterverzameling, maar enigszins verkwistend is in het belangrijke bijzondere geval dat deze karakters decimale cijfers zijn. In plaats van van de programmeur te verlangen dat hij m.b.v. het datatype "FIXED BINARY" overgaat op een zuiver binaire representatie voor getallen als de karaktergewijze representatie niet voldoet, heeft men een stukje hardware in de machine aangebracht om een byte onder te verdelen in twee rijtjes van 4 bits, waardoor twee decimale cijfers in plaats van één in een byte kunnen worden opgeslagen. In de 360-PL/1-implementatie maakt men gebruik van deze gepakt decimale representatie door middel van het data type FIXED DECIMAL. Als hij aan deze representatie zou willen ontkomen, dan zou de PL/1-programmeur niet alleen declaraties zoals

```
DCL F FIXED DEC (6);
```

achterwege moeten laten, maar bovendien decimaal geschreven constanten moeten vermijden. Zelfs zonder nog te spreken over de data types COMPLEX, FLOAT en over bit strings welke laatste onder meer gebruikt worden om true en false te representeren, is het duidelijk dat PL/1 een grote verscheidenheid aan datatypes kent. Het hoeft geen betoog dat men daarom een bijzonder grote zorgvuldigheid in acht had moeten nemen bij het opstellen van regels voor de conversie van het ene datatype naar het andere. Hierin is men tekortgeschoten. De fout die de ALGOL-60-ontwerpers hebben

gemaakt door een geval als

```
integer i ;  i := 3.14
```

toe te staan is een bagatel vergeleken met de absurd schijnende conversieregels die er in PL/1 gedefinieerd zijn. Een enkel voorbeeld ter illustratie hiervan is voldoende.
Schrijft men

```
DCL F FIXED DEC (5);
DCL C CHAR (5);                                    (1)
F = 12345;
C = F;
```

dan verloopt de conversie in de laatste statement als volgt. Eerst wordt het getal 12345 omgezet in de acht karakters lange string

```
bbb12345    (b is een spatie)
```

De drie spaties vooraan worden toegevoegd i.m.v. met de ruimte die in sommige gevallen gevraagd zou worden voor o.a. een decimale punt en een minteken. Vervolgens wordt de waarde van deze string ter lengte 8 toegekend aan C die maar op vijf karakters berekend is. In zo'n geval worden bij string assignment de eerste vijf karakters genomen waardoor C de waarde

```
bbb12
```

krijgt.

Wordt in PL/1 een identifier geassocieerd met "data" dan stelt deze identifier een variabele voor (hetgeen niet vanzelf spreekt; in ALGOL 68 is b.v. pi door de declaratie real pi = 3.14 wel een identifier, maar geen variabele). Constanten zijn steeds skalair; variabelen kunnen ook "aggretaten" zijn, waaronder "arrays" en "structures" worden verstaan. Als componenten van een structure kunnen ook arrays of structures voorkomen.

De elementen van een array kunnen ook willekeurige structures zijn. Procedures worden niet als "data" beschouwd en kunnen niet als component van een array of structure optreden. De bij ALGOL 60 genoemde vrijheden t.a.v. de grenzen en subscripts van arrays bestaan ook in PL/1. Een eenvoudig voorbeeld van een array A en een structure S is het volgende.

```
DCL A (10) FIXED BIN (31);
DCL 1 S,
      2 P FIXED BIN (31),
      2 Q CHAR (5),
      2 R (10) FLOAT BIN (53),
      2 T,
        3 U CHAR (3),
        3 V BIT (5),
      2 W FIXED  DEC (4);
```

Men kan nu b.v. laten volgen

```
S.T.U  = 'ABC';
```

Een vrijwel identieke wijze van structurering was al aanwezig in COBOL. Meer specifiek PL/1-achtig is het begrip "controlled-variable". Declareert men

```
DCL A FIXED BIN (31) CONTROLLED;
```

dan ontstaat elke keer dat de statement

```
ALLOCATE A;
```

wordt uitgevoerd een nieuwe generatie van A waaraan men een waarde kan toekennen zonder dat de vorige generaties van A aangetast worden. Door de executie van

```
FREE A;
```

verdwijnt de laatste generatie en komt automatisch de voorlaatste weer beschikbaar. Een voorbeeld van het gebruik hiervan is het volgende PL/1-programma dat 100 getallen a_1, a_2, ..., a_{100} inleest en deze getallen in omgekeerde volgorde, d.w.z. a_{100}, a_{99}, ..., a_1, afdrukt.

```
P : PROC OPTIONS (MAIN);
      DCL A FLOAT BIN (53) CONTROLLED;
      DO I = 1 TO 100; ALLOCATE A; GET LIST (A); END;
      DO I = 1 TO 100; PUT LIST (A); FREE A; END;
    END;
```

Het gegeven voorbeeld van het in omgekeerde volgorde afdrukken van een rij ingelezen getallen wordt in inleidende programmeercursussen wel eens gebruikt ter illustratie van het begrip array, waarbij vaak wordt gesuggereerd dat men een array hierbij nodig heeft, indien men althans het programma kort wil houden. Er volgt dan zoiets als

```
begin integer i;
      array a [1 : 100];
      for i := 1 step 1 until 100 do a [i] := read;
      for i := 100 step -1 until 1 do print (a[i])
end.
```

De vraag rijst nu of deze programmeeropgave ook zonder het begrip "array" èn zonder het begrip "controlled variable" kan worden opgelost. Inderdaad is dit mogelijk; de volgende oplossing, die in zekere zin zelfs meer elementair is dan de beide vorige, toont dit aan.

```
begin procedure p(n); value n; integer n;
      if n > 0 then
      begin real a;
            a := read; p(n-1); print (a)
      end;
      p (100)
end.
```

Men ziet in dit programma iets van de mogelijkheden die blockstructuur en recursie ook ten aanzien van datastructuren bieden.

4. IETS OVER ENKELE NIEUWERE TALEN

O.a. het tragische gemis aan string-faciliteiten in een verder zo elegante taal als ALGOL 60 is aanleiding geweest tot verdere taalontwikkelingen in de ALGOL-lijn. Enerzijds hebben sommige computerleveranciers waaronder Burroughs, General Electric en Univac, compilers gemaakt voor ALGOL 60 met stringfaciliteiten. Anderzijds zijn er ALGOL-achtige talen ontstaan die wat meer van ALGOL 60 afwijken.

Een taal die op slechts geringe uitzonderingen na ALGOL 60 als deeltaal omvat in SIMULA 67. Deze taal ken de types "character" en "text". Een meer spectaculair aspect van deze taal is het begrip "class", dat opgevat kan worden als een middel om met procedures te manipuleren, zoals blijkt uit het volgende SIMULA 67-programma:

```
begin class c(n); integer n;
      begin procedure p;
            write (n);
      end;
      ref (c) c1, c2;
      c1 :- new c (100); c2 :- new c (200);
      c1.p ; c2.p
end.
```

Door c1 :- new c (100) wordt een nieuw object van de klasse c gecreëerd, d.w.z. een procedure p waarin n = 100. Een referentie naar dit object wordt in c1 opgeborgen; later maakt de statement c1.p hiervan gebruik, waardoor 100 wordt afgedrukt. Evenzo wordt daarna door c2.p het getal 200 afgedrukt. Door DAHL & HOARE [7] worden enige toepassingen van het class-concept gegeven in het kader van het even populaire als vage onderwerp "structured programming". Een andere taal die men in het kader van dit onderwerp wel aanprijst is PASCAL [8]. Deze taal wijkt sterker van ALGOL 60 af dan

SIMULA 67. Twee bijzonder nuttige ALGOL 60-elementen zijn zonder duidelijke reden uit deze taal verdwenen, n.l. de conditionele expressie, zoals b.v.

```
if x > 0 then x else 0
```

en de dynamische arraygrenzen, zoals b.v.

```
begin integer n; n := read;
      begin array a [1 : n];
            - - -
            - - -
```

Daartegenover staat de invoering van enkele minder conventionele datatypes, waaronder de begrippen "*subrange type*", "*pointer type*", "*record type*" en "*set type*". Ten opzichte van de nog te behandelen ALGOL 68 modes vindt men in de PASCAL-datatypes nauwelijks iets nieuws. Dat deze door één man ontwikkelde taal nogal wat belangstelling heeft is vooral te danken aan een goed leesbare documentatie en aan een goede implementatie voor enkele computers. Voor kritisch commentaar op deze taal wordt verwezen naar HABERMAN [9].

5. OVER MODES IN ALGOL 68

Een belangrijke kwestie bij het ontwerpen van een taal is de vraag in hoeverre taalelementen in de taal dienen te worden opgenomen die er, gezien andere aspecten van de taal, logisch in thuis horen, maar die implementatiemoeilijkheden zouden kunnen opleveren en waaraan bovendien van praktische zijde nog niet veel behoefte is gebleken. Het weglaten van deze taalelementen kan men als "pragmatisch", het opnemen ervan als "orthogonaal", (VAN WIJNGAARDEN [10]) ontwerp aanduiden. Een historisch voorbeeld van orthogonaal ontwerp was het toelaten van recursieve procedures in ALGOL 60; een nogal pragmatisch standpunt t.a.v. taalontwerp wordt verdedigd in WIRTH [4]. In de taal ALGOL 68 (VAN WIJNGAARDEN [3], [11]) overheerst

de orthogonale benadering. Het begrip "*mode*" in deze taal kan men opvatten als een zeer ruime interpretatie van de term "*datatype*" die ALGOL 68 niet kent. In de taaldefinitie wordt het metabegrip MODE formeel gedefinieerd met behulp van een stelsel metaproduktieregels. Om hierop in dit korte bestek enig zicht te krijgen volgt hier een gesimplificeerde en niet geheel complete informele beschrijving van dit begrip. Het begrip MODE valt uiteen in de volgende alternatieven:

```
integer
real
boolean
character
structured with ..... mode
FLEXITY ROWS of ..... ("FLEXITY ROWS" staat voor een of meer
                       keren "row" eventueel voorafgegaan door
                       flexible)
reference to .....
procedure with ..... parameter yielding .....
union of ..... mode
```

Op de plaats van de puntjes (.....) wordt een uitdrukking ingevuld waarin wederom een of meer van de voor MODE gegeven alternatieven voorkomen. Men kan zich voorstellen dat dit ingewikkeld geschreven modes toelaat, zoals b.v.

```
union of
row row of procedure with reference to reference to
     row row row of reference to integral parameter yielding
     structured with real field letter r row of boolean field
     letter b mode character mode.
```

In een ALGOL 68-programma kan men bovendien werken met modes die wel volgens bovengenoemde regels kunnen worden opgebouwd, maar dan niet in een eindig aantal woorden kunnen worden uitgedrukt. Met behulp van z.g. modedeclaraties kunnen deze modes wel eindig worden gespecificeerd. Het volgende voorbeeld illustreert dit:

```
mode m = struct (int i, ref m p);
m s .
```

De eerste regel definieert een mode die in de tweede regel wordt gebruikt bij de declaratie van de variabele s. Zonder een afkorting zoals m is de mode van s alleen weer te geven door middel van een oneindig lange rij woorden, waarvan het voornaamste bestanddeel verkregen wordt door

"structured with integral field letter i reference to"

oneindig vaak te herhalen. De mogelijkheid om zelf modes te definieren heeft nog een aantal andere konsekwenties. Een ervan is dat enige standaard modes zoals "complex" en "string" in meer elementaire modes zijn uit te drukken:

```
mode compl = struct (real re, im);
mode string = flex [1 : 0] char .
```

De programmeur moet zich voorstellen dat deze modes evenals een aantal andere modes, functies, operatoren enz. zijn gedefinieerd in de z.g. "standard prelude" die deel uitmaakt van een "range" die zijn "particular program" omvat. Het zelf definieren van modes komt vooral tot zijn recht als men bovendien voor objecten van deze modes zelf operatoren definieert. Het volgende programmadeel laat zien hoe men de mode rat voor positieve rationale getallen kan definieren en hoe men de dyadische operatoren * en + voor deze getallen kan definieren en gebruiken.

```
mode rat = struct (int teller, noemer);
op * = (rat a, b) rat:
begin int t = teller of a * teller of b,
          n = noemer of a * noemer of b;
      int c = ggd (t,n);
      (t÷c, n÷c)
end;
```

```
op + = (rat a, b) rat:
begin int c = ggd (noemer of a, noemer of b);
      int fa = noemer of a ÷ c,
          fb = noemer of b ÷ c;

      int t = fb * teller of a + fa * teller of b,
          n = c * fa * fb;
      int c1 = ggd (t,n);
      (t÷c1, n÷c1)
end;

proc ggd = (int a, b) int:
if b = 0 then a else ggd  (b,a mod b) fi;

rat p, q, r;

p := (7,10) ; q := (5,12);
r := p + q + p * q
```

Men ziet aan dit voorbeeld dat een onderscheid tussen "eenvoudige" en "ingewikkelde" objecten, die dan met "data" resp. "datastructuren" aangeduid zouden moeten worden, in ALGOL 68 niet bijzonder zinvol is: de formule p + q + p * q is voor objecten van de door ons gedefinieerde mode niet ingewikkelder dan voor objecten van b.v. de mode integral.

Het orthogonaliteitsaspect van ALGOL 68 blijkt vooral uit de grote mate van uniformiteit bij het manipuleren met objecten, ongeacht de complexiteit van hun mode. Men lette b.v. op de bijzonder sterke analogie tussen

```
proc (int) real p, q ;
----
p := (int n) real : n + 3/3 ;                    (2)
q := p
```

en

```
int p, q ;
---
p := 5 ;
q := p
```
(3)

De z.g. "routinetext" die na het wordtteken in (2) is vermeld speelt een rol die volkomen analoog is aan de "integral denotation" 5 in (3).

Overigens is de orthogonaliteit zelfs in ALGOL 68 niet zover doorgevoerd als theoretisch denkbaar is, of, anders gezegd, overwegingen van implementatietechnische aard hebben ook in ALGOL 68 geleid tot het verbieden van bepaalde constructies, zelfs als het praktisch nut van deze constructies, indien implementeerbaar, aantoonbaar is. Een voorbeeld hiervan is het verbieden van read(p) in (2), terwijl dit in (3) wel is toegestaan. Het presenteren van een overzichtelijke vergelijking van de verschilpunten tussen datatypes en -structures in PL/1 en modes in ALGOL 68 is vrijwel niet doenlijk. Enige in het oog lopende verschillen zijn:

- De "data"-aspecten van procedures zijn in ALGOL 68 veel sterker aanwezig dan in PL/1 waar immers b.v. een array van procedures onmogelijk is.
- De wijze waarop datastructuren in het geheugen van de machine worden gepresenteerd is in ALGOL 68 volkomen aan de implementator overgelaten. In PL/1 is dit veel vaker in de taal gedefinieerd; dit is daar o.a. nodig i.v.m. de niet ter sprake gekomen attributen "DEFINED" en "AREA" en i.v.m. "record input/output".
- In tegenstelling tot PL/1 is in ALGOL 68 de overgang van de ene mode naar de andere zodanig gedefinieerd dat geen onverwachte verandering van waarde optreedt.

De volgende opmerkingen illustreren dit laatste punt.
Het PL/1-voorbeeld (1) levert, overgezet in ALGOL 68 geen probleem:

```
int f ;
[1 : 5] char c ;
f := 12345;
c := fixed (f,5,0) ;
```

De tweede en derde parameter van fixed geven resp. het totale aantal posities en het aantal posities achter de punt aan. In plaats van de laatste assignation zou c := f syntactisch fout zijn, evenals i := r fout is bij de declaraties

```
int i ; real r .
```

Daarentegen is r := i toegestaan. Hier treedt "widening" op, hetgeen een bijzonder geval van "coercion" is. In de taaldefinitie van ALGOL 68 (VAN WIJNGAARDEN [3], [11]) is het toelaten van r := i en het verwerpen van i := r (in de gegeven context), evenals tientallen andere regels die de schrijver van een ALGOL 68-programma moet kennen, niet in klare engelse tekst, maar in het specialistische formalisme van een z.g. Van Wijngaarden-grammatica vastgelegd. Deze wijze van formuleren is wiskundig volkomen bevredigend, maar voor de praktijk van het programmeren volkomen ongeschikt: slechts voor iemand die meer geïnteresseerd is in de taal dan in de toepassingen ervan dient de bestudering van het ALGOL 68-rapport te worden aanbevolen. In AMMERAAL [13] is een poging gedaan ALGOL 68 op een minder afschrikwekkende wijze te benaderen. Overigens is de populariteit van een taal, hoe belangrijk ook, essentieel iets anders dan de wetenschappelijke bijdrage die bij het ontwerpen van de taal wordt geleverd. Ook een (nog) niet op grote schaal toegepaste taal kan van grote betekenis zijn op het ontwerp van nieuwe talen. Een voorbeeld hiervan is de recentelijk ontwikkelde taal EL1 (WEGBREIT [12]), waarin de datatypes nogal wat trekken van overeenkomst met ALGOL 68 vertonen.

LITERATUUR

[1] NAUR, P. (ed.), *Revised report on the algorithmic language ALGOL 60*, Numerische Mathematik, 4 (1963) 420-453.

[2] *PL/I Language Reference Manual*, Form C28-8201-2. IBM Syst. Ref. Lib. (1969).

[3] WIJNGAARDEN, A. VAN (ed.), B.J. MAILLOUX, J.E.L. PECK & C.H.A. KOSTER, *Report on the algorithmic language ALGOL 68*, Numerische Mathematik, 14 (1969) 79-218.

[4] WIRTH, N., *On the design of programming languages*, Information processing 74, pp. 386-393, North-Holland Publ. Co., Amsterdam (1974).

[5] CODDINGTON, L., *Quick COBOL*, Macdonald London and American Elsevier New York (1971).

[6] DAHL, O.-J., B. MYHRHANG & K. NYGAARD, *The simula 67 Common Base Language*, Norwegian Computing Centre, Forskningsveien 1B, Oslo 3 (1968).

[7] DAHL, O.-J. & C.A.R. HOARE, *Hierarchical Program Structures*, in Structured Programming (Dahl, Dijkstra & Hoare), pp. 175-220, Academic Press (1972).

[8] WIRTH, N., *The programming languages PASCAL*, Acta Informatica, 1 (1971) 35-63.

[9] HABERMAN, A.N., *Critical Comments on the Programming Language Pascal*, Acta Informatica, 3 (1973) 47-57.

[10] WIJNGAARDEN, A. VAN, *Orthogonal Design and Description of a Formal Language*, Mathematisch Centrum, Amsterdam, MR 76 (1965).

[11] WIJNGAARDEN, A. VAN, B.J. MAILLOUX, J.E.L. PECK, C.H.A. KOSTER, M. SINTZOFF, C.H. LINDSEY, L.G.L.T. MEERTENS & R.G. FISKER, *Revised Report on the Algorithmic Language ALGOL 68*, Supplement to ALGOL BULLETIN 36 (1974).

[12] WEGBREIT, B., *The Treatment of Data Types in EL1*, Comm. ACM, 17 (1974) 251-264.

13 AMMERAAL, L., *Syllabus cursus ALGOL 68*, Mathematisch Centrum, Amsterdam, IC 1/75 (1975).

MATHEMATICAL CENTRE SYLLABUS 25, 1976, IV, 59-78

DE HEEL EENVOUDIGE TAAL HET

L. MEERTENS

0. INLEIDING

In het begin van de jaren zestig, met de ontwikkeling van de (zeer geslaagde) programmeertaal ALGOL 60, ontstaat het besef dat we het ontwerpen van een programmeertaal, technisch gezien, aankunnen. ALGOL 60 zou geen eindpunt zijn, maar veeleer punt van vertrek. In deze tijd verschijnen een aantal publicaties met naast bezinnende ook inspirerende en baanbrekende ideeën, gericht op de toekomstige ontwikkeling van programmeertalen. Genoemd mogen worden: *An attempt to unify the constituent concepts of serial program execution* en *On the design of machine-independent programming languages* van DIJKSTRA [1,2] en met name *Generalized ALGOL* van VAN WIJNGAARDEN [3]. De laatste publicatie heeft ten grondslag gelegen aan de programmeertaal EULER (WIRTH & WEBER [4]).

Een grondgedachte van [2] is de volgende: Een programmeertaal bevat, zowel syntactisch als semantisch, een aantal grondbegrippen, "primitiva", en daarnaast middelen om daaruit ingewikkelder begrippen samen te stellen, "compositieschema's". Zorg ervoor dat de compositieschema's zo algemeen mogelijk zijn: iedere combinatie van begrippen die daarvoor in aanmerking komt voor de parameters van een compositieschema moet een zinvol resultaat leveren, ongeveer zoals iedere combinatie van coördinaten langs orthogonale assen als resultante een punt uit de productruimte heeft. Op dit laatste beeld berust de uitdrukking: "orthogonaal" ontwerpen. (In [2] heette een zonde hiertegen" "unnecessary redundancy".) Een hopelijk verhelderend voorbeeld: als ALGOL 60 orthogonaal ontworpen was, dan mochten we schrijven:

```
string procedure teken (z); value z; real z;
teken:= if z > 0 then 'positief' else
        if z = 0 then 'nul' else 'negatief'.
```

Dit is een voorbeeld van de algemeenheid van compositieschema's; in [3] komt -naast deze gedachte- nog een andere zeer belangrijke grondgedachte voor: het aantal primitiva moet niet groot zijn, maar beperkt blijven tot het wezenlijke. Een voorbeeld van het terugbrengen van het aantal primitiva krijgen we als we zouden definiëren dat *string* alleen maar een afkorting is voor *character array*, waarna het zinvol wordt te schrijven

character t; t:= teken (x - y) [1].

Het orthogonaal ontwerpen heeft via VAN WIJNGAARDEN [5] aan de wieg gestaan van ALGOL 68. Toch is uiteindelijk deze methode niet volledig consequent gehanteerd, met name uit efficientie-overwegingen. Men kan overigens van oordeel verschillen over de vraag of ALGOL 68 niet orthogonaal genoeg, of misschien te orthogonaal is ontworpen.

In ieder geval is het interessant bij wijze van experiment het aantal primitiva en compositieschema's in een programmeertaal terug te brengen tot het allerwezenlijkste, door zoveel mogelijk, desnoods tot in het absurde, te generaliseren en door zonder scrupules alles overboord te zetten wat al met bestaande middelen kan worden uitgedrukt. Bij een dergelijke operatie sneuvelt uiteraard alles wat kan worden gekenschetst als "syntactische suiker"; zo vormen formules in een programmeertaal die al procedures kent, een te waarderen, maar niet onmisbare, luxe: *sum (prod (2, x),1)* is wat bitterder, maar niet minder effectief, dan $2 \times x + 1$. En in ALGOL 68 zouden als eerste de "extensies" verdwijnen, nog expliciet aanwezig in het oorsponkelijke ALGOL-68-rapport [6], maar in het herziene rapport [7] verstopt in syntaxis en semantiek. I.p.v. *real x:= 3.14* zou weer prijken *ref real x = loc real := 3.14*, grimmig en grandioos.

HET (Heel Eenvoudig Taaltje) is het resultaat van zo'n experiment. Andere eenvoudige talen zijn LISP 1.5 of het iets minder bekende TRAC® [8]. Deze hebben ook een eenvoudige syntaxis, en met name TRAC heeft een zeer eenvoudige semantiek. Een verschil met HET is dat deze talen zich kenmerken door functie-compositie (zodat al gauw processies van sluithaakjes ontstaan), terwijl bij HET talen van het ALGOL-type uitgangspunt zijn geweest, waardoor de nadruk ligt op het van-links-naar-rechts uitvoeren van het programma. Belangrijker is misschien dat zowel LISP als TRAC bedoeld zijn om

ook gebruikt te worden, terwijl de practische bruikbaarheid bij het ontwerpen van HET geen oogmerk was - dat zou het experiment bij voorbaat zinloos gemaakt hebben. In dit opzicht lijkt HET op GEDANKEN (REYNOLDS [9]).

1. PRIMITIVA VAN PROGRAMMEERTALEN

De eerste opgave is na te gaan welke primitiva aan hogere programmeertalen zoals ALGOL 68 ten grondslag liggen. In *A unified, conceptual, basis of programming*, de eerste voordracht van[10], noemt BAUER: procedures/parameters; free identification; references; sets; structured objects; selection. Ruwweg dezelfde rij zal hier de revue passeren. Opgemerkt zij nog dat de compositieschema's zelf ook primitief of composiet kunnen zijn, en zeker meebeschouwd dienen te worden.

1.1. *Objecten*

Het is zaak te beseffen dat sprake is van objecten uit twee werelden, die uit de "externe" wereld van de programmatekst, verder *TEXT* genoemd, en die uit de "interne" wereld van de computer, verder *VAL* genoemd.

Voorbeelden van elementen uit *TEXT* zijn identifiers (*IDF*), zoals *i* of *marilyn*, denotaties (*DEN*), zoals *3.14*, *true* of *"nul"*, en routineteksten (*RTEXT*).

Dus: $IDF \cup DEN \cup RTEXT \subset TEXT$.

Voorbeelden van elementen uit *VAL* zijn getallen, routines (*ROUT*), arrays (*ARR*) en gestructureerde waarden (*SVAL*).

Dus: $ROUT \cup ARR \cup SVAL \subset VAL$.

1.2. *Gegevensstructuren en hun operaties*

Niet-primitieve objecten worden door compositie opgebouwd. De compositieschema's voor *TEXT* worden gegeven door de syntaxis en zijn te beschouwen als functies

$$TCOMP_i: TEXT \times \ldots \times TEXT \rightarrow TEXT \ ^{*)},$$

---

*) De notatie $F: \mathcal{D} \rightarrow R$ wil zeggen dat F een functie is die elementen uit de verzameling $\mathcal{D}$ afbeeldt op elementen uit R. Zoals gebruikelijk hoeft niet ieder element van R verkrijgbaar te zijn door F toe te passen op een element van $\mathcal{D}$, maar anders dan gebruikelijk is hier evenmin bedoeld dat F noodzakelijkerwijze gedefinieerd is voor alle elementen uit $\mathcal{D}$.

terwijl die voor *VAL*, van de vorm

$$VCOMP_i: VAL \times \ldots \times VAL \to VAL,$$

min of meer voorafgaan aan de semantiek, die een verband legt tussen *TEXT* en *VAL* door aan de hand van *VAL* acties te definieren die de interne wereld beïnvloeden. De functies $TCOMP_i$ en $TVAL_i$ zijn omkeerbaar, m.a.w., uit b.v. de tekst $TCOMP_i(T_1, \ldots, T_n)$ zijn $T_1, \ldots, T_n$ weer af te leiden.

Verdere operaties die bij de semantiek een rol spelen:

a) *REF*: *NAME* → *VAL*.

Een naam (speciaal geval van een waarde) kan verwijzen naar een andere waarde. Essentieel voor *REF* is dat er geen "logisch" verband is tussen naam en waarde - het is een kwestie van opzoeken. Bovendien is *REF* niet constant; er is een actie die namen naar andere waarden kan doen verwijzen.

b) *ACC*: *IDF* → *VAL*.

Een identifier (uit *IDF* ⊂ *TEXT*) geeft toegang tot een waarde. Dit kan nog verfijnd worden door te stellen dat de identifier niet alleen, maar in combinatie met zijn omgeving de toegang geeft, dus *ACC1*: *IDF* × *ENV* → *VAL*. Zoals later zal worden geargumenteerd, is dit niet essentieel. Merk op dat de "waarde van een variabele" in de zin van ALGOL 60 verkregen wordt door de compositie *REF* ∘ *ACC*: bij een gegeven identifier hoort een vaste naam (in *NAME* ⊂ *VAL*) die in de loop der tijd naar verschillende objecten in *VAL* kan gaan verwijzen.

c) *TTV*: *DEN* → *VAL*.

Een denotatie wordt omgezet in een waarde. Dit is de meest rechtstreekse verbinding tussen *TEXT* en *VAL*. Een bijzonder geval is de routinetekst die een routine levert: *TTV*: *RTEXT* → *ROUT*. De mogelijke verfijning *TTV1*: *RTEXT* × *ENV* → *ROUT* wordt hier eveneens buiten beschouwing gelaten.

d) *SUB*: *ARR* × *IND* → *VAL*.

De subscriptie heeft twee argumenten: een array en een index.
Merk op dat er, net als bij *REF*, geen "logisch" verband is tussen array met index en waarde.

e) *SEL*: *SVAL* × *IDF* → *VAL*.

De selectie heeft eveneens twee argumenten: een gestructureerde waarde en een selector (gemakshalve gelijkgesteld met een identifier).

Het gaat erom deze lijst zoveel mogelijk te vereenvoudigen. In de eerste plaats valt de overeenkomst op tussen *SUB* en *SEL*. Het wezenlijke verschil zit in het tweede argument, dat in het ene geval intern, in het andere extern is. De stap die we moeten doen om tot unificatie, tevens generalisatie, te komen, is: *externe objecten kunnen ook dienen als intern object*, m.a.w., $TEXT \subset VAL$. De winst is dat "gewone" arrays en gestructureerde waarden nu gezien kunnen worden als speciale gevallen van gegeneraliseerde arrays. De generalisatie bestaat hieruit dat alle objecten dienst kunnen doen als index (of selector), of het nu getallen, identifiers, of wat dan ook zijn. Een gelijktijdige winst is dat de functie *TTV* uiterst eenvoudig kan worden, nl. de identiteit. In dat geval wordt $ROUT = RTEXT$. De gegeneraliseerde subscriptie/selectie is inmiddels geworden

$$SUB: ARR \times VAL \to VAL.$$

Vergelijk nu deze functie met

$$REF: NAME \to VAL.$$

Ook dit valt te unificeren, als een eeneenduidige compositie

$$PAIR: ARR \times VAL \to NAME$$

mogelijk is, en wel door $SUB = REF \circ PAIR$. Aangezien het aannemelijk is dat met de functies $VCOMP_i$ wel een eeneenduidige functie $PAIR: VAL \times VAL \to VAL$ valt samen te stellen, ligt het voor de hand te stellen $NAME = VAL$, m.a.w., ieder object kan als naam dienst doen, zodat *REF* gegeneraliseerd wordt tot $REF: VAL \to VAL$.

Van het lijstje a t/m e is nu naast *REF* nog *ACC* over. Aangezien echter $TEXT \subset VAL = NAME$, kan zonder bezwaar voor *ACC* de (identieke) functie *TTV* genomen worden (of, wat qua uitwerking op hetzelfde neerkomt, de functie *REF*).

Recapitulerend:

Als $TEXT \subset VAL$, dan kunnen de functies *REF*, *ACC*, *TTV*, *SUB* en *SEL* (a t/m d) worden uitgedrukt met twee functies:

$$REF: VAL \to VAL \text{ en } PAIR: VAL \times VAL \to VAL.$$

1.2.1. *Syntaxis van gegevensstructuren*

Al eerder is gesteld dat voor de compositieschema's van *TEXT* en *VAL* functies $TCOMP_i$ en $VCOMP_i$ nodig zijn. Nu is al gekozen voor $TEXT \subset VAL$. Door nog een stap verder te gaan, en te stellen $TEXT = VAL$, kan in principe één functie, zeg *COMP*, volstaan. Uiteraard zijn ook nog functies nodig om gegeven een composiet de componenten te kunnen terugvinden. Een bijkomstig voordeel van de keuze $TEXT = VAL$ is dat een syntactische definitiemethode, zoals BNF, kan worden gebruikt om te definiëren hoe interne objecten worden samengesteld. In HET is gekozen voor de compositiewijze van lijsten van het LISP-type:

Syntaxis

<object>::= <basisobject> | <lijst>
<lijst>::= (<open lijst>)
<open lijst>::= <object> <open lijst> | <empty>

In woorden: een object is of een basisobject, of een rij van nul of meer objecten, tezamen tussen haakjes geplaatst. De functie *COMP* is nu gedefiniëerd door:

Als O = <object1> en L = (<open lijst1>), dan is
$COMP(O,L)$ = (<object1> <open lijst1>).

Het is dus vereist dat het tweede argument van *COMP* een lijst is. Dit is natuurlijk geen wezenlijke beperking; zo zou de functie *PAIR* b.v. kunnen worden uitgedrukt door $PAIR(A,I) = COMP(I, COMP(A, ()))$.
Als projectiefuncties dienen *HEAD* en *TAIL*, gedefiniëerd door:

Als L = (<object1> <open lijst1>), dan is
$HEAD(L)$ = <object1> en
$TAIL(L)$ = (<open lijst1>).

Nu blijft nog de keuze van de basisobjecten. Nodig zijn basisobjecten die als identifiers kunnen dienen; om vrije keuze van identifiers mogelijk te maken zijn voor diverse operaties een aantal speciale symbolen apart gehouden:

Syntaxis

<basisobject ::= <woord> | <speciaal symbool>
<woord>::= <letter> | <woord> <letter>
<letter>::= *a* | *b* | *c* | *d* | *e* | *f* | *g* | *h* | *i* | *j* | *k* | *l* | *m* |
n | *o* | *p* | *q* | *r* | *s* | *t* | *u* | *v* | *w* | *x* | *y* | *z* |
0 | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9* | -
<speciaal symbool>::= ; | + | * | ↑ | / | = | ! | < | > | ?

Voorbeelden van woorden zijn *a*, *ab*, *-8-* en *1234*. De aandachtige lezer zal opmerken dat b.v. *(ab)* een dubbelzinnig object is: is dit een lijst van één object *ab*, of van twee objecten *a* en *b*? Om verwarring te vermijden geldt de volgende afspraak: Als in een open-lijst twee woorden onmiddellijk op elkaar volgen, dan wordt bij het weergeven van de open-lijst tussen die woorden een spatie geschreven. Op andere plaatsen mogen naar believen spaties worden ingelast, behalve natuurlijk tussen twee letters van een woord. Hiermee stellen *(ab)* en *(a b)* dus verschillende objecten voor. Terwille van de leesbaarheid van woorden als *1ste-element* is het teken - als letter opgenomen.

1.2.2. *Semantiek van gegevensstructuren*

De semantiek wordt gedefinieerd in termen van een denkbeeldige computer, de HET-machine. Deze is uitgerust met een werkstapel, een geheugen en een programmastapel. De aanpak is vergelijkbaar met die van DIJKSTRA [1]. In een eerste benadering zullen we een HET-programma beschouwen als een open-lijst, dus als een rij van objecten, die van links naar rechts worden "uitgevoerd". Als het achtereenvolgens uitvoeren van de objecten O_1, ..., O_m hetzelfde netto-effect op de HET-machine heeft als het uitvoeren van P_1, ..., P_n, dan zeggen we: O_1 ... O_m is "equivalent" met P_1 ... P_n.

1.2.2.1. *De werkstapel*

Voor een taal als ALGOL 60 of ALGOL 68 geldt dat een aanzienlijk gedeelte van de syntaxis dient om de volgorde aan te geven waarin bewerkingen moeten worden uitgevoerd. Bij de implementatie wordt dan gebruik gemaakt van een werkstapel waar de "anonieme" tussenresultaten worden opgeslagen. Bij HET is deze werkstapel, verder met *WS* aangeduid, expliciet onderdeel van de semantische definitie. Kenmerkend voor de algemeenheid van HET is

dat iedere "cel" van *WS* een willekeurig gecompliceerd object kan bevatten. Een operatie vindt zijn operanden bovenop *WS* en levert in plaats daarvan zijn resultaten af. Het gevolg is dat HET-programma's met een minimum aan syntactische suiker toekunnen, waarbij operaties in reverse Polish (of postfix) vorm worden genoteerd. Het is niet nodig voor *WS* een nieuw soort object in te voeren; een lijst levert al alles wat we nodig hebben.

In het vervolg wordt gebruik gemaakt van de volgende afkortingen:
"*WS*:= *COMP*(*O*, *WS*)" wordt afgekort tot "*PUSH O* → *WS*".
"Het is vereist dat *WS* niet gelijk is aan *()*; *O*:= *HEAD*(*WS*); *WS*:= *TAIL*(*WS*)" wordt afgekort tot "*POP WS* → *O*".*)

Semantiek
Een lijst of woord *O* wordt als volgt uitgevoerd:

PUSH O → *WS*.

Aangezien de top van *WS* beschouwd kan worden als het resultaat van een bewerking, kan gesteld worden dat woorden en lijsten zichzelf als resultaat hebben, ongeveer zoals in ALGOL 68 de yield van de integral-denotation *123* de integer *123* is.

Semantiek
Het speciale-symbool *;* wordt als volgt uitgevoerd:

POP WS → *O*. (*O* wordt niet verder beschouwd!)

Deze operatie is te vergelijken met de voiding coercie van ALGOL 68. (Coercies, d.w.z. impliciete transformaties, zijn syntactische suiker en komen dus in HET niet voor.) Merk op dat hiermee "commentaar" kan worden geplaatst; zo is *hier-begint-het-hoofdprogramma ;* equivalent met.

1.2.2.2. *Het geheugen*

Het geheugen, verder aangeduid met *MEM*, is een representatie van de functie *REF*, en wel de grafiek daarvan. *MEM* is een verzameling, mogelijker-

---

*) De uitdrukking "het is vereist dat ..." wil hier en in het vervolg zeggen dat het vereiste voorwaarde is voor zinvolle toepassing van de definitie; als niet aan het vereiste is voldaan wordt het verdere effect hier niet gedefiniëerd.

wijze leeg, van paren, ieder bestaande uit een "naam" en een "waarde". Namen en waarden zijn (willekeurige) objecten. Als *MEM* het paar *N:V* bevat, dan "verwijst" de naam *N* naar de waarde *V*. Een paar *N:V* kan worden "opgeborgen". Het effect is dat *N:V* aan *MEM* wordt toegevoegd, maar als *N* al naar een waarde *W* verwees, dan wordt *N:W* eerst verwijderd.

Semantiek
Het speciale-symbool ↓ wordt als volgt uitgevoerd:

POP WS → N; *POP WS → V*; *PUSH V → WS*; *N:V* wordt opgeborgen.

De rij objecten *v n* ↓ is vergelijkbaar met de assignation *n:= v* uit ALGOL 68, maar merk op dat in HET de waarde i.p.v. de naam als resultaat overblijft. Dit maakt meervoudige toekenningen mogelijk: *v n3* ↓ *n2* ↓ *n1* ↓ *;* is vergelijkbaar met de void-assignation-coercee *n1:= n2:= n3:= v* in ALGOL 68.

Semantiek
Het speciale-symbool ↑ wordt als volgt uitgevoerd:

POP WS → N; het is vereist dat *N* naar een waarde *V* verwijst;

PUSH V → WS.

Deze operatie, die de functie *REF* vertegenwoordigt, is te vergelijken met de dereferencing coercie van ALGOL 68. Als b.v. de assignation *a:= b* van ALGOL 68 inhoudt dat *b* eenmaal wordt gedereferenced, dan is *b* ↑ *a* ↓ een vergelijkbaar stuk HET.

1.2.2.3. *Compositie en decompositie*

Semantiek
Het speciale-symbool + wordt als volgt uitgevoerd:

POP WS → O; *POP WS → L*; het is vereist dat *L* een lijst is;

PUSH COMP(O,L) → WS.

Voorbeeld: *(a) b* + is equivalent met *(b a)*, en *() a* + is equivalent met *(a)*.

Semantiek
Het speciale-symbool / wordt als volgt uitgevoerd:

POP WS → L; het is vereist dat *L* een lijst is, ongelijk aan *()*;

PUSH TAIL(L) → WS; *PUSH HEAD(L) → WS*.

Voorbeeld: *(b a) /* is equivalent met *(a) b*, en *(a) /* is equivalent met *() a*. Met deze operaties is het mogelijk een lijst tot op het bot uit te kleden. Er moet echter ook een methode zijn om, daar aangeland, de basisobjecten te vergelijken. In principe zijn de tot nu gegeven middelen hiervoor al voldoende: als we afspreken dat *t* en *f* de waarheidswaarden *waar* en *onwaar* voorstellen, dan test het volgende stuk HET de gelijkheid van twee operanden: *x ↓ ; y ↓ ; f x ↑ ↓ ; t y ↑ ↓ ; x ↑ ↑* . Aangezien dit nogal een mond vol is voor een operatie die in de eenvoudigste programma's al voorkomt, terwijl bovendien de gegeven oplossing in de practijk niet voldoet vanwege de neveneffecten op *MEM*, is er voor gekozen de test op gelijkheid als primitief op te nemen. In feite is hier een concessie gedaan.

Semantiek

Het speciale-symbool = wordt als volgt uitgevoerd:

POP WS → V1; *POP WS → V2*

Geval A: *V1* is gelijk aan *V2*:

PUSH t → WS;

Geval B: *V1* is ongelijk aan *V2*:

PUSH f → WS.

Ga na dat ongelijkheid van de bovenste twee WS-cellen wordt getest door *= f =*.

1.3. *Besturingsstructuren*

Het is niet zo gemakkelijk een min of meer theoretische uniformiteit aan te brengen in de besturingsstructuren van hogere programmeertalen. In ieder geval kunnen we de volgende gevallen onderscheiden:

a) het van-links-naar-rechts uitvoeren;
b) routines;
c) conditionele statements;
d) case-statements;

e) iteratie;
f) sprongen;
g) collaterale uitvoering, parallelle uitvoering, coroutines e.d..

Zoals al eerder gesteld is de normale besturing in HET het van-links-naar-rechts uitvoeren. Om dit wat preciezer te beschrijven moet de definitie van de HET-machine verfijnd worden: Naast *WS* bestaat ook een programmastapel, *PS*, waarvoor dezelfde afkortingen gelden als voor *WS*.
Er zijn twee mogelijkheden: *PS* is gelijk aan *()*, en dan is de machine "wachtend", of *PS* is niet gelijk aan *()*, en dan is de machine "bezig".

Als de machine wachtend is, is hij gereed een nieuw object te accepteren. Zodra een nieuw object *O* is "ingevoerd", (b.v. via een toetsenbord,) wordt *PUSH O → PS* gedaan (waardoor de status verandert in bezig).

