RA

stichting
mathematisch
centrum

REKENAFDELING CR 21/70 DECEMBER

R.P. VAN DE RIET

INLEIDING IN DE INFORMATICA

Syllabus bij het Oriënterend Colloquium Informatica, 1970 - 1971

RA

2e boerhaavestraat 49 amsterdam

BIBLIOTHEEK MATHEMATISCH CENTRUM
AMSTERDAM

Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O), by the Municipality of Amsterdam, by the University of Amsterdam, by the Free University at Amsterdam, and by industries.

INHOUD

blz.

1. Inleiding 1
   1.1 Globale inhoud van deze cursus 1
   1.2 Welke problemen 1
   1.3 Een kort historisch overzicht 2

2. Algorithmen en de beschrijving ervan 5

3. De werking van een computer 14
   3.1 De drie organen van een computer 15
   3.2 De VERA computer 16
   3.3 Reflecties naar aanleiding van de VERA computer en een introductie van de MIX computer 21
   3.4 Over het ALGOL 60 programma voor de MIX computer 30

4. De werking van een assembler geschreven in ALGOL 60 70
   4.1 Opmerking vooraf 70
   4.2 De VERAL assembler 70
   4.3 Opmerkingen naar aanleiding van de VERAL assembler 79

Literatuur 89

Errata 92

Syllabus bij Oriënterend Colloquium Informatica *)

Inleiding in de Informatica
(programmeren van rekenautomaten)

R.P. van de Riet

1. Inleiding

1.1. Globale inhoud van deze cursus

Rekenautomaten, ook wel rekenmachines of computers genoemd, worden gebruikt om sommige problemen op te lossen.
In deze cursus zullen we kennis maken met
a) de grondprincipes van rekenautomaten,
b) hoe ze worden gebruikt,
c) welke problemen opgelost kunnen worden,
d) wat de oplossingsmethoden zijn.

In een parallelle cursus wordt de programmeertaal ALGOL 60 behandeld. De cursussen samen vormen een "Inleiding in de Informatica".

1.2. Welke problemen

Het gebruik van rekenautomaten kan min of meer verdeeld worden in het administratieve gebruik en het wetenschappelijke gebruik, waarbij niet gezegd is dat het administratieve gebruik nooit wetenschappelijk zou zijn.
Bij het administratieve gebruik, wat zeker 90% van het totale computergebruik omvat, zijn voorbeelden van gebruik meteen aan te wijzen: Girodienst voor het automatisch betalingsverkeer, personeelsadministratie voor de automatische uitbetaling, klantenbestand, voorraadbeheer, vliegtuigstoel reserveringssysteem.
Bij het wetenschappelijke gebruik speelt de wiskunde vaak een belangrijke rol. Allereerst zijn de toepassingen te noemen van de numerieke wiskunde (voorbeelden: inverteren van matrices, berekenen van eigenwaarden van matrices, berekenen van integralen, het oplossen van (partiële) differen-

---

*) *Rapport CR 21, afl. 1*

tiaalvergelijkingen). Vervolgens moeten de toepassingen van de statistiek genoemd worden en die van het nog jonge vak besliskunde. Het vak besliskunde houdt zich bezig met allerlei bedrijfseconomische problemen waarvoor wiskundige modellen kunnen worden opgesteld en uitgerekend m.b.v. de computer.
Enkele voorbeelden van gebieden waarin de wiskunde wel een belangrijke maar geen overheersende rol speelt zijn: "artificial intelligence", "computer aided instruction" en taalonderzoek.

Een globale indeling van administratief gebruik en wetenschappelijk gebruik kan ook als volgt worden gegeven:
Bij administratief gebruik wordt zeer weinig rekenwerk verricht met zeer veel gegevens; de snelheid van de rekenmachine wordt hier bepaald door de snelheid van de input-output (invoer-uitvoer) organen.
Bij wetenschappelijk gebruik wordt zeer veel rekenwerk verricht met betrekkelijk weinig gegevens; de snelheid van de rekenmachine wordt hier bepaald door de snelheid van het rekenorgaan.

Het zal geen verbazing wekken dat computerfabrikanten zowel administratief gerichte machines als wetenschappelijk gerichte machines op de markt brengen. Omdat grote bedrijven hun computers zowel gebruiken voor hun administraties als voor het berekenen van o.a. moeilijke besliskundige problemen, is er wat betreft de zeer grote computers geen twee-deling meer te maken.

1.3. Een kort historisch overzicht

Een rekenapparaat, de "abacus", in de vorm van een voorloper van ons telraam was (waarschijnlijk) reeds bekend bij de oude Egyptenaren.
Nog heden ten dage wordt het Japanse telraam, de "soroban", in het dagelijkse leven van Azië gebruikt.
In 1614 beschrijft de astronoom John Napier zijn "rekenstokken" met behulp waarvan vermenigvuldigingen gemakkelijk zijn uit te voeren.
In 1642 construeert de 18-jarige Blaise Pascal een optelmachine; hij doet dit om zijn vader te helpen die belastingambtenaar was.
In 1694 construeert de 25-jarige Gottfried Wilhelm von Leibniz een machine voor de vier arithmetische bewerkingen: optellen, aftrekken, vermenigvuldigen

en delen. Deze machine werd pas in 1820 door Thomas de Colmar voor commerciële toepassingen geproduceerd.

Voorlopers van de moderne rekenautomaat zijn de "Differentie machine" (1822) en de "Analytische machine" (kwam slechts in concept gereed) van Charles P. Babbage (1792-1871). Hij was van 1828 tot 1839, zonder één college te geven, professor te Cambridge.

Bang, dat de statistische bewerking van het bevolkingsonderzoek van 1880 niet in 1890 gereed zou zijn, construeerde Hermann Hollerith van het "U.S. Bureau of the Census", de eerste op ponskaarten gebaseerde "Hollerith" machine voor o.a. sorteren, classificeren en tabuleren van informatie verponst in kaarten .
Uiteindelijk is de Hollerith firma in de ook nu nog bestaande International Business Machines firma overgegaan.

In 1944 werd de eerste electrische computer, met hulp van IBM, in Harvard (USA) onder leiding van prof. Howard Aiken geconstrueerd; het werd de "Automatic Sequence-controlled Calculator" genoemd, ook wel de "Harvard Mark I". Het was mogelijk om een willekeurige volgorde van de vier arithmetische bewerkingen bij herhaling uit te laten voeren, bovendien konden tabellen automatisch worden geraadpleegd.

De eerste electronische computer werd in 1946 onder leiding van prof. Eckart en prof. Mauchly van de Universiteit van Pennsylvanië geconstrueerd. De naam was: Electronic Numerical Integrator and Calculator (ENIAC).
In Europa kwam de eerste computer, de EDSAC (Electronic Delay Storage Automatic Computer), in Cambridge onder leiding van prof. M.V. Wilkes tot stand en wel in 1949.
Onder leiding van prof. A. van Wijngaarden kwam in 1953 de automatische electronische rekenmachine ARRA (Automatische Relais Rekenmachine Amsterdam) op het Mathematisch Centrum als een van de eersten op het vaste land van Europa te Amsterdam gereed. Na de officiële ingebruikstelling van de ARRA door een hoogwaardigheidsbekleder zijn de relais er uitgehaald en vervangen door electronische circuits. De naam bleef ongewijzigd.

Voor een uitgebreid historisch overzicht van de ontwikkeling van de computers zij verwezen naar:

Saul Rosen, Electronic Computers: A Historical Survey
Computing Surveys vol. 1, no. 1, March 1969, p. 7-36.

2. Algorithmen en de beschrijving er van

De 9-de eeuwse Arabische wiskundige Abu Jafar Mohammed ibn Musa al-Khowarizmi voerde het woord "algorithme" in om een reeks arithmetische handelingen aan te geven welke als een kookrecept uitgevoerd dienden te worden. Een voorbeeld hiervan is de bekende algorithme van Euclides. Dit algorithme luidt als volgt:

<u>Algorithme van Euclides</u>:

> Zij gegeven twee positieve getallen m en n; het grootste positieve getal g dat zowel een deler is van m als van n wordt als volgt gevonden:
> stap 1: Deel m door n; noem de rest r.
> stap 2: Als r ongelijk is aan nul noem dan i.h.v. n:m en r:n en herhaal het proces te beginnen met stap 1.
> Als r gelijk is aan nul dan is g:n. (2.1)

Met opzet zijn de bewoordingen waarin dit algorithme is opgesteld vaag en onduidelijk gehouden. Wat bedoelen we met: "we noemen n:m" ?
In het geval van "we noemen Jan:Piet en Piet:Klaas" bedoelen we: als we i.h.v. de oude Jan tegenkomen dan zeggen we Piet en als we de oude Piet tegenkomen dan zeggen we Klaas. M.a.w. we plakken andere labels, andere namen, aan de personen die zich lieten noemen: Jan en Piet.
Terugkomende op "het getal m", merken we op dat, tijdens onze beschouwing, de naam "m" gehecht is aan een zeker getal.
Ditzelfde getal kan behalve "m" nog meerdere namen hebben; om de zaak ingewikkeld te maken merken we bovendien op dat de naam "m" ook aan meerdere getallen gegeven kan worden.
We concluderen dat bovenomschreven algorithme slecht omschreven is. Bij een wiskundige aanpak worden er net zoveel namen gecreëerd als er getallen voorkomen gedurende de uitwerking van de algorithme:

> $m_0 = m$, $m_1 = n$,
> $m_i = q_i \, m_{i+1} + m_{i+2}$, $i = 0, 1, 2, \ldots$
> als $m_k = 0$ en $m_{k-1} \neq 0$ dan is m_{k-1} de gezochte grootste gemene deler g. (2.2)

Afgezien van de vraag hoe de q_i's berekend moeten worden en afgezien van het feit dat de algorithme niet afbreekt (sommigen noemen dit proces daar-

om niet eens een algorithme), is hier nu toch in duidelijker termen beschreven wat de bedoeling is.

We zijn bezig met computers met eindige geheugens en eindige rekentijden zodat algorithmen die niet afbreken en de introductie van oneindig veel namen complicaties met zich mee brengt.
Het probleem van de oneindige hoeveelheid namen ontstaat door de wiskundige taal die gebruikt wordt; immers in (2.2) staat niet een proces beschreven maar er staan oneindig veel relaties tussen oneindig veel grootheden.
Bij een proces kunnen relaties tijdsgebonden zijn: he. ene moment geldt m = n, het andere moment geldt m = n+1; het ene moment is de naam "m" aan het getal 5 verbonden (op de vraag wat we met het getal 5 bedoelen zullen we nu niet ingaan) het andere moment aan het getal 7.
Teneinde aan te geven dat de naam "m" aan het getal 7 verbonden wordt, voeren we het symbool ":=" in, en we schrijven: "m:= 7".
Nadat de actie omschreven in "m:= 7" is uitgevoerd, is de eventuele verbinding van "m" aan een ander getal (5) verbroken en is "m" nu aan het getal 7 verbonden.
In een sequentie van acties worden de acties onderling gescheiden door het symbool ";".
Om het begin en het einde van een sequentie van acties te onderscheiden worden de symbolen <u>begin</u> en <u>end</u> respectievelijk voor en na de sequentie geplaatst.

<u>Voorbeeld</u>: <u>begin</u> m:= 5; n:= 7; m:= 7 <u>end</u> (2.3)

heeft tot effect dat de naam "m" met het getal 5 verbonden wordt, dat vervolgens de naam "n" met het getal 7 verbonden wordt en dat tenslotte de naam "m" met het getal 7 verbonden wordt. Het netto effect van de samengestelde acties is dus gelijk aan:

<u>begin</u> n:= 7; m:= 7 <u>end</u> (2.4)

Uiteraard betreft dit slechts het netto effect, immers in het tweede geval heeft de relatie m:= 5 nooit bestaan.

Kunnen we nu de formulering van de algorithme van Euclides verbeteren? Het begin luidt nu voor de ggd van 28 en 32:

"<u>begin</u> m:= 28; n:= 32;" (2.5)

De beschrijving van stap 1 is niet evident. We voeren daarom een speciale functie in: "rest (m,n)" welke precies doet wat we willen namelijk de rest uitrekenen bij deling van m door n. We krijgen:

"stap 1: r:= rest (m,n)"

Stilzwijgend veronderstellen we dat de betekenis van het symbool ":=" wordt uitgebreid voor de gevallen dat er rechts niet een getal in cijfers staat maar een uitdrukking (expressie) welke bij evaluatie een getal oplevert. Zo zal "a:= 5+7" een actie bedoelen met als effect dat 5 en 7 worden opgeteld terwijl het resultaat ervan aan de naam "a" wordt toegevoegd.
Tenslotte stap 2, welke we meteen omschrijven:

"stap 2: *if* r ≠ 0 *then*
begin m:= n; n:= r; *goto* stap 1 *end* (2.6)
else g:= n"

We merken in de eerste plaats op dat de betekenis van "m:= n" is: de naam "m" wordt verbonden met het getal met de naam "n". Opgemerkt zij dat dit getal na de actie "m:= n" kennelijk aan twee namen is toegewezen.
N.B. Impliciet wordt aangenomen dat er een getal bestaat verbonden aan de naam "n". Een belangrijk percentage van programmeerfouten ontstaat door een niet vervuld zijn van deze aanname.
We kenden één soort actie namelijk de toekenning van een naam aan een getal, ook wel *assignatie* genoemd, nu zien we een aantal andere acties in werking:

2. "*goto* stap 1" waarvan de actie is: neem als volgende uit te voeren actie, de actie met de label "stap 1" inplaats van de in sequentiële volgorde volgende actie; deze actie heet een *sprong*.
3. "*if* r ≠ 0 *then* de ene actie
else de andere actie".
Afhankelijk van de waarden van "r" (precieser: het getal waaraan "r" (hopelijk) is toegevoegd) worden verschillende acties uitgevoerd.
De algemene gedaante van deze actie heet een *conditionele statement* en heeft de vorm:
"*if* conditie *then* actie 1 *else* actie 2"

We zien meteen het nut van de, als openingshaak en sluitingshaak gebruikte, symbolen *begin* en *end*: hiermee kunnen we een serie acties afsluiten en ze als het ware samen als één nieuwe actie beschouwen.

Zonder begin end is het niet duidelijk wat we bedoelen met:

"if r ≠ 0 then m:= n; n:= r else p:= q; q:= s"

Voor de sprong acties is het nodig namen of labels te hechten aan acties. Elke naam mag echter maar bij één actie horen, zoals ook het geval is met de namen gehecht aan getallen. Er is echter een belangrijk verschil: namen kunnen tijdens het proces aan verschillende getallen gehecht worden (zoals m en n in (2.6)), namen (labels) van acties kunnen echter maar aan één actie toegevoegd worden. De eerste toevoeging is dynamisch, de tweede is statisch.

In het vervolg zullen we de representatie van de acties statements noemen; dus een statement is de beschrijving van de actie.
Een serie statements afgesloten door begin en end wordt samengesteld statement of compound statement genoemd.

We kunnen nu de compound statement opschrijven voor de algorithme van Euclides voor de berekening van de ggd g van de getallen 28 en 32:

```
"begin m:= 28; n:= 32;
        stap 1: r:= rest (m,n);
        stap 2: if r ≠ 0 then
                     begin m:= n; n:= r; goto stap 1 end
                else g:= n
end"                                                    (2.7)
```

Blijft nog het probleem hoe "rest (m,n)" berekend moet worden. Ook hiervoor kan een algorithme worden opgesteld:

```
begin r:= m;
      Herhaal: if r < n then goto Klaar else;
               r:= r-n; goto Herhaal;
      Klaar:
end                                                     (2.8)
```

We zien dat het nodig is acties in te voeren welke niets doen. Deze worden voorgesteld (gerepresenteerd) door een dummy statement welke uit geen enkel symbool bestaat.

Afspraak: De constructie: "if conditie then statement else;"
mag vervangen worden door: "if conditie then statement;".

De volledige algorithme verkrijgt men door in (2.7) de statement "r:= rest (m,n)" te vervangen door de compound statement (2.8).

Eigenlijk is het jammer dat "r:= rest (m,n)" vervangen moet worden. Immers, we zijn vertrouwd met bijvoorbeeld de notatie: "sin (x)" of, algemener, "f(a,b,c)". Met deze notatie bedoelen we: "voer nu het proces uit dat de sinus voor x berekent" of, algemener, "dat de functie f toepast op de parameters a, b en c".
Zo bedoelen we met "rest (m,n)" : "bereken de rest verkregen na toepassing van (2.8) op de parameters m en n".
Het getal dat berekend wordt door (2.8), en dat aan r wordt toegevoegd (geassigneerd), is de waarde van de rest van m bij deling door n. Dit getal kennen we toe aan de naam "rest" door middel van een statement van de vorm:

"rest:= r" in de beschrijving van de algorithme.

Nadat deze toekenning heeft plaats gevonden kunnen we gebruik maken van het getal dat aan "rest" was toegevoegd door een statement van de vorm:

"r:= rest (m,n)".

Vanzelfsprekend is er een notatie nodig waaruit blijkt welk algorithme bij welke naam hoort. Dit is de zogenaamde <u>procedure notatie</u> welke voor het "rest" voorbeeld als volgt loopt:

```
"integer procedure rest (m,n);
 begin r:= m;
       Herhaal: if r < n then goto Klaar;
       r:= r-n; goto Herhaal;
       Klaar: rest:= r
end"                                          (2.9)
```

In de laatste statement wordt aan "rest" het getal, dat toegevoegd is aan r geassigneerd.
De statement "a:= rest (10,3)" zal tot effect hebben dat het getal 1 aan de naam "a" wordt geassigneerd.
Op tweeërlei wijze is het nu mogelijk naamsverwarring te krijgen. Allereerst de naam "rest". Tijdens uitvoering van de algorithme wordt aan deze naam een getal (en wel een integer, vandaar "<u>integer</u>") toegevoegd.

In de beschrijving van de algorithme wordt aan dezelfde naam "rest" een compound statement, de algorithme, toegevoegd. Dit is op zich niet tegenstrijdig maar is wel een bron van verwarring (het nog verzwegen feit dat (2.9) in overeenstemming met ALGOL 60 specificaties is opgesteld is hiervan de oorzaak; de ALGOL 68 terminologie is in dezen beter). Ten tweede is verwarring wat betreft de naam "r" mogelijk omdat "r" zowel in (2.7) als in (2.9) wordt gebruikt. De "r" in (2.9) wordt slechts als hulpgrootheid binnen de compound statement gebruikt; we willen daarom een mechanisme hebben waarmee we een naam kunnen creeëren en waarmee we een naam kunnen laten sterfen.

Dit mechanisme houdt in de "declaratie van r ten opzichte van een blok" en wordt al volgt gebruikt:

De openings begin wordt vervangen door:

"begin integer r;"

Het is alsof er een actie aan het repertoire is toegevoegd:

"creëer een naam die we "r" noemen en die een eigen leven leidt tot dat de laatste statement van de algorithme afgesloten met end is uitgevoerd."

De beschrijving van deze actie noemen we niet statement maar declaratie.

Er zijn meerdere declaraties mogelijk; vandaar de type aanduiding: integer.

Een ander type declaratie is de procedure declaratie (2.9).

Een rij statements voorafgegaan door declaraties, gescheiden door het symbool ";" tussen de symbolen begin en end heet een blok.

Namen, welke gedeclareerd zijn binnen een blok, verliezen hun betekenis buiten dit blok (dus bij overschrijding van de bij het blok behorende end).

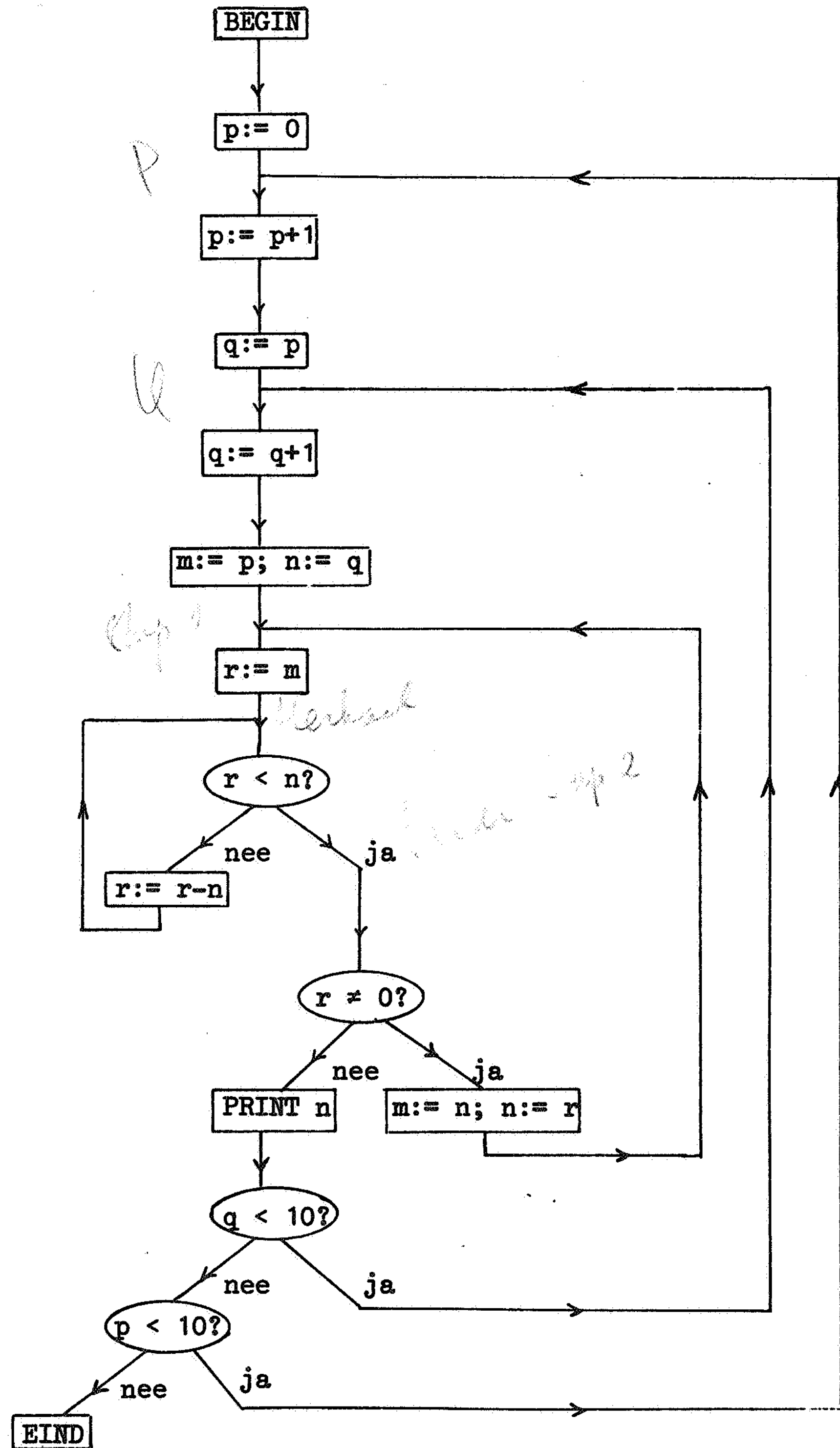
We kunnen nu een volgend blok maken met daarin enkele malen herhaald de berekening en het printen van grootste gemene deler:

```
begin integer p, q;                                                   (2.10)
      integer procedure ggd (m,n);
      begin integer m1, n1, r; m1:= m; n1:= n;
            Stap 1: r:= rest(m1,n1);
            Stap 2: if r ≠ 0 then begin m1:= n1; n1:= r; goto Stap 1 end
                    else ggd:= n1
      end;
      integer procedure rest(m,n);
      begin integer r; r:= m; Herhaal: if r < n then goto Klaar;
            r:= r-n; goto Herhaal; Klaar: rest:= r
      end;
      p:= 0;
      P: p:= p+1; q:= p; Q: q:= q+1; PRINT(ggd(p,q)); if q < 10 then goto Q;
      if p < 10 then goto P
end
```

Het blok dat de grootste is en de gehele algorithme beschrijft wordt <u>programma</u> genoemd.

