MC SYLLABUS

13

MC SYLLABUS

13

PROGRAMMEREN VOOR REKENAUTOMATEN

DE MC ALGOL 60 VERTALER VOOR DE EL X8

DOOR

F.E.J. KRUSEMAN ARETZ

COLLEGEVERSLAG UITGEWERKT DOOR

C.G. VAN DER LAAN

J.P. HOLLENBERG

MATHEMATISCH CENTRUM AMSTERDAM

1971

Inhoud

		blz.
Inleiding		2
I	Opbouw ALGOL 60-systeem	7
I-1	Modulaire structuur van de vertaler	7
I-2	De drie scans van de vertaler	9
I-3	Foutmeldingen	12
I-4	Geheugengebruik tijdens de vertaling	14
I-5	Overzicht van de lengte van de modulen en scans	15
II	De vertaling van aritmetische expressies	17
II-1	Het vertaalschema van aritmetische expressies	17
II-2	De algoritme *Arithexp*	25
II-3	De vertaling van een primary	32
II-4	Voorbeelden ter toelichting van de vertaling van de array-elementen	37
II-5	De macroprocessor: genereren van instructies en optimaliseren van aritmetische expressies	40
III	De vertaling van Boolean expressies	50
III-1	Het vertaalschema van primary's	50
III-2	Optimaliseren van relaties	52
III-3	De vertaling van de Boolean operaties	53
III-4	Moeilijkheden tijdens vertalen van Boolean primary's	54
IV	De vertaling van assignment statements	59
IV-1	Het algemene vertaalschema van assignment statements	59
IV-2	De assignment aan een locale of globale simpele variabele	60
IV-3	De assignment aan een geïndiceerde variabele	61
IV-4	De assignment aan een formele identifier en het parameter-mechanisme	62
V	De vertaling van for statements	65
VI	De vertaling van blokken	67
VI-1	Blokcellen in de stapel	67
VI-2	De vertaling van blokingang en blokverlating	70

		blz.
VII	De vertaling van procedures	74
VII-1	Blokcel en display van een procedure	74
VII-2	De vertlaing van proceduredeclaraties: De macro's *DPTR* en *INCRB*	78
VII-3	Het parameter-mechanisme, *APD* en *APIC*	79
VII-4	De vertaling van de formele perameterlijst: De macro's *CEN*, *CLPN*, *CRV* en *CIV*	82
VII-5	De rest van de vertaling van een proceduredeclaratie: De macro's *TDL*, *ENTR PB*, *EXITP*	90
VIII	Garanties voor de eindigheid van het vertaalproces	93
APPENDIX		95
Literatuur		112

Voorwoord

Dit collegeverslag beslaat het tweede semester van het college "Programmeren voor Rekenautomaten" 1969/1970. In het eerste semester, waarvan geen verslag bestaat, zijn behandeld

a) logische opbouw van een simpele rekenautomaat (poorten, flip flops, opteller,kerngeheugen, basisschema OEBRA naar een ontwerp van Prof. Dr. A. van der Sluis)

b) beschrijving van de EL X8 (registers, aritmetiek, opdrachtenrepertoire, ELAN). Hiervan bestaat een kort verslag, gemaakt in 1967/1968 door Drs. J.V.M. van der Grinten

c) de standaard subroutines voor vierkantswortel en sin/cos voor de EL X8.

Het tweede semester beschreef de ALGOL 60-implementatie voor de X8. Naast het verslag bestaat de volgende documentatie hiervan:

B.J. Mailloux, F.E.J. Kruseman Aretz. "The ELAN source text of the MC ALGOL 60 system for the EL X8", rapport MR 84 van de Stichting Mathematisch Centrum (1966). Dit rapport bevat de volledige tekst van vertaler, run-time subroutines en een eenvoudig bedrijfssysteem.

F.E.J. Kruseman Aretz. "Het objectprogramma, gegenereerd door de X8-ALGOL 60-vertaler van het MC".
MR 121 (1971). Een preprint werd op het college uitgedeeld.

F.E.J. Kruseman Aretz, P.J.W. ten Hagen en H.L. Oudshoorn. "The ALGOL source text of the MC ALGOL 60 translator for the EL X8", dat als tract van het Mathematisch Centrum zal verschijnen. De in de appendix afgedrukte procedures *Arithexp* etc. en *Macro2* etc. zijn hieraan ontleend. Deze ALGOL text is een blauwdruk geweest voor de vertaler versie in ELAN als gegeven in MR 84.

Het procedure stelsel *Arithexp* is ook beschreven in F.E.J. Kruseman Aretz, "ALGOL 60 translation for everybody", Elektronische Datenverarbeitung (1964, blz. 233-244).

Inleiding

090270

In dit college wordt de ALGOL 60 vertaler voor de EL X8 rekenautomaat behandeld.

Deze vertaler is ontwikkeld op het M.C. door een in juli '63 opgerichte werkgroep.
Het ontwikkelen van de vertaler was een onderdeel van een landelijke samenwerking om de destijds in aanbouw zijnde X8 van "software" te voorzien. Het volgende geeft een overzicht van de participanten met hun taak

instelling	taak
R.U. te Utrecht Dr. Neher Laboratorium (P.T.T.) R.C.N. te Petten	} assembler
T.H. te Eindhoven	multiprogrammering
T.H. te Kiel	FORTRAN - compiler
S.M.C.	ALGOL 60 - compiler

Enig idee over het verloop der werkzaamheden op het M.C. geven de jaarverslagen van die instelling.