Als de machine bezig is, wordt de volgende "cyclus" doorlopen:

POP PS → O; *O* wordt uitgevoerd (zie onder Semantiek);

Geval A: *PS* is (nu) gelijk aan *()*:
(de machine is weer wachtend en) de cyclus is voltooid;
Geval B: *PS* is ongelijk aan *()*:
(de machine is nog bezig en) de cyclus wordt (opnieuw) doorlopen.

PS bevat dus steeds de "toekomstige taak" van de machine. Nu is het wezen van het aanroepen van een routine, dat de taak, voorgeschreven door de routine, in de totale taak wordt ingelast. Als we routines voorstellen als een lijst van uit te voeren objecten, b.v. *(P Q R)*, en *PS = (X Y Z)*, dan kan de routine worden aangeroepen door *PS* te vervangen door *(P Q R X Y Z)*.

Semantiek

Het speciale-symbool *!* wordt als volgt uitgevoerd:

POP WS → O;

Geval A: *O* is basisobject:
PUSH O → PS;
Geval B: *O* is een lijst:
O wordt "ingevoegd".

Een lijst *L* wordt als volgt "ingevoegd":
Geval A: *L* is gelijk aan *()*:

(De invoeging is geschied);
Geval B: *L* is ongelijk aan *()*:
TAIL(L) wordt ingevoegd; *PUSH HEAD(L) → PS*.
Merk op dat de operatie *!*, die te vergelijken is met de deproceduring coercie van ALGOL 68, hiermee ook gedefinieerd is voor niet-lijsten. Ga na dat *(O P Q) !* equivalent is met *O P Q*.

Een conditionele statement kan wat zijn semantiek betreft beschouwd worden als een keuze tussen twee stukken programmatekst op grond van een waarheidswaarde:

C: BOOL → TEXT.

Deze vorm voldoet aan het formaat REF: *VAL → VAL*, dat al ter beschikking staat: door de alternatieven als routines onder de namen *t* respectievelijk *f* op te bergen en vervolgens degene op te halen waarnaar het resultaat van de conditie verwijst, wordt het gewenste effect bereikt. Voorbeeld:
if val a = "zzz" then call val z3 else skip fi komt in HET overeen met

```
a + zzz =
    (z3 + !) t + ;
    ()       f + ;
+ !.
```

De generalisatie van deze methode tot willekeurige case-statements is duidelijk.

Voor zowel iteratie als willekeurige sprongen geldt dat er niets mee kan worden uitgedrukt wat niet met routines kan worden uitgedrukt, zodat ze overbodig zijn. Een voorbeeld hoe een routine achtereenvolgens voor de objecten van een lijst kan worden aangeroepen:

```
(arg + () =
    () t + ;
    (arg + /
    behandel + !
    loop + !) f + ;
+ !) loop + !.
```

HET bevat geen enkele mogelijkheid voor collaterale of parallelle uitvoering. Als de operatie *!* niet gedefiniëerd was als het invoegen van de top van *WS*, maar het verwisselen van die top met *PS*, dan zouden coroutines en dergelijke wel in HET uitdrukbaar zijn geweest.

1.4. *Parameters*

Een belangrijk aspect van procedures in hogere programmeertalen is de mogelijkheid parameters door te geven. Dit kan in HET als volgt worden gedaan:

bij de aanroep: stapel eerst de actuele parameters op *WS*;

in de routine: berg één voor één de actuele parameters die op *WS* worden aangetroffen onder de naam van de formele parameters op.

Voorbeeld: een routine die twee namen meekrijgt als actuele parameters, en de waarden waarnaar ze verwijzen omwisselt:

```
(y ↓ ; x ↓ ;
 x ↑ ↑ w ↓ ;
 y ↑ ↑ x ↑ ↓ ;
 w ↑ y ↑ ↓ ;) wissel ↓ ;
```

Deze routine laat het overigens afweten wanneer een van de actuele parameters *x*, *y* of *w* is. Hier komt het principe van de localiteit in het geding. Dit is voor HET geen wezenlijke beperking: het is mogelijk uit een gegeven routine en een gegeven rij van locale namen een nieuwe routine te construeren die bij uitvoering eerst nagaat of zijn locale namen al in gebruik zijn en deze zonodig vervangt door nieuwe namen, vervolgens de locale namen toevoegt aan een verzameling in gebruik zijnde namen, de (eventueel gemodificeerde) oorspronkelijke routine uitvoert en tenslotte de locale namen weer schrapt.

Tot slot van deze sectie een voorbeeld van het construeren van een routine: gegeven is een routine met twee parameters, en we willen hieruit een routine construeren met slechts één parameter, door de tweede te "bevriezen" op een zekere waarde.

Bij voorbeeld: we willen de tweede parameter van de routine waarnaar *wissel* verwijst bevriezen op *xyz*, en het resultaat opbergen onder de naam *wissel-xyz*, zodat dan *pqr wissel-xyz ↑ !* equivalent is met *pqr xyz wissel ↑ !*.

Oplossing: *wissel ↑ xyz ↓ wissel-xyz ↓ ;*.

2. SEMANTIEK VAN DE OVERIGE SPECIALE-SYMBOLEN

Met behulp van het tot nu gebodene is het mogelijk een Turingmachine te simuleren. Dit wil echter niet zeggen dat alle recursieve functies R: *VAL* → *VAL* kunnen worden beschreven in HET; zo is het wel mogelijk een omkeerfunctie voor lijsten te maken die *(aap noot mies)* omkeert tot *(mies noot aap)*, maar niet een voor woorden die *noot* omkeert tot *toon*. De volgende operaties heffen dit gemis op:

Semantiek

Het speciale-symbool < wordt als volgt uitgevoerd:
POP WS → *W*; het is vereist dat *W* een woord is;
zij *L* de lijst waarvan de open-lijst een rij van eenletterige woorden is, een voor elke letter van *W* en wel gelijk aan die letter;
PUSH L → *WS*.

Voorbeeld: *noot* < is equivalent met *(n o o t)*.

Semantiek

Het speciale-symbool > wordt als volgt uitgevoerd:
POP WS → *L*; het is vereist dat *L* een lijst is waarvan de open-lijst een niet-lege rij eenletterige woorden is; zij *W* het woord zodanig dat *W* < equivalent is met *L*; *PUSH W* → *WS*

Voorbeeld: *(t o o n)* > is equivalent met *toon*.

Semantiek

Het speciale-symbool ? wordt als volgt uitgevoerd:
POP WS → *O*;
Geval A: *O* is een woord:
PUSH w → *WS*;
Geval B: *O* is een speciaal-symbool:
PUSH s → *WS*;
Geval C: *O* is een lijst:
PUSH l → *WS*.

Met behulp van deze operatie kan een routine zich ook beschermen tegen eventuele incorrecte actuele parameters, die een niet gedefiniëerde bewerking ten gevolge zouden hebben.

3. HORS D'OEUVRE: ZICHZELF REPRODUCERENDE PROGRAMMA'S

Een aardige opgave is het construeren van programma's die bij uitvoering zichzelf als resultaat afleveren. Dit is volgens de recursiestelling van KLEENE altijd mogelijk voor formalismen waarin alle (partiële) recursieve functies kunnen worden uitgedrukt. Een oplossing voor ALGOL 60 is te vinden in FOKKINGA [11]. In termen van HET is het probleem: geef een programma *P* waarvoor *P !* equivalent is met *P* en eindigt. Een triviale oplossing wordt gegeven door ieder willekeurig woord; zo is *s !* equivalent met *s*. Daarom eisen we ook nog dat *P* een lijst is.

Door stap voor stap een constructief bewijs van de recursiestelling te volgen (VAN EMDE BOAS [12]) wordt de volgende oplossing gevonden:

((s ↓ ; (!) s ↑ + s ↑ +) (s ↓ ; (!) s ↑ + s ↑ +) !).

Zo ingewikkeld hoeft het niet; met weinig moeite kan dit programma vereenvoudigd worden tot

(((s ↓ !) s ↑ +) s ↓ !).

We zullen dit programma stap voor stap volgen, waarbij na iedere stap de open-lijsten van *PS* en *WS* en de inhoud van *MEM* wordt getoond.

PS *((s ↓ !) s ↑ +) s ↓ !*
WS
MEM

PS *s ↓ !*
WS *((s ↓ !) s ↑ +)*
MEM

PS *↓ !*
WS *s ((s ↓ !) s ↑ +)*
MEM

PS *!*
WS *((s ↓ !) s ↑ +)*
MEM *s:((s ↓ !) s ↑ +)*

PS (s ↓ !) s ↑ +
WS
MEM s:((s ↓ !) s ↑ +)

PS s ↑ +
WS (s ↓ !)
MEM s:((s ↓ !) s ↑ +)

PS ↑ +
WS s (s ↓ !)
MEM s:((s ↓ !) s ↑ +)

PS +
WS ((s ↓ !) s ↑ +) (s ↓ !)
MEM s:((s ↓ !) s ↑ +)

PS
WS (((s ↓ !) s ↑ +) s ↓ !)
MEM s:((s ↓ !) s ↑ +)

Door in deze programma's ! overal te vervangen door !! wordt een programma verkregen dat eindeloos doorgaat zich te reproduceren. Het is ook mogelijk een zich explosief reproducerend programma te maken dat WS volschrijft met copieën van zichzelf. Het gegeven korte programma bestaat uit 15 symbolen; aan de lezer wordt overgelaten een nog kortere oplossing te vinden (zo'n oplossing bestaat!).

4. KANTTEKENINGEN

Tot slot van dit verhaal worden enkele opmerkingen gemaakt die een bijdrage kunnen leveren tot de waardering van het experiment HET.

4.1. *HET als taal voor practisch gebruik*

Waarschijnlijk is het niet nodig de lezer ervan te overtuigen dat er voor gebruik in de practijk betere programmeertalen voorhanden zijn. De kritiek die WIRTH op zijn eigen taal EULER heeft geuit [12]: "a high-level Turing machine", gaat voor HET in het kwadraat op. In de eerste plaats hebben de objecten die bij uitvoering verschijnen en verdwijnen een "type", ze stellen een routine voor of een tabel of een getal, maar dat type be-

staat alleen in de geest van de programmeur of kan hoogstens worden afgeleid door analyse van het dynamisch gedrag van het programma. Daarbij komt nog dat de armoede aan syntactische structuur het moeilijk maakt stukken HET als conceptuele eenheid te zien. Weliswaar kan hieraan door zelfdiscipline bij het programmeren enigszins worden tegemoetgekomen, maar in feite betekent deze discipline het zelf opleggen van een syntactische structuur die wezensvreemd is aan HET. Ten slotte: HET is een onverholen invitatie tot het uithalen van "trucjes", slimme kortsluitingen, kortom, het laatste waaraan we in een programmeertaal behoefte hebben. Voor de meeste potentiële gebruikers zal waarschijnlijk nog wel het afschrikwekkendst zijn dat HET niet beschikt over een arsenaal van op toepassingen gerichte standaardoperaties (aritmetiek!).

De typeloosheid heeft HET niet alleen gemeen met EULER, maar ook met GEDANKEN. (Interessant zou zijn na te gaan in hoeverre het mogelijk is bij programma's in deze talen specificaties te geven van de optredende typen, op zo'n manier dat de correctheid van het programma in dit opzicht statisch gecontroleerd kan worden. Een eerste beschouwing van HET in dit opzicht suggereert dat hier reële mogelijkheden liggen.) Verder heeft HET met GEDANKEN gemeen dat deze taal ook ontworpen is als experiment, en wel rond twee gedachten: het "volledigheidsprincipe", wat zoveel inhoudt als semantische orthogonaliteit, en het principe dat namen zelf waarden zijn. Vergelijking van de talen leert wel dat HET niet alleen syntactisch zeer veel simpeler, maar semantisch ook veel "volledíger" is dan GEDANKEN.

Het zou overigens een misverstand zijn te denken dat typeloosheid een voortvloeisel is van semantische orthogonaliteit, getuige ALGOL 68, zeer orthogonaal maar extreem van typen ("modes") voorzien.

4.2. *Implementatie*

Het zal duidelijk zijn dat een HET-implementatie weinig hoofdbreken hoeft te kosten. De voornaamste elementen daarvan zijn: lijstverwerking en een goede zoekalgoritme. Aangezien het blijkt dat het opzoeken (↑) of opbergen (↓) veel vaker voorkomt dan het creëren van objecten (↗), is het verstandig bij de lijsten naast een *HEAD*- en een *TAIL*-veld ook een *VAL*-veld bij te houden, waarin een waarde waarnaar verwezen wordt kan worden opgeborgen. De algemeenheid van de *REF*-functie blijkt dan zeer geringe extra kosten met zich te brengen.

HET heeft de schijn tegen zich dat het een interpretatieve taal is, dus dat de routines zich niet laten compileren, temeer waar deze dynamisch geconstrueerd kunnen worden. De meeste computers laten echter een implementatie van HET toe waarbij de representatie van een object direct uitvoerbare code is, waarop wel zonder veel moeite de compositie- en projectiefuncties toepasbaar zijn. Om de gedachte te schetsen: een composiet wordt normaal gerepresenteerd door een subroutinesprong, maar in het *TAIL*-veld, dat consecutief aansluit op het *HEAD*-veld, door een sprongopdracht naar hetzelfde adres. De lege lijst *()* wordt dan door een terugsprong via een link-stapel voorgesteld.

Onbevredigend is dat HET geen mogelijkheid biedt een eenmaal opgeborgen paar *N:V* te vergeten. Ook al komt het object *N* verder in de gehele machine niet meer voor, de mogelijkheid blijft toch bestaan dat het eens weer geconstrueerd wordt.

4.3. *Leerbaarheid*

Nog sterker dan de gemakkelijke implementeerbaarheid valt de gemakkelijke leerbaarheid van HET op. De vraag dringt zich daardoor op of HET gebruikt zou kunnen worden voor een eerste kennismaking met het verschijnsel programmeertaal. Na de bedenkingen die in 4.1. geuit zijn zal een bevestigend antwoord wellicht als een verrassing komen. Toch hoeven de genoemde bezwaren niet zo zwaar te wegen, als men ook de consequentie overweegt dat HET ongeschikt is voor *practisch* gebruik, maar juist uitgerekend principes als variabelen of routines in hun meest ongegeneerde algemeenheid kan laten zien.

Voor dit doel is de ideale implementatie die, waarbij *PS*, *WS* en *MEM* voortdurend op een beeldscherm worden zichtbaar gemaakt. Als *PS* leeg is, kunnen via een toetsenbord objecten op *PS* worden gestapeld, die dan stante pede worden uitgevoerd. Ter completering dienen nog toetsen met metafuncties voor het in maagdelijke toestand brengen van de machine en voor het stap-voor-stap doorlopen of het tot stilstand brengen van op hol geslagen programma's.

Een bezwaar dat niet is opgeheven is de mate waarin HET zich leent tot codeertrucs. Het kwaad dat hiermee kan worden aangericht is vermoedelijk toch wel een peuleschil vergeleken met de verwoestingen die een taal als BASIC kan aanrichten in het vermogen in abstracte termen over algoritmen te denken bij hen die het ongeluk hebben bij hun eerste schreden op het pad der informatica daaraan te worden blootgesteld.

4.4. *Kan HET niet eenvoudiger?*

Als het moet - ja. Al bij de invoering van = is gesteld dat deze operatie ook kan worden omschreven. Verder kunnen ↓ en *;* tezamen vervangen worden door één nieuwe operatie ⇟, equivalent met ↓ *;* ,want afgezien van neveneffecten is ↓ equivalent met *n* ⇟ *n* ↑ ⇟ *n* ↑ ↑ en *;* equivalent met *n* ⇟.

Een volgende mogelijkheid schuilt in de compositiewijze en de keuze voor basisobjecten. Beschouw:

<object>::= *()* | *(*<object> <object>*)*.

Deze ene regel laat al willekeurig ingewikkelde objecten toe. Anders dan op menselijke gronden is er geen bezwaar tegen het gebruik van objecten als *((()(()()))())* als "identifiers". Deze compositie maakt bovendien de operaties <, > en *?* overbodig.

Als verder nog wordt afgesproken dat enkele objecten speciale semantiek krijgen, en wel

()	voor	↑,
(()())	voor	⇟,
(()(()()))	voor	*!*,
((()())())	voor	/ en
(()(()(()())))	voor	+,

dan is in de resulterende te eenvoudige taal TET een zich reproducerend programma:

```
((((()(()()))())((()())(((()(()()))())(()(((()(()(
)))())(()((()(()(()(()())))())))))))(((()(()()))())((
()())(((()(()()))())(()(((()(()()))())(()((()(()((
)())))()))))))))).
```

LITERATUUR

[1] DIJKSTRA, E.W., *An attempt to unify the constituent concepts of serial program execution*, in Symbolic Languages in Data Processing, Proceedings of an ICC Symposium, Gordon and Breach, 1962, 237-251 (Mathematisch Centrum, MR 46, 1962).

[2] DIJKSTRA, E.W., *On the design of machine independent programming languages*, in R. Goodman (Ed.), Annual review in automatic programming, vol.3, Pergamon Press, 1963, 27-42 (Mathematisch Centrum, MR 34, 1961).

[3] WIJNGAARDEN, A. VAN, *Generalized ALGOL*, in Symbolic Languages in Data Processing, Proceedings of an ICC Symposium, Gordon and Breach, 1962, 409-419, ook in R. Goodman (Ed.), Annual review in automatic programming, vol.3, Pergamon Press, 1963, 17-26.

[4] WIRTH, N. & H. WEBER, *EULER: A generalization of ALGOL and its formal definition*, CACM 9 (1966) 13-25, 89-99.

[5] WIJNGAARDEN, A. VAN, *Orthogonal design and description of a formal language*, Mathematisch Centrum, MR 76, 1965.

[6] WIJNGAARDEN, A. VAN (Ed.), B.J. MAILLOUX, J.E.L. PECK & C.H.A. KOSTER, *Report on the Algorithmic Language ALGOL 68*, Numer. Math. 14 (1969) 79-218 (Mathematisch Centrum, MR 101, 1969).

[7] WIJNGAARDEN, A. VAN, e.a. (Eds.), *Revised Report on the Algorithmic Language ALGOL 68*, Supplement to ALGOL BULLETIN 36, Technical Report TR 74-3, Dept. of Computer Science, Univ. of Alberta, Edmonton, 1974.

[8] POEL, W.L. VAN DER, *De programmeertaal TRAC*, Informatie 13 (1971) 51-58.

[9] REYNOLDS, J.G., *GEDANKEN - A simple typeless language based on the principle of completeness and the reference concept*, CACM 13 (1970) 308-319.

[10] BAUER, F.L., *A course of three lectures on a philosophy of programming*, University of London, 1973.

[11] FOKKINGA, M.M., *A self-reproducing program*, ALGOL BULLETIN 35.4.1 (1973) 24-26.

[12] EMDE BOAS, P. VAN, *Abstract resource-bound classes*. Mathematisch Centrum, 1974.

[13] WIRTH, N., *On the design of programming languages*, in Information Processing 74, 386-393.

MATHEMATICAL CENTRE SYLLABUS 25, 1976, V, 79-91

VERGELIJKING VAN ENKELE PROGRAMMEERTALEN VOOR EEN BASISCURSUS INFORMATICA

R.P. VAN DE RIET

(Vrije Universiteit, Amsterdam)

0. SAMENVATTING

Aan de hand van het probleem van de Torens van Hanoi (met een recursieve en een niet-recursieve oplossing) zullen enkele programmeertalen onderling vergeleken worden.

1. INLEIDING

In een parallel artikel (VAN DE RIET [1]) worden enkele criteria behandeld die betrekking hebben op programmeertalen die gebruikt worden in een basiscursus informatica. In dit parallel artikel is tevens een voorstel opgenomen voor een interactief systeem, BASIS genoemd.

In het voorliggend artikel zullen we concreet, aan de hand van twee oplossingmethoden voor hetzelfde probleem, de Torens van Hanoi, een vergelijking maken tussen een aantal talen. We maken hierbij ruim gebruik van een artikel van PECK [2] over dit onderwerp. PECK geeft een oplossing van de Torens van de Hanoi in de volgende talen: SNOBOL 4, APL, TRAC, LISP 1.5, ALGOL 60, ALGOL W, EULER, ALGOL 68, PL360, XPL, PL/I, PL/C, SPL, PL/I Macro, SIMSCRIPT, PIL, BASIC, FORTRAN, COBOL, 360 Assembler, en 360 Assembler Macros. PECK gebruikt drie verschillende oplossingsmethoden, zodat een vergelijking van de genoemde talen op basis van hetzelfde programma niet volledig mogelijk is. Wij zullen in sectie 2.1 de recursieve versie demonstreren in de talen SNOBOL 4, APL, LISP 1.5., ALGOL 60, ALGOL W, ALGOL 68 en PL/I. In sectie 2.2 zullen we de niet-recursieve versie laten zien in BASIS, (de taal van BASIS is een subtaal van PASCAL), in BASIC en in APL.

2. HET PROBEEM VAN DE TORENS VAN HANOI

Mijns inziens behoort een basiscursus informatica het principe van recursie te behandelen. Voor sommige talen is dit zonder meer mogelijk (bijvoorbeeld ALGOL 60, ALGOL 68, PASCAL, PL/I, APL, SNOBOL, LISP, TRAC), voor andere talen is dit niet mogelijk (bijvoorbeeld BASIC, FORTRAN) en moet een speciale methode gevolgd worden die het mogelijk maakt een recursief algorithme om te schrijven in een iteratief algorithme (= de algorithme bevat slechts loops). Bij dit omschrijven moet een hoeveelheid administratie worden ingevoerd; hierbij komen enkele interessante aspecten van de informatica naar voren.

Als voorbeeld is het probleem van de Torens van Hanoi gekozen en wel om twee redenen:

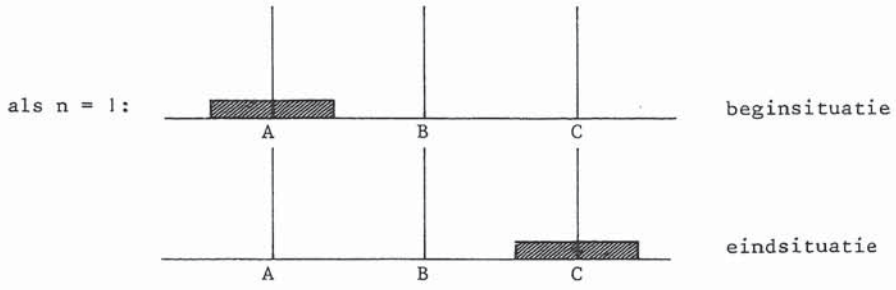
a) De oplossing die het meest voor de hand light is essentieel recursief, in tegenstelling bijvoorbeeld to de berekening van n!. Hierdoor is het een goed voorbeeld voor andere, minder speelse, problemen als bijvoorbeeld quicksort of quad.
b) De formulering en de oplossingmethode kan kort en krachtig zijn.

Voor de volledigheid laten we nu zo'n "kort en krachtige" formulering volgen:

Bij de Tempel van Hanoi stonden drie pennen: A, B en C. Op pen A bevonden zich n (meestal 64) schijven in de vorm van een toren; steeds een kleinere op een grotere gestapeld. De monniken van de bij de Tempel behorende pagode kregen als opdracht de toren van pen A naar pen C te verplaatsen; echter, er mocht slechts één schijf tegelijkertijd getransporteerd worden, bovendien moest die schijf van de top van een toren gehaald worden en op een grotere schijf van een andere toren of op de bodem van een lege pen gedeponeerd worden. De pen B mocht gebruikt worden om tijdelijk een "hulp"toren te bouwen.

2.1. *De recursieve oplossingsmethode*

De recursieve oplossingsmethode kan als volgt in beeld gebracht worden:



dus: "breng schijf 1 van A naar C"

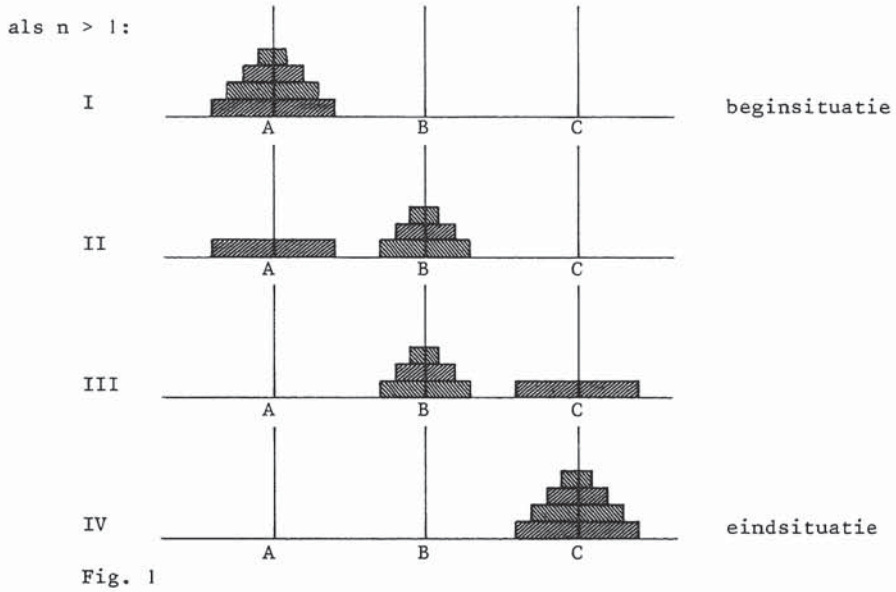


Fig. 1

of in woorden:
los probleem van Hanoi op voor n-1 schijven van A via C naar B,
"breng schijf n van A naar C",
los probleem van Hanoi op voor n-1 schijven van B via A naar C.

De monniken brachten hiermee het volledige probleem terug tot een som van drie problemen die elk eenvoudiger zijn dan het oorspronkelijke probleem.

Het is deze techniek die gedemonstreerd moet worden in een basiscursus informatica.

Om het probleem voor de computer te vertalen behoeven we de formulering slechts zo te veranderen dat de computer het denkwerk moet verrichten, terwijl de monniken het echte werk (het monnikenwerk zogezegd) moeten verrichten. De computer dient namelijk een serie van elementaire opdrachten te printen van de vorm:

"breng schijf 17 van pen B naar A".

In de voorbeeldprogramma's, die nu volgen, laten we een engelse tekst printen; de eenvoudige reden is dat we ze gecopieerd hebben uit PECK [2].

SNOBOL 4

```
      DEFINE("HANOI(N,A,B,C)")
      HANOI(4,"A","B","C")          :(END)
HANOI     LE(N,0)                   :S(RETURN)
      HANOI(N - 1,A,C,B)
      OUTPUT = "MOVE PIECE " N " FROM " A " TO " C "."
      HANOI(N - 1,B,A,C)            :(RETURN)
END
```

ALGOL W

```
BEGIN
PROCEDURE HANOI(INTEGER VALUE N; STRING(1)VALUE A, B, C);
IF N > 0 THEN
  BEGIN HANOI(N-1, A, C, B);
  WRITE("MOVE PIECE",N," FROM ",A," TO ",C,".");
  HANOI(N-1, B, A, C) END;
INTFIELDSIZE := 4;
HANOI(4, "A", "B", "C")
END.
```

ALGOL 68

```
begin priority hanoi = 2;
op hanoi = (int n, [1:3]char t) void:
  ( n > 0 |
    n - 1 hanoi t[1]+t[3]+t[2];
```

```
   out(standout, (n, t[1], t[3]));
   n - 1 hanoi t[2]+t[1]+t[3]
);
format(standout,
   $"MOVE PIECE "2zdx, "FROM"xax, "TO"xa"."$);
4 hanoi "ABC"
end
```

APL

```
       ∇   N HANOI T
[1]    (N≤0)/0
[2]    (N-1)HANOI T[1 3 2]
[3]    'MOVE PIECE ';N;' FROM ';T[1]; ' TO ';T[3];'.'
[4]    (N-1)HANOI T[2 1 3]
       ∇
       4  HANOI  'ABC'
```

LISP 1.5

```
(DEFINE HANOI  (N A B C)  (COND  ((GREATERP N 0)
   (HANOI (SUB1 N) A C B)
   (MAPC  (FUNCTION PRIN1)   (LIST
      'MOVE/ PIECE/  N '/  FROM/  A '/ TO/  C '/.))
   (TERPRI)
   (HANOI  (SUB1 N) B A C)
)))
(HANOI 4  'A 'B 'C)
```

ALGOL 60

```
begin
procedure hanoi (n, a, b, c); value n;
integer n; string a, b, c;
if n > 0 then
   begin hanoi(n-1,a, c, b);
   outstring(1, "MOVE PIECE");
   outinteger(1, n); outstring(1, " FROM ");
   outstring(1, a); outstring(1, " TO ");
   outstring(1, c); outstring(1, "."); sysact(1,14,1);
```

```
        hanoi(n-1, b, a, c)
        end hanoi;
    hanoi(4, "a", "b", "c")
    end
```

PL/I

```
    H:PROC OPTIONS(MAIN);
    DCL HANOI ENTRY(FIXED DECIMAL(3),CHAR(1),CHAR(1),CHAR(1));
       HANOI:PROC(N,A,B,C)RECURSIVE;
       DCL N FIXED DECIMAL(3), (A,B,C)CHAR(1);
       IF N > 0 THEN DO;
          CALL HANOI(N-1,A,C,B);
          PUT FILE(SPRINT)EDIT(' MOVE PIECE ',N,' FROM ',
          A,' TO ',C,'.')(COL(1),A,F(3),5 A);
          CALL HANOI (N-1,B,A,C);END;
       END;
    CALL HANOI(4,'A','B','C');
    END;
```

De outputstatements in ALGOL 60 zijn door hun veelheid weinig fraai, terwijl de declaraties in PL/I opvallen door hun uitgebreidheid. Verder zijn de programma's wel ongeveer equivalent te noemen. De conclusie is dat het proceduremechanisme, dat ongeveer het enige mechanisme is dat in deze versie van de oplossing van het probleem gebruikt wordt, voor de genoemde talen ongeveer gelijk is.

2.2. *De niet-recursive oplossingsmethode*

Het is van belang dat studenten in een basiscursus informatica iets leren over de achtergronden van recursieve procedures. Daarom gaan we nu in het kort een niet-recursieve methode schetsen voor de oplossing van het probleem van de Torens van Hanoi. Deze methode laat verder een interessante techniek uit de informatica zien. Tenslotte, voor sommige talen (BASIC, FORTRAN) is de recursieve methode niet toepasbaar; dan zal een niet-recursieve methode gevolgd moeten worden.

Een doorzichtige, niet van een programmeertaal afhankelijke, redenering is op te zetten waaruit tamelijk eenvoudig een niet-recursief algorithme volgt.

We veronderstellen dat de pagode een vader abt, 64 monniken en één

novice heeft. Elk van de monniken heeft één superieur en één ondergeschikte. De superieur van de monnik van de hoogste rang is de vader abt en de ondergeschikte van de monnik van de laagste rang is de novice.

Vader abt geeft nu aan zijn ondergeschikte, die de monnik van de hoogste rang is, de opdracht om de 64 schijven van pen A via pen B naar pen C te brengen.

Onze monnik roept zijn ondergeschikte en verzoekt hem om de n-1 bovenste schijven van pen A m.b.v. pen C naar pen B te brengen en als hij hiermee klaar is dit te melden; daarna gaat onze monnik mediteren.

Als onze monnik wordt medegedeeld dat de opdracht voltooid is, brengt hij zelf schijf n van pen A naar pen C. Daarna roept hij wederom zijn ondergeschikte en verzoekt hem om de n-1 schijven van pen B m.b.v. pen A naar pen C te brengen en als hij hiermee klaar is dit te melden; daarna gaat onze monnik weer mediteren. Als onze monnik wordt medegedeeld dat de opdracht voltooid is, deelt hij zijn superieur (vader abt) mee dat alles gereed is.

De ondergeschikte monnik gaat op overeenkomstige wijze aan de slag: hij zet zijn ondergeschikte aan het werk. Dit herhaalt zich tot we bij de novice zijn gekomen; de taak van de novice beperkt zich slechts tot het mededelen dat hij klaar is (dit is in overeenstemming met de CAO van de novice: hij mag slechts leren en kijken).

Bij de simulatie van bovengeschetst proces, is het nodig te realiseren dat elke monnik zich in een aantal toestanden kan bevinden welke gescheiden worden door de volgende momenten:

1. hij wordt opgeroepen een karweitje uit te voeren en geeft zijn ondergeschikte een opdracht.
2. hem wordt door zijn ondergeschikte medegedeeld dat de opdracht voltooid is; waarna hij zelf een schijf gaat transporteren; waarna hij zijn ondergeschikte een opdracht geeft.
3. hem wordt ten tweede male medegedeeld dat de opdracht voltooid is.

De drie toestanden: voor moment 1, na moment 1 en voor moment 2 en na moment 2 en voor moment 3, geven we weer door de getallen 1, 2 en 3.

Wij representeren deze toestand in een integer array toestand [0:64]; "toestand [i] = 1" betekent: toestand van monnik i is 1. Toestand [0] geeft de toestand van de novice weer. De opdrachten voor de diverse monniken worden in de character arrays: VAN, VIA en NAAR [0:64] gezet.

De belangrijke idee is verder dat er steeds maar één monnik tegelijkertijd bezig is, zodat we met de computer steeds één monnik kunnen simuleren.

De betreffende monnik wordt aangewezen door de variabele n. Vader abt kan dan gerepresenteerd worden door 65.

Het geven van een opdracht gebeurt door in de arrays VAN, VIA en NAAR de opdracht te zetten, door de TOESTAND = 1 te maken en door vervolgens "n:= n-1" te doen en de computer een andere monnik te laten simuleren.

Het melden dat de opdracht gereed is gebeurt door simpelweg "n:= n+1" uit te voeren.

2.2.1. *BASIS en PASCAL*

In BASIS en in PASCAL ziet de algorithme er als volgt uit:

```
const m = 64;
type  iar = array[0..m] of integer;
      car = array[0..m] of char;
var   n: integer; toestand: iar; van, via, naar: car;
procedure hanoi;
begin n:= m; toestand[n]:= 1;
      van[n]:= 'a'; via[n]:= 'b'; naar [n]:= 'c';
      while n ≤ m do
      if n > 0 then
      case toestand[n] of
      1: begin van[n-1]:= van[n];
               via[n-1]:= naar[n];
               naar[n-1]:= via[n];
               toestand[n-1]:= 1;
               toestand[n]:= 2;
               n:= n-1
         end;
      2: begin writeln(' breng schijf ',n,' van ',van[n],
                       ' naar ', naar[n],'.');
               van[n-1]:= via[n];
               via[n-1]:= van[n];
               naar[n-1]:= naar[n];
               toestand[n-1]:= 1;
               toestand[n]:= 3;
               n:= n-1
         end;
```

```
3: n:= n+1;
   end
   else n:= n+1
   end {hanoi};
   hanoi; {in BASIS kennen we geen programma}.
```

2.2.2. *BASIC*

We zullen het bovenstaande programma in BASIC opschrijven. Hierbij moet opgemerkt worden dat er feitelijk eerst een stroomdiagram gemaakt had moeten worden. Een eerste probleem waar we tegen aan lopen is het feit dat arrays in BASIC tellen vanaf 1 en niet vanaf 0, zodat we een andere oplossing voor de novice moeten zoeken. Dit bereiken we door te zorgen dat n niet gelijk aan nul kan worden.

Een tweede probleem is dat je in BASIC niet simpelweg met characters kunt werken (In Dartmouth BASIC bestaat wel een string <--> integer conversie m.b.v. de $ conventie).

Een derde probleem is dat de keuze van identifiers te beperkt is. Wij zullen de volgende namen gebruiken:

N voor n,
V voor van,
V1 voor via,
N1 voor naar,
T voor toestand.

```
1000    DIM V(64), V1(64), N1(64), T(64)
1010    LET N = 64
1020    LET T(N) = 1
1030    LET V(N) = 1
1040    LET V1(N) = 2
1050    LET N1(N) = 3
1060    REMARK HIER BEGINT WHILE
1070    IF N > 64 THEN 2000
1080    REMARK HIER BEGINT CASE
1090    IF T(N) = 3 THEN 1700
1100    IF T(N) = 2 THEN 1400
1110    REMARK DIT IS CASE 1. N = 1 IS BIJZONDER GEVAL.
1120    LET T(N) = 2
1130    IF N = 1 THEN 1400
```

```
1140        LET V(N-1) = V(N)
1150        LET V1(N-1) = N1(N)
1160        LET N1(N-1) = V1(N)
1170        LET T(N-1) = 1
1180        LET N = N-1
1190        GOTO 1070
1200        REMARK DIT IS CASE 2. LET OP REGELNUMMERS.
1400        PRINT "BRENG SCHIJF"; N; "VAN"
1410        LET K = V(N)
1420        GOSUB 3000
1430        PRINT "NAAR"
1440        LET K = N1(N)
1450        GOSUB 3000
1460        PRINT "."
1470        LET T(N) = 3
1480        IF N = THEN 1700
1490        LET V(N-1) = V1(N)
1500        LET V1(N-1) = V(N)
1510        LET N1(N-1) = N1(N)
1520        LET T(N-1) = 1
1530        LET N = N-1
1540        GOTO 1070
1550        REMARK DIT IS CASE 3.
1700        LET N = N+1
1710        GOTO 1070
2000        STOP
3000        REMARK SUBROUTINE VOOR HET PRINTEN.
3010        REMARK VAN EEN KARAKTER.
3020        IF K > 1 THEN 3050
3030        PRINT "A"
3040        RETURN
3050        IF K > 2 THEN 3080
3060        PRINT "B"
3070        RETURN
3080        PRINT "C"
3090        RETURN
```

Bezien we bovenstaand BASIC programma dan valt het ons niet moeilijk

een groot aantal in het oog springende nadelen op te sommen:

1. De reeds genoemde te korte identifiers.
2. Het gebrek aan syntactische structuur zodat het lastig is de flow-of-control te herkennen.
3. Het invullen van regelnummers bij voorwaartse sprongen (of uitstellen zoals in de print subroutine; of een schatting maken).
4. Naamgeving van een subroutine is onmogelijk.
5. Er is geen parametermechanisme bij subroutines.
6. Het is (vrijwel) onmogelijk een BASIC programma op te stellen zonder een stroomdiagram en zonder een legende voor de betekenis van identifiers.
7. Het is soms vervelend dat arrays vanaf 1 moeten gaan tellen.
8. Het printen van een regel met verschillende onderdelen is lastig; in bovenstaand programma wordt de tekst over 5 regels verdeeld.

Positive punten zijn dat BASIC op vrijwel alle computers, maxi en mini, beschikbaar is en dat het tamelijk snel te leren is. Mijn opvatting is dat BASIC een veredeld soort machine programmeren is; het is jammer dat het onder de interactieve programmeertalen een plaats aan het verwerven is zoals FORTRAN die heeft onder de hogere programmeertalen.

2.2.3. *APL*

Ook in de zeer populaire taal APL [3,4] willen we de niet-recursieve versie van de Torens van Hanoi laten zien.

```
        ∇ PROGRAMMA
[1]     M <- (N <- AANTAL <- 64), 3
[2]     ADM <- M ρ'ABC'
[3]     TOEST <- AANTAL ρ1
[4]  MELD: -> (N > AANTAL)/0
[5]     -> (TOEST[N] = 3)/CASE3
[6]     -> (TOEST[N] = 2)/CASE2
[7]  CASE1: -> (N = 1)/CASE2
[8]     TOEST[N] <- 2
[9]     ADM[N <- N-1;] <- ADM[N; 1 3 2]
[10]    -> MELD
[11] CASE2: 'BRENG SCHIJF';N;'VAN';ADM[N;1];'NAAR';ADM[N;3];'.'
[12]    -> (N = 1)/CASE3
[13]    TOEST[N] <- 3
[14]    ADM[N <- N-1;] <- ADM[N; 2 1 3]
```

```
[15]     -> MELD
[16] CASE3: TOEST[¯1 + N <- N+1] <- 1
[17]     -> MELD
         ∇
         PROGRAMMA
```

Het bovenstaande programma* heeft enkele belangrijke voordelen boven het BASIC programma: goede identifiers, karakter-manipulatie; een aantal bezwaren van BASIC gelden ook hier: geen zichtbare flow-of-control.

In bovenstaand programma hebben we twee typische APL eigenschappen gebruikt, namelijk op regel 9 en regel 14 waar we ten eerste aan een gehele rij van een array de waarde van een andere gepermuteerde (1 3 2) rij van datzelfde array in één keer assigneren terwijl we ten tweede en passant de waarde van N verlagen. We kunnen het programma nog meer "verfraaien" door te bedenken dat wat in regel 14 beschreven staat ook bereikt kan worden door de rij ADM[N-1;] twee plaatsen cyclisch te roteren:

```
[14] ADM[N;] <- 2 ϕ ADM[N <- N - 1]
```

Helaas wordt het programma er door deze trucjes bepaald niet helderder op; integendeel. Ik zou er zelfs voor pleiten om in het geheel niet deze trucjes te behandelen in een basiscursus. Veel van de argumenten om APL te gebruiken gaan dan echter verloren.

3. CONCLUSIE

Het is toch erg droevig te moeten concluderen dat de twee talen BASIC en APL, die zo intensief voor het basisonderwijs in de informatica worden gebruikt, zo weinig voor dit doel geschikt zijn. We kunnen nog refereren naar een interessant artikel van SIME, GREEN & GUEST [5] waarin het verschil werd onderzocht tussen programmeerresultaten met geneste if-then-else statements en met conditionele jumps. Er werd met groepjes onervaren (α - γ) studenten gewerkt. Het is opmerkelijk dat de studenten met de geneste if-then-else statements tot aanzienlijk betere prestaties kwamen: zowel in de

---

*) Dit programma werd in iets gewijzigde vorm gedraaid door A.A. Verrijn Stuart en J.V.M. van der Grinten (Rijks Universiteit, Leiden), waarvoor onze dank.

zin van afvalpercentage als in de tijd die nodig was om de gestelde problemen correct op te lossen.

Enkele psychologen van de Vrije Universiteit zijn ook bezig met plannen op ditzelfde gebied. Dit werk bouwt voort op experimenten met groepjes studenten die een eenvoudig programmeertaaltje leerden dat op de PDP8 werd geïmplementeerd (zie VAN DER VEER & OTTEVANGERS [6]).