Aan de hand van de algorithme van Euclides is een taal ontwikkeld met behulp waarvan dit algorithme beschreven is. Deze taal is ALGOL 60 "in een notedop". Met behulp van deze taal zullen we in het volgende hoofdstuk een proces beschrijven dat een afspiegeling is van de werking van een rekenautomaat. Een groot aantal andere talen staat ter beschikking om algorithmen in te beschrijven; wij kiezen ALGOL 60 omdat het een prettige taal is terwijl een algorithme, er in opgesteld, op de rekenapparatuur hier ter plaatse kan worden uitgevoerd.

Later zullen wij aandacht besteden aan talen als FORTRAN en assembleertalen, nu zullen we de taal van de "blokschema's", "stroomdiagrammen", of ook wel "flow-charts" genoemd, demonstreren aan de hand van hetzelfde voorbeeld:


BEGIN
p:= 0
p:= p+1
q:= p
q:= q+1
m:= p; n:= q
r:= m
r < n?
nee
ja
r:= r-n
r ≠ 0?
nee
ja
PRINT n
m:= n; n:= r
q < 10?
nee
ja
p < 10?
nee
ja
EIND

Het is duidelijk dat deze taal zich niet leent om door de rekenautomaat gelezen te worden. Voor kleine problemen is deze taal erg duidelijk en overzichtelijk. Bij grotere problemen kan men vaak "vanwege de bomen het bos niet meer zien".

Flow charts worden in het bijzonder gebruikt als voorstudie bij het ontwerp van een algorithme in zogenaamde assembleertalen.

3. De werking van een computer

Aan de hand van formele beschrijvingen van twee computers:

de Verschrikkelijk Eenvoudige Reken Automaat (VERA) en

de MIX computer beschreven in:

Donald E, Knuth, Fundamental Algorithmes

The art of computer programming vol. 1

Addison en Wesley 1968, prijs ƒ 91,--.

Beide computers bestaan slechts in hypothetische zin.
Het mechanisme om algorithmen te beschrijven, zoals dat in het vorige hoofdstuk werd ingevoerd, zal nu gebruikt worden om de algorithmen, volgens welke rekenautomaten werken, te beschrijven.
We zullen voor deze algorithmen twee programma's in de taal ALGOL 60 opstellen. Eventuele uitbreidingen van de in het vorige hoofdstuk behandelde begrippen zullen op het moment dat ze nodig zijn worden ingevoerd.
De twee programma's kunnen op de EL X8 computer van het Mathematisch Centrum worden uitgevoerd aldus de hypothetische computers tot echte computers promoverend.

Teneinde meteen te kunnen beginnen met het programma en toch de mogelijkheid van becommentariëring open te houden voeren we de volgende comment conventies in:

1. "comment gevolgd door elke willekeurige rij symbolen uitgezonderd de puntkomma;" wordt als commentaar opgevat en heeft geen actie tot gevolg,
2. "end gevolgd door elke willekeurige rij symbolen uitgezonderd de puntkomma, het end symbool of het else symbool;", waarbij de ";" vervangen mag worden door "end" of "else", heeft hetzelfde effect als "end;", "end end" of "end else".

Mocht het nodig zijn om in het commentaar een puntkomma te willen schrijven dan zullen we daartoe het symbool ; gebruiken.

Met deze comment conventie en de mogelijkheid om meer dan één letter te gebruiken voor een naam zijn we in staat om zichzelf verklarende programma's te schrijven.

3.1. De drie organen van een computer

Functioneel gezien zijn de drie belangrijkste organen waaruit een computer is opgebouwd de volgende:

1. De I/O organen welke het Input- en Output verkeer van informatie tussen de buitenwereld en de computer regelen door middel van (Input:) kaartlezers, bandlezers, (Output:) kaartponsers, bandponsers, snelle regeldrukkers, plotters, (Input en Output:)typemachines, magnetische bandeenheden, magnetische schijveneenheden en magnetische trommels.
2. Het Centrale Reken Orgaan (CRO) ook wel het besturingsorgaan of centrale processor genoemd. Dit orgaan bestuurt de gehele computer, haalt instructies uit het geheugen en voert deze uit, haalt informatie uit het geheugen en brengt informatie in het geheugen.
3. Het geheugen bestaande uit adresseerbare cellen. Elk van deze geheugencellen kan een zekere hoeveelheid informatie bevatten in de vorm van een representatie van een getal, meestal in binaire vorm (waarover later meer). Het hangt van het CRO af of de informatie als instructie, als getal of op nog andere wijze wordt geïnterpreteerd.
   Het adjectief "adresseerbaar" betekent dat elke cel een adres heeft. Dit adres is een geheel getal a met $0 \leq a \leq b$, met b een of andere, o.a. door de prijs van de computer bepaalde bovengrens.
   Voor onze computer zullen wij aannemen dat $b = 3999$.

Behalve genoemde organen zijn er organen die de informatie transporten verzorgen (kanalen),bedieningspanelen en nog vele andere. Deze zullen niet in de volgende programma's beschreven worden.

```
begin   comment VERA computer, RPR 180870, opdrachtnr. 2133

        3.2. De VERA computer

        In deze paragraaf beschrijven we de Verschrikkelijk
        Eenvoudige Reken Automaat.
        Allereerst introduceren we het geheugen dat, in onze
        terminologie bestaat uit een rij van 4000 namen
        waaraan gehele getallen geassigneerd kunnen worden;

        integer array GEH[0:3999];

        comment De i-de naam van deze rij noteren we met GEH[i].
        Het CRO van de VERA beschrijven we met de volgende
        procedure declaratie:;

        procedure CRO;
        begin   comment
                Het CRO beschikt over een aantal, zogenaamde
                registers. Dit zijn zeer snel toegankelijke
                geheugencellen en worden gebruikt om: het
                adres van de volgende uit te voeren instructie
                te onthouden(instr teller), de uit het geheugen gehaalde
                en uit te voeren instructie te onthouden(instr reg),
                het resultaat van rekenkundige bewerkingen te onthouden
                (de accumulator A).
                Het CRO beschikt verder over een procedure BASIS CYCLUS
                en een procedure EXECUTEER INSTRUCTIE.;

                integer instr teller, instr reg, A;

                procedure BASIS CYCLUS;
                begin   BEGIN:  instr reg:= GEH[instr teller];
                                instr teller:= instr teller + 1;
                                EXECUTEER INSTRUCTIE;
                                goto BEGIN
                end BASIS CYCLUS;

                procedure EXECUTEER INSTRUCTIE;
                begin   comment
                        Elke instructie i kan geschreven worden als
                            i = a × 10 + c,
                        met 0 < a < 3999 en 0 < c < 9.
                        De waarde van a geeft het adres aan van
                        de geheugencel G[a] waarop de instructie
                        betrekking heeft. De waarde van c geeft de soort
                        instructie aan volgens onderstaande tabel:

                        c       werking                 "mnemonics"
                        0       A:= G[a]                LDA  a
```

```
1       G[a]:= A                    STA  a
2       A:= A + G[a]                ADD  a
3       A:= A - G[a]                SUB  a
4       if A < 0 then goto a        JAN  a
5       goto a                      JMP  a
6       PRINT(G[a])                 PRI  a
7       goto EIND EXECUTIE          HLT
```

De preciese beschrijving van de instructies
volgt nu:;

```
    integer a,c;
    a:= instr reg : 10;
    comment De operatie : betekent delen met
    na afloop een afronding in de richting
    van 0 naar
    de dichtsbijgelegen integer.
    Voorbeeld: 7 : 10 = 0, 12 : 10 = 1,
    (-7) : 10 = 0, (-12) : 10 = -1.;
    c:= instr reg - a × 10;
    if c = 0 then A:= GEH[a] else
    if c = 1 then GEH[a]:= A else
    if c = 2 then A:= A + GEH[a] else
    if c = 3 then A:= A - GEH[a] else
    if c = 4 then
    begin  if A < 0 then instr teller:= a
    end else
    if c = 5 then instr teller:= a else
    if c = 6 then PRINT(GEH[a]) else
    if c = 7 then goto EIND EXECUTIE
end EXECUTEER INSTRUCTIE;
```

comment Alvorens het CRO gestart kan worden moet
het VERA programma in het geheugen staan en het
start adres, d.i. het adres van de eerste instructie
die uitgevoerd moet worden, bekend zijn.
We nemen aan dat zowel het VERA programma als
dit startadres volgens onderstaande specificatie
op een input band staan:

```
  <input band >::= <VERA machinecode programma>
                 -1 <start adres>
  <VERA machinecode programma>::= <VERA regel>|
                 <VERA machine code programma> <VERA regel>
  <VERA regel>::= <decimaal adres> <decimale instructie>;
                  <commentaar> <Terug Wagen Nieuwe Regel-
                   symbool>
  <decimaal adres>::= <geheel getal>
  <decimale instructie>::= + <geheel getal>| - <geheel getal>
  <geheel getal>::= <digit >|  <geheel getal> <digit>
```

```
        <digit>::= 0|1|2|3|4|5|6|7|8|9
        <start adres>::= + <geheel getal>
      Opmerking: <input band> is evenals de andere tussen
      de tekens "<" en ">" geplaatste objecten een zogenaamde
      syntactische variabele. Het symbool "::=" betekent:
      "is per definitie:" en het symbool "|" betekent "of".
      De notatie, die in de ALGOL 60 cursus een betere
      behandeling krijgt, is de zogenaamde Backus-Naur vorm
      naar twee van de ALGOL 60 ontwerpers: Backus en Naur.;

      procedure START CRO;
      begin   integer semicolon, twnr, adres, instr;
              semicolon:= 91; twnr:= 119; comment Dit zijn de
              interne, volgens MR 81 gespecificeerde,
              representaties van de aangegeven symbolen.;
      LEES:   adres:= READ; instr:= READ; comment De standaard
              procedure READ leest eerst een getal en het
              eerste niet bij dit getal behorende symbool
              van de input band.;
              if adres = -1 then
              begin instr teller:= instr; BASIS CYCLUS end;
              GEH[adres]:= instr;
              comment De standaard procedure RESYM
              leest een symbool van de band en krijgt als
              waarde de volgens MR 81 gespecificeerde
              interne representatie van het gelezen symbool.;
              SKIP: if RESYM ≠ twnr then goto SKIP;
              goto LEES
      end START CRO;

      comment Het CRO behoeft nu alleen nog maar gestart
      te worden door:;
      START CRO;
      comment De volgende label staat er ten behoeve
              van de HLT instructie:;
      EIND EXECUTIE:
   end De VERA computer wordt gestart door:;
   CRO; comment dit was de VERA computer;
end
```

We prepareren nu een VERA programma. Dit beschrijft de in het vorige hoofdstuk ontworpen algorithme. Het stroomdiagram is een goed uitgangspunt om het VERA programma te construeren. Met behulp van de "mnemonics" uit de tabel van de instructies worden eerst de instructies in begrijpelijke taal opgezet terwijl ze later met de hand worden vertaald naar de decimale getalvorm. We beginnen met een handje variabelen en constanten:

```
0     +000;
1     +001;
2     +010;
3     +000;  p
4     +000;  q
5     +000;  m
6     +000;  n
7     +000;  r
8     +010;  LDA 1   )
9     +032;  ADD 3   )
10    +031;  STA 3   ) p:= p+1
11    +041;  STA 4     q:= p
12    +010;  LDA 1   )
13    +042;  ADD 4   )
14    +041;  STA 4   ) q:= q+1
15    +061;  STA 6     n:= q
16    +030;  LDA 3   )
17    +051;  STA 5   ) m:= p
18    +050;  LDA 5   )
19    +071;  STA 7   ) r:= m
20    +063;  SUB 6     A:= r-n
21    +244;  JAN 24    if r < n then goto 24
22    +071;  STA 7     r:= r-n
23    +205;  JMP 20    goto 20 (N.B. A = r)
24    +070;  LDA 7    A:= r
25    +394;  JAN 39    if r < 0 then goto 39
26    +000;  LDA 0   )
27    +073;  SUB 7   ) A:= -r
28    +394;  JAN 39    if r > 0 then goto 39
29    +036;  PRI 3
30    +046;  PRI 4
31    +066;  PRI 6
32    +040;  LDA 4   )
33    +023;  SUB 2   ) A:= q-10
34    +124;  JAN 12    if q < 10 then goto 12
35    +030;  LDA 3
36    +023;  SUB 2
37    +084;  JAN 8     if p < 10 then goto 8
38    +007;  HLT       einde programma
39    +060;  LDA 6   )
40    +051;  STA 5   ) m:= n
41    +070;  LDA 7   )
42    +061;  STA 6   ) n:= r
43    +185;  JMP 18
-1    +008;
```

Bij uitvoering kregen we de volgende printout:

```
+ 1     + 2     + 1     + 1     + 3     + 1
+ 1     + 4     + 1     + 1     + 5     + 1
+ 1     + 6     + 1     + 1     + 7     + 1
+ 1     + 8     + 1     + 1     + 9     + 1
+ 1     +10     + 1     + 2     + 3     + 1
+ 2     + 4     + 2     + 2     + 5     + 1
+ 2     + 6     + 2     + 2     + 7     + 1
+ 2     + 8     + 2     + 2     + 9     + 1
+ 2     +10     + 2     + 3     + 4     + 1
+ 3     + 5     + 1     + 3     + 6     + 3
+ 3     + 7     + 1     + 3     + 8     + 1
+ 3     + 9     + 3     + 3     +10     + 1
+ 4     + 5     + 1     + 4     + 6     + 2
+ 4     + 7     + 1     + 4     + 8     + 4
+ 4     + 9     + 1     + 4     +10     + 2
+ 5     + 6     + 1     + 5     + 7     + 1
+ 5     + 8     + 1     + 5     + 9     + 1
+ 5     +10     + 5     + 6     + 7     + 1
+ 6     + 8     + 2     + 6     + 9     + 3
+ 6     +10     + 2     + 7     + 8     + 1
+ 7     + 9     + 1     + 7     +10     + 1
+ 8     + 9     + 1     + 8     +10     + 2
+ 9     +10     + 1     +10     +11     + 1
```

3.3. Reflecties naar aanleiding van de VERA computer en een introductie van de MIX computer

3.3.1. Diverse soorten talen

De algorithme van Euclides is aan het eind van de vorige paragraaf met behulp van twee talen opgesteld:

1. De machinetaal (of machinecode), welke uit een verzameling van decimale getallen bestaat, aan enkelen waarvan een betekenis kan worden toegekend.

   Het is niet plezierig om in machinetaal te moeten programmeren; een vergissing kan gemakkelijk worden gemaakt.

   Het eerste dat we daarom deden was in het commentaar gebruik maken van:

2. De "mnemotechnische" taal.

   Instructies, in deze taal beschreven, bestaan uit een "mnemonic" en (afgezien van HLT) uit een adres. Een "mnemonic" ontstaat door een aantal karakteristieke letters van de functie omschrijving van de betreffende instructie bij elkaar te zetten. Een "mnemonic" betekent zoiets als "geheugensteuntje". De uitspraak is: ni-'män-ik (althans volgens Webster).

   Ook deze taal had nadere explicatie nodig in de zin van het toevoegen van namen en het bij elkaar voegen van instructies die samen elementaire acties in de zin van ALGOL 60 uitvoeren zoals "p:= p+1".

Teneinde het programmeren op machinetaal-niveau gemakkelijk te maken wordt de computer eerst voorzien van een programma dat assembler heet en dat in staat is om:

1. "mnemonics" te lezen en te vervangen door hun getalcode,
2. in plaats van adressen, namen te lezen welke in adressen worden omgezet.

Het eerste stuk en het laatste stuk van de algorithme van Euclides krijgt dan de volgende vorm:

```
    nul:   0                         LDA q
    een:   1                         SUB tien
    tien:  10                        JAN VERHOOG q
    p:     0                         LDA p
    q:     0                         SUB tien
    m:     0                         JAN BEGIN
    n:     0                         HLT
    r:     0                         LDA n
  BEGIN:   LDA een                   STA m
           ADD p                     LDA r
           STA p                     STA n
           STA q              EIND:  JMP HEEL m door n
```

Deze taal wordt <u>assembleertaal</u> genoemd. De taak van de assembler is niet zo groot. Meestal wordt de programmatektst twee maal door de assembler gelezen. De eerste keer om alle namen op te pikken en in de zogenaamde naamlijst te zetten, de tweede keer om adressen aan de namen te hechten en de instructies te "bakken".

De vraag of de VERA computer uitgerust kan worden met een assembler moet ontkennend worden beantwoord. Door een constructiefout kan deze computer geen informatie van een input medium lezen en kan dus ook geen teksten in assembleertaal analyseren. De situatie met VERA is analoog aan die van de IBM 7094 opgesteld in de Universiteit van Californië te Berkeley welke in het geheel geen input en output organen meer heeft.

Programma's voor de berekeningen van getaltheoretische problemen worden door de wiskundige D.N. Lehmer met de hand bit voor bit (cijfer voor cijfer) in de machine gezet; de machine wordt in werking gesteld en de operateurs die bij een andere machine werkzaam zijn behoeven slechts te letten op het rood aangloeien van een contrôle-lampje in welk geval ze Lehmer moeten bellen. Soms duurt een programma een uur soms 18 dagen.

We kunnen zeggen dat elke instructie van een programma, in assembleertaal geschreven, correspondeert met precies één machinecode instructie.

Het is duidelijk dat dit voor een taal als ALGOL 60 niet het geval is. Het vertalen van een programma, in ALGOL 60 beschreven, is dan ook een tamelijk

moeilijke zaak. Deze vertalers (dus machine programma's) worden <u>compilers</u> genoemd.

3.3.2. Subroutines

Betroffen bovenstaande opmerkingen het uiterlijk aanzien van de VERA taal, de volgende betreffen de instructies zelf.
Heel oorspronkelijk was er een speciale procedure "rest" ontworpen voor het berekenen van de rest bij deling van m op n. Deze procedure is beschreven in de instructies met adressen 18, 19, 20, 21, 22 en 23.
Als we behalve het ggd programma ook nog een programma voor priemgetallen in de VERA hadden opgenomen, die ook gebruik maakt van rest, dan hadden we de 6 instructies moeten copiëren. Teneinde dit copiëren achterwege te kunnen laten maar wel van de reeds aanwezige instructies gebruik te kunnen maken zouden we de instructie JMP 18 in het priemgetallen programma kunnen opnemen. Bijvoorbeeld:

```
108    ...
109    +185;  JMP 18
110    ...
```

We moeten nu bedacht zijn op twee dingen: ten eerste maakt "rest" gebruik van de globale variabelen m en n, deze moeten dus correct gevuld zijn; ten tweede vragen we ons af wat er gebeurt als de laatste instructie: 23 is uitgevoerd. In plaats van de instructie op plaats 110 wordt de instructie op plaats 24, dus in het ggd programma uitgevoerd.
Graag zouden we na de laatste instructie van rest een instructie in de geest van:

"en spring nu weer terug naar waar je vandaan kwam" hebben.