Het is te hopen dat onderzoekingen op dit gebied invloed zullen hebben op toekomstige programmeertalen.

REFERENTIES

[1] VAN DE RIET, R.P., *Some Criteria for elementary Programming Languages*, Syllabus in dit colloquium.

[2] PECK, J.E.L., *Comparison of Languages*, in W.M. Turski (ed.) Programming Teaching Techniques. North Holland Publ. Cy., 1973, 43-60.

[3] GILMAN, L. & A.J. ROSE, *APL an interactive approach*, Wiley, 2nd edition 1974.

[4] MOCK, T.J. & M.A. VASARHELYI, *APL for management*, Wiley - Beckers and Hayes, 1972.

[5] SIME, M.E., T.R.G. GREEN & D.J. GUEST, *Psychological Evaluation of Two Conditional Constructions used in Computer Languages*, Int. J. Man-Machine Studies 5 (1973), 105-113.

[6] VAN DER VEER, G.C. & D.C. OTTEVANGERS, *Problem Solving by Programming*, Afdeling Psychologie, Vrije Universiteit.

MATHEMATICAL CENTRE SYLLABUS 25, 1976, VI, 93-119

SOME CRITERIA FOR ELEMENTARY PROGRAMMING LANGUAGES

R.P. VAN DE RIET

(Vrije Universiteit, Amsterdam)

0. ABSTRACT

In this paper we will draw from some general considerations a number of criteria for programming languages used in elementary (introductory) programming courses. With these criteria some existing and widely used programming languages will be evaluated. A proposal for an interactive system (BASIS) based on a subset of PASCAL will be made.

1. INTRODUCTION

In our Informatics department, a subdivision of the Mathematics department, our introductory informatics courses have been based on ALGOL 60 with batch-oriented exercises.

In the near future we will have to make two important changes in this situation: first we will have a PDP11/45 with a dozen terminals available, so that the exercises can be done on-line in an interactive mode, and second we will abandon ALGOL 60; it being a somewhat old-fashioned language, not so well suited for on-line work, for which our CDC compiler is ultra slow.

We have studied the literature, performed some experiments and done some thinking. The results are presented in this paper since they may be of general interest. To place our study in the proper context, it is necessary to have some general considerations (section 2) from which detailed criteria can be drawn (section 3). These criteria are used (section 4) to evaluate some existing interactive languages and some high-level languages for possible use. A description is given (section 5) of an interactive system, called BASIS, which is now being implemented in our department. Finally, (section 6) BASIS is evaluated using the design criteria of section 3.

Although we have tried to study the most relevant literature on interactive languages, no attempt has been made to study all the literature. We must refer to the SIGPLAN Bulletin where a stream of publications is produced monthly. The following publications were very helpful:

- Proceedings of the IFIP conference on computers and Education, 1970 (SCHEEPMAKER [1])
- Proceedings of the IFIP TC-2 conference in Zakopane, 1972(TURSKI [2])
- Proceedings of the 4-th symposium on Computer Science Education, 1974(SIGCSE [3])

2. GENERAL CONSIDERATIONS

In the following subsections we will discuss:

1. The intention of the course.
2. Organizational aspects.
3. Fundamental aspects.
4. Potential computing power.
5. Effectiveness.
6. Methodological aspects.
7. Teaching aspects.
8. Availability of textbooks.
9. Widespread use.
10. Financial aspects.

2.1. *The intention of the course*

The course is meant as the first introduction to informatics, including programming exercises. The students come from different disciplines (e.g. Mathematics, Physics, Biology, Chemistry, Geology, but also from the humanities and economics departments). One solution for the different disciplines is to give different courses (which is done in many places).
Major disadvantages are: first, inefficient use of educational man- and machine-power, second, insurmountable linguistic barriers between the disciplines so that it is not possible for the interested non-informatics student to follow an advanced programming course, and third, the danger that non-informatics students will not profit from new developments in computer science.

2.2. *Organizational aspects*

Due to the diverse interests of the students, there is a strong need for a short (two or three weeks) introductory course with the emphasis on "Use of computer for potential problem solving" in which there is an intensive contact between the student and the computer in such a way that the computer behaves itself at its best. The communication with the computer (including the programming language) should be as simple as possible.
After this course, it must be possible to organize different courses to fulfill the different needs. For example:

- special programming language courses (e.g. PL/I or FORTRAN),
- a course on how to use the "big" computer,
- special application oriented courses (e.g. simulation),
- advanced programming courses.

2.3. *Fundamental aspects*

Most important, the course must be a sound introduction to programming. About the question of what is programming, a lot of people give different opinions; see, for example, the two panel discussions "What is computer science or informatics" and "Basic concepts in programming" of the Zakopane conference (TURSKI [2]), which reveal that the situation is not much clearer than five years earlier as described in "University Education in Computer Science" (FINERMAN [4]).

Our definition of programming is that it comprises the activity of using a computer to solve a problem, where the emphasis is on problem solving.

For our programming course this means that the course should be student-oriented instead of computer-oriented; this means that we must be very careful with the argument that "something is difficult to implement" (although, of course, the course should be realizable).

The issue of problem solving is a very general one; POLYA [5], for example, has made some interesting remarks on this subject which are being commented by several people (see e.g. GRIES [6]).
Happily enough, we do not have to cope with the general problem of problem solving; we may narrow the scope of our subject to those problems which are solvable by computer. Since our subject is an introductory course we may narrow our scope even further to those problems which involve simple data types and simple operations.
The techniques to be learned, however, should be of general use. Such techniques are for example:

1. Define the problem precisely;
2. Split the problem in subproblems;
3. Concentrate on subproblems;
4. Assemble the solutions of the subproblems;
5. Try to avoid making errors;
6. Devise appropriate tests.

In particular the "Divide and Conquer" technique should be taught in the course and the programming language should facilitate to use this technique. This gives at least two criteria:

1. A simple but powerful procedure mechanism should be available to facilitate "division in the large".
2. Indentation of the individual statements must be provided in the listing of the program, to facilitate "division in the small".

In an on-line interactive system, procedures can be developed and tested independently of each other. Therefore interactiveness is a third criterium derived from the above considerations. We know that other people (e.g. HOARE [7]) are of the opposite opinion: they say that program composition should be done behind the desk by thinking, not by doing. This might be true for large programs, it certainly is not true for the small programs of the introductory course where the thinking part is small compared with the doing part, i.e. removing syntatical errors etc. This has also been ascertained by GALLER [8].

2.4. *Potential computing power*

In order to get a glimpse of the potential computing (thinking) power of the computer, the novice student should learn the possibilities of applying repetition processes on data of simple but adequate type. These types should be numerical as well as textual in order to show mathematical models as well as modern text-processing techniques.

Also the Boolean type is a necessary attribute in the first place to facilitate the construction of repetition (thus making goto's superfluous) in the second place to show the indispensable logic behind programming and in the third place to be able to explain very rudimentarily the functioning of a computer.

Apart from the data types, operations on numbers and booleans are needed.

2.5. *Effectiveness*

It is not possible to learn programming without learning a language; learning the language should be done, however, in the shortest time possible and with a minimum of effort from the part of the instructors, not to mention the students. So, not only should the language be simple, the communication with the computer must also be effective. The computer should give the assistance in the form of comprehensive error signalling. It is also the computer which must provide for rapid and easy text editing.

2.6. *Methodological aspects*

Although the programs made during the course are necessarily small, it is of fundamental importance that they are well-structured. This is not only important for the student who will make large programs later on as a professional; this is of equal importance for those students who will not make frequent use of the computer later on. The reason is that well-structured programming is closely linked to well-structured problem solving.

Furthermore, no student should get the idea that programming is similar to decoding cryptograms or hieroglyphics. Emphasis should therefore be given to readability, documentability, well-structuredness, and so forth.

Another methodological aspect is the proof of correctness (or at least proof of credibility). Programs should be developed in such a way that correctness need not be built in afterwards (which is almost always impossible) but is built in during the development of the program. To this end techniques as stepwise refinement or decomposition should be taught and the language should not be a hindrance but a help. This again pleas for procedures.

A special debugging course, such as suggested by MATHIS [9], should not be necessary.

A suggestion of BASILI & TURNER [10] to teach "structured programming" as a second course can only be understood if one takes into account other than scientific arguments which are used to stick to FORTRAN. Our experience with people who have first learned FORTRAN is that they remain using goto's even if this is not necessary, probably with the argument: "it is so easy".

2.7. *Teaching aspects*

The examples which are used in the course as demonstration material should arouse the interest of the student. Therefore, the language of the

course should be rich enough to facilitate in a natural way the exposition of quite different examples. Sorting techniques, text manipulations, some elementary numerical mathematics, gameplaying and simulations should all be demonstrable.

Another facet is the use of Computer Aided Instruction techniques (see e.g. NIEVERGELT, REINGOLD & WILCOX [11]). Our course should be such that the computer can take over more and more of the teaching load as CAI develops in the future. This is another good reason for interactiveness.

2.8. *Availability of textbooks*

It is very important that good textbooks are available, which on the one hand give a good introduction to the language, and which on the other hand give examples and exercises which show the rich spectrum of computer applications.

2.9. *Widespread use*

The issue of widespread use is always raised to oppose the introduction of new things; as someone said: "compatibility = stagnation".
If new, but better, ideas will get a chance to survive, it is necessary that we will not use the argument of widespread use.

2.10. *Financial aspects*

In the past, batch-oriented programming was frequently argumented with financial computations. With the coming of low-cost terminals and low-cost mini-computers, one can hardly use financial arguments nowadays for maintaining batch-oriented systems. A datalink between a mini-computer and a large computer can be very helpful, as we will show in section 5.

3. DETAILED CRITERIA

From the preceding section we now make a list of criteria to be satisfied by the language of the introductory course. These follow more or less directly from the general considerations.

1. simple,
2. general,
3. well-structured,
4. good program control structures,

5. good basic data types,
6. interactive,
7. versatile,
8. sound basis for other programming language courses,
9. sound basis for application courses,
10. sound basis for advanced programming courses.

3.1. *Simplicity*

This property is necessary, not only from the consideration that the students may have a weak "exact" background, but also since the time for the course is very limited. Simplicity means that the number of concepts is small and that these concepts are more or less orthogonal (i.e. there are but a few exception rules). One should however be careful with using this criterion as the most simple computing machine and programming language in the Turing Machine, which we certainly will not advocate for our course.

3.2. *Generality*

The course should be suited for the "general" student, irrespective of his background and discipline. Also the concepts learned in the course should be general. It would be of little general value to teach e.g. the treatment of arithmetic in SNOBOL, or the own array in ALGOL 60. In this context, a word of caution due to N. WIRTH [12] is in order. He discusses simplicity and generality. Talking on his experience with Euler he said that the combination of generality and simplicity lead to "programs that through their very conciseness and lack of redundancy elude our limited intellectual grasp" making programming "high-level Turing Machine" programming.

3.3. *Well-structured*

The discussion on fundamental concepts as well as on methodological aspects revealed the need for well-structuredness in programming. We mean with this not only goto-less programming, although this is an important subject (see e.g. VAN DE RIET [13], LECARME [14] (in this paper 22 publications on goto's are referred to) and KNUTH [15]) but much more the style of programming which leads to readable, documentable and correct programs which are easy to maintain. The use of global and local variables, for example, must also be given much consideration.
It is tempting to write a dozen pages or so on this subject; we will not do so but only give two citations of O. LECARME [16]:

"The best means to obtain error-free programs is to manage to not put them into it"
and
"We must teach structured programming to everybody (even to those people who seem to be unrecoverable) immediately and as soon as at the beginning of programming teaching".

3.4. *Good program control structures*

This issue has everything to do with the preceding criterium. Conditional statements, conditional repetitions and recursion should be enough for the problems in the introductory course.

It is very unclear and contradictory on the one hand to say that "for i:= 1 step 1 until 100 do S" means that S is executed a hundred times with the values of i being 1,2,...,100, and on the other hand to introduce the goto, providing the possibility of an early exit (we do not even mention the very bad habit to change the value of i within S). The only good way is to use the while - do or repeat - until mechanism together with boolean variables.

```
" for i:= 1 step 1 until 100 do
 begin S1; if bool expr then goto L; S2 end;
 S3; goto H; L:S4; H:"
```

should be coded as

```
" i:= 1; cont:= true;
 while i ≤ 100 and cont do
 begin S1; cont:= bool expr;
     if cont then begin S2; i:= i+1 end;
 end;
 if cont then S4 else S3;"
```

This construction shows already from the outside that we have to do with a conditional repetition. Only when one makes large programs, e.g. a compiler, it becomes necessary to have error-exit-like jumps.
KNUTH [15] describes a loop construction, due to Zahn, which for the example above comes down to:

```
" loop until list empty or end condition:
      i:= i+1; if i ≥ 100 then list empty;
      S1; if bool expr then end condition; S2
      repeat;
      then list empty => S3;
           end condition => S4;
      fi;"
```

Although this construction is very general and also efficiently to implement indeed, we do not like it and prefer the use of a boolean variable.

3.5. *Good basic data types*

From the above discussions it follows that we need as data types: number, boolean and character. It is furthermore necessary to use arrays. For example, the array of booleans can be used to explain number representation, the array of numbers is used to do sorting or linear algebra and an array of characters is just a string used in text manipulation.

3.6. *Interactive*

From several considerations (Fundamental aspects, Effectiveness, Teaching aspects) it became clear that an interactive system is most desirable. We will emphasize now the following not yet mentioned methodological aspect: The student comes in close contact with the computer which must act as a willing computing slave; every legal command is instantaneously executed; every illegal command is immediately signalled.

Students become much more involved in the course while typing at a terminal than, while waiting in a queue near the batch station.

As has been said already (section 2.10) financial cost does not need to be a hindrance for using an interactive system.

3.7. *Versatile*

The course should lay a good foundation for advanced programming courses as well as for application courses.

3.8. *Sound basis for programming language courses*

It should be relatively easy to use this course as a basis for other programming language courses. Therefore the following concepts should in any case be included in the course:

1. value and name of a variable,
2. assignment,
3. procedure calls (recursion included),
4. formal - actual parameters,
5. local and global variables,
6. scope of an identifier,
7. structuring the algorithm,
8. syntax and semantics.

3.9. *Sound basis for application courses*

An advanced course for linguists or for biologists should profit much from our course. One way to achieve this is to choose a sub-language of a powerful high-level-general-purpose-language as the language of our system. In this high-level language adequate data structures and file handling should be available.

3.10. *Sound basis for advanced programming courses*

A course on data structures, on compilers, on artificial intelligence or on operating systems should rest on our course. Again the best way is to use a sub-language of an existing powerful high-level language.

4. EXISTING SYSTEMS

We will very shortly review some existing interactive systems and some existing high-level languages. The latter ones for possible use in our interactive system as language.

4.1. *Some existing interactive systems*

We will review the following systems:

1. BASIC,
2. APL,
3. LSE,
4. TEACH,
5. JOSS,
6. A subset of PASCAL.

4.1.1. *BASIC*

The only positive remarks we can make about BASIC are that it is simple and interactive, that a lot of books are written about BASIC of which the original book of KEMENY & KURTZ [17] can be recommended due to its rich set of examples, and that it is a popular language.

In a comparison PECK [18] demonstrates one program written in 21 languages. About the BASIC program he remarks that "it is curious that BASIC is a popular language". The short identifiers, the absence of procedures and adequate control structure make BASIC completely unsuited for our purposes. LECARME [14] says that BASIC is only suited for learning BASIC pro-

gramming, not for learning basic programming.

4.1.2. *APL*

People are either strong APL fan's or they are fervent opponents. As a proponent of APL, ADAMS [19] claims that APL is suited for structured programming due to its powerful data types and operators. For example, "(+ / ×) : X" is an obviously correct program for calculating the mean of a vector X. The number of symbols in this program is so small that correctness can be seen immediately, whereas the correctness of:

```
" i:= 1; s:= 0;
  while i < n do
  begin s:= s+X[i]; i:= i+1 end;
  s:= s/n"
```

is much more difficult to see, says Adams. It is true that, if one has learned all the special operators, one can write programs if the application lends itself for the operators. However, if the application is non-standard, one could come in real problems not knowing which powerful operators to use.

In our opinion the powerful generators hide too much underlying programming. As regards control structure, not much more is offered than the goto. Another danger lies in the ability of some programmers to write so-called one-liners, i.e. a sequence of symbols filling no more than one line which stand for an APL program, which, when written in a more conventional language, would need one or more pages.
With respect to most of the criteria of the preceding section, APL is not suited. The only one that it evidently satisfies is interactiveness.

4.1.3. *LSE*

The language LSE, developed by HEBENSTREIT [20], is an ALGOL-like interactive language provided with character and string handling and file management. This language comes much closer to our criteria than the preceding one. Due to the use of line numbers as labels, which lead to easy use of goto's, and the absentness of the boolean data type, this language does not fulfill the criterium for well-structuredness, although it has if-then-else and a general for-statement. Its facilities for interaction, such as listing, interruption and step-by-step execution, are very attractive. It is available on two French mini-computers: MITRA15 and T1600.

4.1.4. *TEACH*

A very interesting attempt towards a self-supporting interactive course for an introduction in computer science is the system TEACH developed by FENICHEL, WEIZENBAUM and YOCHELSON [21, 22]. Designed some five years ago at MIT, it has been used there for some years.

We will quote some principles on which TEACH was grounded: "It is more important that programs be readable than that they be writable", "So that students may efficiently find their errors, these must be localized to the narrowest possible context", "Languages for students use must be based on a small number of clear ideas" and finally: "A language for use by students should facilitate good programming practice. In particular, the language may be designed so as to make certain types of bad practice impossible". We refer to the well-written articles for more information about this system. We are happy that a listing of the complete system is in our posession so that we can profit from it.

4.1.5. *JOSS*

One of the very successful and early attempts of an interactive system. It has been designed and developed in 1964 by SHAW [23], ELLIS, NEHANA, NEWELL and UNCAPHER. Its interactive facilities in the sense of direct execution of commands have influenced our own design. As far as program control structure is concerned, the language is poor (nested DO loops and conditional goto's).

4.1.6. *A subset of PASCAL*

SCHILD [24] gives an interesting account of the general considerations about interactive structured programming. He answers questions as: why should the language be structured (being a co-worker of Wirth, he has no problem with this question) and why should it be interactive. The paper is somewhat vague about which subset of PASCAL has been chosen by him and which sort of interaction has been provided (it is really conversational or does it give comprehensive error messages only).

4.2. *Existing higher-level languages*

It is necessary to review briefly some of the well-known high-level languages for possible use in our interactive system. These languages are:

1. FORTRAN,
2. PL/I,
3. ALGOL 60,
4. ALGOL 68,
5. PASCAL.

For other comments on the choice of a language, we refer in particular to LECARME [14].

4.2.1. *FORTRAN*

Despite the fact that FORTRAN is very popular (in particular among physicists and chemists), that in the USA 70 percent of the students get FORTRAN in their introductory courses and that this language for most computers gives very rapidly running programs, it is not suited for our purposes since it does satisfy none of our criteria.

It is very deplorable that RALSTON [25], as president of the ACM in an invited lecture at the International Computing Symposium in Davos, 1973, suggest to consider FORTRAN as an ever-existing language, due to its economic and psychological inertia, and to direct future programming language developments in extending FORTRAN.

The reply of LECARME [26] in the Computing Reviews makes clear that Ralston has been fundamentally wrong.

4.2.2. *PL/I*

CONWAY & GRIES [27] use a subset of PL/I, namely PL/C, to teach programming in a structured way. HOLT & WORTMAN [28] designed 6 nested sublanguages of PL/I namely SP/1, SP/2,..., SP/6 which they use in an introductory course. It thus seems that PL/I might be suited. However, the same HOLT writes also article [29] in which he calls PL/I a "fatal disease". This is not encouraging. Being a hodgepodge of FORTRAN, COBOL and ALGOL, we found it unattractive to use a subset of PL/I for our system. We feared that it is not simple and does not encourage structured programming. Lecarme plainly calls PL/I "dirty".

4.2.3. *ALGOL 60*

ALGOL 60 is somewhat old now, moreover it is very inefficient on our CDC Cyber.

4.2.4. *ALGOL 68*

ALGOL 68 (VAN WIJNGAARDEN [30, 31]) is a very important scientific achievement in the area of programming language design.
The language is, however, complicated and not easy to learn. It is not clear how to choose a suitable subset. The subset chosen by HIBBARD [32] for example, is too large.

Compilers are becoming available now; but if they will be error-proof, fast and fast-code producing is questionable. In any case, the ALGOL 68 compiler which is being made for our CDC computer will be large.

4.2.5. *PASCAL*

This more recent language has been designed by WIRTH [33, 34, 35] taking into account:

modern programming needs,

usability, storage- and time-efficiency,

well-structuredness,

portability (a portable compiler is available).

With respect to the concept structured programming, PASCAL is very suited due to its nice program control facilities.
We do agree with some of the criticism of HABERMANN [36] (Conditional expressions, labels and goto statements); with others we do not agree.
For example, if one has a large procedure it is awkward no to be able to declare a local variable in a small block, procedures should not be large, however; furthermore, scope problems in block-structured programs are difficult to explain, without blocks these scope problems are much easier to explain. For our interactive system dynamic arrays are not very important, so the non-availability of dynamic arrays is no serious problem.

For our CDC Cyber computer a fast and fast-code producing PASCAL compiler of workable size is available. It is remarkable that the code produced can compete with FORTRAN code as far as the execution time is concerned.
Also an efficient use of storage is achieved by using packed arrays and packed records.

Concerning textbooks we note that WIRTH has written a nice introductory book [37, 38] in which some emphasis is given to structured (systematic) programming and to correctness proofs.
We expect that the language will become widely used. (See also LECARME [16]).

Because PASCAL is locally available and fulfills most of our criteria we decided to use a subset of it to be the language of our system.

5. BASIS, AN INTERACTIVE SYSTEM

In this section we shall describe BASIS, which is our interactive system, now under development. In the next section we will evaluate this system using the dsign criteria of sections 2 and 3.

It is of course impossible to describe BASIS in full details; we will therefore describe only some of its interesting characteristics. First, we will give a very brief outline of the course and the role of BASIS in the laboratory part of the course. Second, some aspects of BASIS itself will be treated.

5.1. *Outline of the course*

An introductory course in programming consists of a theoretical part and a laboratory part. Depending on the audience the theoretical part may vary from pure computer appreciation to very technical treatment on number representation or Turing Machines. The laboratory part consists of a set of programming exercises. It is here that the computer system used plays an important role with respect to the time and the amount of assistance needed by the student. One of the main design criteria of BASIS was to minimize these quantities. Considerable thought has been given to the computer-student communication. In short the laboratory part of the course runs as follows:

1. *The very first beginning; the desk calculator*

The student learns to use the computer as a desk calculator and experiments with numbers (real and integers), arithmetic expressions, logical values true and false, with boolean expressions and characters. Moreover, he learns the reaction of the computer on errors. Each error is immediately signaled and a possible suggestion for change is given (if possible). The communication is direct; which means that each command is directly executed.

2. *The use of variables*

Variables of different type are created by a declaration and by an erase. (Here erasing a variable is just the opposite of declaring a variable). The desk calculator experiments are continued with assignments statements. The student now gets a notion of the store of a computer.

3. *The use of simple procedures*

Procedures without parameters and without local variables, are used to introduce the concept of an algorithm; i.e. something that is being written

down (declaration) and that can be called in which case the algorithm is executed (invocation). It is necessary for the student to learn a) that the store of the computer can keep a procedure, b) that the execution of a statement can also be done indirectly and c) that the vocabulary of the computer can be enriched by own-invented capabilities.

4. *Editing of procedures*

As the text of a procedure is kept in the store, it is also possible to change this text. This is performed on a line-by-line basis, not using line numbers but beginning symbols of the lines (so-called line indicators) instead.

5. *Program-control structure*

Some simple program-control structures are introduced in the form of conditional statements and conditional repetitions. It is now possible to do some real programming experiments.

6. *The use of local variables*

7. *The parameter mechanism*

The actual-formal correspondence is explained.

8. *Arrays*

9. *Textmanipulation*

Strings are treated as arrays of characters.

10. *More advanced subjects as e.g. recursion*

5.2. *Some aspects of BASIS*

Having shown the whole course in a nutshell, we will now discuss some important aspects of BASIS a little bit more deeply. These aspects are:

1. The phenomenon "computer".
2. Man-machine communication.
3. The language.
4. The editing facilities.
5. An example of a terminal session.
6. The implementation.

5.2.1. *The phenomenon "computer"*

It is of crucial importance that the computer behaves itself and is as friendly and helpful as possible in an introductory course. Only after this

course, when the students are motivated and know a little bit of the functioning of the computer, it may show its true face with a Job Control Language, a file system, Input/Output, an operating system and so forth. During the course it should demonstrate the following properties:

1. It obeys the student and executes every well-formed command; if the command is not well-formed it says so in friendly and clear words explaining the error suggesting, if possible, a correction.
2. It can store variables and procedures. Storing procedures is particularly useful for enlarging the computers computing vocabulary.

5.2.2. *Man-machine communication*

Unlike BASIC, we decided that every command has to be executed immediately and directly by the computer, for the following reasons:

1. In any interactive system there are always certain commands which must be executed directly and immediately, as e.g. the (BASIC) commands RUN, STOP, LIST or DELETE.
2. In order to show the important concept of a written-down algorithm, in the form of a procedure (See sections 2.3, 2.6 and 3.3), it is necessary to make a sharp distinction between direct execution and indirect execution. This distinction is not sharp in BASIC, where the program may contain procedure declarations (subroutines) and is itself also a (hidden) procedure declaration being called by the RUN command. Actually one has two different procedure declarations in BASIC, one procedure is invoked by a subroutine jump, the other is invoked by RUN. In BASIS we do not explain a procedure as a handsome trick or as a means to shorten the text of a program, but as a written-dow algorithm which can be given a name.
3. Before starting to program, it is possible to accustom the student to the computer by doing some simple desk-calculator experiments to familiarize him with arithmetic and boolean expressions.

As one can conclude from the above considerations, there is not in BASIS a piece of text which is conventionally called a program. In BASIS the program is the collection of variables and procedures.

An important point in an interactive system is the synchronization problem. Student and computer should not be waiting for each other too long. A simple solution, being mostly used, is the one-statement-per-line approach; this has, however, some awkward consequences: first, short statements must also be given on one line and this does not encourage using identation to

to structure the program, second, there is an artificial bound on the length of a statement which does encourage the bad use of cryptographic identifiers.

The solution we have chosen is that the communication is per line, but more than one statement may be placed on one line and one statement may be written on more than one line. The student types in one line and waits for a signal from the computer to go on. If he makes an error, the rest of the line, even if it contains some good statements, is omitted.

5.2.3. *The language*

The language in BASIS is a subset of PASCAL of which we have seen, in section 4.2., that it is suitable for such a purpose. For us, PASCAL has the additional advantage that a fast PASCAL compiler is available on a CDC computer linked to our own PDP11/45 so that a program which is too time-consuming in our interactive BASIS system can be sent to be executed by the CDC computer.

In the direct-execution mode the following "commands" are posssible:
declaration of integer, real, boolean and character variables,
declaration of procedures and functions,
type declaration for arrays of "ordinary" type,
declaration of constants,
assignment statements,
arithmetic and boolean expressions,
procedure and function calls,
erasure of an identifier,
editing of procedure declaration.
In the indirect-execution mode, i.e. within a procedure declaration, the following "commands" may be used:
declaration of integer, real, boolean and character variables,
declaration of array variables,
assignment statements,
arithmetic and boolean expressions,
procedure and function calls,
conditional statement: if ... then ... else ...,
  case i of...end
conditional repetition: while ... do ...; repeat ... until ...,
compound statement: begin ... end.

A peculiarity of PASCAL is the way array variables may be declared using a programmer-defined type. Suppose that the type is erased, what should be done with the variables declared of that type? The only sensible answer, we could give, is to erase them also. For, suppose the variables were not destroyed and that a new type is declared with the same identifier and with this new type also a new variable of that type, then there would exist variables declared with the same type identifier but of a different types.

5.2.4. *The editing facilities*

By means of "erase" one can destroy every global identifier and, if the identifier belonged to a variable or a procedure, the associated storage space is no longer occupied. Another mechanism, namely editing, is therefore necessary to change the procedure declaration of a procedure identifier. It is a similar mechanism as the assignment statement for changing the value of a variable.

The editing is based on the line structure without using line numbers. A problem with line numbers is: who provides them? If it is the student, they are part of the text and as such very attractive to be used as labels; if the system provides them automatically, it is necessary to frequently list the declarations. Instead the first symbols on a line are used to identify the line. These first symbols are called a line-indicator with a simple syntax:

/ text /

After indicating the line, other text may be:

a) inserted after
b) inserted before
e) exchanged with

the indicated line, after which the line with the next line indicator is found by:

c) continuing the search
d) deleting the text.

A facility to systematically change an identifier into another identifier is also envisaged.

5.2.5. *An example of a terminal session*

A first session with BASIS may look like:

```
student:                    response:
print(5);                   +5
print(5 + 6 × 7);           +47
print(3.14);                +.314e1
print(+.314e1);             +.314e1
print(true);                true
print(1 = 1);               true
print(1 = 2);               false
print(1 = 1 or 1 = 2);      true
print(1 < 2 < 3);           wrong operator in
                            boolean expression
print(1 < 2 and 2 < 3);     true
```

A second session with BASIS may look like:

```
var x : real;
    i : integer;
    i:= 5; x:= 3.14;
print(i);                   +5
print(x);                   +.314e1
print(i + x × 2);           +.1128e2
erase x;
print(x);                   unknown identifier : x
x:= 3.14;                   unknown identifier : x
var x : integer;
print(x);                   value of x unknown
x:= 3.14;                   value does not fit
var x : real;               x has been declared
erase x;                                already
var x : real;
x:= 3.14;
print(x);                   +.314e1
```

In this session, a desk-calculator exercise as, for example, calculating the mean and standard deviation of a set of numbers may be done:

```
var mean, standdev : real;
mean:= 6.4; standdev:= sqr(6.4);
mean:= mean + 3.1; standdev:= standdev + sqr(3.1);
```

```
mean:= mean + 9.7; standdev:= standdev + sqr(9.7);
mean:= mean + 4.6; standdev:= standdev + sqr(4.6);
mean:= mean/4; print(mean);
standdev:= sqrt(standdev/4 - sqr(mean));
print(standdev);
```

In a natural way procedures can be introduced here:

```
var mean, standdev, x : real;
    n : integer;
procedure calc;
begin n:= n + 1; mean:= mean + x;
    standdev:= standdev + sqr(x)
end;
mean:= 0; standdev:= 0; n:= 0;
x:= 6.4; calc;
x:= 3.1; calc;
x:= 9.7; calc;
x:= 4.6; calc;
mean:= mean/n; print(mean);
standdev:= sqrt(standdev/n - sqr(mean));
print(standdev);
```

The introduction of a parameter might be done using the above procedure calc after some editing:

```
edit(calc,
     / procedure /,e,/ procedure calc(x : real); /);
list(calc);
```

with the result:

```
procedure calc(x : real);
begin n:= n + 1; mean:= mean + x;
          standdev:= standdev + sqr(x)
end;
```

Now this procedure may be called as follows:

```
mean:= 0; standdev:= 0; n:= 0;
calc(6.4); calc(3.1); calc(9.7); calc(4.6);
```

In following sessions the student may then learn how to solve a problem using more procedures and more complicated program-control structures.

5.2.6. *The implementation*

The system is built up from two main parts:

1. the editor and text-handling mechanism,
2. the executor, for the syntax-checking and execution.

With respect to execution time both parts give conflicting criteria: for the editor it would be most easy to have the literal text, for the executor it would be most efficient to compile the text into machine-code. We have chosen a middle position, being helped by the fact that a fast PASCAL compiler is in the neighbourhood to run the more time-consuming jobs. The original text is split up in symbols, each symbol being a complete keyword, number, identifier or delimiter. These symbols are stored in text cells linked to each other in the reading order and in a line-by-line order. The procedure calc of the preceding example would look like:

```
---> proc ----> id ----> ;
                  1

---> begin ---> id --> := --> id --> +
                  2             2

n --> ; --> id --> := --> id --> + --> id --> ;
1            3             3            4

 --> id -> := -> id -> + -> id -> ( -> id -> )
      5          5          6          4

---> end --> ; --|
```

The search for lines in the edit process can now be done in a rapid manner. The identifiers do not have to be scanned by the executor as their text cells point directly to their information cells (where the type and value are stored). The numbers need also not be scanned by the executor as their text cells contain type and value. Furthermore, the structure of "if ... then ... else", "repeat ... until", and "while ... do" can easily be brought in the structure of the text cells. For example:

```
if ... --> then ... --> else ...
```

while ... --> do ...

After the creation of a procedure, or after an editing command, the procedure body is scanned to fix up the above structure in the text cells. This also helps making the executor reasonably efficient.

Procedure calling is implemented very simply: new information cells are created for formal parameters and for local variables; actual parameters are evaluated, their results are put in the appropriate information cells; the value v of the old text-cell pointer is stored and this pointer is given as new value the value of the first text cell of the procedure declaration; the procedure body is executed and finally, at procedure exit, the new information cells are given free and the execution is resumed with the text cell pointed at by the old value v.
In this scheme recursive calling of procedures is no problem.

At this moment, September 1974, an experimental system is being implemented in PASCAL itself for a CDC Cyber computer to work under Intercom.

After experimentation, BASIS will be implemented on our PDP11/45 computer installation, which has a dozen terminals, virtual memory on disk and an experimental time sharing (TANENBAUM [39]).

As has been said the CDC Cyber and the PDP11/45 are connected with each other by means of a data link. This makes it possible to use the interactive system at the PDP11/45 to conveniently develop a program and to efficiently run it on the CDC computer.

6. BASIS AND THE ORIGINAL DESIGN CRITERIA

In the preceding section we have described BASIS which was designed with the criteria of section 3 in our mind. We will now review these criteria shortly to see how they influenced BASIS.

1. Simplicity has guided us in the man-machine aspects as well as in the choice of the language.
2. The concepts learned: computational power, variables, assignment, procedures (=algorithms), actual-formal correspondence are general indeed.
3. PASCAL has been constructed with well-structuredness under the banner of the designer. It has been our prime intent to maintain this well-

structuredness and even to enlarge it by simply making the use of goto's impossible and by stimulating the use of procedures.

4. For an introductory course, the program-control structures of BASIS are enough.
5. The basic data types are: boolean, character, integer and real.
6. Considerable attention is given to make BASIS as good as possible with respect to interaction.
7. Due to the fact that PASCAL is the mother language, a very small step is necessary to teach more advanced concepts such as files, records and pointers.
8. Teaching of other programming languages on the fundaments of BASIS is easy indeed.
9. An application course can be given in a simple way using PASCAL for either numerical or for text-processing purposes.
10. PASCAL is very suited to be used in advanced programming courses, therefore BASIS gives a good start.

ACKNOWLEDGMENTS

I wish to thank my colleagues Dr. A.S. Tanenbaum and Dr. I. Pohl (from the University of California, Santa Cruz, and temporarily in our department) for their critical remarks.

REFERENCES

[1] SCHEEPMAKER, B. & K.L. ZINN (eds.), *World Conference on Computer Education*, Amsterdam, 1970, Science Associates International, New York, 1971.

[2] TURSKI, W.M. (ed.), *Programming Teaching Techniques*, Proceedings of the IFIP TC-2 Working Conference on Programming Teaching Techniques, Zakopane, Poland, September 18-22, 1972, North Holland Publ. Cy., Amsterdam, 1973.

[3] SIGCSE Bulletin (1974), *Special issue on Fourth Symposium on Computer Science Education*, February 14-15, 1974, Association for Computing Machinery, 1974.

[4] FINERMAN, A. (ed.), *University Education in Computer Science*, Academic Press, 1968.

[5] POLYA, G., *How to solve it?* Princeton University Press, Princeton, N.J., 1945.

[6] GRIES, D., *What should we teach in an introductory programming course?* SIGCSE Bulletin (1974) 81-89.

[7] HOARE, R.A., Private communication (1973).

[8] GALLER, B., Private communication (1973).

[9] MATHIS, R.F., *Teaching Debugging*, SIGCSE Bulletin (1974), 59-63.

[10] BASILI, V.R. & A.J. TURNER, *Experiences with a simple structured programming language*, SIGCSE Bulletin (1974) 144-147.

[11] NIEVERGELT, J., E.M. REINGOLD & T.R. WILCOX, *The automation of introductory computer science courses*, in A. GUENTHER et al. (eds), International Computing Symposium 1973, 495-501, North Holland Publ. Cy., Amsterdam, 1974.

[12] WIRTH, N., *On the design of programming languages*, Proceedings of the IFIP congress 74, August 5-10, Stockholm, North Holland Publ. Cy., 1974, Amsterdam, 386-393.

[13] RIET, R.P. van de, *Over GOTO's en programmacorrectheid*, Informatie 16(1974) 4, 251-260.

[14] LECARME, O., *What programming Language should we Use for Teaching programming?* in TURSKI [2], 61-71.

[15] KNUTH, D., *Structured programming with goto statements*, Report STAN-CS-74-416 (May 1974), Computer Science Department, Stanford University.

[16] LECARME, O., *Structured Programming, Programming Teaching and the Language PASCAL*, SIGPLAN Notices 9(1974) 7, 15-21.

[17] KEMENY, J.G. & T.E. KURTZ, *BASIC programming*, John Wiley and Sons, Inc. New York, 1967.

[18] PECK, J.E.L., *Comparison of Languages*, in TURSKI [2], 43-60.

[19] ADAMS, W.S., *The use of APL in teaching programming*, in TURSKI [2],3-13.

[20] HEBENSTREIT, J., *Informatics in elementary and secondary education*, Proceedings of the Rio Symposium on Computer Education for Development, August 6-12, 1972, AO LIVRO TECNICO S.A.

[21] FENICHEL, Robert R., *Design of languages for elementary programming instruction: Lessons of the TEACH project*, Proceedings of the IFIP Conference on Computers and Education, Amsterdam, August 1970, Book III, 175-177, North Holland Publ. Cy., Amsterdam.

[22] FENICHEL, Robert R., Joseph WEIZENBAUM & Jerome C. YOCHELSON, *A program to teach programming*, Comm. ACM 13(1970) 3, 141-146.

[23] SHAW, J.C., *JOSS: A Designers View of an experimental On-Line Computing System*, Proc. FJCC, Vol. 26, pt 1(1964) 455-464.

[24] SCHILD, R., *Interactive Structured Programming*, in A. GUENTHER et al. (eds.), International Computing Symposium 1973, 81-84, North Holland Publ. Cy., Amsterdam, 1974.

[25] RALSTON, A., *The future of higher level languages (in teaching)*, in A. GUENTHER et al. (eds.), International Computing Symposium 1973, 1-10, North Holland Publ. Cy., Amsterdam, 1974.

[26] LECARME, O., *Computing Reviews*, August 1974, Review 26, 984.

[27] CONWAY, Richard & David GRIES, *An introduction to programming: A structured approach using PL/I and PL/C*, Winthrop Publishers, Cambridge, Mass., 1973.

[28] HOLT, Richard C. & David B. WORTMAN, *A sequence of structured Subsets of PL/I*, SIGCSE Bulletin, 6(1974) 1, 129-132.

[29] HOLT, Richard C., *Teaching the Fatal Disease (or) Introductory Computer Programming Using PL/I*, SIGPLAN Notices 8(1973) 5, 8-23.

[30] WIJNGAARDEN, A. van (ed.), B.J. MAILLOUX, J.E.L. PECK & C.H.A. KOSTER, *Report on the Algorithmic Language ALGOL 68*, Numerische Mathematik 14(1969) 79-218.

[31] WIJNGAARDEN, A. van, B.J. MAILLOUX, J.E.L. PECK, C.H.A. KOSTER, M. SINTZOFF, C.H. LINDSEY, L.G.L.T. MEERTENS & R.G. FISKER, *Revised Report on the Algorithmic Language ALGOL 68*, Supplement to ALGOL BULLETIN 36, Technical Report TR 74-3 (March 1974), Department of Computing Science, The University of Alberta, Edmonton, Alberta, Canada.

[32] HIBBARD, P.G., *A minimum General Purpose Sublanguage of ALGOL 68*, ALGOL Bulletin no. 35 (March 1973) 14-23.

[33] WIRTH, N., *The programming language Pascal*, Acta Informatica 1(1971) 35-63.

[34] WIRTH, N., *The design of a Pascal compiler*, Software practice and experience 1(1974) 4.

[35] WIRTH, N., *The programming language Pascal (revised report)*, Berichte der Fachgruppe Computer Wissenschaften 5, Eidgenoessische Technische Hochschule, Zuerich (1972).

[36] HABERMANN, A.N., *Critical Comments on the Programming Language Pascal*, Acta Informatica 3(1970) 47-57.

[37] WIRTH, N., *Systematisches Programmieren*, Teubner Verlag, Stuttgart (1972).

[38] WIRTH, N., *Systematic programming: an introduction*, Prentice-Hall, Englewood Cliffs (1973).

[39] TANENBAUM, A.S., *Design of a general purpose Time-sharing system for the PDP11/45*, Informatica Rapport IR2, Wiskundig Seminarium, Vrije Universiteit, Amsterdam, 1973.

MATHEMATICAL CENTRE SYLLABUS 25, 1976, VII, 121-132

CORRECTHEIDSBEWIJZEN VOOR ZOEK- EN MARKERINGSALGORITMEN VAN DYADISCHE DATA-STRUCTUREN

W.P. DE ROEVER

1. EEN CORRECTHEIDSBEWIJS VAN EEN ZOEKALGORITME VOOR DYADISCHE BOMEN ZONDER HULPSTACK OF HULPBITS

Vorig jaar is door mij binnen het kader van het colloquium "Programma-correctheid" een lezing gehouden over een zoek- en markeringsalgoritme voor dyadische bomen, waarvoor geen hulpstack nodig was, doch voor zowel het zoekproces als het markeren wel een hulpbit in elke knoop van de te doorzoeken of te markeren dyadische boom (DE ROEVER [1]).

Na afloop van die lezing deelden VAN WIJNGAARDEN, MEERTENS en FOKKINGA mij onafhankelijk van elkaar mee, dat deze algoritme voor zover het het zoekproces betrof kon worden verbeterd, aangezien dáárvoor niet alleen de hulpstack doch ook de hulpbits gemist konden worden. Dit gaf aanleiding tot een nieuwe, tot op dat ogenblik niet in de literatuur voorkomende, zoekalgoritme voor dyadische bomen. Een versie van deze algoritme werd echter een maand later door DWYER [2] gepubliceerd.

Deze toedracht sterkt mij in de overtuiging dat correctheidsbewijzen van algoritmen niet alleen de accountant in de programmeur aanspreken, doch wel degelijk het inzicht in het door de algoritme opgeloste probleem kunnen vergroten. Zodoende kunnen zij tot het vinden van nieuwe, efficiëntere algoritmen inspireren, en daarmee hun practisch nut bewijzen.

De bewuste algoritme kan als volgt worden beschreven:

```
Single(ℓ,r) ⇐ if at(ℓ) then                                          ┐
                                                                     │
                  if ℓ = NIL* then r else single(r,ℓ) fi             }(I)
                                                                     │
                       else single(car(ℓ),cons(cdr(ℓ),r)) fi,        ┘
```

waarbij NIL^* een atoom voorstelt dat niet in ℓ voorkomt, en de aanroep singel(ℓ,NIL^*) tot het doorzoeken van ℓ leidt. Het niet-voorkomen van NIL^* in ℓ kan door een aanroep van de Boolese procedure

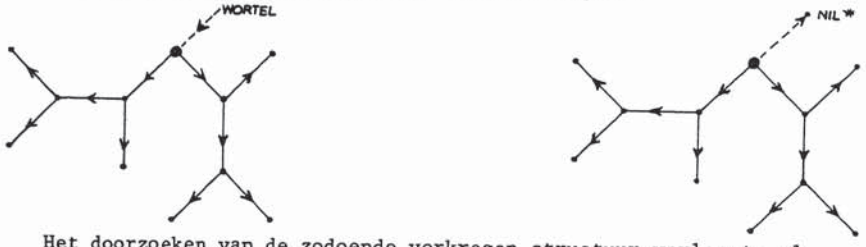
$$Not(NIL^*):in(\ell) \Leftarrow \underline{if}\ at(\ell)\ then\ \neg(\ell=NIL^*)\ \underline{else}$$

$$Not(NIL^*):in(car(\ell)) \wedge Not(NIL^*):in(cdr(\ell))\ \underline{fi}$$

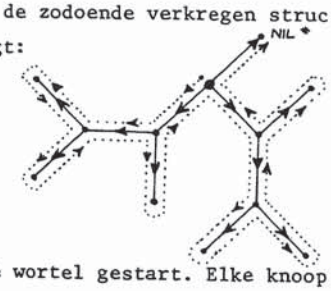
worden vastgesteld.

Hoe werkt algoritme (I) ?

In elke knoop van een dyadische boom, behalve de wortel, heb je met 3 pijlen te maken (zie figuur). De uitzonderingssituatie waarin de wortel verkeert zal worden opgeheven door een extra pijl van de wortel naar het niet in de boom voorkomende atoom NIL^* te laten wijzen:



Het doorzoeken van de zodoende verkregen structuur verloopt, al "backtrackend", als volgt:



De algoritme wordt in de wortel gestart. Elke knoop (niet-atoom) wordt driemaal bezocht:

- éénmaal voor uitvoering van de fase "linksomlaag",
- éénmaal voor uitvoering van de fase "rechtsomlaag", en
- éénmaal voor uitvoering van de fase "terugomhoog".

Tijdens het doorzoeken van de boom bestaat de voor het zoekproces relevante informatie in elke knoop uit 3 pijlen, wier richting bepaald wordt door het aantal malen dat de desbetreffende knoop bezocht is:

- De eerste maal dat een knoop bezocht wordt, moet in de fase "linksomlaag" de linker deelboom en vervolgens in de fase "rechtsomlaag" de rechter deelboom doorzocht worden, en daarna in de fase "terugomhoog" op het spoor teruggekeerd worden (backtracking). Dientengevolge volstaan de oorspronkelijke pijlen naar de linker- en rechter deelbomen, doch moet (behalve in de wortel) de pijl, waarlangs je in de bewuste knoop terechtkwam, van richting omgekeerd worden.
- De tweede maal dat een knoop bezocht wordt, moet de pijl, waarlangs je die tweede maal in de bewuste knoop (vanuit de linker deelboom) terechtkwam, van richting worden omgekeerd (deze pijl is daarvóór al eens van richting omgekeerd teneinde de terugweg vanuit de linker deelboom naar de bewuste knoop aan te geven).
- De derde maal dat een knoop bezocht wordt, moet de pijl, waarlangs je die derde maal in de bewuste knoop (vanuit de rechter deelboom) terechtkwam, van richting worden omgekeerd (deze pijl is daarvóór al eens van richting omgekeerd teneinde de terugweg vanuit de rechter deelboom aan te geven).

Hieruit valt af te leiden dat elke pijl (behalve de toegevoegde pijl van de wortel naar NIL^*) tijdens het doorzoeken t.o.v. zijn oorspronkelijke positie vóór het doorzoeken tweemaal van richting verandert, en dientengevolge ná het doorzoeken weer in zijn oorspronkelijke richting wijst.

Dit wordt nu bewezen.

STELLING. *De volgende implicatie is geldig:*

$$Not(NIL^*):in(\ell) \supset single(\ell,NIL^*) = \ell.$$

Teneinde deze stelling te bewijzen, zal de volgende implicatie bewezen worden:

$$Not(NIL^*):in(\ell) \supset single(\ell,r) = single(r,\ell).$$

Uit de geldigheid van deze laatste implicatie volgt dan, indien $Not(NIL^*):in(\ell)$ geldt, dat $single(\ell,NIL^*) = single(NIL^*,\ell) \stackrel{!}{=} \ell$.

BEWIJS. Laat Not(NIL*):in(ℓ) gelden.

a. ℓ is een atoom, d.w.z. at(ℓ) geldt.
   Uit Not(NIL*):in(ℓ) volgt ¬(ℓ=NIL*).
   Dientengevolge geldt single(ℓ,r) = single(r,ℓ).

b. ℓ = cons(ℓ_1,ℓ_2). Neem het gestelde als inductiehypothese voor ℓ_1 en ℓ_2 aan. Uit Not(NIL*):in(cons(ℓ_1,ℓ_2)) volgt dat zowel Not(NIL*):in(ℓ_1) als Not(NIL*):in(ℓ_2) gelden.
   single(ℓ,r) ≡
   single(cons(ℓ_1,ℓ_2),r) = (1^e-bezoek aan knoop) single(ℓ_1,cons(ℓ_2,r)) =
   = (ind. hyp)
   single(cons(ℓ_2,r),ℓ_1) = (2^e-bezoek aan knoop) single(ℓ_2,cons(r,ℓ_1)) =
   = (ind. hyp.)
   single(cons(r,ℓ_1),ℓ_2) = (3^e-bezoek aan knoop) single(r,cons(ℓ_1,ℓ_2)) ≡
   single(r,ℓ). □

2. EEN CORRECTHEIDSBEWIJS VAN EEN MARKERINGSALGORITME VOOR GERICHTE DYADISCHE GRAFEN ZONDER HULPSTACK

Teneinde ons voor te bereiden op een algoritme die gerichte dyadische grafen afzoekt, moet worden vastgesteld dat dyadische grafen zich van dyadische bomen onderscheiden doordat er circulariteiten en het verschijnsel gemeenschappelijke data ("shared data") kunnen optreden, zoals in de onderstaande figuur:

Om te voorkomen dat circulariteit in de data-structuur ook tot circulariteit tijdens het zoekproces, d.w.z. tot niet-terminatie van de algoritme, aanleiding geeft, moeten de reeds bezochte knopen gemarkeerd worden. Dit wordt duidelijk gemaakt aan de hand van de volgende algoritme voor het markeren van dyadische bomen, dat naderhand in passend gemodificeerde vorm zal worden gebruikt om gerichte dyadische grafen te doorzoeken:

$$\left.\begin{array}{l}
\text{LEFT}(\ell,r) \Leftarrow \underline{\text{if}}\ \text{at}(\ell)\ \underline{\text{then}}\ \text{BACK}(\ell,r) \\
\qquad\qquad \underline{\text{else}}\ \text{LEFT}(\text{car}(\ell),\text{cons}(1,\text{cdr}(\ell),5))\ \underline{\text{fi}}, \\
\text{BACK}(\ell,r) \Leftarrow \underline{\text{if}}\ r = \text{NIL}\ \underline{\text{then}}\ \ell\ \underline{\text{else}} \\
\qquad \underline{\text{if}}\ \text{bitfield}(r) = 1\ \underline{\text{then}}\ \text{LEFT}(\text{car}(r),\text{cons}(0,\text{cdr}(r),\ell)) \\
\qquad\quad \underline{\text{else}}\ \text{BACK}(\text{cons}(1,\text{cdr}(r),\ell),\text{car}(r))\ \underline{\text{fi}}.
\end{array}\right\} \text{(II)}$$

Hoe werkt dit stelsel iteratieve procedures?
Indien de waarde van de Boolese procedure

$$\text{Notmarked}(t) \Leftarrow \underline{\text{if}}\ \text{at}(t)\ \underline{\text{then}}\ \underline{\text{true}}\ \underline{\text{else}}\ \text{Notmarked}(\text{car}(t)) \wedge \text{Notmarked}(\text{cdr}(t)) \wedge \text{bitfield}(t) = 0\ \underline{\text{fi}}$$

aangeeft of alle bitvelden van de gemarkeerde dyadische boom t een 0 bevatten, en de recursieve (niet-iteratieve) markeringsprocedure M(t) als volgt beschreven is:

$$\text{M}(t) \Leftarrow \underline{\text{if}}\ \text{at}(t)\ \underline{\text{then}}\ t\ \underline{\text{else}}\ \text{cons}(1,\text{M}(\text{car}(t)),\text{M}(\text{cdr}(t)))\ \underline{\text{fi}},$$

dan geldt de volgende implicatie:

$$\text{Notmarked}(\ell) \supset \text{LEFT}(\ell,r) = \text{BACK}(\text{M}(\ell),r).$$

Hieruit valt gemakkelijk af te leiden dat

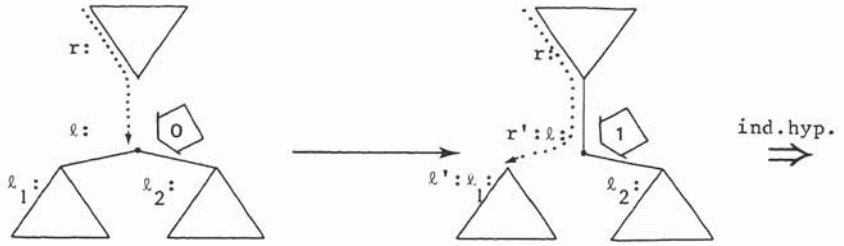
$$\text{Notmarked}(\ell) \supset \text{LEFT}(\ell,\text{NIL}) = \text{M}(\ell)$$

geldt.

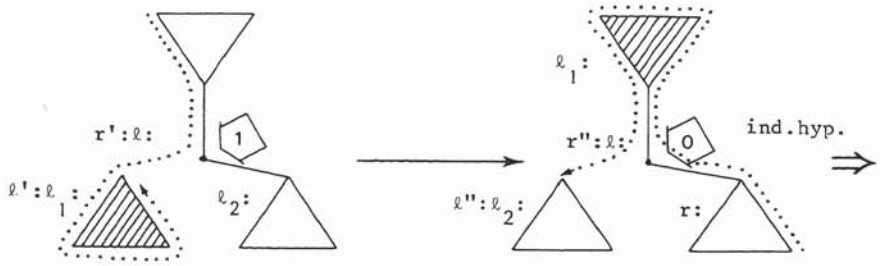
Deze twee implicaties zijn met behulp van de in de vorige paragraaf gegeven bewijsmethode op eenvoudige wijze af te leiden; voor ons doel verdient het echter de voorkeur grafisch na te gaan waarom eerstgenoemde implicatie geldt.

Zij <u>true</u> de waarde van Notmarked(ℓ), $\ell = \text{cons}(\ell_1,\ell_2)$, en laat als inductiehypothese het gestelde voor ℓ_1 en ℓ_2 gelden; dan is het effect van

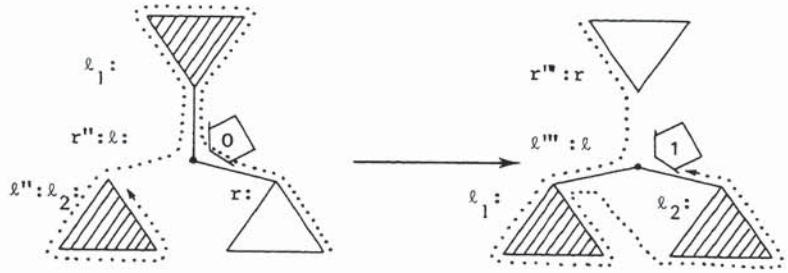
de aanroep LEFT(ℓ,r) als volgt grafisch uit te beelden



LEFT(cons(0,ℓ_1,ℓ_2),r) = LEFT(ℓ_1,cons(1,ℓ_2,r)) =



BACK(M(ℓ_1),cons(1,ℓ_2,r)) = LEFT(ℓ_2,cons(0,r,M(ℓ_2))) =



BACK(M(ℓ_2),cons(0,r,M(ℓ_1))) = BACK(cons(1,M(ℓ_1),M(ℓ_2)),r) ≜
BACK(M(cons(0,ℓ_1,ℓ_2)),r).

Hierbij geeft de stippellijn aan hoe ver het markeringsproces voortgeschreden is, wordt markering d.m.v. arcering aangegeven, en geeft $\langle\boxed{x}$ de waarde x van het markeringsbit van de aangewezen knoop aan. (Voor het geval dat at(ℓ) geldt is het gestelde triviaal.)

Wat valt hierbij op?

Uitgaande van een oorspronkelijk ongemarkeerde data-structuur, wordt LEFT(t_1,t_2) alleen aangeroepen met een ongemarkeerde data-structuur als eerste argument: "met LEFT ga je t_1 dieper in". In het geval van een data-structuur met een circulariteit in ℓ, ga je dus via een aanroep van LEFT met ℓ de mist, d.w.z. een "loop" die tot niet-terminatie van de algoritme leidt, in.

Deze situatie kan worden voorkomen door in LEFT

(i) de bezochte knoop in een extra bit te markeren,

(ii) i.p.v. at(ℓ) de test at(ℓ) ∨ markbit(ℓ) = 1 uit te voeren, waarbij de waarde van marktbit(ℓ) aangeeft of het extra markeringsbit op 0 of op 1 staat.

(Aangezien in algoritme (II) al van één markeringsbit essentieel gebruik wordt gemaakt - het op 0 of op 1 staan bij een aanroep van LEFT houdt geheel verschillende stadia van executie van de algoritme in - moet ter vermijding van circulariteit dus een extra bit aan elke, reeds van één bit voorziene, knoop toegevoegd worden.)

Dit houdt in dat een reeds bezochte knoop net als een atoom wordt behandeld: *je trekt je er a.h.w. tijdens executie van de algoritme van terug en doorloopt dus alleen de "spanning tree" van de gerichte dyadische graaf.*

Deze nieuwe versie van de zojuist besproken algoritme onderscheidt zich echter wezenlijk van de oorspronkelijke versie, *doordat ℓ en r nu niet meer als twee zelfstandige, onafhankelijke, data-structuren beschouwd worden, doch van een representatie van ℓ en r door middel van een geheugen en twee adressen gebruik wordt gemaakt, aangezien ℓ en r d.m.v. doorverwijzing met elkaar in verbinding kunnen staan.*

De verkregen algoritme kan als volgt beschreven worden:

$$
\begin{aligned}
&\mathrm{LEFT}(\ell,r,\sigma) \Leftarrow \underline{\text{if}}\ \mathrm{at}(\ell) \vee \mathrm{markbit}(\ell,\sigma) = 1\ \underline{\text{then}}\ \mathrm{BACK}(\ell,r,\sigma)\\
&\qquad\qquad \underline{\text{else}}\\
&\qquad\qquad \mathrm{LEFT}(\mathrm{car}(\ell,\sigma),\ell,\sigma[\ell \longmapsto \langle 1,1,\mathrm{cdr}(\ell,\sigma),r\rangle])\ \underline{\text{fi}}\ ,\\
&\mathrm{BACK}(\ell,r,\sigma) \Leftarrow \underline{\text{if}}\ r = \mathrm{NIL}\ \underline{\text{then}}\ \langle \ell,\sigma\rangle\ \underline{\text{else}}\\
&\qquad\qquad \underline{\text{if}}\ \mathrm{flagbit}(\ell,\sigma) = 1\ \underline{\text{then}}\\
&\qquad\qquad \mathrm{LEFT}(\mathrm{car}(r,\sigma),r,\sigma[r \longmapsto \langle \mathrm{markbit}(r,\sigma),0,\mathrm{cdr}(r,\sigma),\ell\rangle])\ \underline{\text{fi}}\\
&\qquad\qquad \underline{\text{else}}\\
&\qquad\qquad \mathrm{BACK}(r,\mathrm{car}(r,\sigma),\sigma[r \longmapsto \langle 1,0,\mathrm{cdr}(r,\sigma),\ell\rangle])\ \underline{\text{fi}}
\end{aligned}
\qquad \text{(III)}
$$

Hierbij geldt het volgende:

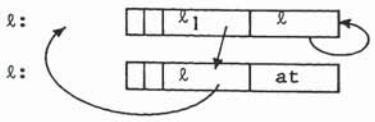
(i) ℓ en r duiden hetzij geheugenadressen, hetzij atomen aan;

(ii) σ duidt een geheugen aan, d.w.z. een verzameling van adressen verwijzend naar viertallen, die elk weer uit een markbit-veld, flagbit-veld, car-veld en cdr-veld bestaan;

(iii) als ℓ een geheugenadres is, isoleren $\mathrm{markbit}(\ell,\sigma)$, $\mathrm{flagbit}(\ell,\sigma)$, $\mathrm{car}(\ell,\sigma)$ en $\mathrm{cdr}(\ell,\sigma)$ de inhoud van de overeenkomstige velden van het viertal waarnaar ℓ in het geheugen σ verwijst;

(iv) $\sigma[x \longmapsto \langle a,b,c,d\rangle]$ duidt het geheugen σ' aan dat uit σ verkregen wordt door adres x naar $\langle a,b,c,d\rangle$ te laten verwijzen (of: het oude viertal waarnaar ℓ verwijst met $\langle a,b,c,d\rangle$ te overschrijven).

Ofschoon het correctheidsbewijs voor algoritme (III) in wezen uit dezelfde onderdelen bestaat als het correctheidsbewijs van algoritme (II), wordt het gecompliceerder doordat er nu geen inductieargument meer naar de complexiteit van de data-structuur opgaat; de data-structuur waarnaar door $\mathrm{car}(\ell,\sigma)$ wordt verwezen behoeft n.l. niet minder complex te zijn dan die waarnaar door ℓ en σ verwezen wordt, aangezien ℓ en $\mathrm{car}(\ell,\sigma)$ hetzelfde adres kunnen aanduiden:



Wél is echter inductie te plegen naar het aantal met 1 gemarkeerde knopen van de data-structuur waarnaar ℓ verwijst.

Het hierna volgende correctheidsbewijs voor algoritme (III) is een

bewerking van dat van TOPOR [3].

Alvorens hiertoe over te gaan worden enige begrippen ingevoerd:

1. $\mathrm{Nodes}(\ell,\sigma) \Leftarrow$ **if** $\mathrm{at}(\ell) \vee \mathrm{markbit}(\ell,\sigma) = 1$ **then** $\emptyset$ **else**

$$\{\ell\} \cup \mathrm{Nodes}(\mathrm{car}(\ell,\sigma),\sigma[\ell \longmapsto \langle 1,\mathrm{flagbit}(\ell,\sigma),\mathrm{car}(\ell,\sigma),\mathrm{cdr}(\ell,\sigma)\rangle]$$

$$\cup\ \mathrm{Nodes}(\mathrm{cdr}(\ell,\sigma),\sigma[\ell \longmapsto \langle 1,\mathrm{flagbit}(\ell,\sigma),\mathrm{car}(\ell,\sigma),\mathrm{cdr}(\ell,\sigma)\rangle]$$ **fi**,

d.w.z., $\mathrm{Nodes}(\ell,\sigma)$ duidt de verzameling van die adressen aan welke vanuit ℓ beschikbaar zijn d.m.v. doorverwijzing via ongemarkeerde (d.w.z. met marbitveld op 0 staande) knopen.

2. $\mathrm{Mark}(\ell,\sigma) \Leftarrow$ **if** $\mathrm{at}(\ell) \vee \mathrm{markbit}(\ell,\sigma) = 1$ **then** σ **else**

$$\mathrm{Mark}(\mathrm{cdr}(\ell,\sigma),\mathrm{Mark}(\mathrm{car}(\ell,\sigma),\sigma[\ell \longmapsto \langle 1,\mathrm{flagbit}(\ell,\sigma),$$
$$\mathrm{car}(\ell,\sigma),\mathrm{cdr}(\ell,\sigma)\rangle])$$ **fi**,

d.w.z., $\mathrm{Mark}(\ell,\sigma)$ duidt het geheugen aan dat resulteert door alle vanuit ℓ via ongemarkeerde knopen bereikbare knopen te markeren (het markbitveld op 1 te zetten).

De volgende eigenschappen van Nodes en Mark zijn m.b.v. inductie naar recursiediepte te bewijzen:

A. $\mathrm{Nodes}(x,\sigma[y \longmapsto \langle 1,a_1,a_2,a_3\rangle]) = \mathrm{Nodes}(x,\sigma[y \longmapsto \langle 1,b_1,b_2,b_3\rangle])$,

B. $y \notin \mathrm{Nodes}(x,\sigma[y \longmapsto \langle 1,a_1,a_2,a_3\rangle])$,

C. $\mathrm{Mark}(x,\sigma[y \longmapsto \langle 1,a_1,a_2,a_3\rangle])[y \longmapsto \langle 1,b_1,b_2,b_3\rangle] =$

$\mathrm{Mark}(x,\sigma[y \longmapsto \langle 1,b_1,b_2,b_3\rangle])$.

We zullen van deze eigenschappen gebruik maken in het bewijs van de hierna volgende stelling:

STELLING. *Zij* $\mathrm{flagbit}(y) = 0$ *voor alle* $y \in \mathrm{Nodes}(\ell,\sigma)$, *en* LEFT *en* BACK *als in* (III) *gedeclareerd, dan geldt:*

$$\mathrm{LEFT}(\ell,r,\sigma) = \mathrm{BACK}(\ell,r,\mathrm{Mark}(\ell,\sigma)).$$

BEWIJS. Door middel van inductie naar $|\mathrm{Nodes}(\ell,\sigma)|$.
(i) Zij $\mathrm{at}(\ell) \vee \mathrm{markbit}(\ell,\sigma) = 1$, dan geldt

$$\mathrm{LEFT}(\ell,r,\sigma) = \mathrm{BACK}(\ell,r,\sigma) = \mathrm{BACK}(\ell,r,\mathrm{Mark}(\ell,\sigma)).$$

(ii) Zij ℓ een geheugenadres met markbit$(\ell,\sigma) = 0$, $\ell_1 = car(\ell,\sigma)$ en $\ell_2 = cdr(\ell,\sigma)$, en neem het gestelde als inductiehypothese aan voor alle $<\ell',\sigma'>$ met $|Nodes(\ell',\sigma')| < |Nodes(\ell,\sigma)|$.

Dan geldt:

$$LEFT(\ell,r,\sigma) = LEFT(\ell_1,\ell,\underbrace{\sigma[\ell \longmapsto <1,1,\ell_2,r>]}_{\equiv \sigma_1}).$$

Op grond van $Nodes(\ell_1,\sigma[\ell \longmapsto <1,1,\ell_2,r>]) =$ (eig. A)
$Nodes(\ell_1,\sigma[\ell \longmapsto <1,0,\ell_1,\ell_2>]) \subseteq$ (def. Nodes) $Nodes(\ell,\sigma)$, en
$\ell \notin Nodes(\ell_1,\sigma[\ell \longmapsto <1,1,\ell_2,r>])$ (eig. B), geldt $|Nodes(\ell_1,\sigma_1)| < |Nodes(\ell,\sigma)|$.
Dientengevolge geldt op grond van de inductiehypothese

$$LEFT(\ell_1,\ell,\sigma_1) = BACK(\ell_1,\ell,\underbrace{Mark(\ell_1,\sigma_1)}_{\equiv \sigma_2}).$$

Daar $\neg(\ell=NIL)$ en $flagbit(\ell,\sigma_2) = 1$, geldt

$$BACK(\ell_1,\ell,\sigma_2) = LEFT(\ell_2,\ell,\underbrace{\sigma_2[\ell \longmapsto <1,0,r,\ell_1>]}_{\equiv \sigma_3}).$$

Wederom valt te bewijzen dat $|Nodes(\ell_2,\sigma_3)| < |Nodes(\ell,\sigma)|$. Dientengevolge geldt op grond van de inductiehypothese:

$$LEFT(\ell_2,\ell,\sigma_3) = BACK(\ell_2,\ell,\underbrace{Mark(\ell_2,\sigma_3)}_{\equiv \sigma_4}).$$

Daar $\neg(\ell=NIL)$ en $flagbit(\ell_2,\sigma_3) = 0$, geldt

$$BACK(\ell_2,\ell,\sigma_4) = BACK(\ell,r,\sigma_4[\ell \longmapsto <1,0,\ell_1,\ell_2>).$$

Tot slot rest ons nog af te leiden dat

$$\sigma_4[\ell \longmapsto <1,0,\ell_1,\ell_2>] = Mark(\ell,\sigma):$$

$\sigma_4[\ell \longmapsto <1,0,\ell_1,\ell_2>] \equiv$

$\mathrm{Mark}(\ell_2,\sigma_3)[\ell \longmapsto <1,0,\ell_1,\ell_2>] \equiv$

$\mathrm{Mark}(\ell_2,\sigma_2[\ell \longmapsto <1,0,r,\ell_1>])[\ell \longmapsto <1,0,\ell_1,\ell_2>] =$ (eig. C)

$\mathrm{Mark}(\ell_2,\ell_2[\ell \longmapsto <1,0,\ell_1,\ell_2>]) \equiv$

$\mathrm{Mark}(\ell_2,\mathrm{Mark}(\ell_1,\sigma_1)[\ell \longmapsto <1,0,\ell_1,\ell_2>]) \equiv$

$\mathrm{Mark}(\ell_2,\mathrm{Mark}(\ell_1,\sigma[\ell \longmapsto <1,1,\ell_2,r>])[\ell \longmapsto <1,0,\ell_1,\ell_2>]) =$ (eig.C)

$\mathrm{Mark}(\ell_2,\mathrm{Mark}(\ell_1,\sigma[\ell \longmapsto <1,0,\ell_1,\ell_2>])) \overset{!}{=} \mathrm{Mark}(\ell,\sigma)$.

Samenvattend hebben we dan bewezen dat geldt:

$$\mathrm{LEFT}(\ell,r,\sigma) = \mathrm{BACK}(\ell,r,\sigma_4[\ell \longmapsto <1,0,\ell_1,\ell_2>]) = \mathrm{BACK}(\ell,r,\mathrm{Mark}(\ell,\sigma)).$$

□

GEVOLG: (Correctheid van de DEUTSCH-SCHORR-WAITE markeringsalgoritme voor gerichte dyadische grafen [4].) *Onder de in de vorige stelling beschreven voorwaarde geldt:*

$$\mathrm{LEFT}(\ell,\mathrm{NIL},\sigma) = <\ell,\mathrm{Mark}(\ell,\sigma)>.$$

Tot slot wil ik opmerken dat KOWALTOWSKI [5] een veelbelovend begin heeft gemaakt met een systematische beschrijving van de representatie van circulaire en deelstructuren gemeenschappelijk hebbende data-structuren. Jammer genoeg beperkt hij zich in zijn werk tot partiële correctheid, en geeft hij geen totale-correctheidsbewijzen, waarvoor de inductieargumenten weer wat gecompliceerder zijn.

LITERATUUR

[1] ROEVER, W.P. DE, *Een correctheidsbewijs van de SCHORR-WAITE markeringsalgoritme voor binaire bomen*, in: Colloquium Programmacorrectheid, Mathematisch Centrum Syllabus MCS 21(1975).

[2] DWYER, S., *Simple algorithms for traversing a tree without auxiliary stack*, Inf. Proc. Lett. 2(1973) 143-145.

[3] TOPOR, R., *The correctness of the Schorr-Waite list marking algorithm*, Memorandum MIP-R-104, School of Artificial Intelligence, University of Edinburgh (1974).

[4] SCHORR, M. & W.M. WAITE, *An efficient machine-independent procedure for garbage collection in various list structures*, CACM 10(1967) 501-506.

[5] KOWALTOWSKI, T., *Correctness of programs manipulating data structures*, Ph.D. Thesis, Berkeley (1973).

MATHEMATICAL CENTRE SYLLABUS 25, 1976, VIII, 133-153

PORTABILITEIT

P. KLINT

0. INLEIDING

Het gebeurt iedere dag opnieuw. Een rekenmachine wordt vervangen door één van een nieuwer type, bij voorkeur van een andere fabrikant. Een elders ontwikkelde vertaler moet snel voor eigen gebruik beschikbaar worden gemaakt. Iemand verandert van werkkring en wil door hemzelf geschreven programma's blijven gebruiken. Op dergelijke momenten wreekt zich het gebrek aan standaardisatie van apparatuur en programmatuur.

Terwijl er voor numerieke en kommerciële toepassingen enige min of meer gestandaardiseerde programmeertalen bestaan (FORTRAN,ALGOL 60,COBOL) is dit voor niet-numerieke problemen nauwelijks het geval. Hierdoor zijn er juist op dit gebied een aantal methoden ontwikkeld om programma's meer *portable* te maken. In het vervolg zullen we een programma portable of *draagbaar* noemen, als het werk vereist voor het overbrengen van dit programma van de ene machine (de *donor*machine) naar een willekeurige andere machine (de *doel*-machine) klein is vergeleken met het werk dat nodig is om het programma op de doelmachine opnieuw te implementeren. Om de gedachten te bepalen: de transportkosten moeten minder dan tien procent van de initiële implementatiekosten bedragen.

Al in 1958 is een voorstel gedaan(SHARE [1], STEEL [2]) om het zogenaamde N * M-vertalersprobleem op te lossen. N probleemgerichte talen implementeren op M machines vereist immers N * M vertalers. Voorgesteld werd om op iedere computer een vertaler van UNCOL (UNiversal Computer Oriented Language) naar de betreffende machinetaal te maken. Zodra voor een probleemgerichte taal PGT een vertaler naar UNCOL beschikbaar is, komt PGT beschikbaar op iedere machine met een UNCOL-vertaler.

De UNCOL-gedachte heeft om twee redenen gefaald; aan de ene kant moet UNCOL efficiënt naar *alle* machinetalen vertaald kunnen worden en aan de andere kant moeten *alle* probleemgerichte talen efficiënt naar UNCOL vertaald

kunnen worden. Dit komt neer op het ontwerpen van een universele probleemgerichte taal.

In het vervolg komen verschillende methoden aan de orde ter oplossing en/of verlichting van het portabiliteitsprobleem. Er is een tamelijk willekeurig onderscheid gemaakt tussen *hardware*-en *software*-oplossingen, dat dient om de presentatie te vereenvoudigen. Ter sprake komen:

1. HARDWARE-OPLOSSINGEN
   1.1 *Netwerken*
   1.2 *Simulatie en emulatie*

2. SOFTWARE-OPLOSSINGEN
   2.1 *T-diagrammen*
   2.2 *Zelfkompilatie*
   2.3 *Implementatietalen*
   2.4 *Tussentalen*
   2.5 *Tussentaal en abstrakte machine*
   2.6 *Abstrakte machine en aktuele machine*
   2.7 *Macroprocessors*
      2.7.1 *GPM*
      2.7.2 *ML/I*
   2.8 *Realizering van Implementatietaal en tussentaal*
   2.9 *Taalhierarchieën*

1. HARDWARE-OPLOSSINGEN

1.1. *Netwerken*

Een benadering van het portabiliteitsprobleem die fundamenteel afwijkt van andere methoden, is het gebruik van *computernetwerken:* het probleem wordt niet opgelost maar omzeild. Door middel van telekommunikatie-technieken verschaft men zich toegang tot de gewenste programmatuur op de donormachine en hierdoor vervalt de noodzaak deze programma's aan de eigen computer aan te passen. Een van de bekendste netwerken is op dit moment het ARPANET (SHER [3]). Ongeveer veertig, merendeels verschillende, rekenmachines in de Verenigde Staten en Engeland zijn in dit net met elkaar verbonden. De aangesloten installaties variëren van ILLIAC IV, IBM 370 en B6700 tot

kleinere machines zoals de PDP11/45.

Een verdere uitweiding over technologie en problemen van netwerken blijft achterwege omdat deze te sterk vallen buiten het kader van de nog te bespreken methoden. Bovendien wordt het probleem niet opgelost voor geïsoleerde computers.

1.2. *Simulatie en emulatie*

Simulatie en emulatie kunnen gedefinieerd worden als het nabootsen van het gedrag van het ene (computer-)systeem door een ander systeem. Men spreekt van simulatie als instrukties van een na te bootsen (donor)machine niet direkt door de hardware van de doelmachine uitgevoerd worden, maar op het moment van executie door een simulatieprogramma, de *simulator*, geïnterpreteerd worden. De simulator heeft onder andere tot taak om

- te simuleren instrukties te dekoderen (operatiekode isoleren, adreskonversie van operanden uitvoeren) en op grond hiervan een simulator-routine aan te roepen.
- in- en uitvoer te simuleren (rekening houdend met record-lengtes, dataformaten), evenals andere faciliteiten die door het operating-systeem van de donormachine geboden worden.

Simulatie heeft echter twee nadelen. In de eerste plaats is het schrijven van een simulator niet triviaal, zeker niet als de architekturen van de donor- en doelmachine sterk van elkaar verschillen. In de tweede plaats leidt simulatie meestal tot een inefficiënt resultaat door de vergrote executietijd van het te simuleren programma. Iedere instruktie van de donormachine wordt namelijk door *meerdere* instrukties van de doelmachine gesimuleerd. Bovendien moeten parallelle processen in de donormachine, zoals instruktiedekodering, sequentieel uitgevoerd worden in de doelmachine.

Een aantal van bovengenoemde bezwaren kan opgelost worden door de na te bootsen machine te *emuleren*. Het enige verschil tussen emuleren en simuleren wordt gevormd door het niveau waarop het emulatie- respektievelijk simulatieprogramma wordt uitgevoerd. De simulator executeert op het niveau van de konventionele computer; de emulator op het daaronder gelegen *microprocessor* niveau (HUSSON [4]). Doordat de cyclustijd van de microprocessor een orde kleiner is dan die van de gewone processor, leidt emulatie tot een aanzienlijk efficiënter resultaat. In sommige microprocessors kan bovendien het parallellisme behouden blijven.

Simulatie en emulatie zijn, afgezien van netwerken, de enige methoden

waarbij het over te brengen programma *ongewijzigd* blijft. Hierdoor treedt er geen instructievermeerdering op zoals het geval is bij de nog te bespreken kompilatie- en macro-expansiemethode, waar het oorspronkelijke programma *vertaald* wordt naar de machinetaal van de doelmachine. In situaties waar het op de doelmachine beschikbare geheugen klein is, kan dit een beslissende faktor zijn. Het is te verwachten dat door het toenemend gebruik van microprogrammering, emulatie van steeds meer belang zal worden.

2. SOFTWARE-OPLOSSINGEN

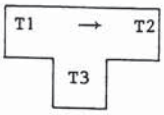
Het bespreken van de nu volgende methoden wordt vereenvoudigd door eerst een notatie in te voeren, die speciaal ontworpen is om interakties tussen verschillende vertalers te beschrijven.

2.1. *T-diagrammen*

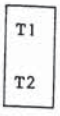
Dit formalisme, T-diagram genoemd (EARLEY & STURGIS [5]) maakt gebruik van de volgende bouwstenen:

- een programma geschreven in taal T ter berekening van funktie F:

- een vertaler van taal T1 naar taal T2 geschreven in T3:

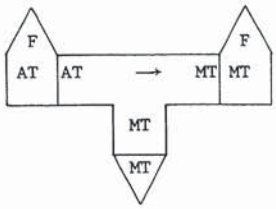


- een interpreter voor taal T1 geschreven in T2:



- een machine die programma's in taal T executeert:

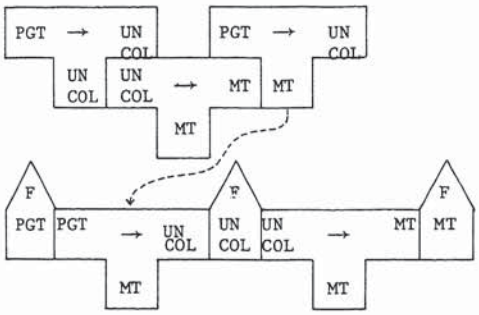
Uitgaande van een machine met machinetaal MT en assembleertaal AT, kan de vertaling van een programma ter berekening van F voorgesteld worden door:



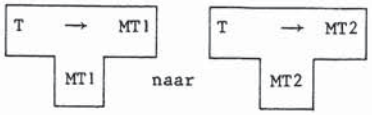
Het resultaat is een programma voor F in machinetaal. In het vervolg zal meestal de driehoek die een machine aangeeft weggelaten worden, als hierdoor geen verwarring ontstaat.

De UNCOL-methode laat zich als volgt beschrijven. Als gegeven zijn een vertaler van de probleemgerichte taal PGT naar UNCOL in UNCOL, en een vertaler van UNCOL naar machinetaal MT in MT, dus

dan verkrijgt men door de volgende operaties zowel een PGT-vertaler in machinetaal, als een vertaling van een in PGT geschreven programma:



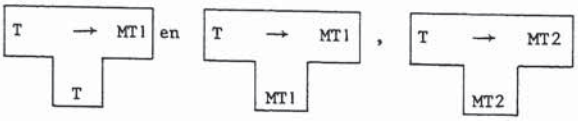
Het portabiliteitsprobleem voor vertalers is nu om



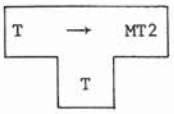
te transformeren. Het emuleren van MT1 is één oplossing. Het schrijven van de vertaler voor T in een hogere *implementatietaal* IT vormt een andere oplossing. We onderscheiden de gevallen IT = T en IT ≠ T. Alleen in dit laatste geval zal vertaling naar een *tussentaal* TT ter sprake komen. Dit houdt echter niet in dat geen tussentaal gebruikt zou kunnen worden voor IT = T.

2.2. *Zelfkompilatie* (IT = T)

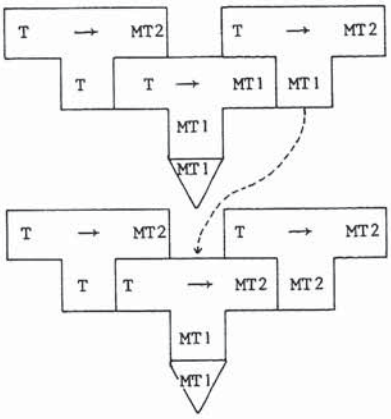
Het schrijven van een vertaler voor een taal in die taal zelf is een bezigheid die al sinds lang tot de folklore behoort. De WISP kompiler (WILKES [6], WAITE [7]) voor een eenvoudige lijstmanipulatietaal (die eigenlijk een macroprocessor blijkt te zijn) is een van de eerste voorbeelden van een zelfkompilerende vertaler. Het overbrengen van een dergelijk type vertaler vindt plaats door met behulp van



te konstrueren. Dit wordt bereikt door met de hand



te schrijven. Meestal zal alleen de kodegenerator in de oorspronkelijke vertaler gewijzigd moeten worden. Het gewenste resultaat wordt als volgt verkregen:

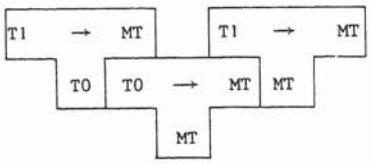


Alle operaties worden op de donormachine M1 uitgevoerd. Als M1 en M2 ver van elkaar verwijderd zijn of onvergelijkbare karakterkodes en data-formaten gebruiken, zal deze methode tamelijk traag zijn, doordat fouten in de kodegenerator voor M2 meestal pas ontdekt worden door vertaalde T-programma's op M2 te executeren. Deze methode faalt (meestal) als men geen toegang tot de donormachine heeft.

Een ander ernstig bezwaar is dat maar weinig talen zich ertoe lenen om hun eigen vertaler erin te schrijven. (Kies voor T bijvoorbeeld een taal die bijzonder geschikt is voor het manipuleren met driehoeksmatrices.) Bovendien wordt het eindresultaat ongunstig beïnvloed door konstrukties in T die niet bij het schrijven van de vertaler gebruikt worden, maar toch hun sporen in de gegenereerde kode achterlaten (zoals dynamische arrays).

Het is interessant om na te gaan hoe de allereerste versie van de

T-vertaler tot stand komt. Op de een of andere manier wordt een vertaler voor T0, een kleine deeltaal van T, gemaakt. Met behulp van de T0-vertaler kan een T1-vertaler, geschreven in T0, verkregen worden, waarbij T1 een deeltaal van T is die T al meer benadert:



Dit proces kan willekeurig vaak herhaald worden tot de T-vertaler verkregen is. De zojuist beschreven methode staat bekend onder de naam *bootstrapping*. Hierbij moet nog opgemerkt worden dat het niet *noodzakelijk* is dat opeenvolgende talen uitbreidingen van hun voorgangers zijn. Vaak kiest men T0 = SNOBOL 4 om een subtaal van T te implementeren.