De VERA computer geeft ons deze mogelijkheid niet omdat in het instructie pakket de mogelijkheid ontbreekt om te vragen naar de waarde van "instr teller".
Bij echte computers is deze mogelijkheid wel aanwezig. In de door Digital Equipment Corporation gefabriceerde PDP8 computer, bijvoorbeeld, bewerkstelligt de instructie "JMS adres" het volgende:

```
begin GEH[adres]:= instr teller * 10;
    instr teller:= adres + 1 end,
```

m.a.w. er wordt een sprong uitgevoerd naar het tweede adres van de procedure. Om weer terug te springen naar de plaats waar de sprong vandaan kwam moeten we springen naar het adres dat opgeborgen is in de eerste cel van de procedure. Daartoe moet er eerst een dergelijke instructie langs arithmetische weg gemaakt worden als volgt:

```
rest:      000
           LDA  m
           STA  r
TREKAF:    SUB  n
           JAN  UIT
           STA  r
           JMP  TREKAF
UIT:       LDA  rest
           ADD  jump
           STA  volgende
volgende:  000
jump:      JMP  0
```

De nu ontstane procedure heeft de vorm van een subroutine.
Voor ons geval is het wel tamelijk ingewikkeld; bovendien hebben we een overhead van 6 instructies op de oorspronkelijke 6.
Indien rest gebruikt wordt op meer dan drie plaatsen dan is de overhead echter al terug verdiend.
In gevallen als deze, waar het dubieus is of er nu een subroutine gemaakt moet worden of niet kan ook met vrucht van het begrip macro gebruik worden gemaakt. In dit geval zal de assembler, op door de programmeur aangegeven plaatsen in het programma, stukken vrijwel gelijkluidende tekst in het programma inlassen. Dit is een faciliteit die het schrijven van programma's vereenvoudigt. Het uiteindelijke, in de machine opgeborgen, programma is echter niet korter.

De reden, waarom de overhead zo groot is, is dat de VERA computer geen indirecte adressering kent. Dat wil zeggen de mogelijkheid om

GEH[a]

als adres te gebruiken. Dit is op de PDP8 wel het geval zodat voor die computer de instructies van "UIT" t/m "jump" vervangen kunnen worden door de ene instructie: JMP I rest, met als effect: instr teller:= GEH[rest] ÷ 10. (NB. het vermenigvuldigen en delen door 10 geldt niet voor de PDP8; voor deze computer is het formaat van de instructie: instr code * 2 ↑ 9 + adres)

In de MIX computer, waarvan de ontwerper Donald Knuth beweert dat het een gemiddelde is van alle gangbare computer typen (het getal MIX of 1009 blijkt het gemiddelde te zijn van getallen als 360, 7094, 6600, 1, 8, 1108, etc.), wordt de subroutine sprong anders behandeld. Er is een speciaal register: J waarin de waarde van instr teller wordt opgeborgen alvorens te springen. De eerste actie die de subroutine moet uitvoeren is nu zijn terugkeersprong-instructie te modificeren met de inhoud van J en wel als volgt:

```
rest:      STJ  UIT
           LDA  m
           STA  r
TREKAF:    SUB  n
           JAN  UIT
           STA  r
           JMP  TREKAF
UIT:       JMP  0
```

Door de eerste instructie "STJ UIT" wordt het adresdeel van de cel: "UIT" gevuld met de waarde van J.

3.3.3. Meer conditionele sprongopdrachten

Om te testen of r = 0 moesten we de instructies 24, 25, 26, 27 en 28 uitvoeren.

In de MIX computer zijn instructies van de volgende vorm aanwezig:

"*if* B *then* *begin* J:= instr teller; instr teller:= a *end*"

met B: "R < 0", "R = 0", "R > 0", "R ≥ 0", "R ≠ 0", "R ≤ 0",

waarin R een register uit een scala van 8 mag zijn, bovendien kan B zijn:

"Comp = -1", "Comp = 0", "Comp = +1",

waarin "Comp" een register is dat slechts de drie aangegeven waarden kan

bezitten. Het register "comp" krijgt een waarde na executie van een instructie CMPR, met R één van de 8 registers. Comp wordt +1 als R > 0, 0 als R = 0 en -1 als R < 0.
Tenslotte kan B zijn:

"Overflow" en "¬ Overflow",

met "Overflow" een register, waarvan de betekenis in sectie 3.4. wordt uitgelegd, en dat twee waarden aan kan nemen: true en false.

Behalve de onconditionele sprong JMP, die ook volgens bovenstaand algorithme werkt met B = true, is er in de MIX nog de onconditionele sprong JSJ, welke beschreven kan worden door "instr teller:= a", m.a.w. het is een Jump met een Saving van J.

3.3.4. Initialiseer, step up en step down instructies

Om "p:= p+1" te programmeren voor de VERA computer waren drie instructies nodig.
Bovendien een geheugencel om het getal 1 op te bergen.
In de MIX computer kunnen we twee van de 8 registers nemen, zeg R1 en R2 om p en q te bewaren. Als we bovendien tevreden zijn met het uitvoeren van de berekeningen in omgekeerde volgorde (p=10,9,...) dan bieden de volgende instructies een grote besparing:

```
            ENT1     10        R1:= 10
pcyclus:    ENT2     -1,1      R2:= -1 + R1
qcyclus:
            "bereken ggd van p en q"
            DEC2     1         R2:= R2-1
            J2P      qcyclus   if R2 > 0 then goto qcyclus
            DEC1     1         R1:= R1-1
            J1P      pcyclus   if R1 > 0 then goto pcyclus
            HLT
```

De boven beschreven 7 instructies komen in de plaats van 14 corresponderende instructies en 3 constanten (ook de constanten 0 en 10 zijn overbodig) voor de VERA computer.

De betekenis van "ENTR a" is "R:= a", waarbij a een gemodificeerd adres kan zijn, zoals in "-1,1". De betekenis van een gemodificeerd adres: "a1,i" is "a1 + if i = 0 then 0 else Ri". De conventie geldt dat "a1,0" vervangen mag worden als "a1".
Het gemodificeerde adres van "-1,1" is nu -1 + R1 zodat de tweede instructie inderdaad R2:= -1 + R1 tot gevolg heeft.
De betekenis van "DECR a" is "R:= R-a", zo is de betekenis van "INCR a": "R:= R+a". De instructies J2P en J1P komen uit het repertoire van de vorige sectie.

3.3.5. De arithmetische instructies en de schuifoperaties

Slechts voor twee van de vier arithmetische operaties heeft de VERA computer instructies. Voor vermenigvuldiging en deling is VERA aangewezen op herhaald optellen en herhaald aftrekken. Deze operaties zullen extra lang duren doordat de VERA ook geen schuifoperaties kent; de laatste operaties dienen om een getal met 10, 100, etc. te vermenigvuldigen of te delen; dit geschiedt door de interne representatie van het getal naar links of naar rechts te schuiven (mits uiteraard de interne representatie decimaal is).
Bij de MIX computer is een geheugencel in een tekenveld V0 en 5 andere velden V1, V2, V3, V4 en V5 verdeeld.
De instructies SLA en SRA schuiven de 5 velden van het register A één plaats naar links respectievelijk, naar rechts.
De instructies SLAX, SRAX, SLC en SRC schuiven de 10 velden, ontstaan door de 5 velden van A voor de 5 velden van X te plaatsen, een plaats naar links, naar rechts, circulair naar links respectievelijk, circulair naar rechts.

De instructies 18-23 van het proces "rest" dienen slechts om een deling uit te voeren en om de rest te berekenen.
Op de MIX computer kan de instructie "DIV" gebruikt worden:

```
ENTA   +0   A:= +0
LDX     m   AX:= m
DIV     n   A:= AX ÷ n; X:= rest
STX     r   r:= X
```

Afgezien van het feit dat we twee instructies minder nodig hebben, betekenen deze instructies pure tijdwinst omdat we geen loop nodig hebben.
Behalve "DIV" is er nog de instructie "MUL a" met het effect: AX:= GEH[a] * A.

3.3.6. De input- output instructies

Op de VERA computer kon je slechts éénmaal een input instructie uitvoeren, namelijk bij het starten van de machine. Verder bestond de mogelijkheid om een getal te printen. Uiteraard zijn deze mogelijkheden voor vrijwel elke computer (de uitzondering is Lehmers computer) ver beneden peil.
De MIX computer zal tot zijn beschikking hebben: een kaartlezer (KL), een kaartponser (KP), een regeldrukker (RD) en een magnetische band eenheid (MB).
Een andere opmerking is geïnspireerd op de drie na elkaar geplaatste instructies PRI. De printer is een apparaat dat vergeleken met de snelheid van de computer zeer langzaam is (voor de EL X8 heeft de printer ongeveer 5 msec nodig om een getal te printen terwijl een optelling 5 μsec behoeft; dus 1000 keer zo langzaam!).Het gevolg van een en ander is dat de VERA computer bijna steeds wacht tot dat de printer klaar is met de getallen p, q en n te printen.
Een gedeeltelijke oplossing kan reeds gevonden worden door p, q en r in een zogenaamde <u>print buffer</u> te stoppen. Dit is een rij geheugencellen waar, aan de ene kant te printen informatie in komt, en aan de andere kant deze informatie naar de echte printer wordt gestuurd. Deze techniek opent de mogelijkheid de computer te laten rekenen terwijl de printer print. Het zal dan nodig zijn dat het CRO van de computer kan zien of kan merken of het printen nu klaar is of niet.
Op de meeste computers gebeurt dit met zogenaamde <u>interrupts</u>, waarbij het CRO geïnterrupeerd wordt door een input-output orgaan met de mededeling "ik ben klaar". Voor de MIX computer gebeurt dit doordat het CRO kan vragen aan het betreffende orgaan: "ben je klaar".
Er zijn twee instructies voor:

JRED met de betekenis "spring als het betreffende orgaan klaar is"

en

JBUS met de betekenis "spring als het betreffende orgaan bezig is".

3.3.7. De vulling van een geheugencel

De VERA geheugencellen zouden wat betreft de instructies in omvang beperkt kunnen blijven tot 5 decimalen. Immers het adres kan niet groter zijn dan 3999 en de instructiecode heeft maar 8 mogelijkheden.
In de MIX computer, met een veel omvangrijker instructieset, moet de omvang van een geheugencel dus ook groter zijn.
Elke MIX geheugencel bestaat uit 6 velden: V0, V1, V2, V3, V4 en V5, waarvan V0 het teken +1 op -1 bevat, waarvan de overige velden elk een geheel getal i: $0 \leq i < 100$ kan bevatten.
Een getal g dat opgeborgen is in een MIX geheugencel is te schrijven als:

$$g = V0 * (V1 * 100 \uparrow 4 + V2 * 100 \uparrow 3 + V3 * 100 \uparrow 2 + V4 * 100 + V5).$$

De grootste waarde van g is dus $100 \uparrow 5 - 1$.

Indien g wordt opgevat als instructie dan is de betekenis van de V's de volgende:

V1	geeft het eerste adresveld A1 aan,
V2	" " tweede " A2 " ,
V3	" " adres modificerende Index register I aan,
V4	geeft de veldspecificatie F aan,
V5	geeft de instructiecode C aan.

(V0,A1,A2,I) wordt het adresdeel genoemd, terwijl (F,C) het instructiedeel wordt genoemd.

3.4. Over het ALGOL 60 programma voor de MIX computer

In de volgende sectie wordt het ALGOL 60 programma voor de MIX computer gegeven. Het is een voorbeeld van een tamelijk groot programma dat uit zeer eenvoudige onderdelen is opgebouwd. Het geeft verder een definitie van de MIX, terwijl, als men de beschrijving van Knuth er naast legt, het tenslotte een studiemogelijkheid biedt.
Bij het opstellen van het programma is de duidelijkheid richtsnoer geweest, terwijl aan efficiëntie slechts weinig aandacht is besteed.

Het gevolg van het beschrijven van de MIX in ALGOL 60 is dat een realistische implementatie van de input-output instructies nogal gecompliceerd zou worden door de onmogelijkheid parallel lopende (simultane) processen te beschrijven. De taal ALGOL 68 leent zich daar wel toe.

De eis dat het ALGOL 60 programma op de EL X8 gedraaid moest kunnen worden leidde ertoe de MIX woorden (maximale grootte 100↑5-1) in real gedeclareerde variabelen op te bergen aangezien gehele getallen in absolute waarde kleiner dan 2↑40-1 exact in een real variabele worden gerepresenteerd.

Vooruit lopende op de bespreking van decimale, octale en binaire getalrepresentaties zij nu reeds opgemerkt dat deze MIX computer volledig decimaal georganiseerd is. Het octaal maken van deze MIX computer vereist slechts het vrijwel overal de getallen 10, 100, etc. als octale getallen te beschouwen en als zodanig te vervangen door hun decimale representaties: 8, 64, etc. Ook de waarden van de instructie code zijn op het decimale getalstelsel gebaseerd zodat bijvoorbeeld de waarden 38 (=4*8+6) voor C volgens Knuth bij ons de waarde 46 is.

Alvorens met de beschrijving van de instructie code te beginnen merken we op dat het niet de elegantie en fraaiheid van deze instructie code zijn die ons deden besluiten de MIX taal te kiezen, er zijn betere machinetalen ontworpen (zoals bijvoorbeeld SERA 69 van de Stichting Studiecentrum voor Informatica te Amsterdam). De enige, maar wel doorslaggevende, reden om MIX te kiezen is de aanwezigheid van de boeken van Knuth met zijn veelheid aan sommen en voorbeelden.

Een zeer vluchtige beschrijving volgt nu.
Uit het adresdeel van een instructie: (V0,A1,A2,I) wordt het adres berekend volgens:

adres:= V0A1A2 + (if I=0 then 0 else REG[I])

waarin V0A1A2 een positief of negatief getal is en REG[I] één van de 6 indexregisters.
Bij het instructiedeel van een instructie: (F,C) worden meestal een aantal aaneengesloten velden van een geheugencel gespecificeerd en wel als volgt.
F wordt geschreven als $F = 10*L + R,\ 0 \leq L \leq R \leq 9$.
Als een geheugencel door het adres wordt aangegeven in de instructie dan worden door F de velden $V_L, \ldots, V_R$ van deze geheugencel gespecificeerd.
Bij lezen worden slechts deze velden gelezen; bij schrijven of bergen worden slechts deze velden gevuld. De waarde van de aangegeven velden zij G.
De instructie code C wordt gesplitst in Code Kop (CK) en Code Staart (CS) als volgt:

$C = 10*CK + CS,\ 0 \leq CK \leq 7,\ 0 \leq CS \leq 7.$

CK geeft een hoofdverdeling van de instructie aan, CS in een aantal gevallen een onderverdeling, en F geeft in enkele gevallen weer een nadere onderverdeling aan.

CK = 0:

CS = 0: NOP, een instructie welke generlei actie tot gevolg heeft.
CS = 1: ADD, A:= A+G
CS = 2: SUB, A:= A-G
CS = 3: MUL, AX:= A*G
CS = 4: DIV, A:= AX÷G; X:= rest.
CS = 5: F = 0: NUM, de 10 niet-teken velden van A en X worden verondersteld elk een representatie van een decimaal te bevatten. Het 10 decimalig getal dat aldus gerepresenteerd is wordt door NUM in een gewoon getal omgevormd en in A opgeborgen.
F = 1: CHAR, van het getal dat in A staat worden door CHAR de 10 decimalen berekend en in de 10 niet-teken velden A en X geplaatst.
F = 2: HLT, de executie van het programma wordt gestopt.

CS = 6: F = 0: SLA, de velden van A worden een aantal malen, door de waarde van adres aangegeven, naar links verschoven. Van rechts worden de velden met nullen aangevuld.

F = 1: SRA, zelfde als bij F = 0 met rechts i.p.v. links.

F = 2: SLAX, zelfde als bij F = 0 met AX i.p.v. A.

F = 3: SRAX, zelfde als bij F = 2 met rechts i.p.v. links.

F = 4: SLC, de velden van AX worden over een aantal plaatsen, door de waarde van adres aangegeven, circulair naar links geschoven (wat links verdwijnt komt er rechts weer in).

F = 5: SRC, zelfde als bij F = 4 met rechts en links verwisseld.

CS = 7: MOVE, een aantal, aangegeven door F, woorden wordt van geheugenplaats met het aangegeven adres als eerste adres naar de geheugenplaats met de waarde van index register 1 als eerste adres verschoven. De waarden van de index register 1 wordt na afloop met F vermeerderd.

CK = 1: LDR, met R: A, 1, 2, 3, 4, 5, 6 of X afhankelijk van CS: 0, 1, 2, 3, 4, 5, 6 of 7.
R:= G.

CK = 2: LDRN, R:= -G, met R dezelfde betekenis als boven.

CK = 3: STR, G:= R, " " " " " " .

CK = 4:

CS = 0: STJ, G:= J.

CS = 1: STZ, G:= 0.

CS = 2: JBUS, als het IO apparaat met nummer F nog bezig is spring dan naar het aangegeven adres, in J het sprongadres achterlatend.

CS = 3: IOC, niet-geïmplementeerde IO besturingsopdracht.

CS = 4: IN, het invoer apparaat met nummer F wordt in actie gezet om een standaard aantal woorden in te lezen, beginnende met het berekende adres. Van een kaart is dat 16 woorden van elk 5 karakters (symbolen). Het nummer van de kaartlezer is 16.

CS = 5: OUT, hetzelfde als boven met uitvoer i.p.v. invoer. Het nummer van de regeldrukker is 18 en de standaardhoeveelheid woorden is 24.

CS = 6: JRED, 't zelfde als bij JBUS met klaar i.p.v. nog bezig.

CS = 7: F = 0: JMP, spring naar het aangegeven adres in J het sprongadres achterlatend.

F = 1: JSJ, spring naar het aangegeven adres zonder J te veranderen (Jump and Save J).

F = 2: JOV, indien het OVERFLOW register aanstaat (= true) doe dan JMP.

F = 3: JNOV, indien het OVERFLOW register niet aanstaat (= false) doe dan JMP.

F = 4:	JL,	indien	het	register	COMP	= -1	doe	JMP.
F = 5:	JE,	"	"	"	"	= 0	"	" .
F = 6:	JG,	"	"	"	"	= +1	"	" .
F = 7:	JGE,	"	"	"	"	≠ -1	"	" .
F = 8:	JNE,	"	"	"	"	≠ 0	"	" .
F = 9:	JLE,	"	"	"	"	≠ +1	"	" .

CK = 5: JRC, met R bepaald door CS, i.e. A, 1, 2, 3, 4, 5, 6 of X, en C, afhankelijk van de waarde van F, één van de volgende condities:
F = 0: N(negatief), F = 1: Z(nul), F = 2: P(positief), F = 3: NN (niet-negatief), F = 4: NZ(niet-nul), F = 5: NP(niet-positief).
De betekenis is: als het aangegeven register aan de conditie voldoet doe dan JMP.

CK = 6: F = 0:	INCR, R:= R + adres;	met	R	bepaald	door	CS.
F = 1:	DECR, R:= R - adres;	"	"	"	"	".
F = 2:	ENTR, R:= adres;	"	"	"	"	".
F = 3:	ENNR, R:= - adres;	"	"	"	"	".