Oppervlakkig gezien leidt bootstrapping tot een steeds minder efficiënt resultaat, naarmate het aantal stappen groter is. De onjuistheid van deze gedachte wordt aangetoond door twee soorten inefficiency te onderscheiden:

(a) de vertaler loopt langzamer dan nodig is;

(b) de vertaler genereert inefficiënte kode.

Type (a) heeft geen invloed op de volgende stap en type (b) leidt tot type (a) en verdwijnt dus in twee stappen.

De familie T0,...,Tn wordt overigens bedreigd door *inteelt*, die kan optreden doordat het ontbreken van bepaalde primitieve operaties in T0 een essentiële (ongunstige) invloed op de struktuur van volgende vertalers heeft.

2.3. *Implementatietalen* (IT ≠ T)

De meest gebruikte methode om draagbaarheid van een programma te verkrijgen is om het in een wijd en zijd verbreide programmeertaal te schrijven. Er zijn verschillende oorzaken waardoor deze methode meestal faalt:

- lokale afwijkingen en uitbreidingen van de standaard-taaldefinitie.
- niet kompatibele subroutine-bibliotheken.
- verschillen in randapparatuur, dataformaten, karakterkodes.
- geheugenkapaciteit: een programma dat geschreven is voor een grote computer zal pas na veel wijzigingen op een kleinere machine te gebruiken zijn.

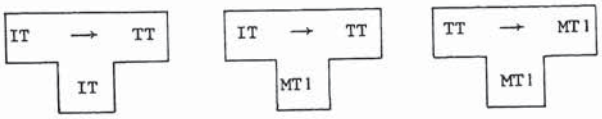
Het is bijzonder moeilijk, zo niet onmogelijk, zich al deze moeilijkheden en beperkingen te realizeren als men gebruik maakt van een bestaande programmeertaal. Bovendien blijft het bezwaar gelden dat er voor niet-numerieke toepassingen nauwelijks geschikte, overal beschikbare, talen bestaan. Misschien vormen LISP en SNOBOL een uitzondering op deze regel.

Het alternatief voor bovenstaande werkwijze is het speciaal voor de implementatie van T ontwerpen van een implementatietaal IT. IT kan dan juist die primitieven en taalkonstrukties bevatten die geschikt zijn om T op efficiënte wijze te implementeren. Het lijkt erop alsof het gebruik van IT alleen maar leidt tot uitstel van executie: eens zal hetzij T, hetzij IT op de doelmachine geïmplementeerd moeten worden. Er zal echter een methode ter sprake komen (DLIMP: Descriptive Language Implemented by means of a Macro Processor) die ontwerp en implementatie van IT aanzienlijk vereenvoudigt. In de volgende paragraaf brengen we eerst nog een verdere strukturering in de vertaler voor T aan.

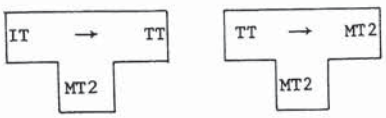
2.4. *Tussentalen*

In veel vertalers wordt gebruik gemaakt van een *interne representatie* van het te vertalen programma, die de kodegenerator stuurt. Denk aan triples, quadruples en parseringsbomen. De machine-onafhankelijkheid van een vertaler kan bevorderd worden door deze interne representatie de status van een taal te geven. De vertaling geschiedt in twee stappen: van te vertalen programma naar de tussentaal TT en van TT naar machinekode. De vertaler kan nu naar de doelmachine overgebracht worden, als voor deze nieuwe machine een vertaler van implementatietaal naar (nieuwe) machinetaal MT2 bestaat. Het probleem kan als volgt geformuleerd worden:

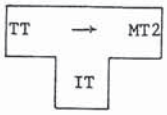
gegeven:



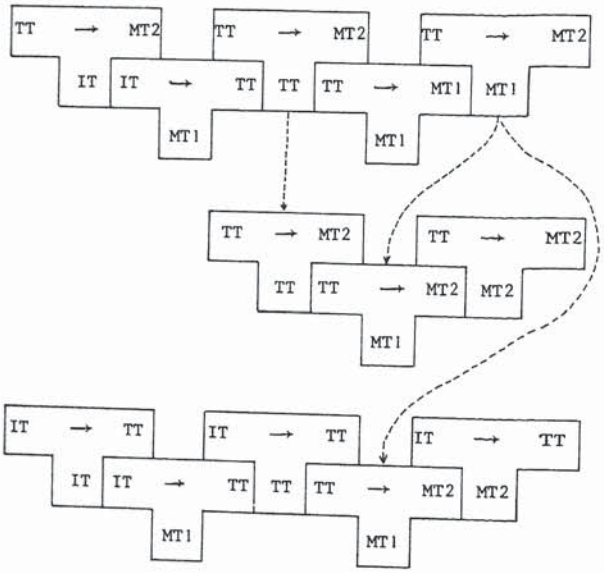
gevraagd:



Het probleem wordt opgelost door een vertaler van TT naar MT2 in IT te schrijven; IT is hier geschikt voor:



De gewenste vertalers kunnen nu in drie stappen verkregen worden:



Zodra de IT-vertaler op de doelmachine beschikbaar is, kan ook de T-vertaler overgebracht worden. Ook deze methode maakt alleen van de donormachine gebruik. Samenvattend komt de beschreven methode erop neer dat:

- de T-vertaler in een speciaal ontworpen taal IT geschreven is;
- T-programma's naar een tussentaal TT vertaald worden;
- overbrengen van de T-vertaler met behulp van de donormachine uitgevoerd wordt;
- iedere overbrenging een vertaler van tussentaal naar machinetaal van de doelmachine vereist.

2.5. *Tussentaal en abstrakte machine*

In het voorafgaande hebben we gezien dat de taal T in eerste instantie vertaald wordt naar de tussentaal TT. Deze tussentaal kan ook op een heel andere manier bekeken worden, namelijk door TT te beschouwen als de besturingstaal van een denkbeeldige *abstrakte machine* AM. Deze interpretatie van TT heeft het voordeel dat het ontwerpen van de abstrakte machine als deelprobleem geïsoleerd wordt. Bij het ontwerpen van AM moet met twee, elkaar vaak tegensprekende eisen rekening gehouden worden:

- De architektuur van AM moet bijzonder geschikt zijn om vertaalde T-programma's te executeren.
- De afbeelding van AM op bestaande computers moet zoniet triviaal, dan toch wel eenvoudig en zeker efficiënt zijn.

Dat het ontwerpen van een abstrakte machine een kunst is, waarvoor op dit moment nog geen regels te geven zijn blijkt uit een vergelijking van enkele machine-organisaties.

2.6. *Abstrakte machine en aktuele machine*

Het opsommen van alle bestaande machine-organisaties is een ondoenlijke zaak, die ons bovendien zou afleiden van het pad dat leidt tot draagbare programma's. De verschillen worden geïllustreerd aan de hand van twee voorbeelden, namelijk de organisatie van register/processor en van procedure-aanroep.

De meest voorkomende vormen van *register/processor*-organisatie zijn:

- geen registers: alle instrukties halen hun operanden uit het geheugen en laten hun resultaat daarin ook weer achter;
- één arithmetisch register;
- meerdere arithmetische registers: operanden mogen zowel in een register als in het geheugen staan;
- register-file: alle operanden moeten in een register staan;
- stack: operanden en resultaat bevinden zich op een stack.

Deze verschillen maken duidelijk dat het bijvoorbeeld onverstandig is om naamloze tussenresultaten die ontstaan bij de evaluatie van een arithmetische expressie expliciet te bewaren. In een stackmachine worden ze automatisch door de hardware bewaard, terwijl in een register-filemachine de topelementen van een (gesimuleerde) stack in een aantal registers bewaard moeten worden.

Het tweede voorbeeld betreft procedure-aanroepen. Afgezien van het doorgeven van parameters, moet bij het aanroepen van een procedure status-informatie gered worden. Dit kan gerealiseerd zijn doordat bij het uitvoeren van de sprong naar de procedure

- de status door de hardware op een stack geplaatst wordt;
- de status door de hardware in het geheugen gezet wordt op een plaats die in een vaste relatie staat tot de procedure;
- er een aparte instruktie bestaat voor het redden van de statusinformatie.

Hopelijk illustreren beide voorbeelden, dat bij het ontwerpen van een abstrakte machine bepaalde konstrukties *vaag of ongespecificeerd* moeten blijven, opdat een toekomstige implementator voldoende vrijheid heeft om deze abstrakte machine zo efficient mogelijk op zijn eigen machine te realizeren. In POOLE & WAITE [8] worden meer aspekten van machine-architektuur met elkaar vergeleken.

Er zijn tot nu toe verschillende pogingen gedaan om abstrakte machines met daarbij behorende talen te ontwerpen. We noemen hier maar enkele:

- FLUB (First Language Under Bootstrap) gebruikt voor de STAGE-2-macroprocessor (WAITE [7]);
- LOWL gebruikt voor de ML/I macroprocessor (BROWN [9]);
- SIL (SNOBOL Implementation Language) (GRISWOLD [10]);
- OCODE gebruikt voor de BCPL-vertaler (RICHARDS [11,12]);
- JANUS, de taal met de twee gezichten: het ene kijkt naar de bestaande hardware, het andere naar mogelijke toepassingen. Met name in dit systeem wordt de "vaagheids"-idee toegepast (COLEMAN, POOLE & WAITE [13]);
- ZCODE gebruikt voor de ALGOL 68 C vertaler (BOURNE [14]).

2.7. *Macroprocessors*

Macroprocessors (CAMPBELL-KELLY [15]) vormen een krachtig hulpmiddel bij het realizeren van een abstrakte machine. Om deze opmerking te illustreren komen nu twee macroprocessors aan de orde; de een om zijn eenvoud en de ander om zijn komplexiteit. Ook zal blijken dat de historisch gegroeide associatie tussen macro's en assembleertalen onjuist is.

2.7.1. *GPM*

GPM (General Purpose Macrogenerator) is een van de eerste programmeertaal-onafhankelijke macroprocessors C en wordt beschreven in STRACHEY [16]. De in GPM gebruikte konstrukties hebben een zeer primitieve syntaktische struktuur, zodat we ons op semantiek en evaluatiemechanisme kunnen koncentreren.

In principe kopieert GPM symbolen van een invoerstroom naar een uitvoerstroom. Van deze regel wordt alleen afgeweken als in de invoerstroom een *macro-aanroep* voorkomt. Een macro-aanroep heeft altijd de vorm:

§ *macronaam, parameter* 1,...,*parameter* n ;

Een voorbeeld is: §mac,123,abc;
Hier is "mac" de macronaam, de string "123" parameter 1 en "abc" parameter 2.
Een macro zonder parameters wordt bijvoorbeeld aangeroepen door:

§macrozonderparameters;

Voordat een macro aangeroepen kan worden, moet met de macronaam een string, de *macro-body*, geassocieerd zijn. Deze string mag de speciale symbolen ~1, ~2, enzovoort bevatten, die staan voor de eerste, de tweede, enzovoort formele parameter. Het symbool ~0 staat voor de macronaam zelf. Als een macro-aanroep geëvalueerd wordt, worden deze symbolen vervangen door een kopie van de evaluatie van de aktuele parameters van de huidige aanroep. Als de macronaam "ABC" geassocieerd is met de string "AB~1C~2AB" dan zal de aanroep "§ABC,XY,PQ;" de string "ABXYCPQAB" produceren.

Macroaanroepen mogen op elke willekeurige plaats voorkomen zoals blijkt uit een aantal voorbeelden. Uitgaande van de associaties:

macronaam	macrobody
A	A~1A
B	B§A,X~1X;B
APA	P~1~1P

zijn de volgende resultaten te verwachten:

macro-aanroep	resultaat
§A,C;	ACA
§A,§A,C;;	AACAA
§A,XDX;	AXDXA
§B,D;	BAXDXAB
§A,P;	APA
§APA,Y;	PYYP
§§A,P;,Y;	PYYP

Voordat we de GPM-evaluatie-algorithme beschrijven komen nog twee punten, ter sprake: *stringquotes* en de *systeemmacro* DEF.

De evaluatie van macro-aanroepen in een string kan onderdrukt worden door deze string te omgeven met de stringquotes "<" en ">". Inplaats van evaluatie wordt één laag stringquotes verwijderd:

string	resultaat
Q<§A,C;>R	Q§A,C;R
§A,<§A,X;>;	A§A,X;A
Q<§>R<;>	Q§R;
A<<B>>C	A<B>C

(Het tweede voorbeeld maakt gebruik van de al eerder geïntroduceerde macro "A").

De systeemmacro DEF wordt gebruikt om een macronaam met een macrobody te associëren. De zojuist weer gebruikte macro "A" wordt gedefinieerd door:

```
§DEF,A,<A~1A>;
```

Het is nuttig de macrobody met stringquotes te omgeven om voortijdige evaluatie op het moment van definitie te voorkomen.

De GPM-evaluatie-algorithme is als volgt:

1. Evalueer de macronaam en aktuele parameters van links naar rechts. Mogelijkerwijs moeten hierbij in de macronaam of parameters voorkomende macro-aanroepen éérst geëvalueerd worden.
2. Zoek de meest recente definitie van de macronaam op.
3. Evalueer de in stap 2 gevonden macrobody op dezelfde wijze als de oorspronkelijke tekst. Voor een aanroep van een formele parameter (~i,i=0,1,2,...) wordt het in stap 1 verkregen resultaat letterlijk gekopieerd.
4. Bij het bereiken van het einde van de macrobody worden de macronaam, de parameters en alle lokaal gedefinieerde macro's verwijderd.
5. Scanning van de (vorige) invoerstroom wordt hervat na de puntkomma van de huidige macro-aanroep.

Een gevolg van bovenstaande algorithme is dat de aanroep

```
§A,X,U,§DEF,A,<~1~2~1>;;
```

als resultaat "XUX" levert. Tijdens evaluatie van de parameters wordt een (lokale) macro met naam "A" gedefinieerd.

Tenslotte nog twee toepassingen van GPM. De eerste betreft het defi-

niëren van de opvolger-funktie voor gehele getallen. Zonder verdere toelichting geven we de definitie van de macro OPVOLGER, zodat de aanroep: "§OPVOLGER,2;" als resultaat "3" heeft:

```
§DEF,OPVOLGER,<§1,2,3,4,5,6,7,8,9,10,§DEF,1,<~>~1;;>;
```

De tweede toepassing laat zien hoe, met zeer beperkte hulpmiddelen, konditionele expressies gerealizeerd kunnen worden. Een met "if A=B then C else D" vergelijkbare konstruktie verkrijgt men in GPM door:

```
§A,§DEF,A,<D>;§DEF,B,<C>;;
```

Als A en B tekstueel aan elkaar gelijk zijn wordt de eerste definitie van A gebruikt, anders de tweede.

2.7.2. *ML/I*

ML/I (Macro Language I) is ontwikkeld door BROWN [17] en verschaft naast vele systeemmacro's ook de mogelijkheid om de syntaktische vorm van een macroaanroep te definiëren. Bovendien is het ML/I systeem zelf draagbaar en verdient om die reden ook onze aandacht. De algemene vorm van een macro-aanroep is:

naam parameter 1 *delimiter* 1 ... *parameter* n *delimiter* n

Zowel *naam* als *delimiters* worden bij definitie van een macro vastgelegd en zijn dus vrij te kiezen. Voorbeelden van naam/delimiter-strukturen zijn:

```
WHILE argument 1 DO argument 2 OD
MOVE argument 1 TO argument 2;
```

In het eerste voorbeeld is "WHILE" de macronaam en zijn "DO" en "OD" de delimiters.

Het invoegen van aktuele parameter i gebeurt met behulp van de *insertie* "%Ai.". (De "A" kontroleert de evaluatiemethode van het argument. De symbolen "%" en "." kunnen willekeurig gekozen worden, maar hierop gaan we niet nader in.)

De body van de macro MOVE zou de volgende vorm kunnen hebben als we een COBOL-achtig assignment statement willen vertalen in het ALGOL equivalent:

```
%A2.:= %A1.;
```

Met de systeemmacro MCDEF kunnen nieuwe macro's gedefinieerd worden. De volledige definitie van de MOVE-macro zou eruit kunnen zien als:

```
MCDEF MOVE TO;
AS < %A2.:= %A1. ; >
```

Hierbij zijn "<" en ">" weer gebruikt om voortijdige evaluatie te voorkomen.

Andere eigenschappen van ML/I die hier niet ter sprake komen zijn onder andere:

- komplexe delimiterstrukturen met optionele argumenten en delimiters (vergelijk de IF THEN FI macro met de IF THEN ELSE FI macro) en alternatieve delimiters. Een konditioneel statement waarin de Boolean expressie een van de operatoren "EQ", "NE", "GE" of "LE" mag bevatten wordt bijvoorbeeld gedefinieerd door:

```
MCDEF IF OPT EQ OR NE OR GE OR LE ALL THEN FI AS < ... >
```

  De alternatieve delimiters staan tussen "OPT" en "ALL" en worden gescheiden door "OR"s.
- konditionele sprongen tijdens macro-expansie
- arithmetiek tijdens macro-expansie.

In TANENBAUM [18] wordt beschreven hoe ML/I gebruikt kan worden om op tamelijk eenvoudige wijze een vertaler te schrijven van de hogere programmeertaal SAL naar MACRO-11, de assembleertaal van de PDP11.

Een macroprocessor als ML/I maakt duidelijk dat het verschil tussen een syntaxgestuurde vertaler en een macroprocessor op sommige punten bijzonder klein is. Er zijn nog verschillende niet of nauwelijks onderzochte problemen op het gebied van macroprocessors. In de eerste plaats vormt *parsering van macroparameters* een probleem. Stel dat de eerder ten tonele gevoerde MOVE-macro aangeroepen wordt als:

```
MOVE 1 TO 2;
```

In de body van de MOVE-macro moet expliciet getest worden of aan de tweede parameter ("2") wel geassigneerd kan worden, omdat anders de kode "2:=1;" gegenereerd wordt. Het zou echter mogelijk moeten zijn om bij de macrodefinitie de syntax van parameters te specificeren. In LEAVENWORTH [19] worden enkele suggesties gedaan.

Een ander nog gedeeltelijk braak liggend terrein is het vóórvertalen

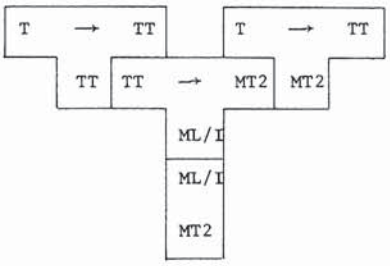
van macrodefinities om de efficiency van de macroprocessor als geheel te vergroten. Er is mij maar één geval bekend (MAURER [20]) waarin deze methode op een macroassembler is toegepast.

2.8. *Realizering van implementatietaal en tussentaal*

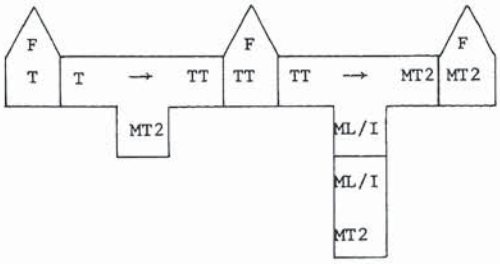
Nu we een krachtige macroprocessor tot onze beschikking hebben is de werkwijze bij het implementeren van en machine-onafhankelijke vertaler min of meer duidelijk:

- ontwerp de implementatietaal IT;
- ontwerp de tussentaal TT;
- schrijf de T-vertaler in IT;
- schrijf ML/I-macro's die IT op TT afbeelden.

Op deze manier verkrijgt men een vertaler van T naar TT geschreven in TT. Het overbrengen van de T-vertaler geschiedt door, alweer met een macroprocessor, TT op de machinetaal MT2 van de doelmachine af te beelden. Als ML/I op de doelmachine beschikbaar is, levert



een vertaler van T naar TT op de doelmachine. Een programma, dat in T geschreven is wordt als volgt vertaald:



Het feit dat ML/I *interpretatief* werkt wordt in deze diagrammen weerspiegeld.

Uiteraard kan de vertaling van TT naar MT2 ook op een andere wijze tot stand komen.

Deze methode wordt DLIMP genoemd (Descriptive Language Implemented by means of a Macro Processor) en is onder andere gebruikt bij de implementatie van ML/I zelf. Als implementatietaal is gebruikt de taal L (BROWN [21]) en als tussentaal LOWL (BROWN [9]).

Aangezien het programma in twee versies beschikbaar is (namelijk in L en LOWL) doet zich de vraag voor of het verschil uitmaakt of men op de doelmachine de L dan wel de LOWL versie implementeert. Metingen in [9] wijzen uit dat het implementeren via LOWL minder dan 10% inefficiency veroorzaakt. Hier staat tegenover dat een implementatie via L ongeveer viermaal zoveel werk kost.

Een voorbeeld zal duidelijk maken dat het implementeren via de (hogere) implementatietaal leidt tot een efficiënter resultaat. In de taal L kan als speciaal geval van het assignment statement voorkomen:

```
SET X = X - 1
```

In een implementatie die van L uitgaat is dit speciale geval gemakkelijk te detekteren, zodat van een eventuele decrement-instruktie op de doelmachine gebruik gemaakt kan worden. De LOWL versie van bovenstaand statement is:

```
LAV  X     laad variabele X in A-register
SAL  1     trek van A-register literal 1 af
STV  X     store A-register in variabele X.
```

Herkenning van hetzelfde speciale geval is nu veel moeilijker, omdat drie opeenvolgende *verschillende* macro's herkend moet worden.

2.9. *Taalhiërarchieën*

Bovenstaande observatie geeft aanleiding tot het uitbreiden van IT en TT tot een hiërarchie van implementatietalen IT1,...,ITn. Met toenemende index neemt de komplexiteit van de bijbehorende taal toe. Ditzelfde geldt voor de primitieven waar een taal gebruik van maakt. ITn kan bijvoorbeeld gedefinieerd zijn voor een abstrakte machine AMn met associatief geheugen, automatische patroonherkenning en ingewikkelde stringoperaties. Het implementeren van ITn op een computer met deze eigenschappen is eenvoudig en

leidt tot een efficiënt resultaat. IT1 kan bijvoorbeeld gedefinieerd zijn voor een abstrakte machine AM1 met één accumulator en een beperkt instruktie-repertoire.

Iedere implementator kan een taalniveau kiezen dat het best geschikt is voor zijn doeleinden. Als IT, de basis vormt voor *meerdere hiërarchieën* is het van groot belang dat IT1 *uitbreidbaar* is, omdat zich anders dezelfde inteelt verschijnselen kunnen voordoen als bij het bootstrappen van een zelf-kompilerende kompiler.

Er is een opvallende overeenkomst tussen taalhiërarchieën en *gestruktureerd programmeren*.

3. BESLUIT

Er is op dit moment een groot aantal technieken beschikbaar om draagbare programma's te konstrueren. Slechts enkele hiervan hebben we de revue laten passeren. Het is misschien duidelijk geworden dat het schrijven van draagbare programma's een *specialisme* is; er bestaan niet een paar truukjes om een willekeurig programma draagbaar te maken.

Bovendien hebben we ons beperkt tot draagbare vertalers, omdat deze een prototype vormen voor grote systeemprogramma's. Portabiliteitstechnieken kunnen natuurlijk ook op andere systeemprogramma's toegepast worden: een operating-systeem bijvoorbeeld. Het laatste woord is over dit onderwerp zeker nog niet gezegd.

LITERATUUR

[1] SHARE Ad-hoc Committee on Universal Languages, *The problem of program communication with changing machines*, CACM 1 (1958) 8, 12-18 en CACM 1 (1958) 9, 9-15.

[2] STEEL Jr., T.B., *UNCOL: the myth and the fact*, in R. Goodman (Ed.), Annual Review in Automatic Programming, Vol. 2 (1960) 325-344.

[3] SHER, M.S., *A case study in networking*, DATAMATION 20 (1974) 3, 56-59.

[4] HUSSON, S.S., *Microprogramming principles and practices*, Prentice-Hall, Englewood Cliffs, N.J., 1970.

[5] EARLEY, J. & H. STURGIS, *A formalism for translator interactions*, CACM 13 (1970) 10, 607-617.

[6] WILKES, M.V., *An experiment with a self-compiling compiler for a simple list-processing language*, in R. Goodman (Ed.), Annual Review in Automatic programming, Vol. 4 (1964) 1-48.

[7] WAITE, W.M., *Implementing Software for non-numeric applications*, Prentice-Hall, Englewood Cliffs, N.J., 1973.

[8] POOLE, P.C. & W.M. WAITE, *Portability and adaptability*, in F.L. Bauer (Ed.), Advanced Course on Software Engineering, Springer-Verlag, Berlin, 1973, 183-277.

[9] BROWN, P.J., *Macro processors and techniques for Portable Software*, Wiley, London, 1974.

[10] GRISWOLD, R.E., *The macro implementation of SNOBOL 4*, Freeman & Co, San Francisco, 1972.

[11] RICHARDS, M., *BCPL: A tool for compiler writing and systemprogramming*, in SJCC 34 (1969) 557-566.

[12] RICHARDS, M., *The portability of the BCPL compiler*, Software-Practice and Experience 1 (1971) 2, 135-146.

[13] COLEMAN, S.S., P.C. POOLE, & W.M. WAITE, *The mobile programming system, Janus*, Software-Practice and Experience 4 (1974) 1, 5-23.

[14] BOURNE, S.R., *Z-CODE: A simple machine*, 1973 en 1974 (revisie), niet gepubliceerd.

[15] CAMPBELL-KELLY, M., *An introduction to macros*, Macdonald/Elsevier, 1973.

[16] STRACHEY, C., *A general purpose macrogenerator*, Computer Journal 8 (1965) 225-241.

[17] BROWN, P.J., *ML/I user's manual (Fourth edition)*, University of Kent at Canterbury, 1970.

[18] TANENBAUM, A.S., *A general purpose macro processor as a poor man's compiler-compiler*, (verschijnt binnenkort).

[19] LEAVENWORTH, B.M., *Syntax macros and extended translation*, CACM 11 (1966) 11, 790-793.

[20] MAURER, W.D., *The compiled macro assembler*, in SJCC 34 (1969) 89-93.

[21] BROWN, P.J., *The use of ML/I in implementing a machine independent language in order to bootstrap itself from machine to machine*, Technical Memorandum No. 68/1, University Mathematical Laboratory Cambridge, 1968.

MATHEMATICAL CENTRE SYLLABUS 25, 1976, IX, 155-176

OBJECT CODE GENERATION

H.J. BOOM

0. INTRODUCTION

A compiler is a program for translating computer programs from one language, the "source language", into another language, the "object language". The object language usually closely resembles that accepted directly by the hardware of a real machine. The "object program", that is, the result of this translation, must usually undergo further processing to link it with other object programs and to place it into the main memory of the computer that is to run it.

The translation process consists of two parts:
- analysis of the "source program" in order to ascertain its meaning, and
- synthesis of the object program, which must express this same meaning in a form that can be run on the real computer.

This paper will discuss the synthesis of the object program. The synthesis process is often called "object code generation".

1. INPUT TO THE CODE GENERATOR

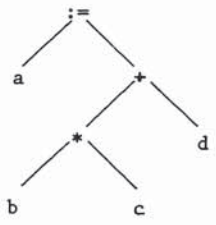
Since the input for the code generator is the output from the analysis phase, let us examine it briefly. The results of analysis are often expressed in "tree form" or in "Polish postfix form", but sometimes in other forms. Some compilers, on the other hand, do not use any intermediate form at all. Instead, routines in the analysis phase directly call the routines in the code generator. This, however, forces a certain inflexibility of approach in the code generator.

1.1. Tree notation

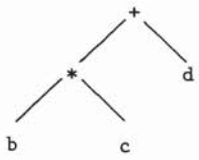
Consider the assignation

a := b * c + d

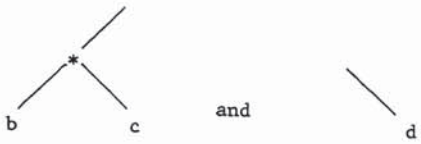
In tree form, this might appear as:



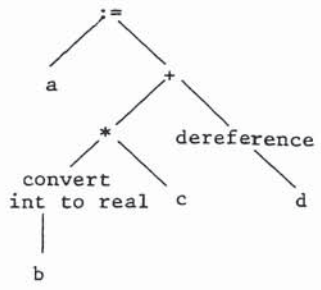
Operators, such as ":=", "+", and "*", are placed at the roots of trees or subtrees, and their operands are the branches. The plus sign, for example, is the root of the subtree



and it has two operands:



For more convenient interpretation of these trees by the code generator, it is desirable that all implicit type conversions in the original program be explicitly represented. If in our example the modes (as in Algol 68) of "a", "b", "c", and "d" are ref real, int, real, and ref int respectively, our tree would have the following appearance:



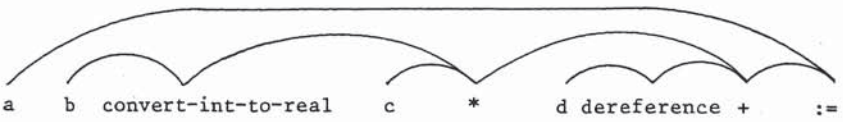
Using list processing techniques, such tree structures can easily be represented in a random access memory. The tree structure provides a high degree of flexibility in processing. Unfortunately, because of the presence of many pointers, these tree structures tend to be quite space-consuming. This is no trouble if the computer has sufficient main memory, or even enough paged virtual memory. If the compiler itself must explicitly use disk storage, processing tends to become complex or slow.

1.2. Polish notation

Another representation, "Polish postfix", is often used. It tends to be more compact, but it is not as easy to manipulate. In this notation, operands are placed first, in order, followed by their operator. The operator is thus a "postfix". Our assignation becomes:

```
a b convert-int-to-real c * d dereference + :=
```

It is still possible to see the tree structure in this form; indeed, there is a formal isomorphism between tress and postfix strings:



Postfix notation is more compact than tree notation because the pointers are not required. The branch structure is implicit, instead of explicit.

An important property of Polish notation, as we shall see later, is that it presents tokens in the order in which some code generation algorithms wish to see them. It is this property that must be preserved when extending the notation to handle untreelike language features. Unfortunately, some compilers wish to process the program parts in a different order, or to process some parts more than once, and this is difficult without explicit pointers.

1.3. *Other forms*

Some compilers use other forms of intermediate text. Intermediate text that resembles the "directive text" described in chapter 5 is very often found. Sometimes translation is broken into many stages, with different forms of text between the various stages.

2. STACK AND NONSTACK MACHINES

The importance of postfix notation for arithmetic expressions is that, on a special kind of machines, it indicates operations in an order in which they can be performed. Such a machine is called a *stack machine*. It has a pushdown stack capable of containing addresses and arithmetic values. This stack can contain many items (much like a stack of papers on a table), but at any moment only the top item is available. Items may be added to the top of the stack, or removed from the top of the stack. Arithmetic operators remove their operands from the stack and replace them by the result.

To execute a postfix string for an arithmetic expression, we execute its items, one at a time, from left to right, as follows:

- to execute an identifier, place its value onto the top of the stack.
- to execute an n-ary operator, remove the n top-most values from the

stack by the result of the operation.

If an identifier is a variable, i.e., it can appear as destination in an assignment, its value is an address of a storage location.

To demonstrate the execution of our example, we shall need a little notation. We shall let α stand for the address of the variable a, and δ for the address of the variable d. For typographical convenience, we write the stack horizontally, with the top on the right.

Let the values of b and c be 2 and 3.1, and let the contents of location δ be 4. Execution proceeds as follows:

stack	action
α	a
α 2	b
α 2.0	convert-int-to-real
α 2.0 3.1	c
α 6.2	*
α 6.2 δ	d
α 6.2 4	dereference
α 10.2	+
α	:=

(but the storage location addressed by α now contains the value 10.2)

If we had machines with hardware arithmetic, and if our programming languages were to consist only of features (such as arithmetic expressions) which fit nicely on stacks, our problems would be nearly over.

Unfortunately, not all machines have such hardware stacks, and most programming languages have unstacklike features, such as globally available variables and large arrays. Global variables must be accessible at all times, not just when they happen to appear atop the stack. There are a number of elegant and inelegant solutions to these problems.

There is another common kind of machine, the so-called *typical third-generation multi-register machine*. Such machines have a number of fast registers, usually between 4 and 32. All or most of the arithmetic can or must be done in these registers. Since the registers are rarely numerous enough to accommodate all necessary values, there is also an addressable main memory. Instructions are provided for transferring values between main memory

and the registers, and it is sometimes permitted to have one or both operands of an arithmetic operation in main storage. If an instruction needs to access main memory, it contains a *memory designator*, whose purpose it is to indicate which piece of main memory is to be accessed. Such a memory designator contains a *base*, which identifies uniquely a register, and a *displacement*, which is an integer. The contents of the register indicated by "base" is added to the integral "displacement". The resulting sum is the memory address.

This addressing scheme is particularly convenient for accessing operands which do not conveniently fit on a stack. Even on a stack machine, some such addressing mechanism is needed in order to find operands when placing them on the stack. In our example, the instruction used to *place the value of* 'c' *onto the stack* would contain such a base-displacement pair in order to address the piece of memory originally containing the value of c.

The stackless register machines must use either their many registers or their main storage to hold temporary values, such as the intermediate values '6.2' and '10.2' in our example.

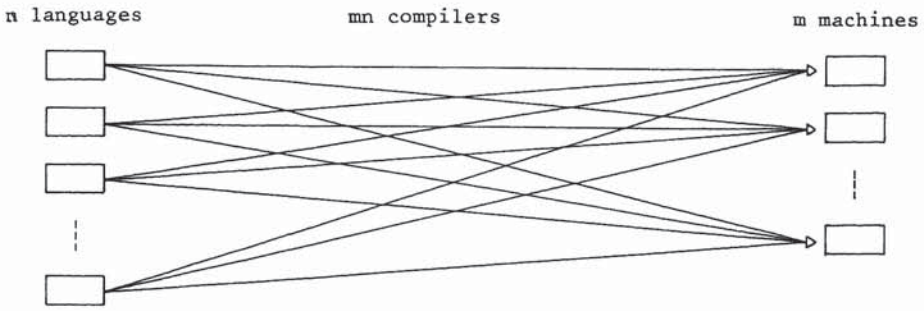
Another necessary use of a stack is for procedure calling. When called, a procedure causes an "activation record" to be placed on the stack. This activation record contains information peculiar to an individual call to the procedure, such as the local variables, or the return address. If the procedure is called recursively, there will arise one activation record for each (recursive) call. Such a stack is indispensable, and non-stack machines must simulate it with software.

Let us summarize the important points about machines for Algol-like languages:

- They need a stack mechanism for procedure calling.
- They need a base-displacement mechanism for accessing local and global variables.
- They need a means for accessing intermediate results from arithmetic expressions. This means may be either of the above two.

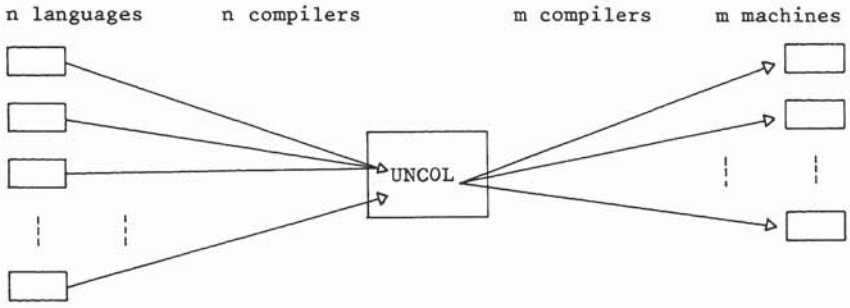
3. PORTABILITY, AND THE UNCOL PROBLEM

It seems at first that, to implement n programming languages on m machines, one must write mn compilers:



We shall not discuss further the languages in which these compilers are written; we assume there are generally available languages appropriate for the writing of compilers.

In order to significantly reduce this number of compilers, it has been proposed to define a Universal Computer Oriental Language (UNCOL). This language would serve as an intermediate step in the translation process:



It would then be necessary to write n+m compilers, and each of these compilers would be simpler than each of the original nm compilers. Furthermore, it appears that the process of compiling can largely be divided into two parts, one purely machine-dependent and language-independent, and one purely language-dependent and machine-independent.

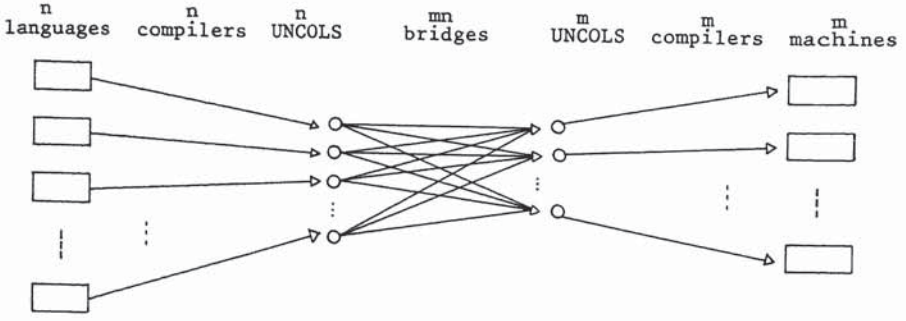
There are several requirements that had to be met by UNCOL:

- It had to have features adequate for the efficient translation of high-level languages.
- It had to have only features that could be efficiently translated to a wide variety of machine languages.
- It had to be standard.

Over the years, it became less and less clear that these goals could be attained. The first two goals conflict with each other. Furthermore, each implementor, seeing the virtue of an UNCOL, quickly designed his own.

The third requirement is political, and not technical. Let us consider the first two. It is clear that there is a set of operations required by most high level programming languages and provided on most machines. This set includes, for example, addition, subtraction, and procedure call. Such a set of operations, supplied with syntax, could form a basis for an UNCOL. Particular languages and machines also require other, highly individual, operations in the UNCOL. Since it is impossible to provide them all in a standard UNCOL (people keep inventing new languages and machines), we must consider such extra operations as creating dialects of UNCOL. Small bridges must be written to translate these dialects to one another.

Our diagram now looks as follows:



Although we now have to write mn bridges, this can be considerably easier than the original problem of writing mn compilers, providing that the UNCOL variants have been chosen with good taste.

WAITE & POOLE [1] have built such a family of UNCOLs, called JANUS. It shows a reasonable chance of being generally accepted in some circles. A JANUS compiler has been built using the STAGE 2 macro processor. The STAGE 2 macro processor can be implemented on nearly any machine in

less than one man-week.

JANUS will not be discussed further in this article.

4. CODE GENERATION FOR ALGOL 68

In order to make the discussion concrete, we shall now consider one method of code generation for ALGOL 68. This code generator operates in a series of stages. Each stage takes the program a little further from ALGOL 68, and a little closer to the final machine language. Nonetheless, a clear division can still be seen between the relatively machine-independent and machine-dependent stages.

Within the code generator, a notation must be used for the intermediate stages of translation of the program. The notation used here is called *directive text*, and will be described later.

(1) *Ingestion*

The program is converted from the Polish or tree form produced by the analysis phase to the directive text used by the code generator.

(2) *Array*

Operations on arrays and on other complicated values requiring descriptors are replaced by simpler operations on the descriptors themselves. For example, a slice, such as a [i,j] will be replaced by code that explicitly contains the arithmetic for the subscripting calculations.

In the output text from this stage, all actions are expressed as operations on so-called *contiguous values*. Each such value can be represented conveniently in a single block of storage, whose size can be determined from the mode of the value. This means that copying such values is a simple operation.

(3) *Simplify* MI

Other operations are broken down to operations that are reasonably likely to appear on real machines. For example, complex arithmetic is replaced by the corresponding real arithmetic. In the code put out by this stage, most operations act on simple scalar values, such as integers, characters, real numbers, and pointers.

Data aggregates also exist, but significant details of the data mapping

algorithms are hidden within special primitive operators. The two means provided for building data aggregates are the *structure* and the *bunch*. A structure is an ordinary, ALGOL-68 structured value. A bunch is a structure whose fields must all be of the same modes, but which are selected by integers instead of tags. Primitive operators are provided for obtaining the distance between adjacent values, and for doing pointer arithmetic.

(4) *Bridge*

The bridge translates the machine-independent, language-dependent text produced by the previous stage into a machine-dependent but language-independent form. Most of the operations will probably be directly available on the machine, but some will require translation. The size and complexity of this phase of the translation can be a fair measure of the appropriateness of the machine for ALGOL 68.

(5) *Instruct*

The stages of the code generator from this point on can also be put to use in a compiler for a different language. The "Instruct" stage assigns storage and machine registers to the identifiers appearing in the intermediate code. This stage must be fully aware of the operations available on the machine and their awkward limitations. It will have to generate instructions to place operands in places where they can be used in other operations. It is in this stage that sophisticated optimization can take place on register allocation.

(6) *Assemble*

The output from the previous stage is no longer directive text, but it is written in an assembler language. This last assembly stage serves to convert the assembler language to a binary object file, suitable for loading by the system loader.

Although it is possible to use the usual assembler provided with the machine for this stage, it is probably more efficient, and perhaps even easier, to write a custom-tailored assembler. Such an assembler would not require its input in character string form, but would accept it in whatever internal form the previous stage finds convenient to produce.

5. DIRECTIVE TEXT

The ALGOL 68 H compiler (version 2) uses a notation called *directive text* inside the object code generator. This notation is similar to that used inside many other compilers. It will be briefly described here, in order to get the flavour of a typical intermediate text. This kind of notation is occasionally used instead of tree or Polish notation. It is more usual to use both this and Polish, but at different stages of the compilation process.

ALGOL 68 H (2) uses directive text in the machine-independent and the machine-dependent parts of the compiler, and also in the bridge. Its design goals are

- to be usable throughout the object code generator as a notation for programs,
- to further the division of the object code generator into clearly specified, comprehensible stages,
- to enable object code optimization but not require it, and
- to simplify the translation task.

The style of the notation is held constant throughout the code generator; it is only the set of available operators and data types that changes. This means that a single general group of routines can serve for the syntactic manipulation of directive text throughout.