CK = 7: CMPR, het register COMP krijgt een waarden volgens:
COMP:= if R < G then -1 else if R = G then 0 else +1; met R bepaald door CS.

```
begin comment MIX computer, RPR 190870, opdrachtnr. 2133.

  3.4.1 De MIX computer.;

  real array GEH[0:3999];

  procedure CRO;
  begin comment Het CRO beschikt over de volgende registers:;
    integer instr teller,COMP; real instr reg;
    real array REG[0:8]; Boolean UVERFLOW;

    comment Omdat de registers REG[0],REG[7] en REG[8] een
    speciale rol spelen voeren we de namen A,X en J in met
    de waarden 0,7 en 8. We merken op dat bij Knuth de
    registers 1,2,3,4,5,6 en J slechts getallen welke in
    absolute waarde kleiner zijn dan 9999 kunnen bevatten.
    Hier behandelen we ze net zoals het A of X register.;
    integer A,X,J;

    procedure BASIS CYCLUS;
    begin BEGIN:
      ERROR(instr teller < 0 ∨ instr teller > 4000,
           ⟨instr teller buiten de grenzen⟩);
      instr reg:= GEH[instr teller];
      instr teller:= instr teller + 1;
      EXECUTEER INSTRUCTIE; goto BEGIN
    end BASIS CYCLUS;

    integer procedure ERROR(B,st); Boolean B; string st;
    if B then
    begin NLCR; PRINTTEXT(⟨FOUT⟩);
      FIXT(4,0,instr teller);
      FIXT(12,0,instr reg);
      PRINTTEXT(st); ERROR:= 1; goto EINDE EXECUTIE
    end ERROR;

    procedure EXECUTEER INSTRUCTIE;
    begin integer adres,VOA1A2,I,F,C,L,R,CK,CS;
      real G;
      comment De volgende zogenaamde switches of schakelaars
      dienen om op efficiente wijze, op grond van de waarden
      van CK,CS en F, te springen naar de algorithme die
      er bij hoort.;
      switch SW1:= NOP,ADD,SUB,MUL,DIV,SW2[F+1],SHIFT,MOVE;
      switch SW2:= NUM,CHAR,HLT;
      switch SW3:= STJ,STZ,JBUS,IOC,IN,OUT,JRED,SW4[F + 1];
      switch SW4:= JMP,JSJ,JOV,JNOV,JL,JE,JG,JGE,JNE,JLE;
      switch SW5:= INCR,DECR,ENTR,ENNR;
      switch SW6:= SW1[CS+1],LDR,LDRN,STR,SW3[CS+1],JRC,
                   SW5[F+1],CMPR;
      comment De ene statement:goto SW6[CK+1] zorgt er voor
      dat we precies bij de goede label uitkomen.
      Alvorens tot de executie over te gaan worden eerst nog
      een paar hulpstukken geintroduceerd.
      De eerste is om van een gegeven cel de gespecificeerde
      velden te lezen, de tweede is om gespecificeerde velden
      van een gegeven cel met een getal te vullen, de derde
      is om het tekenveld VO van een cel te lezen.;
```

```
real procedure LEES VELDEN(cel,L,R); value cel,L,R;
real cel; integer L,R;
if L = 0 ∧ R = 0 then LEES VELDEN:= TEKEN(cel) else
begin real voorstuk,hulp; integer teken;
  ERROR(¬( 0 < L ∧ L < R ∧ R < 5),
       ⪡veld specificatie bij lezen niet OK⪢);
  if L = 0 then begin teken:= TEKEN(cel); L:= 1 end
  else teken:= 1; hulp:= 100 ⩚ (R − L + 1);
  voorstuk:= entier(abs(cel)/100 ⩚ (5 − R));
  LEES VELDEN:= teken × (voorstuk −
  entier(voorstuk/hulp) × hulp)
end LEES VELDEN;

procedure VUL VELDEN(cel,L,R,inf); value L,R,inf;
real cel,inf; integer L,R;
if L = 0 ∧ R = 0 then cel:= TEKEN(inf) × abs(cel) else
begin real celabs,voorstuk,achterstuk,hulp1,hulp2;
  integer teken;
  ERROR(¬ (0 < L ∧ L < R ∧ R < 5),
       ⪡veld specificatie bij vullen niet OK⪢);
  celabs:= abs(cel);
  if L = 0 then begin teken:= TEKEN(inf); L:= 1 end
  else teken:= TEKEN(cel);
  ERROR(abs(inf) > 100 ⩚ (R − L + 1),
       ⪡bij vullen past de informatie niet⪢);
  hulp1:= 100 ⩚ (6 − L); hulp2:= 100 ⩚ (5 − R);
  voorstuk:= entier(celabs/hulp1);
  achterstuk:= celabs − entier(celabs/hulp2) × hulp2;
  cel:= teken × (voorstuk × hulp1 +
                 abs(inf) × hulp2 + achterstuk)
end VUL VELDEN;

integer procedure TEKEN(cel);
TEKEN:= if cel ≥ 0 then +1 else −1;

comment De volgende procedures en variabelen worden nog
voor het gemak ingevoerd, daarmee is de declaratie
beeindigd.;

procedure BA; comment BA Berekent het Adres;
begin ERROR(I > 6,⪡index niet OK⪢);
  adres:= VOA1A2 + (if I = 0 then 0 else REG[I]);
  ERROR(adres < 0 ∨ adres > 3999,
       ⪡adres niet OK⪢);
  NLCR; PRINTTEXT(⪡adres⪢); FIXT(4,0,adres)
end BA;

procedure BG; comment BG Berekent G;
begin BA;
  G:= LEES VELDEN(GEH[adres],L,R);
  NLCR; PRINTTEXT(⪡G⪢); FIXT(12,0,G)
end BG;

procedure SPRING INDIEN(conditie); Boolean conditie;
begin BA; if conditie then
  begin REG[J]:= instr teller; instr teller:= adres end;
  goto EINDE
end SPRING INDIEN;
```

```
  integer i,k,j;
  real hulp1,hulp2;

INITIALISATIE: VOA1A2:= entier(abs(instr reg)/100 00 00);
  i:= abs(instr reg) - VOA1A2 × 100 00 00;
  VOA1A2:= sign(instr reg) × VOA1A2;
  I:= i : 100 00; i:= i - I × 100 00;
  F:= i : 100; C:= i - F × 100;
  L:= F : 10; R:= F - L × 10;
  CK:= C : 10; CS:= C - CK × 10;

  NLCR; PRINTTEXT(⟨reg.:⟩);
  for i:= 0 step 1 until 8 do FIXT(12,0,REG[i]);
  if OVERFLOW then PRINTTEXT(⟨true⟩) else PRINTTEXT(⟨false⟩);
  if COMP = -1 then PRINTTEXT(⟨ L⟩)
                else if COMP = 0 then PRINTTEXT(⟨ E⟩)
                else PRINTTEXT(⟨ G⟩); NLCR; NLCR;
  PRINTTEXT(⟨instr.:⟩); FIXT(4,0,instr teller - 1); FIXT(12,0,instr reg);

  comment Alvorens de strooisprong over de switch SW6 te
  doen worden eventuele fouten eruit gehaald:;
  ERROR(CK > 7,⟨CK niet correct⟩);
  ERROR(CS > 7,⟨CS niet correct⟩);
  ERROR(if CK = 0 ∧ CS = 5 then F > 2 else
        if CK = 0 ∧ CS = 6 then F > 5 else
        if CK = 4 ∧ CS = 7 then F > 9 else
        if CK = 6 then F > 3 else false,
        ⟨F niet correct⟩);
goto SW6[CK + 1];

  comment De volgorde waarin de, in de switches gebruikte,
  labels nu verschijnen is volledig arbitrair. Wij zullen de
  volgorde van Knuth aanhouden.;

LDR: BG; REG[CS]:= G; goto EINDE;
LDRN: BG; REG[CS]:= -G; goto EINDE;

STR: BA; VUL VELDEN(GEH[adres],L,R,
       (if L = 0 then TEKEN(REG[CS]) else 1) ×
       abs(LEES VELDEN(REG[CS],L + 5 - R,5)));
  goto EINDE;

STJ: CS:= J; goto STR;

STZ: BA; VUL VELDEN(GEH[adres],L,R,0);
  goto EINDE;

ADD:
SUB: BG; if C = 2 then G:= -G;
  hulp1:= REG[A] + G;
  if abs(hulp1) > 100000 00000 then
  begin OVERFLOW:= true; hulp1:= hulp1 - sign(hulp1) × 100000 00000 end;
  REG[A]:= hulp1;
  goto EINDE;

MUL: BG; hulp1:= REG[A] × G;
  if abs(hulp1) < 100000 00000 then
```

```
    begin REG[A]:= 0; REG[X]:= hulp1 end else
    begin integer array AV,GV[0:4],RV[0:9];
      for i:= 0,1,2,3,4 do
      begin AV[i]:= LEES VELDEN(REG[A],5 - i,5 - i);
        GV[i]:= LEES VELDEN(G,5 - i,5 - i)
      end;
      for i:= 0 step 1 until 8 do
      begin RV[i]:= SUM(k,0,i,if k < 4 ∧ i - k < 4 then
                             AV[k] × GV[i - k] else 0);
        comment SUM is een standaard procedure welke de som
        van de aangegeven termen berekent.;
      end; RV[9]:= 0;
      for i:= 0 step 1 until 8 do
      begin k:= RV[i]; j:= k : 100;
        RV[i]:= k - j × 100;
        RV[i + 1]:= RV[i + 1] + j
      end;
      REG[X]:= sign(hulp1) × SUM(i,0,4,RV[i] × 100 ↑ i);
      REG[A]:= sign(hulp1) × SUM(i,5,9,RV[i] × 100 ↑ (i - 5));
      comment Het is een goede oefening overal de inefficiente uitdrukkingen
      "SUM(i, ... , 100 ↑ ...)" te vervangen door meer efficiente.;
    end MUL;
    goto EINDE;

DIV: BG; if REG[A] = 0 then
    begin if G = 0 then OVERFLOW:= true else
      begin hulp1:= REG[X];
        REG[A]:= sign(G × hulp1) ×
        entier(abs(hulp1/G));
        REG[X]:= hulp1 - REG[A] × G
    end end else
    begin integer array AV,XV,QV[0:4];
      k:= sign(G); G:= abs(G);
      if abs(REG[A]) > G then
      begin OVERFLOW:= true; goto EINDE end;
      for i:= 0,1,2,3,4 do
      begin AV[i]:= LEES VELDEN(REG[A],5 - i,5 - i);
            XV[i]:= LEES VELDEN(REG[X],5 - i,5 - i)
      end;
      comment De algorithme berust op het volgende:
      Zij A × 100 ↑ 5 + X = Q × G + R. Noodzakelijkerwijs
      moet gelden Q < 100 ↑ 5 - 1 en R < 100 ↑ 5 - 1.
      Uit Q = entier((A × 100 ↑ 5 + X)/G) volgt dus
      A × 100 ↑ 5 + X < G × 100 ↑ 5.
      Aangenomen dat alle getallen A,X en G positief zijn
      volgt dus dat A < G moet zijn. Het geval A > G hebben
      we reeds uitgesloten, dit geeft zeker overflow. Het
      teken van het resultaat wordt achteraf in overeenstemming
      gebracht met de tekens van A,X en G.
      We schrijven A × 10 ↑ 5 + X als:
        (A × 100 + XV[4]) × 100 ↑ 4 + ... ,
      en berekenen hieruit QV[4]. Evenzo bepalen we
      QV[i],i = 3,2,1,0.;
```

```
    hulp1:= abs(REG[A]) × 100 + XV[4];
    for i:= 3,2,1,0 do
    begin hulp2:= entier(hulp1/G);
      QV[i + 1]:= hulp2;
      hulp1:= (hulp1 - hulp2 × G) × 100 + XV[i]
    end;
    k:= k × sign(REG[A]);
    QV[0]:= entier(hulp1/G); hulp2:= entier(hulp1/G);
    REG[X]:= (hulp1 - hulp2 × G) × sign(REG[A]);
    REG[A]:= SUM(i,0,4,QV[i] × 100 ↑ i) × k
  end DIV;
  goto EINDE;

ENTR: BA; REG[CS]:= adres; goto EINDE;
ENNR: BA; REG[CS]:= -adres; goto EINDE;
INCR: BA; REG[CS]:= REG[CS] + adres; goto EINDE;
DECR: BA; REG[CS]:= REG[CS] - adres; goto EINDE;

CMPR: BG; COMP:= if REG[CS] < G then -1 else
                 if REG[CS] = G then 0 else +1;
  goto EINDE;

JMP: SPRING INDIEN(true);
JSJ: BA; instr teller:= adres; goto EINDE;
JOV: SPRING INDIEN(OVERFLOW);
JNOV: SPRING INDIEN(¬ OVERFLOW);
JL: SPRING INDIEN(COMP = -1);
JE: SPRING INDIEN(COMP = 0);
JG: SPRING INDIEN(COMP = +1);
JGE: SPRING INDIEN(COMP ≠ -1);
JNE: SPRING INDIEN(COMP ≠ 0);
JLE: SPRING INDIEN(COMP ≠ +1);

JRC: SPRING INDIEN(
      if F = 0 then REG[CS] < 0 else
      if F = 1 then REG[CS] = 0 else
      if F = 2 then REG[CS] > 0 else
      if F = 3 then REG[CS] ≥ 0 else
      if F = 4 then REG[CS] ≠ 0 else
      if F = 5 then REG[CS] ≤ 0 else false);

MOVE: BA; i:= REG[1];
  for j:= 1 step 1 until F do
  GEH[i + j - 1]:= GEH[adres + j - 1];
  REG[1]:= i + F;
  goto EINDE;

SHIFT: BA;
  begin switch SH:= SLA,SRA,SLAX,SRAX,SLC,SRC;
    integer array AV,XV,AVN[1:5],AXV,AXVN[1:10];
    for i:= 1,2,3,4,5 do
    begin AV[i]:= AXV[i]:= LEES VELDEN(REG[A],i,i);
      XV[i]:= AXV[i + 5]:= LEES VELDEN(REG[X],i,i)
    end;
    goto SH[F + 1];

  SLA: SRA:
    for i:= 1,2,3,4,5 do
    begin j:= i + (if F = 1 then -adres else adres);
      AVN[i]:= if 0 < j ∧ j < 6 then AV[j] else 0
    end;
```

```
    REG[A]:= TEKEN(REG[A]) × SUM(i,1,5,AVN[i] × 100 ↑ (5 − i));
    goto EINDE;

  SLAX: SRAX:
    for i:= 1 step 1 until 10 do
    begin j:= i + (if F = 3 then −adres else adres);
      AXVN[i]:= if 0 < j ∧ j < 11 then AXV[j] else 0
    end;
  zie boven: REG[A]:= TEKEN(REG[A]) × SUM(i,1,5,
                AXVN[i] × 100 ↑ (5 − i));
    REG[X]:= TEKEN(REG[X]) × SUM(i,6,10,AXVN[i] × 100 ↑ (10 − i));
    goto EINDE;

  SLC: SRC:
    for i:= 1 step 1 until 10 do
    begin j:= i + (if F = 5 then −adres else adres);
      if j > 0 then j:= j − (j − 1) : 10 × 10
                 else j:= j − ( j: 10 − 1) × 10;
      AXVN[i]:= AXV[j]
    end;
    goto zie boven;
  end SHIFT;

NOP: goto EINDE;
HLT: goto EINDE EXECUTIE;

IN: BA; if F = 16 then
  begin comment We lezen een kaart en brengen de 16 woorden,
    welke gevormd worden door de alphanumerieke karakters in
    de 80 kolommen, in opvolgende geheugencellen beginnende
    bij het aangegeven adres.
    Om de MIX computer ook geschikt te maken voor invoer van
    paper tape is onderstaand programma een beetje aangepast.;
    integer array symb[1:80];
    for i:= 1, i + 1 while i ≤ 80 do
    begin k:= RESYM;
      comment RESYM is een standaard procedure welke precies
      1 kolom van een kaart leest(1 symbool van een paper tape
      leest) en als waarde de volgens MC specificaties vastgestelde
      interne representatie van het gelezen symbool krijgt.
      Het waarde bereik van RESYM is van 0 tot 127 (dus groter dan
      99). Het is dus nodig dit waarde bereik terug te brengen
      van 0 tot 99.
      Hiertoe laten we de kleine letters a t/m z buiten beschouwing
      en de verticale streep | geven we de ongebruikte waarde 69;
      if k > letter z then k:= k − 27;
      if k = 100 then k:= 69;
      if k = tab then begin symb[i]:=spatie; i:=i + 1;
      for j:= i, j + 1 while j ≠ j : 8 × 8 + 1 do
      begin i:= j; symb[i]:= spatie end end else
      if k = twnr then
      begin for i:= i, i + 1 while i ≤ 80 do symb[i]:= spatie;
        i:= 80
      end else symb[i]:= k
    end; for i:= 0 step 1 until 15 do
    begin hulp1:= 0;
      for j:= 1,2,3,4,5 do hulp1:= hulp1 × 100 + symb[5×i + j];
      GEH[adres + i]:= hulp1
```

```
    end; comment Eventuele symbolen welke teveel zijn (meer dan 80)
    worden geskipt:;
  SKIP: if k = twnr then goto KLAAR; k:= RESYM;
    if k > letter z then k:= k - 27;
    if k = 100 then k:= 69; goto SKIP;
  KLAAR:
  end; goto EINDE;

OUT: BA; if F = 18 then
  begin comment 24 woorden van elk 5 alphanumerieke karakters
    worden op de regeldrukker geprint.;
    for i:= 0 step 1 until 23 do
    for j:= 1,2,3,4,5 do
    k:= LEES VELDEN(GEH[adres + i],j,j);
    if k > 10 then k:= k + 27;
    if k ≡ 96 then k:= 127;
    PRSYM(k);
        comment PRSYM is een standaard procedure en print
        een symbool waarvan de interne representatie meegegeven
        is als parameter.;
  end; goto EINDE;

IOC:; comment Nog niet geimplementeerd.; goto EINDE;

JRED: goto JMP;

JBUS: goto EINDE; comment De vorige instructie en deze
  instructie kunnen niet goed in ALGOL 60 worden beschreven
  daar impliciet wordt aangenomen dat een input of output
  proces simultaan met het rekenproces, dat zich in het CRO
  afspeelt, verloopt. Wij nemen aan dat elke input of output
  opdracht meteen wordt uitgevoerd.;

NUM: hulp1:= 0; for i:= 1,2,3,4,5 do
  begin j:= LEES VELDEN(REG[A],i,i);
    j:= j - j : 10 × 10;
    k:= LEES VELDEN(REG[X],i,i);
    k:= k - k : 10 × 10;
    hulp1:= 10 × hulp1 + j × 100000 + k
  end;
  REG[A]:= TEKEN(REG[A]) × hulp1;
  goto EINDE;

CHAR: begin integer array cijfer[1:10];
    hulp1:= abs(REG[A]);
    for i:= 10 step -1 until 1 do
    begin hulp2:= entier(hulp1/10);
      cijfer[i]:= hulp1 - hulp2 × 10;
      hulp1:= hulp2
    end;
    REG[A]:= TEKEN(REG[A]) × SUM(i,1,5,
      cijfer[i] × 100 ↑ (5 - i));
    REG[X]:= TEKEN(REG[X]) × SUM(i,6,10,
      cijfer[i] × 100 ↑ (10 - i))
  end;
  goto EINDE;

EINDE:
end EXECUTEER INSTRUCTIE;
```

```
procedure START CRO;
begin comment In tegenstelling tot de VERA heeft de MIX
   een uitermate eenvoudige start: slechts 1 kaart wordt
   gelezen met de opdracht IN 0(16).
   Op de kaart moet dan een programma gegeven worden dat
   het verder inlezen van nog meer kaarten bevat. Dit
   programma wordt na het lezen gestart.;
   for instr teller := 0 step 1 until 3999 do GEH[instr teller] := 0;
   for instr teller := 0 step 1 until 8 do REG[instr teller] := 0;
   OVERFLOW := false; COMP := 0;A:= 0; X:= 7; J:= 8;
   letter z:= 35;
   twnr:= 119 - 27;
   tab:= 118 - 27;
   spatie:= 93 - 27;
   instr teller:= 0;
   instr reg:= + 00 00 00 16 44; comment IN 0(16);
   EXECUTEER INSTRUCTIE;
   BASIS CYCLUS
end START CRO;
integer twnr,tab,spatie,letter z;
START CRO;
EINDE EXECUTIE:
  end CRO; CRO
end
```

3.4.2. Het MIX startprogramma

Door op de startknop van de MIX te drukken, dit is hetzelfde als het ALGOL 60 programma in de X8 computer te starten, wordt precies één kaart met 80 symbolen gelezen. De 16 geheugencellen met adressen 0 t/m 15 worden gevuld met elk 5 getallen, welke de interne representaties zijn van die symbolen.

Bijvoorbeeld de kaart met de volgende gegevens

"1A2B3a4b,.."

heeft tot resultaat:

GEH[0]=0110021103
GEH[1]=1004116061
GEH[2]=6161616161

GEH[15]=6161616161

We merken op, dat de interne representatie der symbolen afgeleid is van MR 81, dus de MC-ALGOL-RESYM waarde, met dien verstande, dat kleine letters als hoofdletters worden beschouwd. De volgende algorithme wordt gebruikt.

k:=RESYM; <u>if</u> k>letter z <u>then</u> k:=k-27;
<u>if</u> k=100 <u>then</u> k:=69

met letter z=35.

We merken op, dat het symbool "|" de interne representatie 69 krijgt welke toch ongedefinieerd was.

Nadat die ene kaart gelezen is rijst de vraag: wat nu?

De MIX gaat tot executie van de instructie uit GEH[0] over, vervolgens de instructie uit GEH[1] etc.

In ons geval betekent dit, dat de instructie:

MUL 110,2(1:1)

wordt uitgevoerd, wat natuurlijk zinloos is.

We zien, dat het nodig is de 80 symbolen zodanig te kiezen, dat er een zinvol programma op de plaatsen 1 t/m 15 staat. Dit programma noemen we het

"Koude Start Programma" (KSP).
Wat zal de taak van het KSP moeten zijn?
De eerste taak is het lezen van kaarten; de tweede taak is de symbolen van deze kaarten zodanig te interpreteren, dat we op een prettige wijze een MIX machine taal programma kunnen opstellen en niet meer verplicht zijn om in de tabel van interne representaties die symbolen uit te zoeken (zo ze er zijn) welke bij de gegeven getallen behoren.
Teneinde instructies onder elkaar te kunnen opschrijven zullen we één kaart per instructie nemen en daarvan slechts de eerste 10 kolommen lezen.
De laatste kaart bevat in de eerste 10 kolommen slechts nullen.

We stellen het KSP zowel in mnemotechnische taal als in machine taal op.
Als variabele, welke bijhoudt op welk adres de volgende te lezen instructie geplaatst moet worden, kiezen we REG[1]. De eerste te lezen instructie moet op dat adres geplaatst worden, dat volgt op het adres van de laatste instructie van het KSP. Dan is het effect dat na uitvoering van het KSP, dus nadat alle kaarten zijn gelezen, meteen het zojuist gelezen programma wordt uitgevoerd.

Het eerste KSP programma luidt:			(machine taal):
0:	ENT1	laatste adres +1	00 08 00 02 61
1:	IN	0,1 (16)	00 00 01 16 44
2:	LDA	0,1 (0:5)	00 00 01 05 10
3:	LDX	1,1 (0:5)	00 01 01 05 17
4:	NUM		00 00 00 00 05
5:	STA	0,1 (0:5)	00 00 01 05 30
6:	INC1	1	00 01 00 00 61
7:	JANZ	1	00 01 00 04 50

De symbolen, die samen het KSP vormen zijn dus:

"0802.001G

Helaas, de vlieger gaat niet op want 44 is niet de interne representatie van één of ander symbool, net zo min overigens als 50. De overige getallen komen alle als symbool voor.
Nu zouden we wellicht voor JANZ nog wel een andere vorm kunnen vinden maar

de instructie IN hebben we absoluut nodig.
We zijn dus geneigd het KSP project als onmogelijk ter zijde te schuiven, ware het niet, dat enige slimheid ons hier kan helpen.
We maken gebruik van het feit, dat de inhoud van een geheugencel op twee manieren is te interpreteren: als instructie en als getal.
Als we op de plaats, waar de IN instructie staat, niet het getal 11644 maar het getal 11643 zetten en, alvorens dit getal als instructie te verwerken, de betreffende geheugencel met 1 ophogen, dan hebben we precies het gewenste resultaat. Bovendien behoort bij het getal 43 wel een symbool n.l. "=".
Voeren we dezelfde bewerking uit met de instructie JANZ (bij 49 hoort het symbool "¬") dan komen we tot het volgende definitieve Koude Start Programma:

```
 0:    ENT1     14             00 14 00 02 61
 1:    LDA      7(0:5)         00 07 00 05 10
 2:    INCA     1              00 01 00 00 60
 3:    STA      7(0:5)         00 07 00 05 30
 4:    LDA      13(0:5)        00 13 00 05 10
 5:    INCA     1              00 01 00 00 60
 6:    STA      13(0:5)        00 13 00 05 30
 7:   (IN       0,1(16))-1     00 00 01 16 43
 8:    LDA      0,1(0:5)       00 00 01 05 10
 9:    LDX      1,1(0:5)       00 01 01 05 17
10:    NUM                     00 00 00 00 05
11:    STA      0,1(0:5)       00 00 01 05 30
12:    INC1     1              00 01 00 00 61
13:   (JANZ     7)-1           00 07 00 04 49
```

De bijbehorende symbolen van de eerste kaart zijn:
0E02.0705A0100,0705U0D05A0100,0D05U001G=0015A0115H000050015U0100.0704¬0123456789
De laatste 10 symbolen zijn toegevoegd om de kaart vol te krijgen.
De snelle regeldrukker pagina's 9, 10, 23, 24 en 25 waarop de resultaten te vinden zijn van de MIX simulator met de input gegevens uit deze en de volgende sectie zijn afgedrukt op de pagina's 45, 46, 47, 49 en 51.

```
REG.:           +0        +0        +0        +0        +0        +0        +0        +0        +0 FALSE E

INSTR.:  -1        +1644
ADRES   +0
REG.:           +0        +0        +0        +0        +0        +0        +0        +0        +0 FALSE E

INSTR.:  -0    +14000261
ADRES  +14
REG.:           +0       +14        +0        +0        +0        +0        +0        +0        +0 FALSE E

INSTR.:  +1     +7000510
ADRES   +7
G       +11643
REG.:       +11643       +14        +0        +0        +0        +0        +0        +0        +0 FALSE E

INSTR.:  +2     +1000060
ADRES   +1
REG.:       +11644       +14        +0        +0        +0        +0        +0        +0        +0 FALSE E

INSTR.:  +3     +7000530
ADRES   +7
REG.:       +11644       +14        +0        +0        +0        +0        +0        +0        +0 FALSE E

INSTR.:  +4    +13000510
ADRES  +13
G      +7000449
REG.:     +7000449       +14        +0        +0        +0        +0        +0        +0        +0 FALSE E

INSTR.:  +5     +1000060
ADRES   +1
REG.:     +7000450       +14        +0        +0        +0        +0        +0        +0        +0 FALSE E

INSTR.:  +6    +13000530
ADRES  +13
REG.:     +7000450       +14        +0        +0        +0        +0        +0        +0        +0 FALSE E

INSTR.:  +7       +11644
ADRES  +14
REG.:     +7000450       +14        +0        +0        +0        +0        +0        +0        +0 FALSE E

INSTR.:  +8       +10510
ADRES  +14
G    +101020200
REG.:   +101020200       +14        +0        +0        +0        +0        +0        +0        +0 FALSE E

INSTR.:  +9     +1010517
ADRES  +15
G       +30600
REG.:   +101020200       +14        +0        +0        +0        +0        +0    +30600        +0 FALSE E

INSTR.:  +10          +5
REG.:  +1122000360       +14        +0        +0        +0        +0        +0    +30600        +0 FALSE E

INSTR.:  +11      +10530
ADRES  +14
REG.:  +1122000360       +14        +0        +0        +0        +0        +0    +30600        +0 FALSE E

INSTR.:  +12    +1000061
ADRES   +1
```

```
REG.:  +1122000360        +15     +0     +0     +0     +0     +0     +30600      +0 FALSE ■

INSTR.:  +13      +7000450
ADRES    +7
REG.:  +1122000360        +15     +0     +0     +0     +0     +0     +30600     +14 FALSE ■

INSTR.:   +7        +11644
ADRES  +15
REG.:  +1122000360        +15     +0     +0     +0     +0     +0     +30600     +14 FALSE ■

INSTR.:   +8        +10510
ADRES  +15
@   +100000000
REG.:   +100000000        +15     +0     +0     +0     +0     +0     +30600     +14 FALSE ■

INSTR.:   +9      +1010517
ADRES  +16
@       +20300
REG.:   +100000000        +15     +0     +0     +0     +0     +0     +20300     +14 FALSE ■

INSTR.:  +10            +5
REG.:  +1000000230        +15     +0     +0     +0     +0     +0     +20300     +14 FALSE ■

INSTR.:  +11        +10530
ADRES  +15
REG.:  +1000000230        +15     +0     +0     +0     +0     +0     +20300     +14 FALSE ■

INSTR.:  +12      +1000061
ADRES    +1
REG.:  +1000000230        +16     +0     +0     +0     +0     +0     +20300     +14 FALSE ■

INSTR.:  +13      +7000450
ADRES    +7
REG.:  +1000000230        +16     +0     +0     +0     +0     +0     +20300     +14 FALSE ■

INSTR.:   -7        +11644
ADRES  +16
REG.:  +1000000230        +16     +0     +0     +0     +0     +0     +20300     +14 FALSE ■

INSTR.:   +8        +10510
ADRES  +16
@   +100000000
REG.:   +100000000        +16     +0     +0     +0     +0     +0     +20300     +14 FALSE ■

INSTR.:   +9      +1010517
ADRES  +17
@     +3050300
REG.:   +100000000        +16     +0     +0     +0     +0     +0   +3050300     +14 FALSE ■

INSTR.:  +10            +5
REG.:  +1000003530        +16     +0     +0     +0     +0     +0   +3050300     +14 FALSE ■

INSTR.:  +11        +10530
ADRES  +16
REG.:  +1000003530        +16     +0     +0     +0     +0     +0   +3050300     +14 FALSE ■

INSTR.:  +12      +1000061
ADRES    +1
REG.:  +1000003530        +17     +0     +0     +0     +0     +0   +3050300     +14 FALSE ■
```

```
101070 -    200

INSTR.:  +12       +1000061
ADRES    +1
REG.:         +205          +42          +0          +0          +0          +0          +0      +20005        +14 FALSE ■

INSTR.:  +13       +7000450
ADRES    +7
REG.:         +205          +42          +0          +0          +0          +0          +0      +20005        +14 FALSE ■

INSTR.:   +7         +11644
ADRES  +42
REG.:         +205          +42          +0          +0          +0          +0          +0      +20005        +14 FALSE ■

INSTR.:   +8         +10510
ADRES  +42
@            -0
REG.:           -0          +42          +0          +0          +0          +0          +0      +20005        +14 FALSE ■

INSTR.:   +9       +1010517
ADRES  +43
@            -0
REG.:           -0          +42          +0          +0          +0          +0          +0          -0        +14 FALSE ■

INSTR.:  +10             +5
REG.:           -0          +42          +0          +0          +0          +0          +0          -0        +14 FALSE ■

INSTR.:  +11         +10530
ADRES  +42
REG.:           -0          +42          +0          +0          +0          +0          +0          -0        +14 FALSE ■

INSTR.:  +12       +1000061
ADRES    +1
REG.:           -0          +43          +0          +0          +0          +0          +0          -0        +14 FALSE ■

INSTR.:  +13       +7000450
ADRES    +7
REG.:           -0          +43          +0          +0          +0          +0          +0          -0        +14 FALSE ■

INSTR.:  +14    +1122000360
ADRES+1122
REG.:        -1122          +43          +0          +0          +0          +0          +0          -0        +14 FALSE ■

INSTR.:  +15    +1000000230
ADRES+1000
REG.:        -1122          +43          +0          +0          +0          +0          +0          -0        +14 FALSE ■

INSTR.:  +16    +1000003530
ADRES+1000
REG.:        -1122          +43          +0          +0          +0          +0          +0          -0        +14 FALSE ■

INSTR.:  +17    +1000000010
ADRES+1000
@            -1
REG.:           -1          +43          +0          +0          +0          +0          +0          -0        +14 FALSE ■

INSTR.:  +18    +1000000210
ADRES+1000
@        -1122
REG.:        -1122          +43          +0          +0          +0          +0          +0          -0        +14 FALSE ■
```

3.4.3. Een test programma

Het spreekt vanzelf, dat de MIX computer behoorlijk getest moet worden. Dank zij het feit, dat onze MIX computer zo vriendelijk is om alle tussenresultaten zichtbaar te maken, kunnen we nu een eenvoudig programmaatje en daarmee de MIX zelf testen.
Aan de hand van het ALGOL programma van de MIX computer zullen we nagaan welke onderdelen hoe getest moeten worden.

a) De switches SW1-SW6 worden automatisch getest mits alle instructies een beurt krijgen.

b) LEESVELDEN kan als volgt getest worden:

```
1122000360    14: ENNA    1122          "A:= -1122
1000000230    15: STA     1000 (0:2)
1000003530    16: STA     1000 (3:5)
1000000010    17: LDA     1000 (0:0)
1000000210    18: LDA     1000 (0:2)
1000003510    19: LDA     1000 (3:5)
```

c) VUL VELDEN wordt met bovenstaand programmaatje getest

d) BA wordt getest met:

```
1000000263    20: ENT3    1000
0000030510    21: LDA     0,3 (0:5)     "A= -1122001122
```

e) SPRING INDIEN wordt getest met:

```
0024000250    22: JAP     *+2 "(betekent: deze regel + 2)
0026000050    23: JAN     *+3 "(betekent: deze regel + 3)
0918273645    24: AAP
5463728190    25: NOOT
```

```
INSTR.:  +19    +1000003510
ADRES+1000
@          +1122
REG.:        +1122          +43        +0        +0        +0        +0        +0           -0      +14 FALSE

INSTR.:  +20    +1000000263
ADRES+1000
REG.:        +1122          +43        +0     +1000        +0        +0        +0           -0      +14 FALSE

INSTR.:  +21          +30510
ADRES+1000
@  -1122001122
REG.:  -1122001122          +43        +0     +1000        +0        +0        +0           -0      +14 FALSE

INSTR.:  +22       +24000250
ADRES  +24
REG.:  -1122001122          +43        +0     +1000        +0        +0        +0           -0      +14 FALSE

INSTR.:  +23       +26000050
ADRES  +26
REG.:  -1122001122          +43        +0     +1000        +0        +0        +0           -0      +24 FALSE

INSTR.:  +26          +30501
ADRES+1000
@  -1122001122
REG.:  -2244002244          +43        +0     +1000        +0        +0        +0           -0      +24 FALSE

INSTR.:  +27    +1000001202
ADRES+1000
@          +1122
REG.:  -2244003366          +43        +0     +1000        +0        +0        +0           -0      +24 FALSE

INSTR.:  +28          +34503
ADRES+1000
@          +1122
REG.:         -251          +43        +0     +1000        +0        +0        +0  -7771776652      +24 FALSE

INSTR.:  +29    +1001000533
ADRES+1001
REG.:         -251          +43        +0     +1000        +0        +0        +0  -7771776652      +24 FALSE

INSTR.:  +30    +1001000504
ADRES+1001
@          +1000
REG.:  -251777176           +43        +0     +1000        +0        +0        +0         -652      +24 FALSE

INSTR.:  +31        +4000106
ADRES   +4
REG.:          -25          +43        +0     +1000        +0        +0        +0         -652      +24 FALSE

INSTR.:  +32          +31204
ADRES+1000
@          +1122
REG.:   -222816399          +43        +0     +1000        +0        +0        +0         -974      +24 FALSE

INSTR.:  +33       +15000506
ADRES  +15
REG.:         -974          +43        +0     +1000        +0        +0        +0   -222816399      +24 FALSE

INSTR.:  +34       +10030063
```

f) ADD en SUB worden getest met:

```
0000030501    26: ADD     0,3          "A= -2244002244
1000001202    27: SUB     1000(1:2)"A= -2244003366
```

g) MUL, DIV en SHIFT worden getest met:

```
0000034503    28: MUL     0,3(4:5)
1001000533    29: ST3     1001
1001000504    30: DIV     1001
0004000106    31: SRA     4
0000031204    32: DIV     0,3
0015000506    33: SRC     15
```

h) INCR en DECR worden getest met:

```
0010030063    34: INC3    10,3
0000030163    35: DEC3    0,3       "REG(3):=0
```

i) CMPR testen we met:

```
1000000573    36: CMP3    1000      "COMP:=1
```

j) MOVE en CHAR testen we tenslotte met:

```
1100000261    37: ENT1    1100
1000000207    38: MOVE    1000(2)
1100000510    39: LDA     1100
0013000105    40: CHAR    13
0000000205    41: HLT
0000000000
```

De auteur is dank verschuldigd aan de heer M. Rem voor de hulp bij de tot standkoming van de MIX simulator.

```
ADRES+1010
REG.:        -974        +43        +0        +2010        +0        +0        +0   -222816399        +24 FALSE 6

INSTR.:  +35        +30163
ADRES+2010
REG.:        -974        +43        +0           -0        +0        +0        +0   -222816399        +24 FALSE 6

INSTR.:  +36   +1000000573
ADRES+1000
6  -1122001122
REG.:        -974        +43        +0           -0        +0        +0        +0   -222816399        +24 FALSE 6

INSTR.:  +37   +1100000261
ADRES+1100
REG.:        -974      +1100        +0           -0        +0        +0        +0   -222816399        +24 FALSE 6

INSTR.:  +38   +1000000207
ADRES+1000
REG.:        -974      +1102        +0           -0        +0        +0        +0   -222816399        +24 FALSE 6

INSTR.:  +39   +1100000510
ADRES+1100
6  -1122001122
REG.:  -1122001122     +1102        +0           -0        +0        +0        +0   -222816399        +24 FALSE 6

INSTR.:  +40     +13000105
REG.:   -101020200     +1102        +0           -0        +0        +0        +0     -1010202        +24 FALSE 6

INSTR.:  +41          +205
```

3.4.4. Een Print Programma *)

Dankzij de faciliteit van MIX om bij de uitvoering van elke instructie de inhoud van de registers en de inhoud van de aangewezen geheugencel uit te printen, konden we de MIX testen. Dit is eenzelfde soort faciliteit als de mogelijkheid op een echte computer om de instructies stap voor stap uit te laten voeren en om de inhoud van de registers zichtbaar te laten maken door lampjes.

Natuurlijk is dit niet de manier om uitvoer, output, te produceren. Daarvoor moeten we de instructie OUT gebruiken voor de printer.
Nu wil het ongeluk dat de printer slechts symbolen kan printen en dan ook nog slechts in porties van 120 tegelijk.
Er zullen dus speciale subroutines moeten komen voor
a) het printen van één symbool,
b) het printen van één getal.

3.4.4.1. De subroutine PRSYM

Voor het printen van één symbool moeten we onderscheid maken tussen:
a) de initialisatie van de routine PRSYM; deze routine noemen we PRSYMINIT
b) het plaatsen van een symbool in een buffer, waar 120 symbolen in kunnen, welke geleegd moet worden als hij vol is; deze routine heet PRSYM,
c) het afsluiten van het printproces; dit komt neer op het met spaties aanvullen van de buffer en deze te legen; deze routine heet PRSYMSLUIT.
Omdat PRSYM de kern is waar alles om draait beginnen we hiermee.
Er is een buffer van 24 woorden met de adressen BUF tot BUF+23.
Er is een teller BUFT welke aangeeft tot hoever de buffer gevuld is: dit zijn de eerste BUFT÷ 5 woorden en van het volgende woord de eerste (BUFT-BUFT÷5*5) velden.
Na vulling van een volgend veld wordt BUFT met één opgehoogd; blijkt BUFT de waarde 120 te hebben bereikt, dan wordt de buffer aan de printer aangeboden om te worden geprint.

---

*) Rapport CR 21, afl. 4

Wij zullen in een volgende paragraaf zien hoe voor een echte computer met vruchtgebruik kan worden gemaakt van meer dan één printbuffer.

De instructies worden in een kleine variant van de door Kruth gespecificeerde taal MIXAL geschreven. Op de specificaties van deze taal wordt later ingegaan.

3.4.4.1.1. Het MIXAL programma.

De in de marge geplaatste regelnummers horen niet bij het MIXAL programma maar zijn toegevoegd om gemakkelijker over het programma te kunnen praten.

```
1    PRSYM:         STJ  TERUG(0:2)      " Maak terugsprong in orde
2                   STA  SYMB            " Het te printen symbool bevindt
3                                        " zich in A; dit moet gered worden.
4    "We bepalen nu op welke plaats het te printen symbool
5    "in de buffer moet worden geplaatst:
6
7                   ENTA 0               ")A:= AX : G(= BUFT : 5)
8                   LDX  BUFT            ")X:= rest
9                   DIV  vijf            ")
10
11   "Het symbool moet nu in veld (X + 1 : X + 1) van
12   "de geheugencel met adres BUF + A worden geplaatst.
13   "Van de vele manieren kiezen wij de volgende:
14   "Gebruik register 1 om het adres te bewaren en
15   "voer de gemodificeerde (voorgebakken) instructie:
16   "STA BUF,1(X + 1 : X + 1) uit.
17   "Helaas is een instructie als REG[1]:= A onmogelijk.
18
19                  STA  hulp
20                  LD1  hulp
21
22   "Vervolgens berekenen we (X + 1) × 10 + (X + 1) = FF.
23
24                  INCX 1
25                  STX  hulp            " hulp:= F
26                  ENTA 10
27                  MUL  hulp            " AX:= A × G(= 10 × F)
28                  LDA  hulp            " A:= F
29                  STX  hulp            " hulp:= 10 × F
30                  ADD  hulp            " A:= A + G(= F + 10 × F = FF)
31                  STA  INSTR(4:4)
```

```
32
33      "De instructie INSTR is nu gebakken.
34
35                      LDA  SYMB              ")GEH[BUF,1]:=
36      INSTR:          STA  BUF,1(0:0)        ")          SYMB
37
38      "Nu moet BUFT met een worden opgehoogd:
39
40                      LD1  BUFT              " REG[1]:= BUFT
41                      INC1 1                 " REG[1]:= REG[1] + 1
42                      CMP1 honderd20         " REG[1] < 120?
43                      ST1  BUFT              " BUFT:= REG[1]
44                      JL   TERUG             " Zo ja, dan klaar
45                      STZ  BUFT              " BUFT:= 0
46                      OUT  BUF(18)           " Stuur buffer weg
47
48      "Door toevoeging van de laatste, door de eerste gemodificeerde
49      "instructie, is de subroutine klaar:
50
51      TERUG:          JMP  0
52
53      "Wel moeten nog de variabelen en de buffer een
54      "plaats toegewezen krijgen:
55
56      BUFT:           CON  0                 ")CON betekent: de constante.
57      vijf:           CON  5                 ")CON is niet een echte instructie
58      SYMB:           CON  0                 ")zoals ADD. Met CON kan
59      hulp:           CON  0                 ")elk getal in een geheugencel
60      honderd20:      CON  120               ")worden geplaatst
61      BUF:            CON  0
62                      CON  1
63                      CON  2                 " Er volgen nu nog 20 cellen
64
65                      CON  22
66                      CON  23
67
68      "De subroutines PRSYMINIT en PRSYMSLUIT volgen nu:
69
70      PRSYMINIT:      STJ  TERUGINIT(0:2)
71                      STZ  BUFT
72      TERUGINIT:      JMP  0
73
```

```
74      PRSYMSLUIT:     STJ   TERUGSLUIT(0:2)
75                      LD2   BUFT
76                      J2Z   TERUGSLUIT
77                      DEC2  120
78      OPNIEUW:        ENTA  66              " A:= interne representatie spatie
79                      JMP   PRSYM
80                      INC2  1
81                      J2N   OPNIEUW
82
83      "De verleiding was groot om de label OPNIEUW een
84      "regel lager te zetten. PRSYM verknoeit A.
85
86      TERUGSLUIT:     JMP   0
```

3.4.4.1.2. Het programma kritisch bezien

We maken nu enige kritische opmerkingen betreffende de details van de gekozen instructies.

3.4.4.1.2.1. Twee tellers

Betreffende de instructies op regels 7-9 kan opgemerkt worden dat we ook twee tellers hadden kunnen bijhouden: BUFT1 en BUFT2, de eerste lopend van 0 tot 23 en de ander van 1 tot 5. In aantal benodigde geheugenplaatsen zou dit inefficiënter zijn, efficiënter zou het echter met betrekking tot de tijd zijn (de DIV operatie is duur).

3.4.4.1.2.2. Gebruik van de schuifinstructie

In plaats van de instructie INSTR op regel 36 te "bakken" was de volgende techniek mogelijk geweest:

- Zet de reeds half gevulde buffercel in A;
- Schuif A één veld naar links;
- Plaats door middel van ADD het te printen symbool in het meest rechtse veld;
- Zet A in de betreffende geheugencel.

Het gevolg van deze techniek is dat de waarde van BUFT - BUFT+5*5 niet meer wordt gebruikt.

De regels 22 - 36 kunnen dan vervangen worden door:

```
22a     LDA     BUF,1
23a     SLA     1          "Het meest rechtse veld wordt 0.
24a     ADD     SYMB       "In dit veld wordt het symbool geplaatst.
25a     STA     BUF,1
```

Dit leverde een belangrijke vereenvoudiging op:
tijdwinst èn geheugenwinst door gebruikmaking van de schuifinstructie SLA. Merkwaardig is dat we ons kennelijk geen zorgen hoeven maken over de oude vulling van GEH[BUF,1]. noch over de volgorde waarin de symbolen worden opgeborgen, noch over het feit dat na vijf symbolen een volgende geheugencel aan de beurt is.