Directive text consists of a sequence of *directives*. Most directives represent actions to be carried out at run time. A few directives have other functions, such as to indicate the beginning or end of a routine, or to place a label.

An action directive consists of an operator, perhaps some input operands. We shall write such a directive in the form:

OPERATOR input operands → output operands.

Of course, inside the object code generator, directives are not encoded as characters, but as easily decoded sequences of bits and pointers.
In this paper, however, notation is used that is more accessible for the human reader.

Each input or output operand may be an identifier, or an indirect reference via a pointer. In addition, an input operand may be a denotation of

a value. For examples, we have the following:

```
abc
pluck
indirect  pqr
real 7.539
```

Between directives, it is permitted to place labels. Sometimes, a label will be written as a prefix to a directive.

There are three directives associated with procedure calling. Procedures may have input parameters and output parameters. The CALL directive is used at the point of call; the ENTER directive marks the entry point to the procedure, and EXIT marks the end:

```
CALL  procedure-name;  input-parameters → output-parameters
procedure-name:
ENTER  → parameters
EXIT   parameters →
```

Notice that the roles of input and output parameters are interchanged between the CALL and the ENTER and EXIT. This is because parameter passing is like a pipe: what goes in at one end comes out at the other.

Conditional, unconditional, and indexed jumps are also available.

6. EXAMPLE: COMPLEX ARITHMETIC

Consider the statement:

```
v := z * 6 + w
```

z and w are both complex identifiers, and v is a complex variable. In Polish notation, this would appear as

```
v z 6 convert-to-real  convert-to-complex * w + :=
```

INGEST would convert this to directive text:

```
INT-TO-REAL       6 → t1
REAL-TO-COMPLEX   t1 → t2
COMPLEX - *       z, t2 → t3
COMPLEX - +       t3, w → t4
ASSIGN            v, t4
```

Since this code contains no arrays or other ungainly objects requiring descriptors, it will be unchanged by the ARRAY stage.

The next stage, SIMPLIFY-MI, has the task (among others) of breaking down complex arithmetic. Furthermore, it can take advantage of the behaviour of arithmetic involving special values, such as zero, in order to simplify some operation.

First, by expanding the complex arithmetic, we get the following:

```
INT-TO-REAL   6 → t1
COMBINE       t1, 0.0 → t2
BREAK         z  → realz, imz
BREAK         t2 → realt2, imt2
REAL*         realz, realt2 → t5
REAL*         imz, imt2 → t6
REAL-         t5, t6 → t7
REAL*         realz, imt2 → t8
REAL*         imz, realt2 → t9
REAL+         t8, t9 → t10
COMBINE       t7, t10 → t3
BREAK         t3 → rt3, it3
BREAK         w → rw, iw
REAL+         rt3, rw → t11
REAL+         it3, iw → t12
COMBINE       t11, t12 → t4
ASSIGN        v, t4
```

COMBINE is an operation that constructs a structure, given its fields as operands. In this example, it is used to combine pairs of real values to make complex values. Similarly, BREAK separates the complex value into its two components.

The expansion of an operation such as COMPLEX* is done by substituting parameters into a standard template, creating new local variable names when-

ever necessary. In the above example, mnemonic newly-generated names (e.g.,realz) were used for clarity. In reality, they would likely be obscure. These templates are kept in a table, and fetched out whenever necessary. Letting parameters begin with "p", and local variables with "v", we have the following template for complex multiplication:

```
COMPLEX*  p1, p2 → p3
BREAK     p1 → v1, v2
BREAK     p2 → v3, v4
REAL*     v1, v3 → v5
REAL*     v2, v4 → v6
REAL-     v5, v6 → v7
REAL*     v1, v4 → v8
REAL*     v2, v3 → v9
REAL+     v8, v9 → v10
COMBINE   v9, v10 → p3
```

The SIMPLIFY-MI stage is also capable of performing elementary local optimizations. A COMBINE operator, for example, whose yield is used only in a later BREAK operation, is redundant.

The combination

```
COMBINE  (t1,0) → t2

BREAK     t2 → realt2, imt2
```

will be replaced by the simpler code

```
SET  t1 → realt2
SET  0  → imt2
```

The constants 6 and 0.0 can also be propagated to their eventual uses. Each "t1" can thus be replaced by "6.0", and each "imt2" by "0.0":

```
BREAK    z → realz, imz
REAL*    realz, 6.0 → t5
REAL*    imz, 0.0 → t6
```

```
REAL-     t5, t6 → t7
REAL*     realz, 0.0 → t8
REAL*     imz, 6.0 → t9
REAL      t8, t9 → t10
SET       t7 → rt3
SET       t10 → it3
BREAK     w → rw, iw
REAL+     rt3, rw → t11
REAL+     it3, iw → t12
COMBINE   t11, t12 → t4
ASSIGN    v, t4
```

Multiplication by zero yields zero. This will give us more zeros (t6 and t8) to push forward. Moreover, the operations SET t7 → rt3 and SET t10 → it3 are unnecessary, since we can simply use the old names in later code. We now get:

```
BREAK     z → realz, imz
REAL*     realz, 6.0 → t5
REAL*     imz, 6.0 → t9
BREAK     w → rw, iw
REAL+     t5, rw → t11
REAL+     t9, iw → t12
COMBINE   t11, t12 → t4
ASSIGN    v, t4
```

Notice the great simplification which results from extremely simple operations on directive text. It is easy to take advantage of systematically propagated special cases with this notation, because data relationships between operations are explicit.

The possibility of using macro-expansion to implement complicated operations simplifies the compiler. A macro-expander for the kind of expansions needed is easy to write, and the rest is table-lookup.

It is, however, unfortunate that the directive text contains no traces of the Polish text format so useful for a stack machine. Furthermore, the optimizations discussed here tend to destroy any stack structure that might have been kept.

7. TAME AND WILD DATA

Most serious optimizing compilers use intermediate code similar in principle to this directive text. The important fact about directive text is that it emphasizes data relationships. It is made explicitly clear which directives use the yielded values of which other directives. It is this that furthers the simplification of complicated code sequences. The paths along which directives transmit data to other directives in a program are called the "data flow" of that program.

Features in programming languages can be divided into two classes: those which hinder data flow analysis, and those which do not (some actually help it). Examples of hindrances are array subscripting, procedures with global identifiers, and pointer variables. The point is that each of these features increases the amount of dynamic choice the programmer has as to when to access particular pieces of data.

Optimizing compilers for languages with such features often divide data into two classes:

- "tame" data, on which straightforward data flow analysis is possible, and
- "wild" data, on which it is impossible or inordinately difficult.

Wild data rarely takes part in optimization efforts. For a datum to be tame, it must be clear to the compiler when it is defined and when it is used.

In ALGOL 68, it is possible to regard several classes of values as tame:

- values yielded by subexpressions, awaiting further computations.
- values possessed by identifiers as a result of identity declarations.

Values referred to by variables can be tame if the variables themselves satisfy certain restrictions. Such restrictions essentially imply that pointers to these variables are not created and passed elsewhere in manners which the compiler cannot follow. BOOM [2] describes an algorithm to domesticate values.

8. REGISTER AND STORAGE ALLOCATION

After the text has reached the point where only operations available on the machine occur, it is time for register and storage assignment. We shall here consider only the treatment of tame values: the treatment of

wild values is similar, but tends to require more run-time address calculation and less optimization.

An identifier (appearing in directive text) is called "busy" along any path of control that leads from a definition to a use of that identifier, without passing through any definitions. Identifiers (if any) for wild variables are always considered busy, since it is not known precisely when their values are accessed. We shall sometimes speak of a value as being busy, meaning that its identifier is.

At any point in the program where an identifier is busy, it is necessary that either:

- its value is known to the compiler (i.e., a constant), or
- the compiler knows the addressing calculations necessary to find the value at run-time.

Furthermore, it is required that at joints in the flow of control the one identifier must either be known to have the same constant value along both paths of control, or else the same addressing algorithm.

We shall now speak mainly of the non-stack machines mentioned in chapter 2. These machines have a set of fast registers in which many operations are performed. For our purposes, the important facts about these registers are the following:

- They are directly addressable; i.e., without indexing calculations.
- Some of them can be used in indexing calculations for accessing main storage.
- They are scarce. Usually there are between four and thirty-two of them.
- It is important or mandatory that certain classes of operations be performed in these register, either to improve efficiency or because the operations are not available otherwise.

We may pause to notice that modern stack machines also use registers for addressing calculations. Such registers are, however, not usually available for arithmetic, which is done on the stack. It is possible to use dynamic programming techniques to determine optimal sequences of operations to translate a given piece of directive text; it may be better not to lose the stack structure inherent in the original program in the first place.

The consequence of such a (non-stack) register structure is that values must frequently be loaded into registers or stored into storage to permit operations to be performed on them, and sometimes stored in order to make room for other values.

Most compilers handle this problem in the following way. First, main storage is allocated for the various directive-text identifiers in a procedure in such a way that no two identifiers that are ever busy at the same time are allocated the same storage. Then the registers are considered as transient storage for temporary placement of values. This is handled in much the same way as the paging mechanism of a modern operating system, with two important exceptions:

- Sometimes it is required that a page (i.e., a value) is in backing store and not in cache store.
- The sequence of future page requests is not completely unknown.

Let us consider an algorithm for generating code for such a register machine. As the algorithm considers directives one after another, it maintains records describing which values currently reside in which registers and in which storage cells. Each directive corresponds to some machine instruction. This machine instruction may require that its operands be in particular locations, such as in certain classes of registers, or in store. Specific registers are chosen, code is generated to ensure that operands arrive in these registers, and then the instruction to perform the operation is generated. Afterward, the records are updated to reflect the new contents of registers and storage.

One significant detail of the above algorithm remains unclear: which registers are chosen for the operands. If the register is poorly chosen, it may be necessary that a value that is required later be stored.

The choice of register depends on several facts:

- the class of registers acceptable for the required operand. One would not use, for example, a register incapable of containing floating point operands for a floating point operand.
- the class of registers currently holding no values of significance
- the pattern of future uses of values.

An algorithm has been proved optimal by BELADY [3] under certain assumptions:

```
if   the required value is already in an acceptable place
then leave it there
elif there is an acceptable empty register
```

```
then choose it
else use that acceptable register whose value will not be
     required for the longest time
fi.
```

If an already full register is chosen, its value will of course have to be stored into main storage, unless a duplicate is available.

The assumptions under which this algorithm optimal are:

- all operations except loading and storing must be performed in registers.
- all registers are equally available for all operations.

These conditions are not satisfied on many machines. Nevertheless, many compilers use algorithms like this one for determining which registers to store, sometimes combined with some other heuristics.

Mathematical programming techniques have sometimes also been considered, but some feel that this is like using a steam roller for cracking eggs. In a system as complex already as a optimizing code generator, one must look closely before introducing new complexity.

A very commonly used heuristic, also used in regular paging systems, is to store the value that has not been used for the longest time (past tense instead of future). This is usually done in implementations for which lookahead in the intermediate code is impossible.

There are further complications; programs rarely consist of straight-line code. There are often loops and other conditional branches. Good optimizing compilers use topological flow of control analysis, or even performance statistics from previous runs, to estimate the most frequently used paths of control. These paths are then used for the above analysis, and interface code is generated on the less frequently used paths to make the register and storage assignment match. In the absence of performance statistics from actual runs, it is often assumed that program loops are executed much more frequently than surrounding code. Register allocation for loops is then done from the inside out.

It must be emphasized that, in an optimizing compiler, extremely simple techniques, used at the right places, can have enormous effects. Much human time has been wasted on complicated techniques used at inappropriate places.

9. STORAGE ADDRESSING

For some languages on some machines, it is possible for the compiler to pre-allocate fixed storage for program variables, and then to place the absolute addresses of such storage into machine instructions. (The system linking loader may modify these addresses, but in any case, storage is completely allocated and addresses finally compiled before the first instruction of the program is executed.)

Such solid compile-time calculation of addresses has a number of drawbacks:

- It is difficult to handle recursion, because for recursive calls each call must have its own distinct local variables.
- It is impossible to have arrays whose size is computed at run time.
- It is expensive to have all local variables for all procedures occupying real storage at all times, if they are needed only when the procedure is invoked.

For these reasons, other methods are used. Computers usually come with special addressing hardware to overcome these problems. On such machines, the address of a storage operand is specified by placing two numbers in an instruction, a "base" and a "displacement". The base is the number of an "index register". The contents of the index register is added to (on some machines, subtracted from) the displacement to obtain the storage address.

This addressing hardware means that storage can conveniently be thought of as divided into "segments". Each segment is accessible using a particular base value. Storage can be allocated by the compiler within a segment by determining a displacement. The position of the entire segment in main memory can be determined independently at run time by computing a base value and placing it in an index register.

Each procedure invocation, for example, can have its own local variables in a segment of its own. On procedure entry, storage is obtained from a pool of generally available storage. The address of this storage is placed in an index register, and the object code can thereafter access its private memory by using the displacements computed at compile time. Upon return from the procedure, the storage is returned to the general pool.

When a procedure is recursively invoked, each invocation has its own base value and hence its own local variables.

The same mechanism can be used by a procedure to access its global variables, using a different base value. After all, one procedure's global variables are another procedure's locals.

We can also use this mechanism for list processing. The minor difference is that in list processing, the pointer manipulation is usually done by the programmer. Once again we get objects accessed by using a pointer to a segment and a displacement within it.

10. OPERATING SYSTEM COMPATIBILITY

Most operating systems provide a bewildering variety of features, options, and bugs. It is necessary to choose ways to communicate with it wisely in order to minimize hindrance to the user. Points where a compiler must be compatible with operating system conventions include the following:

- input and output
- subroutine linkage conventions
- program overlays
- format of the binary relocatable object code
- main storage management
- the form of specification of compiler options.

These matters are very easy to get wrong. Often incorrectness does not show up until a user tries an unusual application. P.C. POOLE [4] advises that in any compiler project it is wise to assign one man at the beginning to the task of becoming familiar with the operating system(s).

REFERENCES

[1] COLEMAN, S.S., P.C. POOLE & W.M. WAITE, *The mobile programming system, Janus,* Software-Practice and Experience 4 (1974) 1, 5-23.

[2] BOOM, H.J., *Optimization analysis of programs in languages with pointer variables,* Ph. D. thesis, University of Waterloo, 1974.

[3] BELADY, L.A., *A study of replacement algorithms for a virtual storage computer*, IBM Systems Journal 5 (1966) 2,78.

[4] POOLE, P.C., *Portable and adaptable compilers*, in the Lecture Notes of the Advanced Course in Compiler Construction, Munich, 1974.

MATHEMATICAL CENTRE SYLLABUS 25, 1976, X, 177-195

NAAR EEN FOUTENWERENDE PROGRAMMEERTAAL

L. MEERTENS

0. INLEIDING

In het verleden zijn een aantal methoden aangedragen om tot correcte programma's te komen. De meeste van deze methoden gaan er mank aan dat praktische toepassing grote problemen oproept. Een euvel van al deze methoden blijkt, dat het al te gemakkelijk blijft ertegen te zondigen. Het is niet voldoende methoden te bestuderen waardoor betrouwbaarder programma's kunnen worden verkregen, maar we moeten ook gereedschappen verschaffen waarmee de willige geest zo'n methode aan zijn zwakke vlees kan opleggen. Het gereedschap bij uitstek daarvoor is de programmeertaal.

Om de gedachte te bepalen: de ideale programmeertaal is die waarin alleen correcte programma's kunnen worden uitgedrukt. Iedere poging een fout programma te schrijven wordt afgestraft doordat de tekst als syntactisch ongeldig wordt verworpen.

Eerst zal enige aandacht worden geschonken aan de methode van de inductie-asserties; geschetst zal worden welke problemen verbonden zijn aan het inbouwen in een programmeertaal. Daarna zal een weg worden aangegeven die beter aansluit bij wat nu praktisch mogelijk is, en die toch een behoorlijke vooruitgang belooft ten opzichte van het thans bestaande.

1. DE METHODE VAN DE INDUCTIE-ASSERTIES

Door FLOYD [1] is een methode aangegeven om de correctheid van een programma te bewijzen: plaats rond de statements waaruit het programma is opgebouwd asserties, zoals in $\{A_1\}\ S_1\ \{A_2\}\ S_2\ \{A_3\}$, en laat zien dat geldigheid van A_1 vóór de uitvoering van S_1 de geldigheid van A_2 na uitvoering inhoudt, enz. Het volgt dan, dat $\{A_1\}\ S_1;\ S_2\ \{A_3\}$ ook klopt. Op die wijze kunnen ook voor conditionele en iteratie-constructies axiomatische regels

worden opgesteld (HOARE [2]). (Voor een uitvoeriger overzicht raadplege men DE BAKKER [3].) Het grote voordeel van deze methode boven andere om de correctheid van programma's te bewijzen is daarin gelegen, dat de asserties in de programmatekst thuishoren.

De gedachte uit de inleiding toepassend op deze methode, komen we tot het volgende: een programma is pas dan syntactisch correct, wanneer het op de juiste wijze is voorzien van inductie-asserties. Dit brengt met zich dat de beschrijving van de programmeertaal tot op zekere hoogte de beschrijving van een stellingen-bewijzer, of althans - verificator, moet zijn.

Hieronder volgen enkele schema's die in een volgend voorbeeld een rol spelen. Daarin betekent "X: Y_1, Y_2, ..., Y_n": om na te gaan of X correct is, moet worden nagegaan of Y_1, Y_2, ..., Y_n alle correct zijn. Verder betekent "$A[v:= e]$": de assertie die verkregen wordt door in A voor alle voorkomens van de variabele v de expressie e te substitueren. "$P \to Q$" betekent gewoon: Q volgt logisch uit P. "$\{A_1, A_2, \ldots, A_n\}$" is hetzelfde als $\{A_1 \wedge A_2 \wedge \ldots \ldots \wedge A_n\}$.

$\{P\}\ \{Q\}$: $P \to Q$.
$\{P\}\ v:= e\ \{Q\}$: $\{P\}\ \{Q[v:= e]\}$
$\{P\}$ *if* B *then* S_1 *else* S_2 *fi* $\{Q\}$:
$\{P, B\}\ S_1\ \{Q\}$, $\{P, \neg B\}\ S_2\ \{Q\}$
$\{P\}$ *while* B *do* S *od* $\{Q\}$:
$\{P, B\}\ S\ \{P\}$, $\{P, \neg B\}\ \{Q\}$.

1.1. *Een voorbeeld*

Uit een rij getallen $a[1 : n]$ willen we het kleinste opzoeken. We krijgen dan het volgende programma:

$\{1 \leq n\}$
$i:= 1$
$\{i \leq n,\ i = 1\}$
$x:= a[i]$
$\{i \leq n,\ \forall k \in [1 : i]: x \leq a[k],\ \exists k \in [1 : i]: x = a[k]\}$
while $i \neq n$
do $i:= i + 1$
$\{i \leq n,\ \forall k \in [1 : i - 1]: x \leq a[k],\ \exists k \in [1 : i - 1]: x = a[k]\}$
if $a[i] < x$

```
        then x:= a[i]
        else
        fi
    od
    {∀k ∈ [1 : n]: x ≤ a[k], ∃k ∈ [1 : n]: x = a[k]}.
```

Merk op dat zelfs voor dit uiterst eenvoudige voorbeeld het vermogen van de stellingen-bewijzer groot genoeg moet zijn om, naast trivialiteiten als $(i + 1) - 1 = i$ of $i \leq n \wedge i \neq n \rightarrow i + 1 \leq n$, ook te kunnen bewijzen $i = 1 \rightarrow (\forall k \in [1 : i]: a[i] \leq a[k])$, en zelfs $(\forall k \in [1 : i - 1]: x \leq a[k]) \wedge a[i] < x \rightarrow (\forall k \in [1 : i]: a[i] \leq a[k])$.

Dit probleem is wellicht nog te ondervangen doordat de programmeur in zijn programma aanwijzingen zou kunnen geven, b.v. in de vorm van tussenasserties. Triviaal is dit echter niet, en het vereist tevens dat de programmeur inzicht heeft in de strategie van de stellingen-bewijzer. Er kleven bovendien nog wel grotere bezwaren aan gedwongen toepassing van deze methode.

1.2. *Problemen verbonden aan de inductie-assertiemethode*

Een fundamentele begrenzing aan iedere mechanische verificatie van bewijzen wordt gegeven door de onvolledigheidsstelling van GÖDEL: gegeven een voldoende machtig formeel systeem om stellingen in te formuleren en te bewijzen, laat zich altijd wel een stelling formuleren die juist is, maar in dat systeem onbewijsbaar. Dit houdt in dat er dan ook correcte algoritmen moeten zijn, waarvan de correctheid ook bewijsbaar is, maar niet met de door de programmeertaal verschafte middelen. Met andere woorden: zulke algoritmen kunnen in de programmeertaal niet worden uitgedrukt.

Dit is een ernstig bezwaar, hoewel het niet duidelijk is hoe zwaar het in de praktijk zou wegen. Maar daar doemen, bij de huidige state of the art van de inductie-assertiemethode, al veel eerder problemen op. De voorbeelden die door auteurs op dit gebied gegeven worden beperken zich bijna geheel tot trivialiteiten als het berekenen van de grootste gemene deler of het rangschikken van een rij getallen. Kortom, het manipuleren met gehele of reële getallen in een algoritme die zich in de vorm van een stroomdiagram laat gieten. Voor het geval van willekeurige recursieve parameterloze procedures is een aanzet tot het kunnen gebruiken van inductie-asserties gegeven in DE BAKKER & MEERTENS [4], een aanzet overigens die niet onaanzienlijke

complicaties met zich brengt. En als het gaat om procedures met parameters (die zelf weer procedures kunnen zijn), is het vooralsnog niet duidelijk of deze met een verder uitgebreide inductie-assertiemethode aangepakt kunnen worden.

Vervolgens worden hieronder twee algoritmen gegeven: een voor het berekenen van het n-de getal uit de rij van FIBONACCI:

```
old:= 1; new:= 0;
for i to n
do int sum = old + new; old:= new; new:= sum od,
```

en een voor het vergelijken van twee woorden om te zien welk alfabetisch eerder komt:

```
op < = (string a, b) bool
if b = "" then false
elif a = "" then true
elif a[1] = b[1] then a[2 :] < b[2 :]
else a[1] < b[1]
fi
```

De lezer die probeert deze algoritmen van inductie-asserties te voorzien zal zich onmiddelijk een ander probleem realiseren: hoe drukken we hier überhaupt het correctheidscriterium in een assertie uit?

Tenslotte kan de vraag gesteld worden of van een programmeur in het algemeen verwacht mag worden dat hij bij een algoritme waarvan hij de correctheid inziet, ook een formeel bewijs levert. Een voorbeeld (ontleend aan een voordracht van E.W. DIJKSTRA): Gevraagd in een boom (een ongerichte graaf van één component zonder lussen) de grootste afstand tussen twee knopen te vinden, wanneer bij iedere tak de lengte, een niet-negatief getal, gegeven is. Stel je nu een fysische realisatie voor, waarbij kralen verbonden zijn met touwtjes van de goede lengte. Til de boom nu bij een willekeurige kraal op en laat de rest naar beneden hangen. Het is niet moeilijk in te zien dat de laagste kraal een uiteinde van het langste pad moet zijn. (Stel dat een andere kraal nog lager hangt. Beschouw (a) het pad omhoog van die kraal naar de vastgehouden kraal en (b) het langste pad plus het pad omhoog vanaf de hoogste kraal daarvan naar de vastgehouden kraal. Via de laagste gemeenschappelijke kraal van (a) en (b) is direct een langer pad te

construeren dan het veronderstelde langste pad.) Til de boom aan die kraal op, pak nu de laagste kraal beet, en beide uiteinden van het langste pad zijn gevonden.

Uit deze beschouwing volgt een algoritme: neem een willekeurige knoop x, zoek een knoop a zodat $d(a, x)$ maximaal en vervolgens een knoop b zodat $d(a, b)$ maximaal. Het is niet moeilijk hieruit een programma te maken dat correct is. De lezer wordt verzocht bij zo'n programma eens een formeel correctheidsbewijs te trachten te leveren.

De conclusie kan voorlopig niet anders luiden dan dat de inductie-assertiemethode nog onvoldoende ontwikkeld is om voor het gestelde doel bruikbaar te zijn. Dit is geen kritiek op de methode op zich; het is eerder een aansporing deze waardevolle methode verder te ontwikkelen.

2. PROGRAMMA EN BETEKENIS

De sleutel tot de volgende gedachten is dat een algoritme niet alleen een rij handelingen voor een automaat beschrijft, maar dat die handelingen ook een bedoeling, een *betekenis* hebben. Die betekenis bestaat in de geest van de programmeur. Als op een of andere manier een aanduiding van die betekenis in de programmatekst te vinden zou zijn, dan is het in principe mogelijk sommige programmeerfouten aan de hand daarvan op te sporen, louter door een statische beschouwing van de tekst.

Wat besturingsstructuren betreft, is in hogere programmeertalen in dit opzicht al veel bereikt. Ieder zal het er over eens zijn dat de betekenis van

```
        i:= 1;
test:   if i > n then go to finish fi
        a[i]:= b (i)
        i +:= 1;
        go to test;
finish:
```

veel duidelijker wordt weergegeven

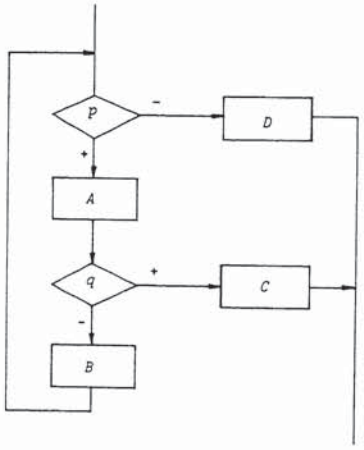
```
for i to n
do a[i]:= b (i)  od.
```

Een voorbeeld van een op dit punt zeer vergaande programmeertaal is ALEPH (BOSCH, GRUNE & MEERTENS [5], GRUNE [6]). In de praktijk blijken inderdaad veel programma's die een programmeerfout (vaak een denkfout) bevatten, bij syntactische analyse door de mand te vallen.

Gezien het op dit punt al bereikte zal dit aspect van programmeertalen hier niet verder behandeld worden. Eén opmerking nog: een aantal recente voorstellen voor nieuwe notaties op het punt van besturingsstructuren (b.v. MARTIN [7], FRIEDMAN & SHAPIRO [8]; zie ook KNUTH [9]) geven wel de indruk dat te mechanistisch gezocht wordt naar het vermijden van sprongen. Het gaat hierbij om gevallen als

```
        for e ∈ S
        do if tag of e = name then go to found fi od;
        S ∪:= namecell (name, 1);
        go to finish
found:  freq of e +:= 1;
finish:.
```

De betekenis hiervan laat zich niet weergeven door een willekeurige constructie die voor een stroomdiagram als hieronder



een notatie geeft zonder *go to*, zoals *R*: *while p do A*; *if q then C leave R fi*; *B od D elihw*, maar wel b.v. door

```
case ∃e ∈ S: tag of e = name
in freq of e +:= 1
out S ∪:= namecell (name, 1)
esac
```

2.1. *Mode en betekenis*

Vergeleken bij besturingsstructuren, zijn gegevensstructuren in hogere programmeertalen in het algemeen maar mager bedeeld. Daarom zal nu alle verdere aandacht op dit punt gericht worden.

Gegevensstructuren dienen om in een algorithmisch proces waarden voor te stellen. Binnen een computer worden deze waarden gewoonlijk door een rij bits weergegeven. De betekenis van zo'n bitrij kan in het algemeen niet louter uit die rij zelf worden afgeleid. Zo zou het b.v. kunnen gebeuren dat een integer en een pointer door precies dezelfde rij nullen en enen blijken te worden voorgesteld, ook al is hun betekenis voor de programmeur totaal verschillend. Van de nieuwe ideeën die de komst van hogere programmeertalen hebben gekenmerkt, is een der belangrijkste het idee van een aan waarden verbonden "mode" (of "type"). Dit kwam neer op de erkenning van het feit dat bij de vele door programmeurs aan bitrijen toegekende betekenissen, enkele algemeen gangbaar waren. Zo heb je reële getallen, met de mode "real", en waarheidswaarden, met de mode "boolean", en de syntaxis voorkomt dat waarden van deze twee modes per ongeluk met elkaar worden verward. Bovendien wordt de representatie ontoegankelijk voor de programmeur, wat op zich heel nuttig is om van de toevallig gebruikte representatie te kunnen abstraheren - met alle voordelen die een dergelijke abstractie met zich brengt.

Het begrip mode heeft zijn hoogtepunt (tot nu toe) bereikt in ALGOL 68 [10]. Daarom zal ALGOL 68 hier ook als uitvalsbasis gebruikt worden om te laten zien hoe dit idee nog verder kan worden doorgevoerd teneinde de betrouwbaarheid door uitgebreidere statische controle te verhogen en zelfs om nog hogere abstractieniveaus te bereiken. Omdat het hier alleen om het schetsen van ideeën gaat, zullen af en toe zonder voorafgaande waarschuwing nieuwe syntactische constructies ad hoc worden ingevoerd, waarvan de semantiek alleen gesuggereerd wordt door de context. Het gaat dus niet om voorgestelde uitbreidingen van ALGOL 68, maar om de aanduiding van een benadering die in nieuw te ontwerpen talen zijn weg zou kunnen vinden.

2.2. *De grotere betrouwbaarheid van ALGOL 68 - een voorbeeld*

De mode van een waarde in ALGOL 68 is een volwassen versie van het type uit ALGOL 60. Het Revised Report zegt hierover: "*Each value has an attribute, termed its ""mode"", which defines how that value relates to other values and which actions may be applied to it.*" Hoewel ook het ALGOL-60-type zoiets suggereert, verschaft het niet voldoende informatie om statisch na te gaan of een gegeven bewerking zinvol zal zijn. Neem als voorbeeld het volgende ALGOL-60-programma:

```
l: begin integer i;
        procedure var (op); procedure op; op (i)
        procedure spot (op); procedure op; op (l);
        procedure zero (v); integer v; v:= 0;
        procedure jump (s); label s; go to s;
        procedure do (op, arg); procedure op, arg; arg (op);
        do (zero, var);
        do (jump, spot)
   end.
```

De uitvoering van de twee procedure statements die *do* aanroepen is volstrekt zinvol. Maar met hetzelfde gemak kunnen we schrijven: *do* (*jump*, *var*), *do* (*zero*, *spot*) of zelfs *do* (*do*, *do*), waarvan de uitvoering respectievelijk zou uitmonden in de zinloze bewerkingen *go to i*, *l*:= *0* en *do* (*do*). Merk op dat het feit dat dit in ALGOL 60 kan, niet het gevolg is van de mogelijkheid de parameter-specificatie weg te laten, maar van de uniforme specificatie *procedure*, die geen informatie verschaft over het type van parameters die acceptabel zijn (LANGMAACK [11]).

In ALGOL 68 zouden *zero* en *jump* verschillende modes hebben, aangezien ze parameters van een verschillende mode accepteren. Evenzo verschillen de parameters *op* van *var* en *spot* van mode, omdat ze voor hun eigen parameters verschillende modes accepteren, en dus hebben ook *var* en *spot* een verschillende mode. Bijgevolg moet de parameter *arg* van *do* "gespecificeerd" worden als twee verschillende modes accepterend, d.w.z., de "union" van die modes. Hetzelfde geldt natuurlijk voor de parameter *op*. Voordat *arg* mag worden aangeroepen met *op* als parameter, moeten de verschillende gevallen worden onderscheiden om na te gaan of de feitelijke modes bij elkaar passen. Een vergelijkbaar ALGOL-68-programma zou er daarom als volgt uitzien:

```
l: begin int i;
        proc var = (proc (ref int) void op) void: op (i);
        proc spot = (proc (proc void) void op) void: op (go to l);
        proc zero = (ref int v) void: v:= 0;
        proc jump = (proc void s) void: s;
        proc do = (union (proc (ref int) void,
                          proc (proc void) void) op,
                   union (proc (proc (ref int) void) void,
                          proc (proc (proc void) void) void) arg)
        void:
        case arg in
             (proc (proc (ref int) void) void pprivv):
                  case op in
                       (proc (ref int) void priv): pprivv (priv)
                  out error 1
                  esac,
             (proc (proc (proc void) void) void pppvvv):
                  case op in
                       (proc (proc void) void ppvv): pppvvv (ppvv)
                  out error 2
                  esac
        esac;
        do (zero, var);
        do (jump, spot)
   end.
```

De definitie van *do* ziet er, vergeleken bij de ALGOL-60-versie, nogal ingewikkeld uit. Wat er gebeurd is, is dat de dynamische type-controle, die in een "waterdichte" ALGOL-60-implementatie impliciet aanwezig is, hier expliciet is gemaakt. Aanroepen zoals *do* (*jump*, *var*) en *do* (*zero*, *spot*) geven bij elaboratie aanleiding tot de elaboratie van resp. *error 1* en *error 2* (hier niet verder gedefinieerd). Een would-be aanroep als *do* (*do*, *do*) is zelfs syntactisch ongeldig. Zodoende is het onmogelijk een (syntactisch geldige) aanroep te construeren met een parameter waarvan de waarde mogelijkerwijs niet acceptabel is, behalve in het geval waar een expliciete tekst wordt uitgevoerd eer die waarde echt gebruikt wordt. (Wellicht is ook illustratief dat de auteur bij een eerdere versie van dit verhaal *op* (*arg*) op-

schreef voor het procedure-lichaam van *do* zonder daar erg in te hebben en de fout pas bemerkte bij het omzetten van de ALGOL-60-versie naar ALGOL 68.) Bovendien zijn er geen impliciete tests meer nodig. De ALGOL-68-benadering is dus superieur aan die van ALGOL 60, zowel uit het oogpunt van veiligheid als uit dat van efficientie.

2.3. *De mode als statische samenvatting van zinvolle bewerkingen*

De mode van een waarde kan beschouwd worden als een statische samenvatting van de zinvolle bewerkingen op die waarde. Bijvoorbeeld, als een lijst bestaat uit twee velden, het veld *head*, dat een string is, en het veld *tail*, dat een pointer naar een lijst is, dan wordt deze recursieve definitie weerspiegeld in de volgende mode-declaration:

mode list = *struct* (*string head*, *ref list tail*).

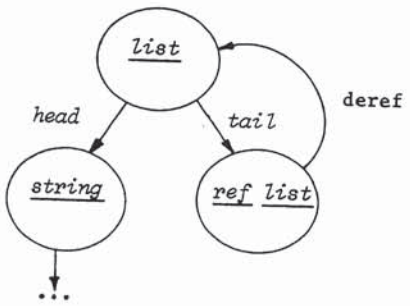
Deze declaratie kan dan ook worden opgevat als mededeling dat zinvolle bewerkingen op een lijst zijn:

- het selecteren van het veld *head*, gevolgd door zinvolle bewerkingen op een string;
- het selecteren van het veld *tail*, gevolgd door zinvolle bewerkingen op een pointer naar een lijst.

Als we als zinvolle bewerkingen op een pointer naar een lijst nemen:

- het volgen van de pointer (dereferencing), gevolgd door zinvolle bewerkingen op een lijst,

dan kan deze verbale uiteenzettingen schematisch door het volgende diagram worden afgebeeld:

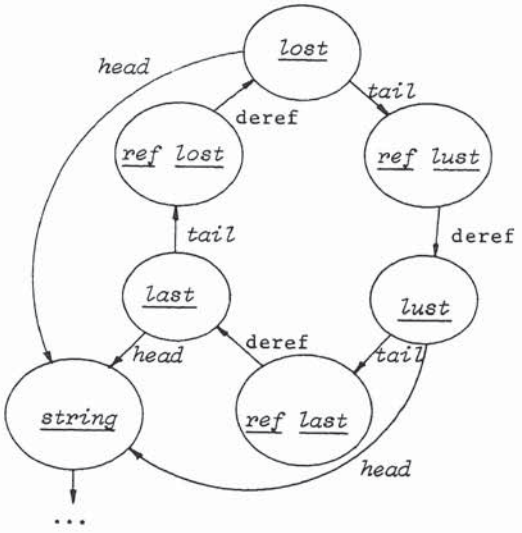


Dit diagram correspondeert met een eindige-toestandsautomaat die, met *list* als begintoestand, een (reguliere) taal accepteert waarvan iedere zin correspondeert met een zinvolle bewerking op een lijst.

Voor de ingewikkelder mode-declaration

```
mode lost = struct (string head, ref lust tail),
     lust = struct (string head, ref last tail),
     last = struct (string head, ref lost tail)
```

krijgen we het volgende diagram:



De corresponderende eindige-toestandsautomaat accepteert vanuit b.v. de begintoestand *lost* dezelfde taal als de automaat voor *list*: de zinvolle bewerkingen op *list*- en *lost*-waarden vallen samen. Inderdaad specificeren *list* en *lost* (of *lust* of *last*) dezelfde mode, wat klopt met de boven aangehaalde zienswijze op modes in ALGOL 68. Op het verband tussen de equivalentie van modes en van eindige-toestandsautomaten is al gewezen door KRÁL [12].

2.4. *Het vullen van de resterende gaten*

Tot nu toe is min of meer als vanzelfsprekend aangenomen dat de door een syntactisch geldige uitdrukking voorgeschreven bewerkingen in ALGOL 68 ook altijd (dynamisch) zinvol zijn. Maar dat hoeft niet het geval te zijn.

We zullen twee gevallen beschouwen waarin de verdere elaboratie niet gedefineerd is:

- Een naam die moet worden gedereferenced, of waaraan een waarde moet worden toegekend, blijkt nil te zijn (zoals in *ref real* (*nil*):= *3.14*).
- De subscript in een slice ligt buiten het subscriptbereik (zoals in [*1* : *3*] *int a*; *a*[*4*]).

Het is een wenselijke eigenschap van programmeertalen dat zulke gevallen zich niet kunnen voordoen, om meer dan een reden: Als we erin slagen al zulke gevallen uit te sluiten, dan verdwijnt eenvoudigweg de mogelijkheid (syntactisch geldige) programma's te schrijven waarvan de elaboratie pas tijdens de uitvoeringsfase ongedefinieerd blijkt. Als de programmeur niet statisch kan aantonen dat een bepaalde bewerking zinvol zal zijn, komt de last daarop te testen op zijn schouders te rusten. Dat zal in het algemeen het aantal tekens dat hij moet opschrijven vergroten (vgl. de definitie van *do* in ALGOL 68). Daardoor wordt de programmeur aangemoedigd zijn probleem opnieuw te overwegen en zijn implementatie van de algoritme zo te herformuleren dat het duidelijk wordt dat ongedefinieerde bewerkingen niet kunnen optreden. Het programma zal dan (hopelijk) begrijpelijker worden; bovendien is hij verplicht ieder geval te overwegen waar de zaak spaak zou kunnen lopen. Een extra, niet onbeduidend voordeel is het verdwijnen van kostbare impliciete tests in een "waterdichte" implementatie.

Ook al worden hieronder slechts twee bijzondere gevallen besproken, de meeste resterende gaten kunnen in vrijwel dezelfde geest worden aangepakt. Zo is bijvoorbeeld een syntactische controle op het gebruik van ongeïnitialiseerde variabelen aanwezig in ALEPH [5, 6].

Zelfs het probleem van onjuiste invoergegevens kan misschien worden opgelost door, als onderdeel van het programma, een syntactische beschrijving te geven aan de hand waarvan de input tevoren wordt gecontroleerd.

2.4.1. *Het probleem van namen die nil zijn*

Binnen het kader van ALGOL 68 is een volledige en afdoende oplossing mogelijk voor het geval van het dereferencen van of toekennen aan een naam die nil is. Het volstaat namen die nil zijn uit te sluiten (b.v. door geen representatie voor het nil-symbol ter beschikking te stellen). Als een programmeur uitdrukkingen wil gebruiken waarvan de waarde mogelijk een naam is, maar mogelijk ook niet naar een waarde verwijst (d.w.z., nil is), dan

kan hij een constructie gebruiken als

```
mode list = struct (string head, union (ref list, void) tail).
```

Een lijst die nil is wordt dan voorgesteld door de waarde empty, de enige waarde van de mode void. Een stukje programma als

```
ref list a:= z;
while tail of a :≠: ref list (nil)
do a:= tail of a od
```

zou dan herschreven moeten worden als

```
ref list a:= z
while case tail of a in
          (ref list b): (a:= b; true
      out false
      esac
do skip od.
```

Merk op dat het aantal expliciet verrichte tests het zelfde blijft, maar dat het totale aantal tests afneemt, aangezien in een niet hyper-intelligente implementatie de assignation *a:= tail of a* een test op het niet nil zijn van *ref list (a)* tot gevolg heeft (vanwege de selectie) en misschien zelfs nog twee tests op *a* zelf (vanwege de toekenning en het impliciete dereferencen van het tweede voorkomen).

2.4.2. *Het probleem van het subscriptbereik*

Met de behandeling van subscripts zijn we niet zo gemakkelijk klaar. En juist dat geval is typerend zowel voor het efficientieverlies dat hogere programmeertalen veroorzaken ten opzichte van met de hand coderen - in de meeste gevallen weet de programmeur heel goed dat de subscript binnen het bereik ligt, maar hij is niet in staat deze wetenschap te benutten - als voor de zelfs door hogere programmeertalen verschafte gelegenheid om programma's te schrijven waarvan de zinledigheid pas tijdens de verwerking aan het licht treedt - lijkt niet gewaagd te gissen dat "subscript buiten het bereik" verantwoordelijk is voor pakweg de helft van de fouten die in de

verwerkingsfase optreden -. Een eerste stap naar een oplossing is in PASCAL gezet door "ranges" als type in te voeren. Het hoofdgebrek van deze aanpak is het feit dat ranges al in de compilatiefase worden bepaald, zodat we weer in het FORTRAN-tijdperk belanden. Een gedetailleerder kritiek is te vinden bij HABERMANN [13]. De voor de hand liggende volgende stap vereist modes die dynamisch kunnen ontstaan, maar in een bepaald gedeelte van het programma bevroren zijn.