3.4.4.1.2.3. Betere jumpinstructie

Ook de instructies op de regels 40 - 45 zijn nog niet optimaal gekozen. Het doel van deze instructies is
a) BUFT één ophogen;
b) over de OUT instructie springen als de buffer nog niet vol is.
Op zich zouden we slechts met één register als bufferteller toekunnen, ware het niet dat we het gebruik van dit register dan voor de rest van het programma verbieden. Vandaar de geheugencel BUFT.
Nu is een sprong op het negatief, nul, of positief zijn van een register eenvoudiger uit te voeren dan een sprong op grond van een comparitie met een willekeurig getal (zoals 120).
Vandaar dat we liever tellen vanaf 120 naar nul of vanaf -120 naar nul. We kunnen dan met een "J1N" volstaan in plaats van "CMP1;JL". We onderzoeken daarom de mogelijkheid om BUFT te laten starten op -120 om hem bij 0 weer -120 te maken.
Het resultaat van de instructies op de regels 7 - 20 is dan dat register 1 een waarde krijgt die in een verkeerde cadans verspringt
(: -24, -23, -23, -23, -23, -23, -22,...) . Starten we BUFT met de waarde -119, dan gaat het precies goed; d.w.z. register 1 krijgt de waarden:
-23, -23, -23, -23, -23,....,0, 0, 0, 0, 0.

De bijbehorende waarde van X is dan: -4, -3, -2, -1, 0,..., zodat de instructies 24 - 36 niet ongewijzigd kunnen blijven.
De instructies 22a - 25a komen er dan als volgt uit te zien:

```
22b     LDA       BUF+23,1 *)
23b     SLA       1
24b     ADD       SYMB
25b     STA       BUF+23,1 *)
```

Terwijl de instructies 40 - 45 de volgende gedaante krijgen:

```
40a     LD1       BUFT
41a     INC1      1
42a     ST1       BUFT
43a     J1NP      TERUG
44a     ENN1      119
45a     ST1       BUFT
```

Ook regel 71 moet gewijzigd worden in:

```
71a     ENN1      119
72a     ST1       BUFT
```

En de regels 75 - 81 moeten gewijzigd worden in:

```
75a     ENN2      119
76a     CMP2      BUFT
77a     JE        TERUGSLUIT
78a     LD2       BUFT
79a OPNIEUW: ENTA   66
80a     JMP       PRSYM
81a     INC2      1
82a     J2NP      OPNIEUW
```

De conclusie is dat het netto effect niet zit in geheugenwinst, we zijn er zelfs twee plaatsen op achteruitgegaan. Wel hebben we tijdwinst geboekt omdat de instructies 40 - 44 steeds voor elk symbool herhaald moeten worden en van deze instructies hebben we er één af kunnen halen.

De instructies 44a en 45a worden slechts één op de 120 keer uitgevoerd terwijl de instructies van PRSYMINIT en PRSYMSLUIT slechts één maal per programma worden uitgevoerd.
De moraal van bovenstaande beschouwingen is deze:
Onderzoek de instructies die de meeste keren worden gebruikt en ga na of hier tijdwinst is te boeken eventueel ten koste van geheugenwinst of tijdwinst voor andere instructies.

3.4.4.1.2.4. Bijzondere situatie uitzonderen?

We gaan ook nog na of de instructies 22a - 25a zijn te verbeteren. Eén op de vijf keer wordt er voor niets geschoven namelijk wanneer X = 0 (of in de 40a - 45a versie wanneer X = -4).
We zouden 22a - 25a nu kunnen wijzigen in:

```
22c          ENTA    0
23c          JXZ     TELOP
24c          LDA     BUF,1
25c          SLA     1
26c  TELOP:  ADD     SYMB
27c          STA     BUF,1
```

of in:

```
22d          LDA     SYMB
23d          JXZ     STORE
24d          LDA     BUF,1
25d          SLA     1
26d          ADD     SYMB
27c  STORE:  STA     BUF,1
```

Om een keus te maken moeten we weten hoeveel tijd de verschillende instructies in beslag nemen. Een tabel van Knuth leert ongeveer het volgende:
De jump en absolute instructies (ENT, ENN, INC en DEC) duren 1 tijdseenheid (te) de overige instructies 2 te met als belangrijke uitzonderingen MUL: 10 te en DIV: 12 te.

Aldus kunnen we een schatting maken van de benodigde rekentijd voor 1000 symbolen.
Voor de instructies 22a - 25a : 1000 * 8 = 8000 te.
Voor de instructies 22b - 27b : 800 * 10 + 200 * 6 = 9200 te.
Voor de instructies 22d - 27d : 800 * 11 + 200 * 5 = 9800 te.

We merken op dat de cijfers net een ander beeld zouden geven wanneer we, zoals heel in het begin van PRSYM, het symbool nog in A hadden.

3.4.4.1.2.5. Fout detectie

Er kleeft nog een ernstig mankement aan PRSYM: hij test niet of het in A meegegeven symbool wel netjes is zodat een test vooraf gewenst is:

```
JAN     ALARM
CMPA    honderd
JGE     ALARM
```

In dit geval is ALARM een label van het programma dat als nooduitgang voor moeilijke situaties dienst doet.
Een andere meer rigoureuse mogelijkheid zou zijn:

```
STZ     SYMB
STA     SYMB(5:5)
```

3.4.4.1.3. Algemene opmerkingen n.a.v. het programma

Bij het maken van subroutines komt altijd het probleem naar voren: hoe behandelen we de registers.
Bezien we de aanroep van PRSYM in de routine PRSYMSLUIT, dan merken we op dat we zeer zorgvuldig A steeds weer opnieuw de waarde 66(spatie) geven en dat we register 2 als teller hebben gebruikt. (in plaats van register 1).
Geen wonder, immers A verliest zijn waarde in PRSYM en niet alleen A, ook de registers 1 en X, èn (althans in de oude versie) de register COMP.
Bij sommige computers (als de Philips-Electrologica X8) is het gebruikelijk dat een subroutinesprong gepaard gaat met het vooraf redden van, niet

*) *De later te bespreken MIXAL assembler eist dat BUF niet na maar voor de subroutine staat.*

de instructieteller, maar ook de "kleine" registers zoals hier COMP en OVERFLOW. Dit is gemakkelijk uit te voeren omdat in de geheugencel waar de instructieteller opgeborgen wordt toch plaats genoeg is.
Uiteraard gaat het terugspringen uit de subroutine gepaard met het herstellen van die kleine registers.

In ons geval staan we voor de keus:

a) bij binnenkomst van de subroutine de registers A, X en 1 redden d.m.v.:

```
PRSYM:  STJ  TERUG(0:2)
        STA  SYMB
        STX  REDX
        ST1  RED1
```

en voor terugkeer de registers te herstellen door:

```
        LDA  SYMB
        LDX  REDX
        LD1  RED1
TERUG:  JMP   0
```

b) de registers worden niet gered maar er moet rekening gehouden worden met dit feit.
In dit geval is het van groot belang in een begeleidend commentaar te vermelden welke registers worden gebruikt en welke andere subroutines worden gebruikt, zodat het niet nodig is om alle instructies van de routine na te gaan.

3.3.4.1.4. <u>Wensen ten aanzien van een assembler</u>

Er zijn een aantal wensen die rijzen n.a.v. de besproken programma's:

1. Zonder dat we er erg in hebben is een groot aantal namen ingevoerd. Deze moeten natuurlijk alle verschillend zijn. Het is niet eenvoudig steeds weer nieuwe namen te bedenken. Daarom bestaat in MIXAL de volgende mogelijkheid:
Als label mag men een onbeperkt aantal keren gebruik maken van de 10 wijzers: 0H, 1H, 2H,...,9H.
Vanuit het "adresdeel" van een instructie kan men nu 20 verschillende instructies aanwijzen. Men kan 10 voorgaande instructies aanwijzen d.m.v. 0B, 1B, 2B,...,9B, dit zijn de instructies die aangewezen worden door

de 10 laatste, door voorafgaande wijzers aangewezen, instructies. Men kan 10 volgende instructies aanwijzen d.m.v. 0F, 1F, 2F,...,9F, dit zijn de instructies die aangewezen worden door de 10 dichtstbijzijnde, door volgende wijzers aangewezen instructies.
Hoe eigenaardig ook, de regel geldt dat de eigen instructie telt als een volgende instructie.
Wil men de eigen instructie aanwijzen dan wordt het symbool "*" gebruikt.
In tekening gebracht (slechts voor één wijzer 1H):

```
                              equivalent met:
1H: -                         A: -
1H: - 1B                      B: - A
    - 1F                         - C
1H: - *                       C: - *
```

Een andere, in navolging van ALGOL 60 gekozen, methode is een blokstructuur met <u>begin</u>, declaraties en <u>end</u> in te voeren, zodat namen slechts een locale betekenis hebben.

2. De assembler moet expressies, zoals voorkomend op regel 22b en 25b, i.c. BUF+23, kunnen ontcijferen.

3. De assembler moet zelf hulpvariabelen als hulp, SYMB en BUFT een plaats in het geheugen toewijzen.

4. In plaats van de regels 61 - 66, moet het mogelijk zijn in één regel te vermelden dat de buffer BUF 24 woorden lang is.
   In MIXAL bestaat hiertoe de mogelijkheid d.m.v. de <u>pseudo instructie</u> ORIG.

   ```
           ORIG   3000
   ```

   betekent: plaats de volgende instructie op adres 3000.

   ```
   BUF:    CON    0
           ORIG   BUF+24
   ```

   betekent: plaats de volgende instructie op adres BUF+24, zodat er 24 plaatsen voor de buffer zijn gereserveerd.
   We merken op dat ook CON een soort pseudoinstructie is, welke echter in tegenstelling tot ORIG, zorg draagt voor de vulling van een geheugencel.

5. De assembler moet voor constanten als "vijf" en "honderd20" zelf geheugenplaatsen reserveren en deze op 5 en 120 initialiseren. In MIXAL worden deze constanten genoteerd tussen twee symbolen "=".
   Dus regels 9 en 42 zouden luiden:

   ```
               DIV =5=
   en          CMP1 =120=
   ```

3.4.4.2. De subroutine PRNUM

Met behulp van de in de vorige secties besproken subroutine PRSYM maken we nu een subroutine voor het printen van een getal dat in het register A is meegegeven.
De 10 decimalen van het getal worden in de 10 velden van de geheugencellen D1 en D2 opgeborgen. Eerst wordt een teken geprint vervolgens 5 decimalen, dan een spatie, vervolgens 5 decimalen, tenslotte weer een spatie.

```
1    PRNUM:   STJ  TERUGPRNUM(0:2)
2             STA  GETAL              " GETAL:= getal
3             JAN  1F                 ")if A > 0 then
4             ENTA 37                 ")   A:= plus
5             JMP  2F
6    1H:      ENTA 38                 " else A:= minus
7    2H:      JMP  PRSYM
8    "Vervolgens worden D1 en D2 gevuld
9
10            LDA  GETAL
11            CHAR
12            STA  D1
13            STX  D2
14
15   "In register 2 worden de vijf velden afgeteld; register 3
16   "gebruiken we om D1 of D2 aan te geven.
17
18            ENT3 0
19   2H:      ENT2 5
20   1H:      LDA  D1,3(1:1)          " Print het voorste symbool
21            JMP  PRSYM
22            LDA  D1,3(1:5)          " A:= D1
23            SLA  1                  " Schuif A een naar links
24            STA  D1,3               " D1:= A
25
26   "We testen of we er 5 gehad hebben:
27
28            DEC2 1
29            J2P  1B                 " Nu pas vullen we 1H in
```

```
30
31        "We printen een spatie:
32                            ENTA 66
33                            JMP  PRSYM
34
35        "Wellicht zijn we klaar:
36
37                            J3P  TERUGPRNUM
38
39        "Als dit niet het geval is dan herhalen we de riedel:
40
41                            ENT3 1
42                            JMP  2B              " Nu vullen we 2H in
43
44        "N.B. Dank zij de modificatie D1,3 nemen we
45        "de tweede keer D2.
46
47        TERUGPRNUM:         JMP  0
48        GETAL:              CON  0
49        D1:                 CON  0
50        D2:                 CON  0
```

Opmerking: de subroutine kan aanzienlijk efficiënter worden beschreven.

3.4.5. Een geavanceerd buffersysteem

In sectie 3.4.4. werden printprocessen beschreven die gebaseerd zijn op de gedachte dat een OUT instructie meteen wordt uitgevoerd, d.w.z. de volgende instructie wordt pas dan uitgevoerd wanneer alle 120 symbolen geprint zijn. Zo werkt ook de ALGOL 60 MIX simulator. In werkelijkheid duurt het printen van de 120 symbolen erg lang vergeleken met de snelheid van het CRO en het zou jammer zijn om het CRO te laten wachten op het gereedkomen van het printen.
In sommige gevallen kan dat niet anders; wanneer er nauwelijks rekenwerk is te verrichten maar er is veel input/output, dan moet het CRO wachten.

De oplossing kan namelijk gevonden worden als de computer aan een gevarieerde klantenkring service moet verlenen. Sommige klanten vragen veel rekenwerk, sommige vragen veel input/output. De kunst is nu de computer zo te programmeren dat hij simultaan zowel het zware rekenwerk van de ene klant als het input/output werk van de andere klant verzorgt.

Het is dan nodig dat de computer over veel achtergrondgeheugen (magnetische trommels, magnetische schijven) beschikt zodat hij een teveel aan input en output hierop kwijt kan.

Teneinde enig inzicht te krijgen in de problematiek van input/output buffers, en hoe deze efficient bespeeld kunnen worden volgt hieronder een programma dat met een willekeurig aantal buffers symbolen print.
Wij moeten onderscheid maken tussen twee processen:

1. het proces dat een buffer vult met symbolen en
2. het proces dat een buffer naar de printer stuurt.

Het vulproces verloopt als volgt:

Als er een niet-volle buffer aanwezig is dan wordt deze opgevuld. Blijkt hij vol te zijn dan wordt hij ter leging naar de printer gestuurd.
Als er geen niet-volle buffer aanwezig is, dan wordt gekeken of er een lege buffer is welke dan weer gevuld kan worden; blijkt er geen lege buffer te zijn dan moet er gewacht worden, waarbij wel steeds het leegproces moet worden geactiveerd.

Het leegproces verloopt als volgt:

Als de printer bezig is dan valt er niets te legen.
Als de printer klaar gekomen is met een buffer dan wordt de volgende buffer bekeken; is deze vol dan wordt hij naar de printer gestuurd.

We gebruiken een ketting van buffers BUFi. De ketting ontstaat door te eisen dat (Wij nemen 3 buffers):

abs(GEH[BUF1]) = BUF2
abs(GEH[BUF2]) = BUF3
abs(GEH[BUF3]) = BUF1

Van elke buffer moeten we kunnen aangeven of hij vol is of leeg. Hiervoor gebruiken we het teken van GEH[BUFi]; d.w.z. als het teken +1 is, dan is de buffer leeg, als het teken -1 is, dan is de buffer vol.

We gaan nu na wanneer het teken omgezet moet worden. Aanvankelijk zijn alle buffers leeg dus de tekens zijn dan alle +1. Op het moment dat een buffer vol is geraakt moet het teken op -1 gezet worden (de buffer wordt op slot gezet). Dit is dus duidelijk een taak van het vulproces. Vervolgens wordt een transport gestart, zo er geen transport aan de gang was tenminste, dit gebeurt door het leegproces uit te voeren.
De vraag komt nu aan de orde: wie zet het teken weer op +1 (van 't slot af) en wanneer?
Het is duidelijk dat dit moet gebeuren meteen nadat de buffer geleegd is. Dit wil zeggen dat, wanneer blijkt dat de printer niet bezig is, er onderzocht moet worden of de printer juist klaar gekomen is met het legen van een buffer. M.a.w. het leegproces moet alvorens met het legen van eén buffer te beginnen ergens noteren (in BUFLEEG) welke buffer aan de printer is gegeven; het leegproces kan dan bovendien als taak krijgen om, in het geval dat de printer niet bezig is, de door BUFLEEG aangewezen buffer weer van 't slot af te zetten en BUFLEEG naar de volgende buffer te laten wijzen.
BUFLEEG krijgt een positieve waarde indien hij wijst naar een naar de printer gestuurde buffer; hij krijgt een negatieve waarde indien hij slechts wijst naar een volgende eventueel nog te printen buffer.
De algorithmen tekenen zich nu duidelijk af:

```
procedure VULBUF;
begin A: if door BUFVUL aangewezen buffer in op slot then
         begin   if printer is klaar then LEEGBUF;
                 BUFVUL := abs(GEH[BUFVUL]);
                 goto A
         end else
         begin   zet symbool in de door BUFVUL aangewezen buffer;
                 BUFT := BUFT+1;
                 if  BUFT = 120 then
                 begin zet de buffer op slot;
                       BUFT := 0;
                 if printer is klaar then LEEGBUF
end  end  end  VULBUF;
```

```
procedure LEEGBUF;
begin comment LEEGBUF wordt slechts dan aangeroepen wanneer
      de printer klaar is;
      if BUFLEEG > 0 then
      begin Zet de door BUFLEEG aangewezen buffer van slot af;
            BUFLEEG := -abs(GEH[BUFLEEG])
      end;
      if door abs(BUFLEEG) aangewezen buffer is op slot then
      begin BUFLEEG := abs(BUFLEEG);
            OUT buffer(18)
end  end  LEEGBUF;
```

Met deze voorstudie in ALGOL 60 als achtergrond is het niet moeilijk om de MIXAL subroutines te schrijven.

```
1       PRSYM2:          STA   SYMB
2
3       VULBUF:          STJ   TERUGVULBUF(0:2)
4       1H:              LD1   BUFVUL               " REG[1]:= BUFVUL
5                        LD2   0,1                  " REG[2] is de inhoud van
6                                                   " de eerste buffercel
7                        J2P   NIETOPSLOT
8
9       "De door REG[1] aangewezen buffer is op slot zodat
10      "de printer eventueel geactiveerd kan worden. Eerst
11      "doen we REG[2]:= -REG[2].
12
13                       ENN2 0,2
14                       ST2   BUFVUL               " BUFVUL:= -GEH[BUFVUL]
15                       JRED LEEGBUF(18)
16                       JMP   1B
17
18      "We hadden ook de waarde van BUFVUL ongewijzigd kunnen
19      "laten en in plaats daarvan de instructies:
20      "                ENN1 0,2
21      "                JRED LEEGBUF(18)
22      "                JMP   1B + 1
23      "kunnen uitvoeren waarbij we dan zouden uitgaan van
24      "de veronderstelling dat LEEGBUF niet aan REG[1]
25      "knoeit. Het wel gekozene is echter veiliger.
26
27      NIETOPSLOT:      ENTA 0                     ")
28                       LDX   BUFT                 ")A:= AX : 5
29                       DIV   =5=                  ")X:= rest
```

```
30
31        "Voor het vullen van een woord met een symbool
32        "kiezen we uiteraard de instructie vlgs 22a - 25a.
33        "Eerst moet de geheugencel bepaald worden uit
34        "de waarde van REG[1], die het begin van de buffer aanwijst
35        "en A die de betreffende cel in de buffer aanwijst.
36
37                         STA  hulp              ")
38                         LD2  hulp              ")REG[2]:= A
39                         INC2 1,1               ")REG[2]:= A + 1 + REG[1]
40
41        "De betreffende geheugencel wordt aangegeven door REG[2].
42
43                         LDA  0,2
44                         SLA  1
45                         ADD  SYMB(5:5)         " Voor de veiligheid
46                         STA  0,2
47
48        "Nu moet BUFT met 1 worden opgehoogd en er moet
49        "gekeken worden of er een buffer vol is.
50
51                         LD2  BUFT              ")
52                         INC2 1                 ")
53                         ST2  BUFT              ")BUFT:= BUFT + 1
54                         CMP2 =120=             ")if BUFT < 120 then
55                         JL   TERUGVULBUF       ")goto klaar
56
57                         STZ  BUFT              " BUFT:= 0
58
59        "De buffer aangewezen door REG[1] moet op slot worden
60        "gezet, door de eerste cel negatief te maken.
61
62                         LD2N 0,1               " REG[2]:= -GEH[REG[1]]
63                         ST2  0,1               " GEH[REG[1]]:= REG[2]
64        "Nu moet de printer nog geactiveerd worden:
65
66                         JRED LEEGBUF(18)
67
68        TERUGVULBUF:     JMP  0
69        SYMB:            CON  0
70        BUFVUL:          CON  x + 1
71        1H:              CON  x + 50
72                         ORIG x + 24
73        1H:              CON  1B
74                         ORIG x + 24
75                         CON  1B
76                         ORIG x + 24
77
78        BUFT:            CON  0
79
80        "N.B. Slechts op deze plaats blijkt dat er drie
81        "buffers zijn.
82
```

```
83    LEEGBUF:        STJ  TERUGLEEGBUF(0:2)
84                    LD1  BUFLEEG            " REG[1]:= BUFLEEG
85                    J1N  1F
86
87    "Zet de door BUFLEEG aangewezen buffer van slot af:
88
89                    LD2  0,1(1:5)           " REG[2]:= abs(GEH[REG[1]])
90                    ST2  0,1                " GEH[REG[1]]:= REG[2]
91                    ENN1 0,2                " REG[1]:= -REG[2]
92                    ST1  BUFLEEG            " BUFLEEG:= -abs(GEH[BUFLEEG]]
93
94    "Het register 1 is nu negatief.
95
96    1H:             ENN1 0,1                " REG[1]:= -REG[1]
97                    LD2  0,1                " REG[2]:= GEH[REG[1]]
98                    J2P  TERUGLEEGBUF       " niet op slot, dan klaar
99
100   "Er is een volle buffer die geprint wordt:
101
102                   ST1  BUFLEEG
103                   OUT  1,1(18)
104
105   TERUGLEEGBUF:   JMP  0
106   BUFLEEG:        CON  - BUFVUL - 1
```

De subroutine PRSYMSLUIT moet nu zodanig veranderd worden dat, ten eerste het register 2 gered wordt en ten tweede de printer zolang geactiveerd wordt tot alle buffers leeg zijn. De volgende instructies zijn geschikt om de tweede activiteit uit te voeren:

```
JBUS   * (18)
JMP    LEEGBUF
JBUS   * -2 (18)
```

3.4.6. Een goede computer heeft interrupt mogelijkheden

Een kritische beschouwing van de vorige sectie leert ons dat het systeem niet waterdicht is. Wat kan er gebeuren:
Aan PRSYM worden in hoog tempo 3 * 120 symbolen aangeboden als gevolg waarvan de printer één keer aan de gang wordt gezet om de eerste 120 symbolen te printen.
Vervolgens is het CRO een zeer intensief rekenwerk begonnen zodat de subroutine LEEGBUF niet wordt aangeroepen, zodat geen maatregelen worden getroffen op het klaar komen van het printen van de eerste 120 symbolen,

zodat de andere 240 symbolen voorlopig ongeprint blijven, zodat, bij het klaarkomen van het intensieve rekenwerk waarna bijvoorbeeld weer 360 symbolen moeten worden geprint, het CRO moet wachten tot de printer gereed is met het printen van de van vroeger overgebleven 240 symbolen.
De enige manier om dit probleem te omzeilen is zo nu en dan tijdens het intensieve rekenwerk de instructie JRED uit te voeren.