Stel dat we kunnen schrijven

```
mode r3 = 1 : 3;
[r3] real a;
for i ∈ r3 do a[i]:= 0 od.
```

Op deze wijze gehanteerd is dit alleen een afkorting, maar we kunnen ook verder gaan:

```
mode r = 1 : n; [r] real p; read (p);
for i ∈ r
do r imin:= i; real min:= p[imin];
   mode s = succ i : upb r;
   for j ∈ s
   do if p[j] < min then min:= p[imin:= j] fi od;
   p[imin]:= p[i]; p[i]:= min
od.
```

Dit vereist wel dat de syntaxis op een of andere manier het bereik dat wordt opgeleverd door *succ i : upb r* herkent als deelbereik van *r*, aangenomen dat *i* van de mode *r* is. Dit lijkt wellicht wat ad hoc, maar het is in principe beslist niet onmogelijk. In feite kan alleen bepaald worden wat wel en wat niet door de syntaxis herkend kan worden door een aantal representatieve, uiteenlopende algoritmen te bestuderen. Waarschijnlijk is het teveel gevraagd dat herkend wordt dat $(i + j) \div 2$ in hetzelfde bereik ligt als i en j. Voor zulke gevallen bestaat altijd nog de uitweg via

```
case (i + j) ÷ 2 in (r mid): p[mid] out skip esac.
```

Dat is aanvaardbaar zolang het aantal gevallen waar dergelijke omslachtige uitdrukkingen niet te omzeilen zijn, voldoende klein is. Een vluchtige blik

in enkele ALGOL-60-programma's suggereert dat tenminste zo'n 80% van de subscripts rechtstreeks kan worden afgehandeld, zelfs nog zonder eventuele herformuleringen te beschouwen. Als voor de resterende 20% wordt onderzocht hoe statisch kan worden aangetoond dat de subscript binnen het bereik ligt, is het aannemelijk dat daaruit enkele principes naar voren komen die een substantieel gedeelte van deze gevallen bestrijken en daarom de moeite van inlijving in de syntaxis waard zijn.

2.5. *Modes die door een karakteristiek predikaat zijn gedefinieerd*

BAUER & GNATZ [14] hebben de mogelijkheid geopperd van definities als

mode posreal = (*real x*) *bool*: $x > 0$.

Dit lijkt erg op *mode r* = $1 : n$, wat uitgedrukt zou kunnen worden als

mode r = (*int i*) *bool*: $1 \leq i \wedge i \leq n$.

In de context van de declaratie *real x*, *posreal y,z* zijn uiteraard toekenningen als $x := y$ en $y := z$ toegelaten, maar $y := x$ niet, aangezien niet gegarandeerd is dat x positief is. Dan moet je schrijven:

case x in (*posreal p*): $y := p$ *out error esac*.

De reden om deze constructie hier te vermelden, is dat deze een gereedschap verschaft om de betrouwbaarheid te verhogen in bepaalde gevallen waar andere eenvoudige syntactische middelen geen uitkomst bieden. Stel dat er een functie gedefinieerd is die alleen convergeert op een positief argument. Een verstandige programmeur kan bij de routine-ingang een controle inlassen. Maar als deze een niet-positieve waarde ontdekt beginnen zijn moeilijkheden pas. Waar ontstond de gewraakte waarde? Hoe werd die doorgegeven? Door de mode *posreal* voor te schrijven op alle plaatsen waarvan hij weet dat de betrokken waarden positief horen te zijn, kan de programmeur de fout bij zijn oorsprong vangen.

2.6. *Het abstraheren van de gegevensrepresentatie d.m.v. primitieve modes*

Ook al is een bepaalde bewerking volstrekt zinvol waar het de semantiek

van een programmeertaal betreft, dan nog kan deze, bezien vanuit een menselijk standpunt, pure onzin zijn. Onderwijzers plegen erop te staan dat je appels en peren niet bij elkaar kunt optellen, maar hoeveel programmeerfouten zouden er niet gemaakt zijn waarbij geboortedata bij salarissen werden opgeteld, of relatieve fouten bij absolute grootheden? De auteur heeft zelf een pijnlijke herinnering aan een programma dat over muziek ging en waarin twee verschillende systemen gehanteerd werden om tonen door gehele getallen voor te stellen. Enkele zorgvuldige maatregelen ten spijt bleven waarden in het ene systeem opduiken op plaatsen waar waarden in het andere systeem werden verwacht.

Het is merkwaardig dat de betrouwbaarheid, verschaft door het abstraheren van de gegevensrepresentatie, wel op een laag niveau geboden wordt, maar de gebruiker op hogere abstractieniveaus onthouden wordt. Een uitzondering is te vinden in de "class" van SIMULA 67, met behulp waarvan we kunnen definiëren:

```
class stack;
begin integer ptr; integer array cell [1 : large];
      procedure push (i); integer i;
      begin ptr:= ptr + 1; cell[ptr]:= i end;
      integer procedure pop;
      begin pop:= cell[ptr]; ptr:= ptr - 1 end;
      ptr:= 0
end.
```

De identifiers *ptr* en *cell* kunnen buiten deze definitie onzichtbaar worden gemaakt, zodat na de declaratie *ref* (*stack*) *s* de enige toelaatbare operaties op *s* zijn: *s.push* en *s.pop*.

De voordelen die het gebruik van classes biedt om het correctheidsbewijs te vergemakkelijken door het te factoriseren, zijn door HOARE [15] uiteengezet. Een sterk hierop gelijkende constructie is voorgesteld door LISKOV & ZILLES [16], die er de nadruk op leggen dat een abstract gegevenstype gekenmerkt moet worden door de ter beschikking staande bewerkingen zoals door de gebruiker gedefinieerd. In de kiem is deze gedachte al te vinden bij WILKES [17].

Zulke constructies zijn wel mooi, maar nog onvoldoende om in de behoeften te voorzien, aangezien zo alleen monadische bewerkingen kunnen worden voorgeschreven. Een andere mogelijkheid wordt wellicht geboden door een uit-

breiding van "definitie-modulen", voorgesteld door SCHUMAN [18]. Daarin kunnen identifiers en andere indicators ontoegankelijk gemaakt worden in de omgeving waar het definitie-moduul wordt aangeroepen door aan hun definitie *implicit* te laten voorafgaan. Als evenzo de mogelijkheid wordt ingevoerd voor een mode-definition *primitive* te zetten, wat dan aanduidt dat in de omgeving van de aanroep de gedefinieerde mode als primitief wordt behandeld, dan wordt het gewenste effect bereikt. We kunnen b.v. definiëren:

```
definition prelude
begin mode primitive int = [0 : 15] bool,
          primitive char = [0 : 15] bool,
      op + = (int a, b) int: ...,
      ...,
      op abs = (char a) int: a,
      op repr = (int a) char: a
end,
```

en dan de definitie aanroepen in

```
begin env prelude;
      int i, char c;
      ...
end,
```

waar in de laatste omgeving *i*:= *abs c* en *c*:= *repr i* geldige assignations zijn, terwijl *i*:= *c* en *c*:= *i* syntactisch ongeldig zijn. De mode van *abs* is *proc* ([] *bool*) [] *bool* in de omgeving van het definitie-moduul, maar *proc* (*char*) *int* in de omgeving van de aanroep.

LITERATUUR

[1] FLOYD, R.W., *Assigning meanings to programs*, Proc. of a Symposium in Applied Mathematics, Vol 19 - Mathematical Aspects of Computer Science (J.T. Schwartz, ed.), 19-32 (1967).

[2] HOARE. C.A.R., *An axiomatic basis for computer programming*, Comm. ACM 12 (1969) 576-580.

[3] BAKKER, J.W. DE, *Inleiding bewijsmethoden*, Syllabus van het Colloquium Programmacorrectheid 1, Mathematisch Centrum, 1973.

[4] BAKKER, J.W. DE & L.G.L.T. MEERTENS, *On the completeness of the inductive assertion method*, Mathematisch Centrum, IW 12/73, 1973.

[5] BOSCH, R., D. GRUNE & L. MEERTENS, *ALEPH, A Language Encouraging Program Hierarchy*, The International Computing Symposium 1973, Davos, North-Holland 93-99 (1974) (Mathematisch Centrum, IW 9/73, 1973).

[6] GRUNE, D., *ALEPH, een grammaticale aanpak van programmacorrectheid*, Syllabus van het Colloquium Programmacorrectheid 4, Mathematisch Centrum, 1973.

[7] MARTIN, J.J., *The 'natural' set of basic control structures*, SIGPLAN Notices 8 (1973)12, 5-14.

[8] FRIEDMAN, D.P. & S.C. SHAPIRO, *A case for while-until*, SIGPLAN Notices 9 (1974)7, 7-14.

[9] KNUTH, D.E., *Structured programming with* go to *statements*, Computing Surveys 6 (1974) 261-301.

[10] WIJNGAARDEN, A. VAN, e.a. (Eds.), *Revised Report on the Algorithmic Language ALGOL 68*, Supplement to ALGOL BULLETIN 36, Technical Report TR 74-3, Dept. of Computer Science, Univ. of Alberta, Edmonton, 1974.

[11] LANGMAACK, H., *On correct procedure parameter transmission in higher programming languages*, Acta Informatica 2 (1973) 110-142.

[12] KRÁL, J., *The equivalence of modes and the equivalence of finite automata*, ALGOL Bulletin 35.4.5 (1973) 34-35.

[13] HABERMANN, A.N., *Critical comments on the programming language Pascal*, Acta Informatica 3 (1973) 47-57.

[14] BAUER, F.L. & R. GNATZ, *Mengen in algorithmischen Sprachen oder: Arten und Prädikate*, Abt. Mathematik der TU München, Bericht Nr. 7202, 1973 (?).

[15] HOARE, C.A.R., *Proof of correctness of data representations*, Acta Informatica 1 (1972) 271-281.

[16] LISKOV, B. & S. ZILLES, *Programming with abstract data types*, Proceedings of a Symposium on Very High Level Languages, SIGPLAN Notices 9 (1974)4, 50-59.

[17] WILKES, M.V., *The outer and inner syntax of a programming language*, Comp. Journal 11 (1968) 260-263.

[18] SCHUMAN, S.A., *Toward modular programming in high-level languages*, ALGOL Bulletin 37.4.1 (1974) 30-53.

MATHEMATICAL CENTRE SYLLABUS, 25, 1976, XI, 197-218

PROGRAMMING LANGUAGES AND HARDWARE

A. TANENBAUM

0. INTRODUCTION

The relationship between programming languages and hardware is not unlike the relationship between the chicken and the egg. Neither would be viable without the other, and in their respective evolutionary processes, changes in one tend to lead to changes in the other. It is the purpose of this article to investigate some aspects of the interaction of programming languages and hardware upon one another. We will begin by considering how hardware has affected software, and then we will examine how software has affected hardware.

1. ANCIENT HISTORY

In a number of ways programming languages resemble the notation of classical mathematics. Constants, variables, expressions and functions are only a few of the concepts common to both. However, the nature of computer hardware has also forced programming languages to differ from classical mathematics in some very fundamental ways. These influences of hardware on programming languages are so basic that they are now always taken for granted. Nevertheless, it is worth briefly pointing some of them out.

1.1. *Computers execute instructions*

Result: Nearly all programming languages are algorithmic (rather than descriptive) in nature. They consist of instructions to be carried out, whereas classical mathematics consists of describing static relations between objects. Pure mathematicians have been known to begin foaming at the mouth upon seeing the (Fortran or PL/I) statement: N = N+1, which obviously does

not describe a static relationship.

1.2. *Each instruction takes a finite time*

Result: Nearly all programming languages are concerned about the order in which steps are carried out. In classical mathematics N := 0; N := 1 is not allowed. There is a tacit assumption that assignments are eternal.

1.3. *No computer is infinite*

Result: Programming languages must, of necessity deal with finite sets. When programmers forget this limitation they invariably get into trouble, e.g., overflow, roundoff error, storage capacity exceeded. In classical mathematics one can consider the set of all real numbers whose first decimal digit is 1. No programming language allows that.

2. SOME INFLUENCES OF HARDWARE ON THE DESIGN OF PROGRAMMING LANGUAGES

2.1. *Machine code*

Back in the (good?) old days, people programmed directly in machine code. Machine code was the only programming language available. In this sense the hardware completely determined the programming language. There was an exact correspondence between every feature of the programming language and the hardware. As an example of a program in machine code, here is a complete program for the IBM 1401 to read and list a card deck:

```
1M0012012001
```

2.2. *Assembly language*

The next step forward was the development of assembly languages. This was an attempt to write machine code in a somewhat less obnoxious form. The assembler took over the responsibility for looking up the numerical value of opcodes and labels, etc. Since each assembly language was specific to one particular machine, the hardware had an overwhelming influence on the assembly language. If the machine instructions had one opcode and two addresses, the syntax of the assembly language called for exactly one opcode and two addresses per statement. If the machine instructions had an

opcode and three addresses, the assembly language statements did too.

2.3. *Machine oriented languages*

These languages are intended for systems programming. In such applications the programmer needs complete access to all the machine features, although he usually does not need complete control over the machine instructions executed, at least not all the time. PL/360[1] is the spiritual progenitor of this type of language.

Typical machine features to which access is needed are general and special purpose registers, protected addresses and memories, interrupt and trap vectors, memory mapping facilities, i/o channels and controllers, and in some cases certain machine instructions. To be of any use, the MOL's must allow the programmer to examine and modify the appropriate hardware registers, etc. In turn, this requires that the MOL's allow them to be desribed somehow. Thus the need to have access to the hardware forces the programming language to make such access possible.

2.4. *Fortran*

Although Fortran is only slightly more popular than the bubonic plague in certain circles these days, it was an enormous advance at the time it was conceived. At that time, most people thought it would never be possible to replace assembly language. (Nowadays some people fear it will never be possible to eradicate Fortran).

In order to give Fortran a fighting chance for survival in a world full of hostile assembly language programmers, its designers tried to make it as easy as possible to produce an efficient compiler for it. To do this, they designed the Fortran statements with their ultimate translation to IBM 704 machine code in mind. Thus the hardware of the IBM 704 had a profound effect on Fortran, and on programming and programmers for years to come.

Specific features and instructions of the IBM 704 gave rise to specific features in Fortran as illustrated below.

IBM 704 INSTRUCTIONS	FORTRAN II STATEMENTS
load, store, arithmetic	expressions and assignment
conditional skip (CAS)	IF (expression) n1, n2, n3
count and loop (TXL, TXH)	DO n I = k1, k2, k3
transfer (TRA n)	GO TO n
indexed jump (TRA n,1)	GO TO I, (n1,n2,...,nk)
transfer and set index (TSX P)	CALL P
read/write n characters (IOCD)	READ, PRINT

Furthermore, storage in Fortran was static and allocated at compile time, the same as in 704 assembly language programming. In short, nothing was put into Fortran unless it had a very close counterpart in 704 machine code. The hardware heavily influenced the programming language, and at the same time insured that Fortran would be a success.

It is interesting to speculate what would have happened had the Fortran designers not been so strongly influenced by the 704 hardware. It is very likely that Fortran programs would have been unacceptably large and slow, and that all the assembly language programmers (fearing obsolesence and unemployment) would have said: *"I told you so"*, resulting in high level languages being abandoned altogether as impractical.

2.5. *Memory size, overlays, and files*

It often happened, and in fact still sometimes happens, that a program is too big to fit into memory. Even more frequently it happens that the data is too big to fit into memory. To accomodate this unpleasant aspect of the hardware, certain features have been introduced into programming languages. In the case of excessively large programs, overlays; in the case of excessively large data, files. Note that if the machine's address space were large enough to hold the entire program and all the data, there would be no need for overlays or file manipulation statements.

At the time Fortran IV was developed, the hardware base for files was magnetic tape. Magnetic tapes are inherently sequential devices. Not surprisingly, files in Fortran are also sequential. If random access disks had been more widely available then, it is likely that Fortran file i/o would have been random access too. In contrast, PL/I and ALGOL 68 were developed later, and both of these languages reflect the improved hardware by allowing random access files.

On a computer system with a very large address space, files and overlays are not needed. One might expect that on such systems, the programming languages developed would reflect this. In the MULTICS system for the Honeywell 66/80, the hardware allows up to 2^{18} segments of up to 2^{16} 36-bit words. A collection of data is stored as a segment. To access external data, a PL/I programmer need only append a "$" to the segment name.

For example, to count the number of periods in a segment, INPUT, of length 100 000 characters, the following PL/I program could be used:

```
DECLARE INPUT$ CHAR(1:100 000), I FIXED, COUNT FIXED INITIAL(0);
DO I = 1 TO 100 000
  IF INPUT$(I) = "." THEN COUNT = COUNT + 1;
END;
```

Note that there are no "read" statements operating on files. The segment INPUT$ is simply addressed as any other variable.

2.6. *Interactive computing*

The development of time shared computers has given rise to the birth of programming languages especially suited for on line work, e.g., BASIC and APL. Compilers intended for interactive work often attempt incremental compilation, compiling statements as they are typed in. Furthermore, when statements are changed after having been entered, incremental compilers try to recompile the new statement all by itself, without having to recompile all the other statements that were not changed. Recompiling the entire program wastes CPU time.

Interactive computing is most popular for teaching purposes and for use by non programmers (e.g., engineers). These users tend to make many mistakes, and tend to take a dim view of having to wait several minutes only to discover that they left out a comma somewhere. They want to be told about any errors immediately, something which is much easier to achieve with incremental compilation than with conventional compilation.

The influence from the hardware here is toward programming languages that are easy to compile incrementally. In practice this means that features which make incremental compilation difficult are avoided. Such awkward features include block structure, scope rules, typed variables (i.e. declarations), nesting of procedures and if statements, and complicated statements in general. Statements in BASIC are short, contained on one line, and highly inde-

pendent of one another, leading to a language that is well suited to interactive computing. BASIC is far and away the most widely used language for interactive computing, and its simple structure is at least partly responsible for its popularity.

2.7. *Interrupts and traps*

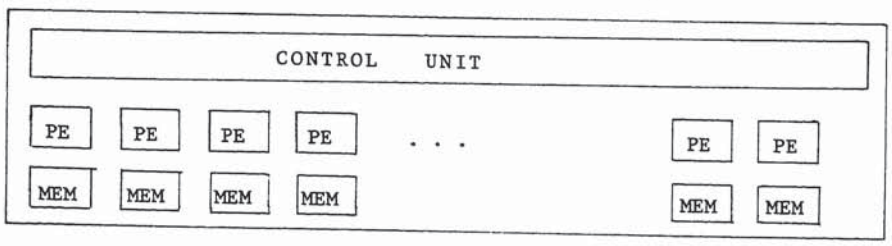
Many computers have hardware facilities for interrupting a running program when an exceptional condition occurs, and beginning execution of a specified routine to handle the situation. Among the conditions that are often recognized by the hardware are i/o completion, integer overflow, floating point overflow and underflow, attempt to divide by zero, stack limit and addressing errors.

Some programming languages, for example, PL/I, provide a mechanism for programmers to handle certain kinds of hardware traps (and also software traps such as format conversion error) themselves. In PL/I these are called "on conditions". A PL/I programmer can specify that a certain procedure is to be invoked whenever an overflow occurs. This procedure can do whatever it wants to, and then return to the point at which the trap occurred, to continue execution.

2.8. *Special languages and peculiar hardware*

Some computers are so strange that conventional programming languages are hopeless at worst, and grossly inadequate at best. The Illiac IV is an example of a machine whose hardware is so different from the usual garden variety computer, that normal programming languages cannot be used. A brief description of the Illiac IV may make this point clearer.

A complete Illiac IV consists of four quadrants, each of which has 64 processing elements (PE's). All 64 PE's in a quadrant are controlled by a single control unit. Each PE has a private memory



of 2K words. The most unusual feature of the hardware is that when the control unit issues an instruction, all 64 PE's carry it out simultaneously, using data from their own memories. The average instruction time is 312.5 ns, but in this time up to 256 computations could be performed, one per PE. A full Illiac IV can carry out floating point arithmetic operations at an effective speed of nearly one operation per nanosecond. It is said that at the time the Illiac IV was first proposed, one such machine would have doubled the computing power of the entire world!

It is clear that any language in which only sequential processes can be described, will waste 63/64 of the computing capacity of an Illiac IV quadrant.

Furthermore, the parallel processing facilities offered by languages such as Simula, PL/I, and Algol 68 are of no use, because each process must execute precisely the same program, synchronized down to the level of individual machine instructions. An odd beast like the Illiac IV essentially forces its keeper to develop a new programming language to suit it.

At least two programming languages have been developed for the Illiac IV: IV tran and Glypnir. These languages have been described in the literature (MILLSTEIN [2] and LAWRIE et al. [3]), but one small example may suggest some of the features needed.

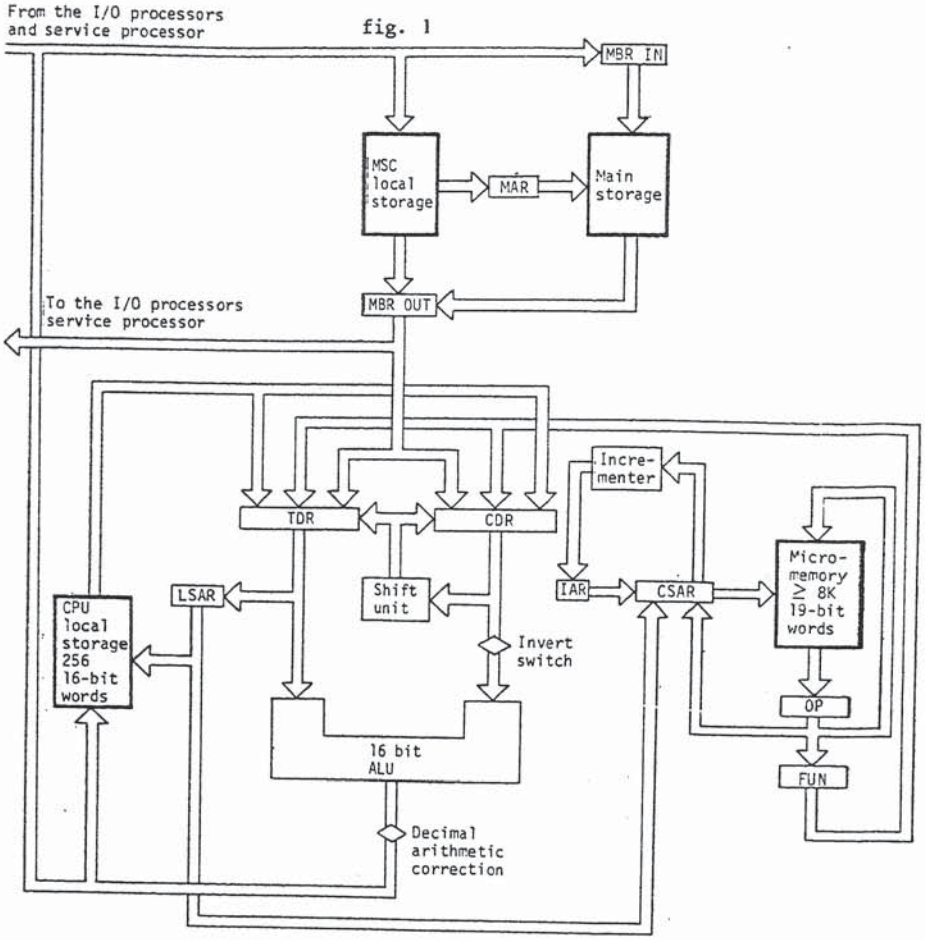
FORTRAN	IV TRAN
`  DO 1 I = 1,3`	`  DO 1 FOR ALL (I,J,K)/[1...3].C.[1...7].C.[1...9]`
`  DO 1 J = 1,7`	`  A(I,J,K) = SQRT(A(I,J,K))`
`  DO 1 K = 1,9`	`1 CONTINUE`
`  A(I,J,K) = SQRT(A(I,J,K))`	
`1 CONTINUE`	

3. THE HARDWARE SOFTWARE INTERFACE

Most machine languages are secret. And terrible. The actual hardware of most modern computers consists of an arithmetic logical unit (ALU), some registers, one or more memories and a collection of buses to connect the pieces. The ALU can usually perform addition, subtraction, shifts, and some boolean operations on 8, 16, 32, or 64 bit binary or decimal numbers.

The instruction set typically consists of 10 - 20 instructions, mostly concerned with register transfer, memory read/write, operation of the ALU,

fig. 1



REGISTERS

Name	Width	Use
TDR	16	Left input to ALU (true data register)
CDR	16	Right input to ALU (complement data register)
MBR IN	16	Halfwords to be stored in main storage are loaded here
MBR OUT	16	Halfwords read from main storage arrive here
CSAR	15	Micromemory MAR (control storage address register)
OP	19	Micromemory MBR (for reading and writing)
FUN	19	Holds microinstruction while it is being executed
LSAR	4 x 4	Used to address CPU local storage (local storage address registers)
IAR	3 x 15	Micro program counters (IAR0, IAR1, IAR2)

Notes: LSAR is 4 registers of 4 bits each
IAR is 3 register of 15 bits each

FUNCTIONAL UNITS

Name	Output width	Use
ALU	16	Binary addition, AND, OR, and EXCLUSIVE OR
Shift unit	16	Shifts 0-15 bits left or right with zero fill
Incrementer	15	Increments current micro program counter (IAR) by 1
Invert switch	16	Controls ALU right input: CDR, 1's complement of CDR, 16 ones or 16 zeroes
Decimal arithmetic correction	16	Performs correction needed to convert binary addition to decimal addition

and conditional branching. Anyone programming in machine code must be aware of the existence and use of all the registers, memories, functional units, as well as their timing properties, which are generally unpleasant.

Fig. 1 shows the hardware organization of an IBM 370/125. The ALU has four functions: ADD, AND, OR, and EXCLUSIVE OR. There is also a shift unit. The ALU is fed by the two 16-bit registers TDR and CDR. The right ALU input can be CDR, the 1's complement of CDR, all zeroes or all ones.

The 370/125 has four memories. The micromemory (or control store) holds the machine language program, which consists of 19-bit machine instructions. The CPU local storage consists of 256 16-bit words of scratch storage for the machine language program's variables and temporaries. The main storage consists of 98K to 128K 8-bit bytes, and contains the data upon which the machine language program operates. The MSC local storage consists of 32 23-bit words used to address main storage.

The 370/125 has 13 machine instructions, each 19 bits long. They can roughly be divided into four groups: loading TDR or CDR, ALU operation, conditional branch, and miscellaneous, as follows:

Load TDR or CDR

1. Load 1 byte of TDR/CDR with immediate data. Zero or retain other byte.
2. Load TDR/CDR from CPU local storage.
3. Load TDR/CDR with byte or word from main storage.
4. Load TDR/CDR from MSC local storage.

ALU operation (ADD, AND, OR, or EXCLUSIVE OR)

5. ALU operation. Store result in CPU local storage.
6. ALU operation. Store result in main storage.
7. ALU operation. Store result in MSC local storage.

Test and conditional branch

8. Test for presence or absence of one out of 48 conditions.
9. Branch if previous test yielded true.
10. Conditional branch on one out of 10 conditions.

Miscellaneous

11. Shift CDR left or right 0 - 15 bits with zero fill, and store in TDR or CDR.
12. Sense and control. Read or set clocks, timers, other devices.
13. Table look up. Effectively a 256 way branch.

We have described the 370/125 hardware in some detail to demonstrate two points:

1. The 370/125 is a general purpose computer.
2. The 370/125 bears no resemblance whatsoever to the machine described in the IBM publication "IBM System/370 - Principles of Operation".

Not only does the 370/125 bear no resemblance to the familiar 370 machine language with 16 32-bit registers and over 200 instructions, but the hardware of the other models of the 370 series do not either. Furthermore they are all totally different from each other. The model 115 instruction set has about as much relation to the model 168 instruction set as the PDP-8 does to the Cyber 73. The various models of the 370 series are simply unrelated computers produced by the same manufacturer.

Each model of the 370 series comes with an interpreter (microprogram) for "370 machine language". All models of the series interpret exactly the same language which we will call level 2 language. Thus programs written in this language can be moved from model to model without any changes, and will produce the same results on any model. Of course, they will produce them much faster on the larger models than on the smaller ones.

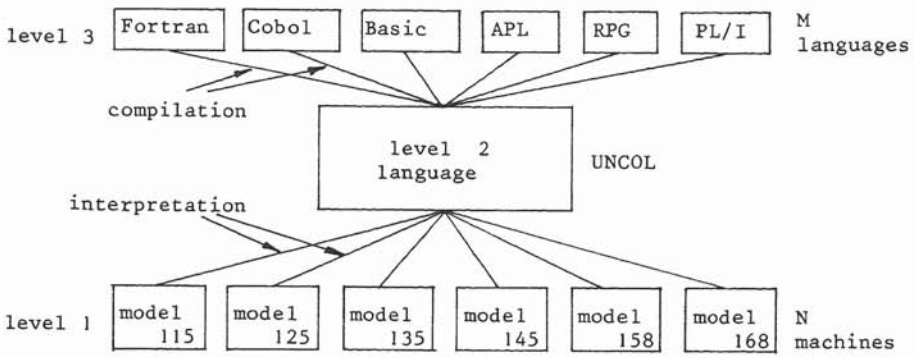
It is worth emphasizing again that the language most people think of as 370 machine language, level 2 language, is simply an arbitrary language which all 370 models can interpret in software. By changing the interpreters one can change the "machine language". In fact IBM supplies interpreters for several other "machine languages" including the 1401, 1410, 1440, 1460, 7040, 7044, 7070, 7080, 7090 and 7094.

The curious reader may be wondering at this point *"why bother with interpreted pseudo machine languages? Why not just program each model in its own machine language?"*. There are at least three reasons. First, the true machine languages of all the models are very simple. It may take 10 - 20 or more machine language instructions to carry out one level 2 instruction. If each level 2 instruction were expanded to the equivalent sequence of level 1 instructions, the size of main storage would have to be increased by a factor of 10 - 20 or more to hold the new programs. This would vastly increase the price of all the machines.

Second, writing programs for, or compiling programs to, the level 2 language is hard enough, but using the level 1 language would be far, far worse, due to its complexity. In particular, timing considerations are often

a problem. Thus the use of level 1 language simplifies compiler construction.

Third, the level 2 language functions as an UNCOL as shown below.



For M languages and N machines, only N + M compilers and interpreters are needed, instead of M * N. To introduce a new computer, IBM need only write a single program for it - an interpreter for level 2 language. To introduce a new language on all the existing (and totally different remember) machines, they need produce only one compiler.

Although the above discussion has been in terms of the IBM 370, nearly all the manufacturers (except CDC) use the UNCOL idea to produce portable software. DEC, Honeywell, Univac, XDS and others all have computer series whose models all interpret a common level 2 language on different CPU's with different architecture's and instruction sets, in order to provide for portability across models.

From the above figure it is clear that the level 2 language serves as an interface between the level 1 languages (the true hardware machine languages) and the level 3 langues (the programming languages). The compilers convert programs into the level 2 intermediate language for interpretation by the interpreters running at level 1. It is also clear that the interface, level 2, is neither dictated by the hardware nor the programming languages. Rather, it is chosen as a compromise between the needs of the compiler writer and the interpreter writer.

Once one has realized the arbitrary nature of the interface, certain questions come to mind:

1. *Is it a good idea to have just one interface level, or should there be more than one?*
2. *Are current interface levels optimal? Are they adequate? Are they tolerable?*

In a certain vague sense the design of the interface level is the design of the hardware. The "hard"-ness comes from the fact that writing new interpreters (microprograms) is hard - many manufacturers strongly discourage it. Also, on some older machines, like the CDC Cyber 70 series, the interface level really is the hardware machine language; its instructions are carried out by very complicated digital timing circuits, in effect constructing the interpreter out of transistors.

To the extent that the interface level is hardware, influences upon its design by software may be regarded as effects of programming languages on hardware design. The remainder of this paper will be concerned with such influences.

4. THE MU5

When designing a new computer, the designers must determine the architecture of the interface level. Traditionally the interface level was kept very simple, making it easy to write the interpreters, but difficult to write the compilers. In as much as there are usually more compilers than interpreters, and in as much as a compiler is far more difficult to write than the equivalent interpreter, there is considerable reason to try to make the interface level as "high" as possible. The main advantages of a high level interface are ease of compiler construction, and better object program efficiency due to the compiler's ability to generate "good code" (due to the lack of peculiar instructions). Also, the steady drop in hardware costs may make it feasible to implement the interface level directly in hardware in the future.

In this section we will examine the architecture of a computer intended specifically for running high level languages, the MU5. The MU5 was designed by Manchester University (in 1968) and built by ICL. The following description of its interface level is just a brief sketch. For more details [4,5,6] should be consulted.

The MU5 is basically a single address machine. Unlike many contemporary computers it does not have a large collection of accumulators whose use requires extensive optimization (e.g., the Cyber 70 has 3 classes of registers to be optimized, A,B,X, with variations within the classes, i.e. A0 is different from A6). The MU5 has only one of each class of register.

In order to achieve speed, the MU5 has a cache or buffer memory. This

is an associative memory that holds the most recently used instructions and operands in high speed storage. Whenever an operand or instruction is needed from memory a check is made to see if it is in the cache. If so, it is taken from the cache, and no memory reference is needed. If the cache is sufficiently large, most memory references can be satisfied, and the effective memory speed is the cache speed rather than the main memory speed, Because the operation of the cache is completely transparent, the compiler needs not concern itself with it.

The instruction set has been designed to be small and simple. The operation and operand fields are orthogonal. Most instructions are 16 bits, except those containing long literals (up to 64 bits), or long offsets (16 bits).

The MU5 has a segmented virtual memory. There are 2^{14} segments of 2^{18} bytes. A 32 bit address can select any byte in the address spaces. Integers are 32 bits and reals are 64 bits.

One segment is special, called the name segment. It is used for non-array variables, and for descriptions of array variables. It also holds certain administrative information. For block structured languages the name segment operates like a stack, with one frame per currently active range (or procedure).

4.1. *Registers*

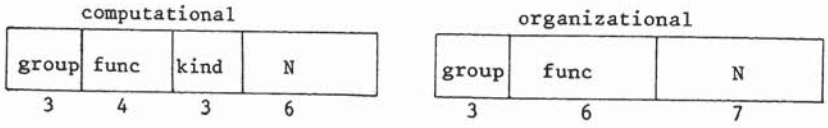
The principal MU5 registers and their lengths are:

NB (16) - Name base. Points to bottom of current name segment frame.
XNB (16) - Extra name base. Used to access other frames.
SN (14) - Segment name. Specifies the name segment.
SF (16) - Stack front. Points to the last word in the name segment.
A (64) - Accumulator for floating point and decimal arithmetic.
X (32) - Fixed point accumulator.
B (32) - Index register. Used for counting and subscripting.
BN (1) - Boolean accumulator.
T (3) - Test register. Stores result of previous compare.
D (64) - Descriptor register. Used for array access.
XD (64) - Extra descriptor register, for a second array.
MS (16) - Machine status (PSW) concerned with interrupts, etc.

The current range (or procedure) frame is bounded by NB and SF(SF > NB). The first word contains the return address, previous NB and previous MS, stacked by the hardware. Then come the actual parameters, local variables, and scratch storage. SF points to the last word of scratch storage. Parameters and local variables are accessed by giving their offset from NB in units of 32 bits for integers and 64 bits for reals and descriptors. A 6 bit offset is used, allowing up to 64 integers and 32 reals/descriptors per frame. If this is insufficient a 32-bit instruction with a 16-bit offset is possible. To access data in another frame, the XNB must first be set to that frame's base. Then the reference uses XNB instead of NB.

4.2. *Instructions*

The MU5 has two instruction formats, each 16 bits:



There are 8 instruction groups. Group 0 uses the organizational format; groups 1 - 7 use the computational format.

Group	Use
0	Register manipulation, branching, boolean, special
1	B register arithmetic
2	Array accessing using XD
3	Array accessing using D
4	Fixed point arithmetic on signed 32-bit numbers
5	Logical operations on unsigned 32-bit numbers
6	Decimal arithmetic on 64-bit numbers (sign, 15 digits)
7	Floating point arithmetic on 64-bit numbers

Groups 1 - 7 each have 16 instructions, determined by func. Groups 1,4,5,6, and 7 have similar instructions, but on different registers. A brief summary of groups 1 - 7 follows.

Func	Instr.	Func	Instr.	Func	Instr.	Func	Instr.
0	load	4	add	8	x or	12	rev. subtract
1	load double	5	subtract	9	or	13	compare
2	stack & load	6	multiply	10	shift	14	convert
3	store	7	divide	11	and	15	rev. divide

The func = 2 instructions first increment SF by 1, then put the B, X, or A register on the stack, then perform the load. The reverse subtract and divide instructions subtract or divide the operand by the accumulator, putting the result in the accumulator.

4.3. *Addressing*

The kind field determines the kind of operand as follows.

kind	meaning of N	kind	meaning of N
0	6 bit literal	4	B-th array element
1	register	5	B-th array element
2	offset for 32-bit quantity	6	Zeroth array element
3	offset for 64-bit quantity	7	extended

For the extended kinds, N is broken up into two 3-bit fields, N_1 and N_2, as follows.

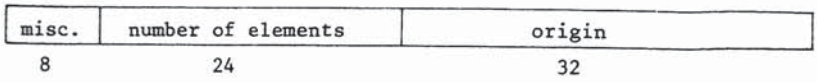
N_1	N_2 for $N_1 = 0,1$	N_2 for $N_1 > 1$
0 literal	signed 16-bit	use SF as base
1 literal	signed 32-bit	use 0 as base
2 32-bit quantity	signed 64-bit	use NB as base
3 64-bit quantity	signed 64-bit	use XNB as base
4 B-th array element	unsigned 16-bit	pop, and retard SF
5 B-th array element	unsigned 32-bit	D
6 Zeroth array element	unsigned 64-bit	NB
7 Special	unsigned 64-bit	XNB

For $N_1 = 0$ or 1, N_2 specifies the sort of literal. The literal itself follows the instruction. For $N_2 = 0 - 3$, a 16-bit signed offset follows the instruction, allowing data to be referred relative to SF, NB, XNB or 0 (absolute addressing). $N_2 = 4$ pops an operand off the top of the name segment and reduces SF by 1. $N_2 = 5,6$ or 7 are the same as 1, 2 or 3, but

with an implied offset of 0.

4.4. *Array access*

MU5 has special hardware for accessing arrays, including strings. Each array variable is represented in the name segment by a 64-bit descriptor:



The miscellaneous bits determine the element size (1,4,8,16,32 or 64 bits), enable bounds checking etc. The origin can point into any segment.

To access an array element the instruction sequence is:

1. Load the subscript into B.
2. Subtract the lower bound from B. (if 1, can be combined with step 2)
3. Perform the operation (e.g., add element to accumulator).

Both upper and lower bounds are checked. If the subscript is out of range a trap occurs.

For higher dimensional arrays a dope vector is needed, with 3 elements (lower bound, stride, upper bound) per dimension. The sequence to extract M[i,j,k] from $M[\ell_1:u_1, \ell_2:u_2, \ell_3:u_3]$ is:

1. B := i	load first subscript into B
2. SUB1 descriptor	load XD with descriptors; bounds check i; $D := (i-\ell_1) * S_1$
3. B := j	load second subscript into B
4. SUB2	$D := (i-\ell_1) * S_1 + (j-\ell_2) * S_2$; bounds check j
5. B := k	load third subscript into B
6. SUB2	$D := (i-\ell_1) * S_1 + (j-\ell_2) * S_2 + (k-\ell_3) * S_3$; bounds check k
7. B = D	$B = (i-\ell_1) * S_1 + (j-\ell_2) * S_2 + (k-\ell_3) * S_3$
8. perform operation using B-th array element	

The constants S_1, S_2, and S_3 are the strides taken from the dope vector. The complete access, including bounds checking requires 8 16-bit instructions.

Arrays of arrays can be handled by having the "elements" of the array be descriptors instead of primitives. Strings are handled as character arrays. The string operations expect the source descriptor in XD and the destination descriptor in D. There are operations to move a string, compare two strings, perform all 16 boolean operations on strings, and translate a string using table look up.

4.5. *Boolean operations*

There are instructions to combine the result in the T register with the boolean accumulator, BN, to simplify evaluating boolean expressions. For example, consider, if i - k * m < j ∧ n ≥ k then goto ℓ

```
X := k              load k into X register (fixed point accumulator)
X := k*m            multiply X by M
X := i-X            reverse subtract i
compare X,j         compare instruction sets T
BN := T<            load BN from T, with "less than" as true
X := n              load n into X
compare X,k         compare instruction sets T
BN := BN ∧ T ≥      boolean AND of BN and T, with ≥ as true
if BN then goto ℓ   branch on BN true
```

4.6. *Use of segmented address space for Algol 68*

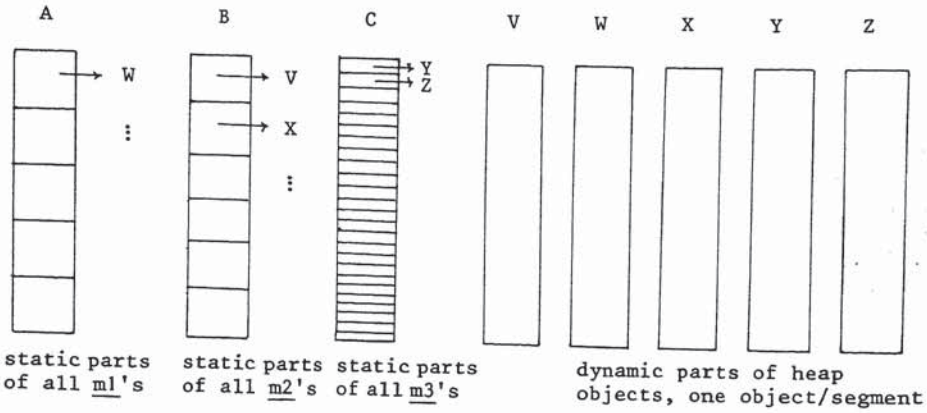
For each declared array, only a 64-bit descriptor is kept in the name segment. The elements themselves are kept in another segment, the vector stack segment. Each frame in the name segment has a corresponding frame in the vector stack segment. When a range (or procedure) is exited, both the top frame in the name segment and the top frame in the vector stack segment are released.

Local generators for non-flexible arrays use the vector stack segment. All of this storage is recovered upon exit.

Each local flexible array is put into a separate segment. This allows it to grow and shrink in an arbitrary way. To keep track of which segments are currently in use for local flexible arrays, a segment stack, called the pile, is maintained (in a separate segment), with one frame per range (or procedure). Upon exit, all the segments used for local flexible arrays can be put back on the free segment list.

For each heap generator in the program a segment is allocated for the static part of the created objects. The dynamic parts (e.g., array elements) of each object are kept in a separate segment. For example

(... heap m1 ... heap m2 ... heap m3 ...)



Consequently, all the objects in A are the same size and format, and the format is known at compile time. All the free slots in A are chained together. Similarly for B, C, etc.

To garbage collect, start from the name segment and follow all pointers, marking active storage. Then scan A, B, C, etc. linearly to find all unused slots. Put the unused slots on the free list for that segment, and release the segments containing the dynamic parts. Notice that no compacting or updating is needed. Furthermore, since storage does not suddenly run out (unless you use 2^{14} segments) garbage collection can always be postponed until it is convenient, e.g., until the working stack is empty.

4.7. *Parallel processing*

Each subprocess has its own name segment. However, its static chain may point back to its parent. By using XNB, a subprocess can access any of its ancestors' variables.

To implement semaphores there is a read and mark instruction. It reads an operand into an accumulator and simultaneously zeroes the operand. Sema-

phores can be implemented by having a single data process containing all semaphore information. As long as only one process can access the data base at a time, all is well. Have a flag variable free. Free = 0 when the data base is in use, Free = 1 when it is available. Before accessing the data base, a process executes the following program:

```
ℓ:  A := free; free := 0; ¢ read and mark -1 instruction ¢
    if A = 0 then goto ℓ fi;
    use data base;
    free := 1 .
```

5. BURROUGHS B1700

An alternative to having a single interface level used by all compilers and interpreters is to have a different interface level for each programming language. Thus the Fortran compiler would produce an intermediate program in a language suited to Fortran to be interpreted by the Fortran interpreter. Likewise for other languages. This method, used on the B1700, has the advantage that each intermediate level can be tailored to the specific programming language, rather than forcing all compilers to use a single "machine language" that is probably not well suited to any of them.

On the B1700, the intermediate levels are intended to be as close to the programming languages as possible. To allow the intermediate levels to be arbitrarily defined, the architecture of the true hardware (level 1) must be very flexible. In the following sections we will examing the B1700 machine language. The reader should contrast it with the 370/125 machine language described earlier. See LINDSEY [7] and WILNER [8] for details.

5.1. *B1700 level 1 architecture*

The most unusual feature of the B1700 is its ability to address main memory to the ~~bit~~. Every bit has a specific address. An arbitrary bit string of 1 to 24 bits can be read out of main memory beginning at any bit. This allows the interpreted machines called S-machines to use arbitrary sized fields wherever needed.

The B1700 has about 60 registers and pseudo registers, each with a special function. All registers are visible and accessible, including the microprogram counter, microinstruction register etc. Some of the principal

registers and their lengths are:

X	(24)	- left ALU input
Y	(24)	- right ALU input
L	(24)	- used for testing and skipping
T	(24)	- used for bit extraction and shift/rotate
F	(48)	- holds a main memory address (specified to the bit)
A	(24)	- a 16 word hardware stack for microprocedure calls
C	(24)	- determines ALU width, mode (binary/decimal) carry bit, etc.
XYST	(4)	- holds bits for least signif. unit of X, interrupt pending, $Y \neq 0$, $X \neq 0$
XYCN	(4)	- holds sign of X, result if $X = Y$, $X < Y$, $X > Y$

X, Y, L and T can all be loaded from or stored into memory directly. All four are also used as scratch registers. There is also a CPU local storage of 32 24-bit words, used for temporaries and S-machine registers.

The ALU continuously computes 9 functions of X and Y and stores the results in 9 pseudoregisters. The 9 functions are: add, subtract, and, or, exclusive or, complement x, complement y, mask x and mask y. The mask functions copy the low order n bits (n is in the C register) and zero the rest. The ALU width is adjustable from 0 to 24 bits under program control. Its radix (binary or decimal) is also under program control. To add X and Y and put the result in L, one moves the SUM pseudoregister to L. There are no microinstructions "ALU operation" specifying a function.

There are a number of 4 bit condition registers, such as XYST and XYCN whose bits give a continuous readout of certain conditions. For example, one bit in XYCN is set whenever $X < Y$ holds. There is a microinstruction for conditionally branching on any bit in any register, so *if* $X < Y$ *then goto* ℓ *fi* requires only one microinstruction.

There are 32 microinstructions, but some are similar, e.g., branch forward and branch backward are distinct. The principal operations possible are:

1. Register to register move
2. Move literal (8 or 24 bits)
3. Shift/rotate (X,Y,T, or X combined with Y)
4. Extract 0 - 24 bits from the middle of T, move to any register right justified with zero fill
5. Conditional branch on bit (any bit in any register may be tested)

6. Unconditional branch
7. Microprocedure call (return address is pushed onto A stack)
8. Read/write main memory to/from X,Y,L, or T
9. Read/write CPU local storage to/from any register
10. Manipulate F (several instructions)

5.2. *B1700 S-machines*

The considerable flexibility of the B1700 architecture makes it possible for compiler writers to design their own target machines by successive approximation. First an S-machine is designed and implemented. For example, in the System Development Language (SDL) used by Burroughs for writing compilers, four bits were initially allocated to the display level. Later it appeared that no programs were ever nested deeper than 7 levels, so the display field was changed to three bits. On nearly all other machines such a change would be impractical.

Because the extract microinstruction can be used to quickly analyze variable sized fields, a variety of opcode lengths are used in the S-machines, to improve performance. The most common opcodes are given the shortest codes, in an approximation to a Hoffman code.

Address field lengths can be varied from program to program. For example, if a compiler discovers that a particular program uses only one level of block nesting and no more than 16 variables per level, it can compile 5-bit address into the object program and inform the interpreter of this fact. In effect the S-machine can be adjusted at compile time to fit the particular program being run.

As another example of the B1700 flexibility, a program can begin running, read some input data, and then decide how many bits an integer should have. The interpreter can dynamically adjust the ALU width in one microinstruction.

In some cases the S-machine is so close to the programming language being used, that absence of goto's in the programming language implies that there need be no goto's in the S-machine's instruction set. This is the case for the SDL machine, for example, which has instructions for if, case, loop, call, etc., but no branch instruction.

REFERENCES

[1] WIRTH, N., *PL 360 - A Programming Language for the 360 Computers*, JACM 15 (1968), 37-74.

[2] MILLSTEIN, R., *Control Structures in Illiac IV Fortran*, CACM 16 (1973), 621-627.

[3] LAWRIE, D., T. LAYMAN, D. BAER & J. RANDAL, *Glypnir - A Programming Language for Illiac IV*, CACM 18 (1975), 157-164.

[4] LAWRIE, D., T. LAYMAN, D. BAER & J. RANDAL, *MU5 Basic Programming Manual*, Dept. of Computer Science, University of Manchester, 1974.

[5] KILBURN, T., D. MORRIS, J.S. ROHL & F.H. SUMNER, *A System Design Proposal*, IFIP Cong. 68, 806-811.

[6] LINDSEY, C.H., *Making the Hardware Suit the Language*, in proceedings of the Algol 68 Implementation Conference, J.E.L. Peck (ed.), North Holland Publ. Co., 1971.

[7] LINDSEY, C.H., *Burroughs B1700 Systems Reference Manual*, Burroughs Corp., 1972.

[8] WILNER, W., *Design of the Burroughs B1700*, Proc. FJCC (1972), 489-497.

MATHEMATICAL CENTRE SYLLABUS 25, 1976, XII, 219-236

CONCURRENT PASCAL, EEN TAAL VAN P. BRINCH HANSEN

F.E.J. KRUSEMAN ARETZ

(Philips Research Laboratories, Eindhoven)

0. INTRODUKTIE

Concurrent Pascal is een door BRINCH HANSEN ontworpen programmeertaal, bestemd voor algorithmische beschrijving van operating systems. Centraal staan derhalve de begrippen proces en procesinteraktie via gemeenschappelijke gegevens. Processen en kollekties gemeenschappelijke gegevens worden in Concurrent Pascal "system components" genoemd. De regels van de taal leggen hierarchische relaties tussen system components op en daarmee dwingt de taal een min of meer hierarchische wijze van systeemontwerp af.

Het aantal algemeen toegankelijke publikaties over Concurrent Pascal is (nog) tamelijk klein (BRINCH HANSEN [1,2], VERVOORT [3]); om die reden probeert dit verhaal in eerste instantie inzicht in de taal en zijn gebruik te geven en slechts sekundair enige kritische evaluatie te bieden.

Concurrent Pascal sluit nauw aan bij Pascal (WIRTH [4]) voor wat betreft deklaratiestruktuur en de uitdrukkingsmiddelen voor flow-of-control in de sekwentiële processen. Het is echter een geheel andere taal dan de eveneens op Pascal gebaseerde taal die BRINCH HANSEN in zijn boek [5] over operating systems als voertuig bezigt.

1. SYSTEM TYPES EN SYSTEM COMPONENTS

In het algemeen bestaat een programma in Concurrent Pascal uit een drietal onderdelen:

1. een beschrijving of blauwdruk van alle verschillende types van system components; hiervoor wordt de type-definitie van Pascal gebruikt, waarbij aan elk system type een naam wordt toegekend;
2. een opsomming van (de namen van) alle system components, waarbij voor elk het type (d.w.z. een verwijzing naar bovengenoemde beschrijvingen)

wordt gegeven, en waarbij diverse components van hetzelfde type mogen zijn (voorbeeld: een aantal processen, elk voor de besturing van een terminal); hiervoor wordt de deklaratie van variabelen in Pascal gebruikt, waarin aan elke system component een naam wordt toegekend;

3. een aktivering van alle system components, waarbij tevens de hierarchische betrekkingen tussen die components worden vastgelegd; hiervoor wordt een speciale statement, beginnend met het symbool "init", gebruikt.

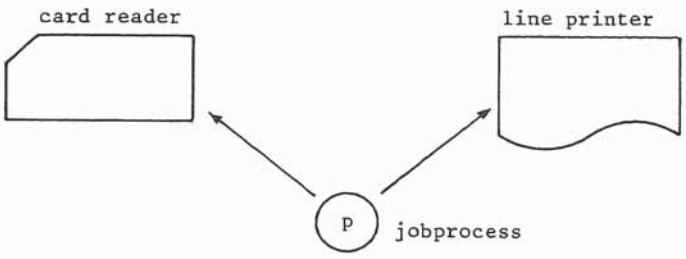
Uit het voorgaande volgt dat het aantal processen van een in Concurrent Pascal beschreven systeem in de deklaratiestruktuur wordt vastgelegd en dus geen dynamisch karakter heeft. Eveneens worden de onderlinge relaties tussen de processen en data-kollekties in de programmatekst vastgelegd en deze zijn dus evenmin aan dynamische wijzigingen onderhevig. Dit alles geeft in Concurrent Pascal geschreven systemen een statische struktuur. De system components, hoewel geïntroduceerd als variabele, zijn allerminst variabel. Ze worden geacht permanent te bestaan, en we kunnen geen assignments doen met de naam van een system component als linkerlid.

Als een zeer elementair voorbeeld van het voorafgaande beschouwen we een programma met slechts één component, een proces dat kaarten leest van een kaartlezer en deze afdrukt op een regeldrukker:

```
type page = array [1 .. 512] of char;
     job = process
           var block: page;
           begin cycle readcards(block);
                       printlines(block)
                 end
           end;
var jobprocess: job;
begin init jobprocess end.
```

Hierin is job (de naam van) een system type en jobprocess een system component. Een process type bestaat uit "process", mogelijk gevolgd door een parameterlijst (waarover later meer) en een puntkomma, en een blok. De lokale variabelen van het blok van een process type zijn "private variables" voor dat type: elke system component van dat type zal over zijn eigen kollektie daarvan beschikken en deze is voor geen enkele andere system component bereikbaar. Gezien het eeuwige leven van system components worden de lokale variabelen daarvan ook "permanent variables" genoemd.

Een goed hulpmiddel om de struktuur van een programma in Concurrent Pascal weer te geven is de "access graph"; voor bovenstaand programma ziet deze er als volgt uit:



De pijlen hierin geven acces aan en niet de richting van de informatiestromen.

Op details van "readcards" en "printlines" gaan we niet verder in.

2. SHARED DATA

Processen in Concurrent Pascal kunnen samenwerken door operaties uit te voeren op een kollektie gemeenschappelijke gegevens. Zo'n kollektie heet "monitor" en is een system component. Het is de bedoeling dat een monitor in feite een door de programmeur gedefinieerde abstract data structure implementeert, te zamen met de op die data structure gedefinieerde operaties. Een eenvoudig voorbeeld hiervan is een buffer waarin een portie informatie kan worden opgeslagen. De operaties daarop zijn dan bijvoorbeeld "send", om informatie in de buffer te bergen, en "receive", om daaraan informatie te ontlenen. De wijze van opslag, en dus ook de algorithmen voor send en receive, zijn voor de gebruiker van zo'n buffer irrelevant en kunnen het beste in de monitor "verborgen" worden.

Volgens een aan het class-concept van Simula 67 (BIRTWISTLE e.a. [6]) ontleend schema bestaat een monitor type uit: "monitor", mogelijk gevolgd door een parameterlijst (waarover later meer) en een puntkomma, en een blok. De lokale variabelen van het blok konstitueren met elkaar de abstract data structure; ze zijn individueel onzichtbaar buiten de monitor. Op de lokale variabelen (de "shared data") volgen de "procedure entries" en de "function entries". De namen hiervan zijn de enige lokale namen van een monitor die extern zichtbaar zijn; de bijbehorende bodies implementeren de operaties op de abstract data structure. De compound statement van het blok, tenslotte, wordt één maal uitgevoerd voor elke system component van het beschouwde monitor type en dient voor initialisatie van de abstract data structure.

Voor een buffer waarmee een proces blokken informatie naar een ander proces kan oversturen zouden we dan in nulde benadering tot de volgende type-definitie komen:

```
type corebuffer = monitor
                  var mem: page; full: boolean;
                  procedure entry send(block: page);
                  begin if ¬ full then
                        begin mem:= block; full:= true end
                  end;
                  procedure entry receive(var block: page);
                  begin if full then
                        begin block:= mem; full:= false end
                  end;
                  begin full:= false end;
```

We zien hier direkt een aantal problemen die om een oplossing vragen:

1. Willen we kunnen garanderen dat de variabelen mem en full een koherente waarde hebben bij het aanroepen, door een proces, van een van de procedure entries, dan zullen de aanroepen van de procedure entries van een monitor elkaar in de tijd moeten uitsluiten: de monitor verleent aan de diverse, een procedure entry van die monitor aanroepende processen beurtelings een exklusief acces, en zonodig worden processen even stilgelegd bij het aanroepen van een procedure entry om deze mutual exclusion te realiseren. Aangezien het de bedoeling is dat processen relatief weinig tijd in de monitor doorbrengen spreekt men hier van "short term scheduling" en wordt in de taal niet vastgelegd in welke volgorde een aantal op exklusief acces wachtende processen tot de monitor wordt toegelaten,
2. De send-operatie bij een volle buffer, en de receive-operatie bij een lege buffer moeten nog gedefinieerd worden. Het zendende proces zal, indien de buffer vol is, moeten wachten tot het ontvangende proces de buffer heeft geleegd. Voor deze proces-synchronisatie of "long term scheduling" kent Concurrent Pascal variabelen van het type "queue", lokaal ten opzichte van een monitor, waarin een proces zichzelf kan "ophangen" om later, door een ander proces, te worden "verlost". Een proces hangt zichzelf in een queue-variabele "q" op door een statement "delay(q)" binnen een procedure entry uit te voeren. Daarbij verliest dat proces tevens het exklusieve accesrecht. Een ander proces kan eerdergenoemd proces daarna verlossen door een statement "continue(q)". Daarbij verlaat dat andere

proces automatisch de (procedure entry en de) monitor en wordt het accesrecht overgedragen aan het verloste proces dat de procedure entry waarin het zichzelf had opgehangen voortzet achter de delay-statement. Dit in de veronderstelling dat er een proces uit de queue-variabele te verlossen viel. Anders heeft de statement "continue(q)" geen ander effect dan het verlaten van de monitor door het die statement uitvoerende proces.

We komen hiermee tot de volgende, definitieve tekst voor "corebuffer":

```
type corebuffer = monitor
                  var mem: page; full: boolean;
                      sender, receiver: queue;
                  procedure entry send(block: page);
                  begin if full then delay(sender);
                        mem:= block; full:= true;
                        continue(receiver)
                  end;
                  procedure entry receive(var block: page);
                  begin if ¬ full then delay(receiver);
                        block:= mem; full:= false;
                        continue(sender)
                  end;
                  begin full:= false end;
```

Variabelen van het type queue kunnen slechts één proces vasthouden. Een monitor component van bovenstaande type kan dus niet veilig door twee zendende processen als buffer gebruikt worden, tenzij andere synchronisatiemaatregelen, buiten de buffer om, het gelijktijdig gebruik verhinderen.

Een fundamenteel verschil tussen processen en monitors is dat monitors geen zelfstandige aktiviteit ontplooien en geen eigen proces doorlopen. Een procedure entry van een monitor wordt uitgevoerd door een of ander proces dat die procedure entry aanroept. Eventuele lokale variabelen van een procedure entry ("temporary variables" genoemd) worden in dat aanroepende proces ondergebracht en in geval van een "delay" ook daar veilig bewaard. De "initial operation" van een monitor, dwz. de compound statement van zijn blok, wordt uitgevoerd door het hoofdprogramma bij de aktivering van die monitor in dat hoofdprogramma. Er is dus een groot verschil tussen aktivering van een proces en aktivering van een monitor: het eerste betekent het starten van een verder zelfstandig doorwerkend proces, het tweede slechts het uitvoeren van zijn initial operation.

3. ACCESS RIGHTS

De scope rules van Concurrent Pascal wijken in een enkel opzicht af van die van (sequential) Pascal: binnen een system type zijn als regel slechts de eigen lokale namen bekend en dus geen globale namen. Op deze regel zijn twee uitzonderingen:

1. namen van constants en types diffunderen wel in system types naar binnen,
2. een system type mag een parameterlijst hebben met daarin formele parameters van een (in de tekst voorafgaand gedefinieerd) monitor type. Dan zijn in het blok van dat system type de namen van de procedure entries van die monitor bekend (mits voorafgegaan door de formele identifier en een punt).

   Voor elke system component van dat system type wordt dan bij de aktivering middels een aktuele parameter vastgelegd, tot welke monitor component die system component access heeft, of anders gezegd, met welke monitor component zijn formele parameter korrespondeert.

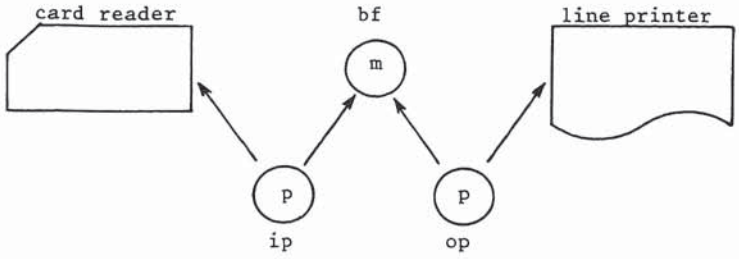
Het zijn deze regels die, met de regel dat de definitie van een type aan zijn gebruik moet voorafgaan, de hierarchische betrekkingen tussen de system components van een programma afdwingen.

Als voorbeeld geven we een uitbreiding van het in 1. gegeven programma tot een programma waarin het lezen van kaarten en het afdrukken van regels door aparte processen gedaan wordt (waardoor kaartlezer en regeldrukker tegelijkertijd aktief kunnen zijn) en de informatie van het ene proces aan het andere wordt overgedragen met behulp van de in 2. gegeven monitor:

```
type page = array [1 .. 512] of char;
     corebuffer = monitor
                  var mem: page; full: boolean;
                      sender, receiver: queue;
                  procedure entry send(block: page); begin ... end;
                  procedure entry receive(var block: page); begin ... end;
                  begin full:= false end;
     inputprocess = process (buffer: corebuffer);
                  var block: page;
                  begin cycle readcards(block);
                             buffer.send(block)
                        end
                  end;
```

```
outputprocess = process (buffer: corebuffer);
                var block: page;
                begin cycle buffer.receive(block);
                            printlines(block)
                      end
                end;
var bf: corebuffer; ip: inputprocess; op: outputprocess;
begin init bf, ip(bf), op(bf) end.
```

De bijbehorende access graph ziet er als volgt uit:



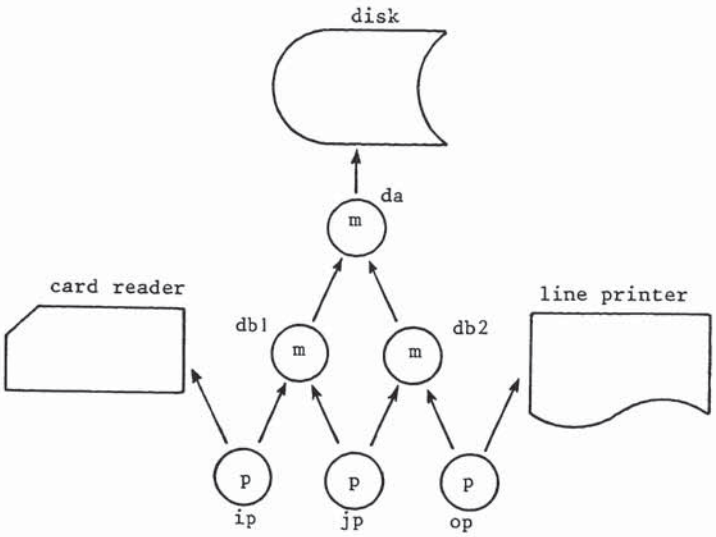
4. EEN SPOOLING SYSTEM

We geven nu een beschrijving in Concurrent Pascal van een eenvoudig spooling system met drie processen: "jp" van een type "jobprocess", dat invoer uit een inputbuffer verwerkt en uitvoer voor een outputbuffer genereert, "ip" van een type "inputprocess", dat kaarten leest en de gelezen informatie in de inputbuffer opslaat, en "op" van een type "outputprocess", dat informatie aan de outputbuffer onttrekt en op een regeldrukker afdrukt.

Voor deze processen is de wijze van informatieopslag in de buffer irrelevant; deze wordt beschreven in een monitor type "diskbuffer", met procedure entries "send" en "receive". Dit type wordt zowel gebruikt voor de inputbuffer "db1" als voor de outputbuffer "db2". De feitelijke opslag gebeurt in sporen van 512 karakters op een schijf; db1 gebruikt de sporen 0 t/m 99, db2 de sporen 100 t/m 199, en beide buffers vullen en legen deze sporen in cyclische volgorde. Om te voorkomen dat de twee buffers tegelijk een transportopdracht voor de schijf plaatsen verpakken we de transport-

opdracht in een monitor "da" van het type "diskaccess", met procedure entries "write" en "read".

De access graph hiervoor heeft de volgende gedaante:



Het programma ziet er in grote lijnen als volgt uit:

```
type page = array [1 .. 512] of char;
     diskaccess = monitor
                  var ... ;
                  procedure entry write(track: integer; block: page);
                  begin ... end;
                  procedure entry read(track: integer; var block: page);
                  begin ... end;
                  begin ... end;
     diskbuffer = monitor (disk: diskaccess; base, limit: integer);
                  var head,tail,length: integer;
                      sender,receiver: queue;
                  procedure entry send(block: page);
                  begin if length = limit then delay(sender);
                        disk.write(base+tail,block);
                        tail:= (tail+1) mod limit;
                        length:= length + 1; continue(receiver)
```

```
                end;
                procedure entry receive(var block: page);
                begin if length = 0 then delay(receiver);
                      disk.read(base+head,block);
                      head:= (head+1) mod limit;
                      length:= length - 1; continue(sender)
                end;
                begin head:= 0; tail:= 0; length:= 0 end;
    inputprocess = process (buffer: diskbuffer);
                var block: page;
                begin cycle readcards(block);
                           buffer.send(block)
                      end
                end;
    jobprocess = process (inputbuffer,outputbuffer: diskbuffer);
                var ... ;
                begin ... inputbuffer.receive(...) ...
                      ... outputbuffer.send(...) ...
                end;
    outputprocess = process (buffer: diskbuffer);
                var block: page;
                begin cycle buffer.receive(block);
                           printlines(block)
                      end
                end;
var da: diskaccess; db1, db2: diskbuffer;
    ip: inputprocess; jp: jobprocess; op: outputprocess;
begin init da, db1(da,0,100), db2(da,100,100),
          ip(db1), jp(db1,db2), op(db2)
end.
```

De parameters van een proces of monitor, voorzover niet van een monitor type, zijn altijd value parameters. Hiermee wordt aan db1 en db2 een trajekt van 100 sporen toegewezen, beginnend bij spoor 0 respektievelijk 100.

Concurrent Pascal stelt ons in staat een dergelijk spooling system in een vloek en een zucht in elkaar te zetten. Maar een zorgvuldige analyse toont aan dat bovenstaand programma wel een aantal bezwaren heeft:

1. de mutual exclusion die een monitor biedt is bedoeld voor short term scheduling: elk proces wordt geacht slechts relatief kort een monitor bezet te houden. De aanroep "disk.write(base+tail,block)" in de procedure entry "send" van "diskbuffer" duurt echter tenminste één diskaccess, echter mogelijk langer als net een ander transport in behandeling is.
2. gedurende de tijd dat "ip" met een aanroep van "send" de buffer "dbl" bezet houdt kan "jp" zich daar niet als kandidaat voor een schijftransport aanmelden. Als het schijfkanaal de flessenhals in het systeem is kan dan niet gebruik gemaakt worden van het feit dat lees- en schrijfopdrachten voor een zelfde buffer veelal op dicht bij elkaar liggende sporen plaatsvinden, en is het niet mogelijk een strategie op de volgorde van schijftransporten te ontwikkelen.
3. gedurende een schijftransport voor "ip" of "op" staat de kaartlezer dan wel de regeldrukker stil.

In de volgende sectie trachten we de eerste twee bezwaren te ondervangen. Het laatste bezwaar kan verholpen worden door het tussenschakelen van extra processen en corebuffers in analogie met het voorbeeld in 3.

5. CLASSES

Een oplossing van de efficiency-problemen van het in de vorige paragraaf beschreven spooling system moet gevonden worden in vergroting van het parallelisme in de toegangswegen tot de schijf. Laten we daarom voor de procedure entry "send" van de monitor "diskbuffer" eens nagaan op welke plaatsen het vitaal is om exklusief acces tot de monitor te hebben. Kennelijk is het noodzakelijk de statement "if length = limit then delay(sender)" en het statement-paar "length:= length + 1; continue(receiver)" ondeelbare handelingen te laten zijn. Maar zodra "length" kleiner dan "limit" is geworden betekent dit dat het spoor met spoornummer "base+tail" leeg is en gevuld kan worden, en hierin wordt geen wijziging gebracht door welke aktie dan ook van de enige tegenspeler in de buffer, die immers slechts "receive" aanroept en derhalve ten hoogste andere, nog gevulde sporen kan legen. We kunnen dus de statements van "send" en "receive" grotendeels in de processen zelf uitvoeren en slechts voor de synchronisatie via de variabele "length" een monitor gebruiken. We komen dan tot de volgende oplossing:

```
type page = array [1 .. 512] of char;
     diskaccess = monitor
                  ...
     diskbufferstate = monitor (limit: integer);
                  var length: integer; sender,receiver: queue;
                  procedure entry requesttosend;
                  begin if length = limit then delay(sender) end;
                  procedure entry requesttoreceive;
                  begin if length = 0 then delay(receiver) end;
                  procedure entry sendingcompleted;
                  begin length:= length + 1; continue(receiver) end;
                  procedure entry receivingcompleted;
                  begin length:= length - 1; continue(sender) end;
                  begin length:= 0 end;
     inputprocess = process (bufferstate: diskbufferstate;
                             disk: diskaccess; base,limit: integer);
                  var block: page; tail: integer;
                  begin tail:= 0;
                        cycle readcards(block);
                              bufferstate.requesttosend;
                              disk.write(base+tail,block);
                              tail:= (tail+1) mod limit;
                              bufferstate.sendingcompleted
                        end
                  end;
     jobprocess = ...
     outputprocess = ...
var da: diskaccess; dbs1,dbs2: diskbufferstate;
    ip: inputprocess; jp: jobprocess; op: outputprocess;
begin init da, dbs1(100), dbs2(100),
          ip(dbs1,da,0,100), jp(dbs1,dbs2,da,0,100,100,100),
          op(dbs2,da,100,100)
end.
```

Hiermee is het mogelijk geworden dat tenminste drie aanvragen voor een schijftransport, één voor elk proces, simultaan geplaatst worden. We hebben daarbij echter prijsgegeven dat de processen geen kennis behoeven te hebben over de aard van de buffers. We kunnen van die oude verworvenheid iets terugwinnen door de introduktie van classes.

Een class is in Concurrent Pascal een implementatie van een abstract data structure met de bijbehorende operaties daarop, lokaal in een proces of een monitor (of zelfs in een class). Indien een class lokaal is t.o.v. een proces, behoort de erdoor geïmplementeerde data structure tot de private data van dat proces, en is daardoor alleen toegankelijk, via zijn procedure entries, voor dat proces. Indien een class lokaal is t.o.v. een monitor, behoort de erdoor geïmplementeerde data structure tot de shared data van een aantal processen, n.l. van al die processen die access rights tot die monitor hebben. Het is in dit geval de monitor die ervoor zorgt dat slechts één proces tegelijk, via de monitor, access heeft tot de (procedure entries van) de class. Indien een class lokaal is t.o.v. een class, wordt de toegankelijkheid rekursief bepaald door die van de omvattende class.

Een class type heeft syntaktisch dezelfde gedaante als een monitor type (met "monitor" vervangen door "class"), en wordt tot de system types gerekend. Classes zijn echter geen echte system components, al noemt Brinch Hansen ze wel zo. Ze mogen niet aan processen en monitors als aktuele parameter worden meegegeven: access rights tot een class heeft slechts die system component waarin die class lokaal is, of, bij wijze van uitzondering die class waaraan eerstgenoemde class als aktuele parameter is meegegeven. De implementatie van een class is dus veel eenvoudiger dan die van een monitor.

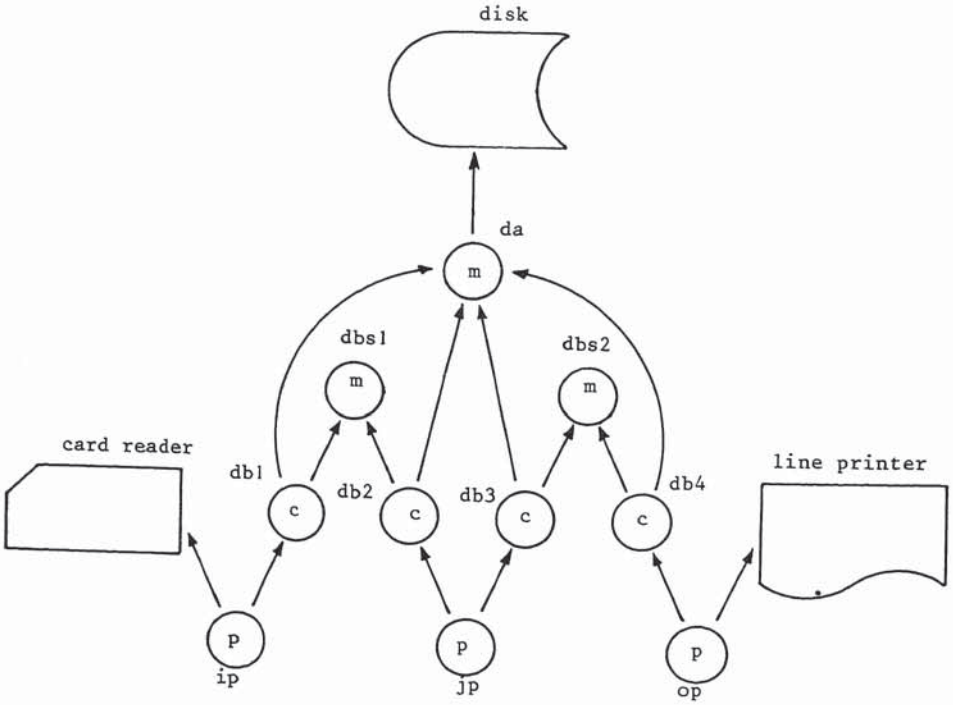
In het hieronder volgende programma voor ons spooling system is gebruik gemaakt van een class type om de struktuur van de buffers weer goeddeels voor de processen verborgen te houden. Helaas moeten wel die processen hun lokale class(es) zelf met een init-statement initialiseren, en helaas moeten wel aan die processen access rights tot de monitors "da" en "dbs1" en/of "dbs2" verleend worden, opdat de processen deze kunnen doorgeven aan hun lokale class(es). De base- en limit-parameters hebben we geëlimineerd door "head" en "tail" in het monitor type "diskbufferstate" onder te brengen.

```
type page = array [1 .. 512] of char;
     diskaccess = monitor
                  ...
     diskbufferstate = monitor (base, limit: integer);
                  var head,tail,length: integer; sender,receiver: queue;
                  function entry requesttosend: integer;
                  begin if length = limit then delay(sender);
                        requesttosend:= base + tail
                  end;
```

```
                function entry requesttoreceive: integer;
                begin if length = 0 then delay(receiver);
                      requesttoreceive:= base + head
                end;
                procedure entry sendingcompleted;
                begin tail:= (tail+1) mod limit;
                      length:= length + 1; continue(receiver)
                end;
                procedure entry receivingcompleted;
                begin head:= (head+1) mod limit;
                      length:= length - 1; continue(sender)
                end;
                begin head:= 0; tail:= 0; length:= 0 end;
    diskbuffer = class (dbs: diskbufferstate; da: diskaccess);
                procedure entry send(block: page);
                var track: integer;
                begin track:= dbs.requesttosend;
                      da.write(track,block);
                      dbs.sendingcompleted
                end;
                procedure entry receive(var block: page);
                var track: integer;
                begin track:= dbs.requesttoreceive;
                      da.read(track,block);
                      dbs.receivingcompleted
                end;
                begin end;
     inputprocess = process (bufferstate: diskbufferstate;
                             disk: diskaccess);
                var block: page; db1: diskbuffer;
                begin init db1(bufferstate,disk);
                      cycle readcards(block);
                            db1.send(block)
                      end
                end;
     jobprocess = ...
     outputprocess = ...
var da: diskaccess; dbs1,dbs2: diskbufferstate;
```

```
    ip: inputprocess; jp: jobprocess; op: outputprocess;
begin init da, dbs1(0,100), dbs2(100,100),
           ip(dbs1,da), jp(dbs1,dbs2,da), op(dbs2,da)
end.
```

De access graph voor bovenstaand programma ziet er als volgt uit:

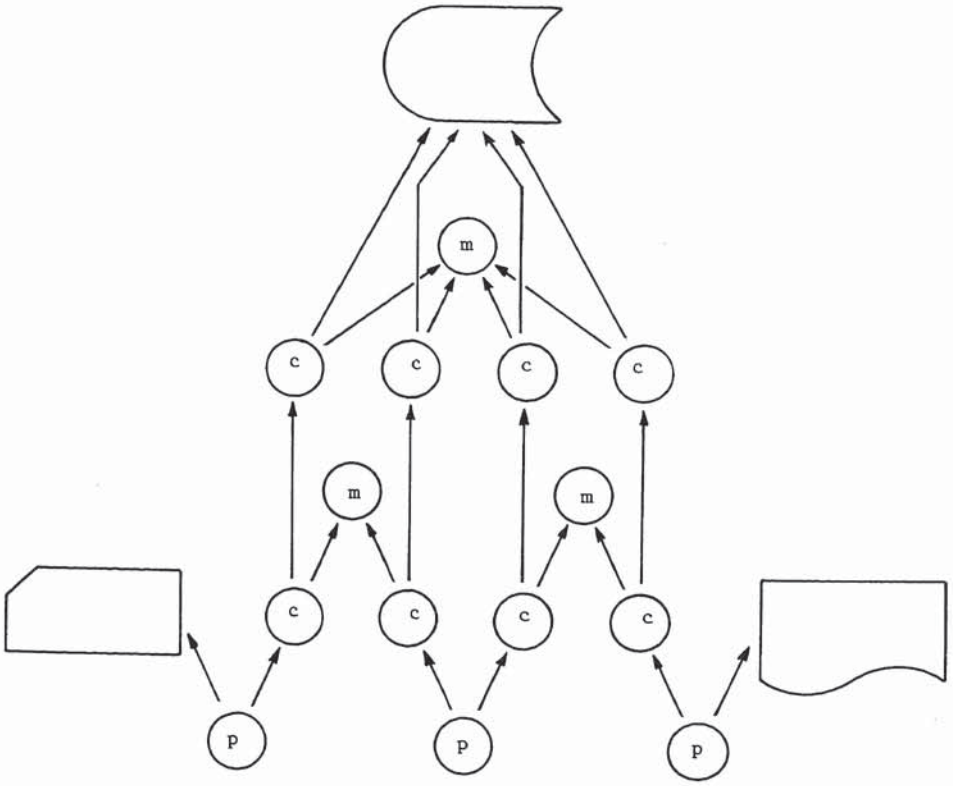


In deze access graph komt echter in het geheel niet tot uitdrukking dat de classes ieder voor zich lokaal zijn t.o.v. een van de drie processen, en dat alle processen een direkt accessrecht hebben tot de monitor da, ook al maakt geen van die processen van dat recht gebruik. In feite zouden de classes binnen hun moederprocessen getekend moeten zijn, maar daarmee wordt de figuur misschien wel juister maar niet duidelijker, en Brinch Hansen doet dat dan ook niet.

Naar mijn smaak wordt de class in dit voorbeeld op een wijze gebruikt die niet geheel overeenkomt met het concept van lokale abstract data struc-

ture, waarvoor classes uiterst bruikbaar zijn. Merk op dat de classes db1, ..., db4 in het geheel geen lokaal permanent geheugen hebben, tenzij men de identifikatie van de met hun formele parameter "dbs" korresponderende monitor hiertoe wil rekenen. Hun funktie is hier het aanreiken van een aanroepritueel voor de monitoren dbs1 of dbs2, en da, in de vorm van de procedures "send" en "receive", zonder dat dat ritueel echt wordt afgedwongen, aangezien de monitoren ook aan de processen zelf bekend zijn. Bij gebrek aan beter bevordert deze toepassing van classes echter toch de strukturering van het programma.

Tot slot van deze sectie merken we op dat, zolang het feitelijke access tot de schijven door de monitor da beheerst wordt, de scheduling daarvan aan de implementatie van Concurrent Pascal wordt overgelaten. Het is echter mogelijk om deze scheduling zelf ter hand te nemen, door de monitor da te vervangen door een monitor en vier classes (de laatste lokaal t.o.v. de classes db1, ..., db4). De monitor beheerst dan de voorrangsregels, en elk access tot de schijven verloopt dan volgens een vast, in het class type beschreven ritueel, bestaande uit aanmelding bij de monitor, data transport en afmelding bij de monitor, analoog aan het eerder beschreven ritueel in het class type "diskbuffer". De bijbehorende access graph ziet er als volgt uit:


m
c
c
c
c
m
m
c
c
c
c
p
p
p

6. BESLUIT

In de voorafgaande schets van Concurrent Pascal zijn een aantal nieuwe, dan wel niet algemeen gangbare taalconcepten aan de orde geweest:

1. scope rules en een parameter-mechanisme voor accessrechten, die hierarchische betrekkingen tussen system components afdwingen,
2. abstract data structures,
3. het onderscheid tussen permanent data en temporary data, en dat tussen private data en shared data,
4. het onderscheid tussen long term scheduling (voor process synchronization) en short term scheduling (mutual exclusion).

Het is een grote verdienste van Brinch Hansen, deze concepten in een min of meer logisch geheel verenigd te hebben tot een implementeerbare taal. Hierdoor behoeven dergelijke concepten niet tot gedachten-experimenten beperkt te blijven, maar kunnen ze hun waarde in, en voor de praktijk bewijzen. Verder blijkt de taal, door zijn abstractie, inderdaad bij het ontwerpen van systemen als denkdiscipline van nut te zijn.

Aan de andere kant bevat de taal allerlei ad-hoc oplossingen, b.v. voor de interfaces met randapparatuur, of het universal type (BRINCH HANSEN [7]), terwijl het marginale onderscheid tussen monitors en classes, en de ingewikkelde scope-regels voor classes op de noodzaak van nadere bezinning wijzen.

Al met al een interessante taal.

LITERATUUR

[1] BRINCH HANSEN, P., *A programming methodology for operating system design*, in J.L. Rosenfeld (ed.), Information processing 1974, North-Holland Publ. Co., Amsterdam, 1974.

[2] BRINCH HANSEN, P., *The programming language Concurrent Pascal*, I.E.E.E. Transactions on Software Engineering 1 (1975) 199-207.

[3] VERVOORT, W.A., *Concurrent Pascal and the design of operating systems*, te verschijnen in Informatie.

[4] WIRTH, N., *The programming language Pascal (revised report)*, E.T.H., Zurich, 1972.

[5] BRINCH HANSEN, P., *Operating system principles*, Prentice-Hall, 1973.

[6] BIRTWISTLE, G.M., O.J. DAHL, B. MYRHAUG & K. NYGAARD, *Simula begin*, Studentenlitteratur, Sweden, 1973.

[7] BRINCH HANSEN, P., *Universal types in Concurrent Pascal*, Information processing letters 3 (1975) 165-166.