Op een moderne computer is deze zaak anders geregeld. Daar is het niet het CRO dat kijkt, d.m.v. een instructie JRED, of een apparaat klaar is, integendeel: het is het apparaat dat als het klaar is gekomen zich afmeldt bij het CRO.
Dit betekent dat het CRO geïnterumpeerd wordt tijdens de uitvoering van een programma. Na afwerking van de in bewerking zijnde instructie wordt dan namelijk een speciale instructie uitgevoerd.
De speciale instructie kan de volgende gedaante hebben:

```
JMP    0
```

Als men dan op geheugenplaats 0 de instructie

```
JSJ    INTERRUPTROUTINE
```

plaatst, dan kan men vanaf de vrij te kiezen geheugenplaats INTERRUPTROUTINE een routine plaatsen die:

1° zorgt dat alle registers worden gered,
(Merk op dat het register J, al verknoeit is door de instructie JMP 0; feitelijk is de MIX instructieset dan ook te arm voor een MIX met interrupts)

2° passende maatregelen neemt op grond van de speciale interrupt (waarvoor vaak aparte registers aanwezig zijn),

3° het CRO weer "horend" (ook wel: ontvankelijk voor interrupts) maakt.

Het laatste is nodig omdat met de uitvoering van de speciale instructie het CRO "doof" wordt voor interrupts. Het is wel duidelijk dat dit nodig is.

4. De werking van een assembler geschreven in ALGOL 60

4.1. Opmerking vooraf

In dit hoofdstuk wordt de VERAL assembler besproken, welke in ALGOL 60 is opgeschreven.
De belangrijke reden voor het behandelen van deze assembler is dat het één van de meest eenvoudige assemblers is die men zich kan denken, waaraan echter de werking van het syntactisch lezen van tekst voortreffelijk gedemonstreerd kan worden.
Een logisch vervolg van dit hoofdstuk is de behandeling van de MIXAL assembler. Daarvoor moeten we echter verwijzen naar een MC rapport van G. ten Velden dat hierover binnenkort zal verschijnen.

4.2. De VERAL assembler

De taal VERAL wordt in het ALGOL 60 programma gedefinieerd. Het is een rechtstreekse afspiegeling van de VERA machinetaal. Bovendien is VERAL een subset van MIXAL.

begin **comment** VERAL assembler. R.P. v.d. Riet
opdrachtnummer 2133, codenummer RPR 190970.

De syntactische definitie van een VERAL programma
is als volgt:

```
<VERAL programma>::= <regel> | <VERAL programma> <regel>
<regel>::= <VERAL instructie> <scheider>
<VERAL instructie>::= <eventuele label locatie>
                      <eventuele kale instructie>
<eventuele label locatie>::= <leeg> | <naam> :
<eventuele kale instructie>::= <leeg> | <kale instructie>
<kale instructie>::= <pseudo instructie> | <machine instructie>
<pseudo instructie>::= <pseudo mnemonic> <eventuele expressie>
<machine instructie>::= <machine mnemonic> <eventuele expressie>
<eventuele expressie>::= <leeg> | <een of meer spaties> <expressie>
<expressie>::= <term> | <expressie> <binaire operator> <term>
<term>::= <atoom> | <unaire operator> <atoom>
<atoom>::= <naam> | <getal>
<scheider>::= ; | <twnr> | " <commentaar> <twnr>
<leeg>::=
<pseudo mnemonic>::= CON | ORIG | END
<machine mnemonic>::= LDA | STA | ADD | SUB | JAN | JMP | PRI | HLT
<binaire operator>::= + | -
<unaire operator>::= + | -
<naam>::= <letter> | <naam> <letter> | <naam> <digit>
<getal>::= <digit> | <getal> <digit> | <getal> <spatie>
<letter>::= a | b |...| y | t | A | B |...| Y | Z
<digit>::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

De laatste regel van een VERAL programma bevat de pseudo mnemonic END gevolgd door een expressie, waarvan de waarde het beginadres aangeeft waar het programma bij executie moet worden gestart.

De strategie van het leesproces kan gebaseerd worden op de volgende opvattingen over de tekst van een programma:

a) De tekst is een rij symbolen (zie F.E.J. Kruseman Aretz: ALGOL 60 translation for everybody, Electronische Datenverarbeitung Heft 6/1964, pp. 233-244.)
b) De tekst wordt opgesplitst in een rij van syntactische eenheden, welke elk voor zich bestaat uit een rij symbolen.

Voordelen en nadelen komen nu niet ter discussie. Wij gaan uit van de tweede methode.
Een syntactische eenheid wordt gelezen door de aanroep van de procedure Lees synt eenheid, waarna de betekenis van de gelezen syntactische eenheid als waarde wordt achtergelaten in de variabele synt eenheid.
Deze betekenis kan zijn: scheider, naam, getal, plus operator of minus operator.
We merken op dat Lees synt eenheid geen verschil ziet tussen een pseudo mnemonic, machine mnemonic of een naam.
In alle drie gevallen wordt de betekenis: naam in synt eenheid afgeleverd.

De stukken tekst behorende bij de "hogere" syntactische variabelen: VERAL programma, ... , term, kunnen ook als rijen van syntactische eenheden worden opgevat.
De procedures Lees programma, ... , Lees term, welke de corresponderende syntactische variabelen moeten herkennen zijn nu zodanig geconstrueerd dat:
a) bij ingang van de procedure blok is de eerste syntactische eenheid van de (niet leeg veronderstelde) tekst behorende bij de corresponderende syntactische variabele reeds gelezen.
b) bij uitgang van de procedure blok is de eerstvolgende syntactische variabele niet meer behorend tot genoemde tekst reeds gelezen.
De belangrijke reden voor deze strategie is dat het einde van een expressie bepaald wordt door de syntactische eenheid welke niet meer behoort tot die expressie.
Nu volgen de leesprocedures:;

```
procedure Lees programma;
begin laatste regel:= false;
L: Lees regel;
   if ¬ laatste regel then goto L else
   begin Beeindig assembleer proces;
      BASIS CYCLUS
end end Lees programma;

procedure Lees regel;
begin integer i;
   comment Alvorens de regel te gaan lezen wordt het regelnummer
      geprint:; PRINT regelnummer;
   Lees instructie; PRINT regel;
   ERROR(synt eenheid ≠ scheider, ⪡regel niet correct afgesloten⪢);
   Lees synt eenheid
end Lees regel;

procedure Lees instructie;
begin Lees eventuele label locatie;
   Lees eventuele kale instructie;
end Lees instructie;

procedure Lees eventuele label locatie;
if synt eenheid = naam then
begin Skip layout; if symbool = dubbele punt then
   begin ERROR(oude naam,⪡twee keer zelfde naam⪢);
      BERG OP(plaats van naam,instr teller);
      Lees symbool; Lees synt eenheid
end end Lees eventuele label locatie;

procedure Lees eventuele kale instructie;
if synt eenheid = naam then
begin integer adres,type; type:= type van naam;
   Lees synt eenheid;
   adres:= Lees eventuele expressie;
   if type < 8 then
LDA: begin ERROR(abs(adres) > 3999,⪡adres te groot⪢);
      GEH[instr teller]:= (if adres < 0 then -1 else +1) ×
         (abs(adres) × 10 + type);
```

```
        PRINT instructie;
        instr teller:= instr teller + 1
      end else
      if type = 8 then
CON:  begin GEH[instr teller]:= adres; PRINT instructie;
        instr teller:= instr teller + 1
      end else
      begin PRINT instr teller; instr teller:= adres;
        if type = 10 then
END:  laatste regel:= true else
ORIG: ERROR(type ≠ 9,⪡type van mnem niet OK⪢)
end end Lees eventuele kale instructie;

integer procedure Lees eventuele expressie;
begin real expr;
  expr:= if is optel operator(synt eenheid) ∨
  synt eenheid = naam ∨ synt eenheid = getal
  then Lees term else 0;
L: if is optel operator(synt eenheid) then
  begin expr:= expr + Lees term; goto L end;
  ERROR(abs(expr) > 99999,⪡expressie te groot⪢);
  Lees eventuele expressie:= expr
end Lees eventuele expressie;

integer procedure Lees term;
L: if synt eenheid = plus operator then
begin Lees synt eenheid; goto L end else
if synt eenheid = minus operator then
begin Lees synt eenheid; Lees term:= — Lees term end else
if synt eenheid = naam then
begin ERROR(¬ oude naam,⪡waarde van naam onbekend⪢);
  Lees term:= waarde van naam; Lees synt eenheid
end else
if synt eenheid = getal then
begin Lees term:= waarde van getal; Lees synt eenheid end else
Lees term:= ERROR(true,⪡foute term⪢);

comment Het moeilijke werk is klaar, nu volgen nog de hulpprocedures:

. Lees synt eenheid, voor het lezen van een syntactische eenheid
  welke de waarde: scheider, naam, getal, plus operator of minus
  operator kan hebben,
. Skip layout, welke layout symbolen skipt,
. Lees symbool, welke een symbool leest,
. PRINT regel, PRINT instr teller, PRINT instructie,
. Beeindig assembleer proces,
. ERROR,
. BERG OP, welke op het "adres van(de gelezen) naam" in de
  naamlijst de waarde ervan opbergt,
en:;

Boolean procedure is optel operator(s);
is optel operator:= s = plus operator ∨ s = minus operator;
```

```
procedure Skip layout;
begin integer i; for i:= symbool while i = spatie do
  Lees symbool
end Skip layout;

procedure Lees symbool;
begin comment Eerst zetten we het oude symbool in het array
  "te printen regel" welke in zijn geheel aan het eind van een
  regel wordt afgedrukt.;
  if symbool ≠ twnr then te printen regel[positie]:= symbool else
  begin regelnummer:= regelnummer + 1;
    positie:= voorraad spaties:= 0
  end;
  positie:= positie + 1; ERROR(positie > 100,⟨regel te lang⟩);
  if voorraad spaties > 0 then
  begin voorraad spaties:= voorraad spaties - 1;
    symbool:= spatie
  end else
  begin symbool:= RESYM;
    comment we lezen 1 symbool van band of kaart;
    if symbool = tab then
    begin voorraad spaties:= voorraad spaties + 8 -
      (positie - positie : 8 × 8);
      symbool:= spatie
end end end Lees symbool;

integer procedure Lees synt eenheid;
begin Skip layout;
  if symbool = puntkomma ∨ symbool = twnr then
  begin Lees symbool; synt eenheid:= scheider end else
  if symbool = aanhalingsteken then
  begin integer i; for i:= symbool while i ≠ twnr do Lees symbool;
    Lees symbool; synt eenheid:= scheider
  end else
  if is letter(symbool) then
  begin Lees naam; synt eenheid:= naam end else
  if is digit(symbool) then
  begin Lees getal; synt eenheid:= getal end else
  if symbool = plus then
  begin Lees symbool; synt eenheid:= plus operator end else
  if symbool = minus then
  begin Lees symbool; synt eenheid:= minus operator end else
  begin ERROR(true,⟨synt eenheid fout⟩);
    symbool:= aanhalingsteken; Lees synt eenheid
  end;
  Lees synt eenheid:= synt eenheid
end Lees synt eenheid;

procedure Lees getal;
begin waarde van getal:= symbool;
L: Skip layout; Lees symbool;
  if is digit(symbool) then
  begin waarde van getal:= 10 × waarde van getal + symbool;
    goto L
end end Lees getal;
```

```
comment De vorm van de volgende procedures is niet de
definitieve. In een volgende sectie komen we hierop terug.
Wij nemen hier aan dat een naam uit maximaal 8 symbolen
(letters of digits) bestaat.
Deze naam wordt opgeborgen in een naamlijst de integer array
"naamlijst", en wel als volgt:
Per naam wordt in de naamlijst een vaste hoeveelheid ruimte
gereserveerd (voor zover het deze sectie betreft), deze ruimte
noemen we een informatie cel.
De informatie cel bestaat uit de array elementen:
    naamlijst [plaats van naam + j], j = 0,1,2,3
    met j = 0,1 voor de symbolen,
    met j = 2 voor het type,
    en  j = 3 voor de waarde.
Behalve de door de programmeur gekozen namen bevat de naamlijst
ook de mnemonics met hun type.
Bij de initialisatie moet de naamlijst dus met de mnemonics
gevuld worden.;

procedure Lees naam;
begin integer i,j,eerste helft,tweede helft;
  eerste helft:= tweede helft:= j:= 0;
  for i:= symbool,symbool while is letter(i) ∨ is digit(i) do
  begin j:= j + 1; Lees symbool;
    if j < 4 then eerste helft:= eerste helft × 64 + i + 1 else
    if j ≤ 8 then tweede helft:= tweede helft × 64 + i + 1
  end;

comment Onder de namen zoeken we naar de zojuist gelezene.;

  for i:= 1 step 4 until laatste plaats do
  begin if eerste helft = naamlijst[i] ∧
           tweede helft = naamlijst[i + 1] then
    begin oude naam:= true;
      plaats van naam:= i;
      type van naam:= naamlijst[i + 2];
      waarde van naam:= naamlijst[i + 3];
      goto klaar
  end end;

  comment Als blijkt dat er nog geen naam was gelijk aan de
  gelezene, dan komen we hier terecht:;

  oude naam:= false;
  i:= plaats van naam:= laatste plaats:= laatste plaats + 4;
  ERROR(i + 3 > max van naamlijst,⟨naamlijst te klein⟩);
  naamlijst[i]:= eerste helft; naamlijst[i + 1]:= tweede helft;
  type van naam:= naamlijst[i + 2]:=
  waarde van naam:= naamlijst[i + 3]:= -1;
klaar:
end Lees naam;
```

```
procedure BERG OP(p,waarde); value p,waarde; integer p,waarde;
naamlijst[p + 3]:= waarde;

comment De volgende procedures verzorgen het printen van een
regel.;

procedure PRINT instr teller;
begin integer i; integer array decimaal[1:4];
  Boolean teken;
  Bereken decimalen van (instr teller) aantal:(4)
  berg ze op in: (decimaal) met het teken in:(teken);
  for i:= 1,2,3,4 do
  te printen regel[-21 + i]:= decimaal[i]
endPRINT instr teller;

procedure PRINT instructie;
begin integer i; integer array decimaal[1:5];
  Boolean teken; PRINT instr teller;
  Bereken decimalen van (GEH[instr teller]) het aantal is: (5)
  berg ze op in: (decimaal) met het teken in: (teken);
  te printen regel[-14]:= if teken then plus else minus;
  for i:= 1,2,3,4,5 do
  te printen regel[-14 + i]:= decimaal[i]
end PRINT instructie;

procedure PRINT regelnummer;
begin integer i; integer array decimaal[1:4];
  Boolean begin;
  Bereken decimalen van (regelnummer) n is: (4)
  stop ze in: (decimaal) met het teken in: (begin);
  begin:= true;
  for i:= 1,2,3,4 do
  if ¬ (decimaal[i] = 0 ∧ begin) then
  begin begin:= false; te printen regel[-6 + i]:= decimaal[i] end;
  if begin then te printen regel[-2]:= 0;
end PRINT regelnummer;

procedure PRINT regel;
begin integer i;
  NLCR;
  for i:= -20 step 1 until 100 do
  begin PRSYM(te printen regel[i]);
    te printen regel[i]:= spatie
end end PRINT regel;

procedure Bereken decimalen van(getal,n,arr,t);
  value getal,n; real getal; integer n; integer array arr;
  Boolean t;
begin integer i,s,tn;
  tn:= 10 ↑ (n - 1);
  t:= getal > 0; getal:= abs(getal);
  for i:= 1 step 1 until n do
  begin arr[i]:= s:= getal ÷ tn;
    ERROR(s > 9,⊀te printen getal fout⊁);
    getal:= getal - s × tn; tn:= tn/10
  end;
```

```
  ERROR(getal ≠ 0,<niet alle decimalen gekregen>)
end Bereken decimalen van;

comment De volgende procedures krijgen meer inhoud indien de
  VERAL assembler wordt ingebouwd als "START CRO" in de
  VERA simulator.;

procedure Beeindig assembleer proces;
begin integer i,j,t,w,s,k,m;
  NLCR; PRINTTEXT(<naamlijst>); NLCR;
  for i:= 1 step 4 until laatste plaats do
  begin k:= m:= 0;
  L: t:= 262144; w:= naamlijst[i + m];
    for j:= 1,2,3,4 do
    begin s:= w : t; w:= w - s × t; t:= t/64;
      if s ≠ 0 then
      begin PRSYM(s - 1); k:= k + 1 end
    end;
    if m = 0 then begin m:= 1; goto L end;
    for k:= k step 1 until 7 do PRSYM(spatie);
    FIXT(2,0,naamlijst[i + 2]);
    FIXT(10,0,naamlijst[i + 3]); NLCR;
end end Beeindig assembleer proces;

procedure BASIS CYCLUS; EXIT;

integer procedure ERROR(B,st); Boolean B; string st;
if B then
begin NLCR; PRINTTEXT(<FOUT >);
 PRINTTEXT(st); ERROR:= 1; EXIT
end ERROR;

comment Afgezien van de initialisatie en declaratie van de gebruikte
variabelen zijn we nu klaar.
We maken eerst een lijst van alle nog niet gedeclareerde variabelen:;

integer synt eenheid,scheider,naam,symbool,dubbele punt, plaats van
naam,instr teller,type van naam,getal,plus operator,minus operator,
waarde van naam,waarde van getal,spatie,twnr,positie,voorraad spaties,
tab,puntkomma,aanhalingsteken,plus,minus,laatste plaats,max van
naamlijst,regelnummer;
integer array te printen regel[-20:100],naamlijst[1:4000],GEH[0:3999];

Boolean oude naam,laatste regel;

Boolean procedure is digit(s); is digit:= s < 10;

Boolean procedure is letter(s);
is letter:= letter a ≤ s ∧ s ≤ letter Z;

integer letter a,letter Z;

procedure Initialiseer assembleer proces;
begin comment Eerst worden de bijzondere symbolen ingevuld:; integer i;
  dubbele punt:= RESYM;
  spatie:= RESYM;
  puntkomma:= RESYM;
```

```
            tab:= RESYM;
            aanhalingsteken:= RESYM;
            plus:= RESYM;
            minus:= RESYM;
            letter a:= RESYM;
            letter Z:= RESYM;
            twnr:= RESYM;
            scheider:= 1; naam:= 2; getal:= 3; plus operator:= 4;
            minus operator:= 5;
            symbool:= twnr; positie:= 0; voorraad spaties:= 0;
            regelnummer:= 0; Lees symbool;
            instr teller:= 0;
            laatste plaats:= -3; max van naamlijst:= 4000;
            for i:= 0 step 1 until 10 do
            begin Skip layout; Lees naam; naamlijst[plaats van naam + 2]:= i
            end; for i:= -20 step 1 until 100 do te printen regel[i]:= spatie;
            symbool:= spatie;
            positie:= voorraad spaties:= 0; regelnummer:= 0;
            Lees symbool; Lees synt eenheid; Lees programma
          end Initialiseer assembleer proces;

          Initialiseer assembleer proces
        end : ;    "+-aZ
LDA STA ADD SUB JAN JMP PRI HLT CON ORIG END

          BEGIN:  ORIG 0                          "Eenvoudig
          p:      CON 0; CON 1; CON 2; CON 3;
          q:      CON 0
          m:      LDA p                           "test
          n:      STA q - p +12 - 1
          r:      JAN n
                  END m                           "programma


                     1
0000                 1          BEGIN:  ORIG 0                              "Eenvoudig
0000  +00000         2          p:      CON 0;
0001  +00001         2                          CON 1;
0002  +00002         2                                   CON 2;
0003  +00003         2                                            CON 3;
                     3
0004  +00000         3          q:      CON 0
0005  +00000         4          m:      LDA p                               "test
0006  +00151         5          n:      STA q - p +12 - 1
0007  +00064         6          r:      JAN n
0008                 7                  END m                               "programma
naamlijst
LDA       +0           -1
STA       +1           -1
ADD       +2           -1
SUB       +3           -1
JAN       +4           -1
JMP       +5           -1
PRI       +6           -1
HLT       +7           -1
CON       +8           -1
ORIG      +9           -1
```

END	+10	−1
BEGIN	−1	+0
p	−1	+0
q	−1	+4
m	−1	+5
n	−1	+6
r	−1	+7

4.3. Opmerkingen n.a.v. de VERAL assembler

Aan de assembler van de vorige sectie kleven nog een aantal feiten welke we nu zullen bespreken.

4.3.1. Foute programma's

We hebben ons op het standpunt gesteld dat een fout programma fout is, zodat constatering van één fout de assemblage van de rest van het programma stopzet.
Nu zijn er fouten en fouten: het plaatsen van meer dan 100 symbolen op een regel is minder erg dan het gebruiken van de label END niet gevolgd door een dubbele punt.

Een goede strategie is om de assemblage zover mogelijk voort te zetten omdat dan zoveel mogelijk fouten geconstateerd worden. Het is dan waarschijnlijk dat schrijf- en ponsfouten er in één of twee runs uitgehaald zijn. Hiertoe is het noodzakelijk verschillende fout-routines te hebben:

1. Voor het constateren van een fout welke verder geen actie tot gevolg heeft (voorbeeld: twee keer eenzelfde naam als label gebruikt).
2. Voor het constateren van een fout waarna de rest van de instructie geskipt wordt omdat deze oninterpreteerbaar is geworden (voorbeeld: LDB LDA B waarin het ontbreken van de dubbele punt tot desastreuse gevolgen leidt).
3. Voor het constateren van een fout welke een stopzetting van de assemblage ten gevolge heeft (voorbeeld: er worden meer dan 1000 namen gebruikt).

Een grote verbetering wordt reeds bereikt door de procedure EXIT uit de body van ERROR weg te laten en tegelijkertijd de statement:

"ERROR (synt eenheid ≠ scheider, ⍃ regel niet correct afgesloten ⍄)"

te vervangen door:

"<u>if</u> synt eenheid ≠ scheider <u>then</u>
<u>begin</u> ERROR (<u>true</u>, ⍃ regel niet correct afgesloten ⍄);
L: Lees synt eenheid; <u>if</u> synt eenheid ≠ scheider <u>then</u> <u>goto</u> L
<u>end</u>"

4.3.2. Voorwaartse verwijzingen

Het blijkt dat het eenvoudige programma:

```
"    ORIG 1000
     LDA B
  B: CON 0
     END 1000 "
```

De foutmelding "waarde van naam onbekend" geeft, door de voorwaartse verwijzing "LDA B".

Een mogelijke oplossing voor dit probleem is de zogenaamde 2 scan methode, d.w.z. in plaats van één keer de tekst van het programma door te nemen (scannen) doen we het twee keer: de eerste keer om alle namen op te pikken en hun waarde in te vullen, aldus de naamlijst in orde makend, de tweede keer om m.b.v. de naamlijst de instructies te bakken.

Een andere oplossing voor het probleem is de zogenaamde bijwerk keten methode. Deze methode zullen we nu beschrijven.
Bezien we de instructie "LDA B", dan kunnen we wel de instructie code (0) bepalen, echter niet het adres gedeelte (1001). M.a.w. we kunnen GEH[1000] niet invullen.
Pas wanneer "B: CON 0" is gelezen, kan GEH[1000] de waarde +10010 krijgen.
Er zijn dus twee handelingen uit te voeren:

a) Bij het verwerken van "LDA B", noteer dan dat GEH[1000] niet gevuld is, dat de instructiecode 0 is en dat het adres die van de later als label verschijnende naam B is.
   Noteer bij de naam B in de naamlijst dat het gelezen is niet als label maar als adres van de instructie die geplaatst zal worden in GEH[1000].
b) Bij het verwerken van "B: CON 0", vul dan GEH[1000] in en verander de informatie over B in de naamlijst zodanig dat B een gewone label is.

De methode die we zullen kiezen moet ook werken voor het geval:

```
       ORIG 1000
       LDA B
       ADD B
       SUB B
    B: CON 0
       END 1000
```

We gaan eerst na of de procedure "Lees naam" moet veranderen opdat "LDA B" verwerkt kan worden. B wordt gelezen en blijkt nieuw te zijn, verder kan "Lees naam" niets doen. Deze procedure hoeft dus, in eerste instantie, niet veranderd te worden.

Vervolgens bestuderen we de procedure "Lees term", welke door "Lees eventuele expressie" wordt aangeroepen. De procedure komt er achter dat B een nieuwe naam is. Welke waarde moet de procedure identifier "Lees term" nu krijgen? Als aan deze waarde te zien moet zijn dat het de waarde van een toekomstige label is, dan is dit onmogelijk aan te geven, daar "Lees term" elk mogelijk getal moet kunnen afleveren (Voorbeeld: CON -5000). M.a.w. ook "Lees term" behoeft geen verandering te ondergaan. Ditzelfde geldt voor de procedure "Lees eventuele expressie".
Aldus komen we als vanzelf bij de procedure "Lees eventuele kale instructie" terecht, en we ontdekken dat zeker de eerst drie statements veranderd moeten worden teneinde de situatie baas te worden, immers we moeten apart testen of we een situatie met de vooruitverwijzing hebben. Indien we een vooruitverwijzing hebben moet er ook nog onderscheid gemaakt worden tussen echte mnemonics (LDA-HLT) en pseudomnomonics (CON, ORIG en END).
Het programma:

```
"   ORIG B
 B: END B  "
```

is zinloos.
Over het programma:

```
"   ORIG 1000
    CON B+B
 B: CON 0
    END 1000  "
```

zijn nog wel zinnige opmerkingen te maken; voor ons is het echter onassembleerbaar (bij een 2-scan-methode levert dit programma geen moeilijkheden).

Er moet dus getest worden of na een echte mnemonic een naam komt die nieuw is, maar niet gevolgd wordt door een optel operator.

Teneinde notities te maken hebben we tot onze beschikking GEH[1000] en de ruimte in de naamlijst bij de gelezen naam B; in de eerste noteren we de instructie code.
De eerste drie statements van "Lees eventuele kale instructie" moeten nu in eerste instantie gewijzigd worden in:

```
type:= type van naam;
if type < 8 then
begin Lees synt eenheid;
      if synt eenheid = naam then
      begin Lees synt eenheid;
            if synt eenheid = scheider then
               begin if ¬ oude naam then
                     begin naamlijst [plaats van naam + 3]:=
                           instr teller;
                           GEH[instr teller]:=
                           -10 + type; PRINT instructie;
                           instr teller:= instr teller + 1;
                           goto EIND
               end end;
               adres:= waarde van naam + Lees eventuele expressie
      end else adres:= Lees eventuele expressie
end else
begin Lees synt eenheid; adres:= Lees eventuele expressie end;
```

Behalve deze wijziging moet er nog een label EIND voor de laatste end van "Lees eventuele kale instructie" geplaatst worden. Waarom GEH[instr teller] aan -10 + type gelijk wordt gemaakt wordt later duidelijk.

De andere handeling, bij het tegenkomen als label van de reeds gebruikte naam B, zullen we nu beschrijven. Daartoe beschouwen we de procedure "Lees eventuele label locatie". De foutmelding "ERROR(¬ oude naam, {twee keer dezelfde naam})" zou er uit kunnen; het ligt echter meer voor de hand de Boolean variabele "oude naam" iets anders te gebruiken: n.l. in de zin van "nog geen waarde verkregen".
Dit betekent voor de procedure "Lees naam" de verandering:

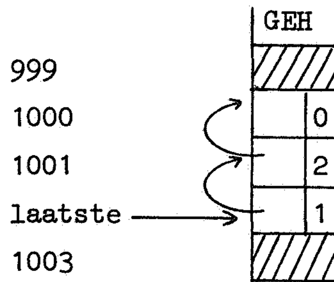
"oude naam:= true" wijzigen in:

"oude naam:= naamlijst[i+2] ≠ -1".

Het effect is dat bij het lezen van de tweede B in het programma:

```
       ORIG 1000
       LDA B
       ADD B
       STA B
    B: CON 0
       END 1000
```

de waarde van "oude naam" false is.
Een andere noodzakelijke verandering in de procedure "Lees eventuele label locatie" betreft de aanroep: "BERG OP (plaats van naam, instr teller)". Nu moeten we immers kijken of er nog geheugencellen, zoals GEH[1000] zijn, die nog bijgewerkt moeten worden. Zoals bovenstaand voorbeeld aangeeft kunnen dat er meerdere zijn. Hoe wordt dat aangegeven?
Als het er één is, dan wijst naamlijst [plaats van naam + 3] naar die geheugenplaats.
Als er meerdere zijn, dan wijst naamlijst [plaats van naam + 3] naar de laatste, de laatste geheugencel (GEH[1002]) wijst naar de voorlaatste (GEH[1001]), deze, op zijn beurt, wijst weer een vorige aan, etc. Tot we bij de eerste komen die geen vorige aanwijst. We krijgen aldus een keten of bijwerkketen. In een plaatje kunnen we deze keten aanschouwelijk voorstellen:



De verwijzingen geven we aan met de waarde van GEH[1002] ÷ 10; de laatste decimaal van GEH[1002] wordt gebruikt om de instructiecode te bewaren. De vorige geheugencel is dus GEH[GEH[1002]÷10]. Behalve de doorverwijzing is het nodig dat we de eerste geheugencel kunnen herkennen. Als we nu hiervoor het negatief zijn van GEH[1000] gebruiken dan blijkt dit een voldoende maar ook noodzakelijk kenmerk te zijn. Voldoende: geen enkele doorverwijzing die een niet-negatief adres aanwijst levert een negatieve ge-

heugencel op. Noodzakelijk: elk niet-negatief getal in GEH[1000] kan als adres van een doorverwijsgeheugencel opgevat worden (we zien af van het feit dat er maar 4000 geheugencellen zijn).

N.B. Het is nu ook duidelijk waarom, enkele bladzijden terug, de uitdrukking -10 + type werd gebruikt.

Aldus komen we tot de volgende vervanging van de aanroep "BERG OP" in "Lees eventuele label locatie".

```
     if type van naam = -1 then
     begin integer i,j; naamlijst[plaats van naam + 2]:= -2;
        i:= waarde van naam;
     L: if GEH[i] ≥ 0 then
        begin j:= GEH[i] ÷ 10;
              GEH[i]:= instr teller * 10 + (GEH[i]-j*10); i = j;
              goto L
        end; GEH[i]:= GEH[i] + 10 + instr teller * 10
     end; naamlijst[plaats van naam + 3]:= instr teller
```

De vervanging van de eeste drie statements van "Lees eventuele kale instructie" moet nu ook weer (subtiel) gewijzigd worden in (dit betreft slechts de kern):

```
         if ¬ oude naam then
         begin GEH[instr teller]:= waarde van naam * 10 + type;
               naamlijst[plaats van naam + 3]:= instr teller;
               PRINT instructie;
               instr teller:= instr teller + 1;
               goto EIND
     end end;
```

Hierbij maakten we listig gebruik van de initiële waarde van naamlijst [plaats van naam + 3], namelijk -1.

Tenslotte moet de procedure "Beëindig assembleer proces" nagaan of er nog namen over zijn die geen waarde gekregen hebben, in welk geval een foutmelding moet komen.

Vraag: Waarom is het lastig om in de bijwerkketen techniek expressies met nog onbekende namen te behandelen.

Probleem: Het volgende programma geeft aanleiding tot moeilijkheden. Onderzoek deze en ga na welke wijzigingen in de assembler moeten worden aangebracht:

```
   ORIG 1000
   LDA B
   LDA B
   ORIG 1000
   LDA B
B: END 1000
```

4.3.3. Efficiënter namen zoeken

Als de naamlijst n namen bevat, dan zal er gemiddeld $(n+1)/2$ keer onderzocht moeten worden of een gegeven naam in de naamlijst voorkomt.
Immers, de kans dat we in één keer de naam vinden is $1/n$, de kans dat we hem in twee keer vinden is de kans dat we hem de eerste keer niet vinden $((n-1)/n)$ maal de kans dat we hem in één keer in de, met één verminderde, rij aantreffen $(1/(n-1))$; dus ook $1/n$.
Conclusie: de kans dat we hem in precies i slagen aantreffen is $1/n$. Het gemiddelde aantal keren is dus $\sum_{i=1}^{n} \frac{i}{n} = (n+1)/2$.

Zou de lijst met namen volledig gealfabetiseerd zijn, dan ziet de zaak er anders uit.
Het gemiddeld aantal vergelijkingen voor een rij van n namen is $v(n)$.
We nemen de "middelste" naam d.w.z. de naam met index entier $((n+1)/2)$. We onderzoeken of dit de goede is, zo nee dan gaan we verder zoeken in één van de twee helften, zodat $v(n) = 1 + v(\text{entier}(\frac{n+1}{2}))$.

De functionaalvergelijking $f(x) = 1 + f(\frac{x}{2})$, heeft als oplossing $f(x) = {}^2\log(x)$, zodat $v(n) \approx {}^2\log n$.

Om een illustratie te geven van het verschil tussen $(n+1)/2$ en ${}^2\log n$:
Het door prof.dr. F.E.J. Kruseman Aretz ontworpen bedrijfssysteem MILLI voor de EL X8 bevat ongeveer 1800 namen. In gemiddeld elke instructie komt één naam voor. Er zijn ongeveer 15000 instructies.

$$(1800+1)/2 * 15000 \approx 13500000 \text{ vergelijkingen}$$
$${}^2\log 1800 * 15000 \approx 162000 \text{ vergelijkingen}$$

Aangezien elke vergelijking nogal wat tijd vergt, betekenen bovenstaande getallen een verschil tussen anderhalf uur assembleertijd en een acceptabele assembleertijd van een kwartier. (N.B. behalve het zoeken in de naamlijst moet er nog meer gebeuren!)

Het ligt dus voor de hand om de naamlijst alfabetisch te ordenen. We zullen dit in een volgend hoofdstuk nader beschouwen.

4.3.4. Willekeurig lange namen

De eis dat een naam uit niet meer dan 8 letter of digits mag bestaan is een zeer onnatuurlijke eis (vanuit het standpunt van de gebruiker, niet vanuit het standpunt van de assemblermaker). Om van deze eis af te komen is het nodig te kunnen werken met informatie-eenheden van variabele grootte.
Een onderdeel van de informatie is dan de grootte zodat ook de grootte opgeborgen moet worden.

In ons geval moet de informatie inhouden:

het aantal letters
de letters
het type
de waarde.

We laten het als een opdracht aan de lezer om dit voor de VERAL assembler verder uit te werken.

4.3.5. De leesstrategie en de laatste regel

Na de laatste regel van het programma moeten er nog tenminste één syntactische eenheid en één symbool volgen. Dit is de consequentie van de leesstrategie: één syntactische eenheid te ver en één symbool te ver lezen.
Een eenvoudige remedie is de procedurebody van "Lees synt eenheid" te laten voorafgaan door:

"<u>if</u> ¬ laatste regel <u>then</u>" .

Bovendien moet de laatste regel bij de initialisatie de waarde <u>false</u> krijgen.

Literatuur

1. Naur, P. (ed.): Revised Report on the Algorithmic Language ALGOL 60
Numerische Mathematik 4 pp. 420-453
In dit rapport moet elke "informaticus" zijn weg weten te vinden.

2. Ekman, T; Fröberg, C-E.: Introduction to ALGOL programming
Oxford University Press London 1967 (Prijs: ± ƒ17,--)
Een zeer goede introductie in ALGOL 60; bevat naast het ALGOL 60 rapport enkele interessante foto's.

3. Seidel, J.J. (red): Computer wiskunde
Aula-boeken Uitgeverij Het Spectrum 1969 (Prijs: ± ƒ5,--)
Een aanbevolen inleiding in het gebruik van rekenautomaten (de vijf hoofdstukken: Algorithmen, Numerieke toepassingen, Non-numerieke toepassingen, Over de structuur van rekenautomaten, Vraagstukken en oplossingen).

4. Phillips, G.M; Taylor, P.J.: Computers
Methuen & Co. Ltd., London 1969 (Prijs: ± ƒ10,--)
Een uitstekende inleiding waarin behalve het programmeren ook de computer zelf op elementaire wijze behandeld wordt. Zelfs Turing machines worden behandeld.

5. Fry, T.F.; Computer Appreciation
Butterworths, London 1970 (Prijs: ± ƒ14,--)
Een uitstekende algemene inleiding met de nadruk op computer hardware en administratieve toepassingen.

6. Rice, John K.; Rice John R.: Introduction to Computer Science
Holt, Rinehart & Winston Inc. 1969 (Prijs: ± ƒ55,--)
Een uitstekende, zij het kostbare introductie in het programmeren (in diverse talen: machine, natuurlijke, stroomdiagrammen, ALGOL 60, FORTRAN), het oplossen van problemen in het algemeen en computer organisatie.

7. Wegner, P.: Programming Languages, Information Structures and Machine Organization
McGraw-Hill, 1968 (Prijs: ± ƒ54,--)
Een zeer gedetailleerd boek voor de gevorderde en meer geïnteresseerde.

8. Knuth, D.E.: The Art of Computer Programming Volume 1 / Fundamental Algorithms
Addison-Wesley, 1968 (Prijs: ± ƒ91,--)
Het is onnodig over dit boek iets anders te zeggen dan "goede wijn behoeft geen krans".

9. Knuth, D.E.: The Art of Computer Programming Volume 2 / Seminumerical Algorithms
Addison-Wesley, 1969 (Prijs: ± ƒ81,--)
Het tweede boek in de serie waarin 7 delen zijn gepland.

10. Chandor, A.: A dictionary of Computers
Penguin reference books 1970 (Prijs: ± ƒ7,--)

11. De "computer monograph" serie van MacDonald/Elsevier; betrekkelijk goedkope boekjes (15-20 fl) met als titels: List processing (Foster), A comparative study of programming Languages (Higman), Recursive techniques in programming (Barron), Basic machine principles (Iliffe), Time-sharing computer systems (Wilkes), Assemblers and Loaders (Barron), Executive programs and operating systems (Cuttle, Robinson ed.), Computer-based library and Information systems (Henley), Compiling techniques (Hopgood).

12. Flores, I.: Computer Software
Prentice-Hall Inc., 1965 (Prijs: ± ƒ70,--)
Geeft een zeer brede beschrijving van assemblers, loaders, macro's en van operating systemen.

13. Flores, I.: Computer Organization
Prentice-Hall Inc., 1969 (Prijs: ± ƒ60,--)
Geeft een breed opgezette beschrijving van computer hardware.

14. Rosen, S. (Ed): Programming Systems and Languages
McGraw-Hill, 1967 (Prijs: ± ƒ60,--)
Bevat een aantal "klassieke" artikelen over programmeertalen, compilers en operating systemen.

15. Minsky, M.L.: Computation finite and infinite machines
Prentice-Hall Inc. 1967 (Prijs: ± ƒ60,--)
Behandelt de wiskunde van computers: Theorie van machines, Turing machines, recursieve functies, berekenbaarheid.

16. Alle nummers van Computing Surveys; een 3-maandelijks tijdschrift uitgegeven door de Association for Computing Machinery, New York.

Errata by CR 21.

Uitgangspunt bij het produceren van de syllabus is geweest: "Beter op tijd een syllabus met fouten dan te laat een foutloze syllabus" met het onderstaande gevolg:

p.	regel	staat:	moet staan:
2	1 v.o.	1694	± 1671
	1 v.o.	de 25-jarige	
	9 v.o.	.	. (zie Fry)
	3 v.o.	.	. (zie Fry)
3	11 v.b.	.	. (zie Phillips & Taylor)
	12 v.o.	.	. Reeds in 1941 construeerde Konrad Zuse in Duitsland de eerste electrische rekenmachine Z3. (AFIPS conference proceedings 27, part 2 1965 pp. XV-XVII)
4	4 v.b.	.	. Hoffman, W.: Digitale Informations Wandler Braunschweig, Vieweg, 1962 740 blz.
5	2 v.b.	De	Van de naam van de
	3 v.b.	voerde	werd
	3 v.b.	in	afgeleid
14	5 v.b.	mes	ms
	7 v.b.	.	. zullen we de werking van een computer demonstreren.
16	7 v.o.	9.	7.
18	11 v.b.	semicolon	
	12 v.b.	semicolon:= 91;	
22	1 v.b.	0	CON 0
	2 v.b.	1	CON 1
	3 v.b.	10	CON 10

p.	regel	staat:	moet staan:
22	4 v.b.	0	CON 0
	5 v.b.	0	CON 0
	6 v.b.	0	CON 0
	7 v.b.	0	CON 0
	8 v.b.	0	CON 0
	3 v.b.	VERHOOG q	VERHOOGq
	12 v.b.	Heel m door n	Heelmdoorn
	16 v.b.	moet luiden: "zetten en adressen aan namen te hechten, en de tweede keer om de instructies".	
25	2 v.o.	Comp	COMP
	1 v.o.	Comp	COMP
26	1 v.b.	Comp	COMP
	2 v.b.	Comp	COMP
	5 v.b.	Overflow	OVERFLOW
27	3 v.b.	if i=0 then 0 else Ri	(if i=0 then 0 else Ri)
	12 v.o.	één plaats	een aantal plaatsen
	9 v.o.	een plaats	een aantal plaatsen
28	9 v.b.	zal	kan
	10 v.b.	.	. (in ons geval slechts een KL en een RD)
30	12 v.o.	het	
32	14 v.b.	den	de
33	6 v.b.	.	, na het register op false te hebben gezet.
	8 v.b.	.	, zet anders het register op false.
38	22 v.b.	moet luiden: "JOV: i:= if OVERFLOW then 1 else 0; OVERFLOW:= false; SPRING INDIEN (i=1);"	
	23 v.b.	moet luiden: "JNOV: i:= if OVERFLOW then 1 else 0; OVERFLOW:= false; SPRING INDIEN (i=0);"	
39	10 v.o.	1 do	1 $\wedge$ i $\leq$ 80 do
42	16 v.b.	kleine	hoofd-
	17 v.b.	hoofd	kleine

p.	regel	staat	moet staan:
53	3 v.b.	Kruth	Knuth
	5 v.b.	later	later helaas niet meer
	5 v.b.	.	. Deze specificaties vindt men in het boek van Knuth en in een binnenkort te publiceren MC rapport van G. ten Velden.
57	1 v.o.	.	. *) Zie de noot onder aan p. 59
65	12 v.o.	in	is
66	13 v.b.	.	De variabelen BUFLEEG en BUFVUL zijn in het ALGOL programma namen die nog verschillende waarden kunnen aannemen. In het MIXAL programma zijn het echter de vaste adressen van geheugencellen die op hun beurt verschillende waarden kunnen aannemen. Met andere woorden: $BUFVUL_{ALGOL} \sim GEH[BUFVUL_{MIXAL}]$.