"De in 1963 gevormde interne werkgroep, bestaande uit de heren Zonneveld, Kruseman Aretz, Nederkoorn, Van der Laarschot en Barning hield zich in 1964 uitvoerig bezig met de verdere ontwikkeling van een ALGOL 60-systeem voor de toekomstige X8 rekeninstallatie.
Deze werkzaamheden bestonden o.a. uit

a) de verdere ontwikkeling van een "running system" en een voorlopige load-and-go ALGOL 60 vertaler voor de X8;
b) het programmeren van standaard functies voor de X8;
c) het bestuderen van het monitor-probleem, in het bijzonder wat betreft de verslaglegging van de bedieningshandelingen op de verreschrijver;
d) de verdere ontwikkeling van X8 simulator."

(jaarverslag S.M.C. 1964).

"Van de vertaler voltooide de heer Kruseman Aretz de versie, geschreven

in ALGOL 60. Deze versie kon uitvoerig getest worden met een bestaand ALGOL 60 systeem voor de X1 (de MC II vertaler, geschreven door de heren Nederkoorn en Van de Laarschot). Hierna kon de eigenlijke codering van de vertaler in ELAN, de machine-code-taal voor de X8, door hem ter hand genomen worden.
Ook het complex werd door de heer Kruseman Aretz aangevuld en afgerond. Het schrijven van input- en outputroutines werd toevertrouwd aan de heren Mailloux en Van Berckel. Als bedrijfssysteem ontwikkelden deze laatsten de PICO-monitor, die het werken in ALGOL 60 op een kleine installatie (zonder achtergrondgeheugen en zonder meervoudige in- en output-kanalen) bijzonder eenvoudig en zo efficient mogelijk doet verlopen."
(jaarverslag S.M.C. 1965).

"running system" is een verzameling subroutines, aangeroepen tijdens de executiefase van een programma voor diverse administratieve taken zoals in- en uitgaan van een blok, het opzoeken van arrayelementen e.d.
Een load-and-go vertaler maakt het mogelijk het objectprogramma (resultaat van de vertaal arbeid), zonder eerst te zijn opgeslagen in een achtergrond geheugen direct voor uitvoering in het geheugen op te slaan.
De X8 simulator simuleert X8 opdrachten op de X1 (zie hiervoor rapport R917).
Complex is een andere naam voor running system.

De participanten in het X8-project hadden de volgende (elkaar soms tegensprekende) eisen aan het MC ALGOL 60 systeem van de X8 gesteld

1) het MC ALGOL 60 systeem moest gebruikt kunnen worden op een minimum X8-configuratie (d.w.z. 16 K geheugenplaatsen, in- en output door één bandlezer respectievelijk bandponser)
2) het moest zo efficient mogelijk zijn bij ingewikkelde berekeningen
3) het moest programmafouten detecteren waar mogelijk
4) het moest een modulaire opbouw hebben, zó dat het gemakkelijk aan toekomstige eisen aangepast kon worden
5) de opzet moest "didactisch" zijn dat wil zeggen de structuur moest duidelijk en gemakkelijk te begrijpen zijn om eventuele fouten snel op te kunnen sporen.

De eisen spreken elkaar soms inderdaad tegen.
Neemt men b.v. de ALGOL-statement (*A* is een real array)

```
for i := 1 step 1 until 1000 do A[i] := 0
```

dan kan men deze als volgt in ELAN vertalen:

```
        "neem adres van A[1] in register S"
        F = 0
        A = 1000
loop:   MS = F
        S + 2
        A - 1, Z
     N, GOTO (: loop).
```

Opm: *F* wordt opgeslagen in 2 woorden, men moet dus *S* telkens met 2 ophogen.

Dit is zeer snel omdat binnen de loop de adressen van de elementen niet uit de basisgegevens worden uitgerekend maar voor opeenvolgende elementen uit elkaar volgen door een simpele optelling.

```
for i := 1 step 1 until 1000 do
begin A[i] := 0; i := i * i end;
```

Nu zijn de adressen niet meer zo makkelijk in de loop te genereren. Omdat een vertaler, die kan analyseren, wanneer for loops met geïndiceerde variabelen in de statement na *do* wel of niet versneld kunnen worden, lang en complex is (in strijd met doelstellingen 1) en 5)), wordt zo'n analyse in de X8-vertaler niet gedaan en de adresberekeningen binnen of buiten for loops op dezelfde, standaard manier gedaan.
We zullen later zien dat deze oplossing tot gevolg heeft dat bewerkingen met arrays nogal tijdrovend zijn (in strijd met doelstelling 2)). Het is zó duur dat het beter is een arrayelement niet 2 maal in één blok op te zoeken, doch te assigneren aan een simpele parameter.

De controletaak bevat o.a. syntaxcontrole tijdens vertaling en controle tijdens runtime. De run-timecontrole bevat o.a.

a) testen subscripts
b) controle op het "overlopen" van de werkruimte
c) controle op de correspondentie formele/actuele parameters.

De controle op de correspondentie formele/actuele parameters kan niet uitputtend uitgevoerd worden tijdens de vertaling zonder het programma partieel in executie te nemen.
Echter de runtime-controle hierop is ook onvolledig. Dit heeft tot gevolg dat bijvoorbeeld het programma

```
begin procedure P(Q); procedure Q;
      begin Q (true) end;
      P (print)
end
```

toegelaten en in executie genomen wordt. Want de controle tijdens runtime onder c) bevat slechts de controle:
staat de formele parameter links van := ?
zo ja, is dan de aangeboden actuele parameter waaraan geassigneerd wordt wel een variabele?
Als een ALGOL-vertaler onderdeel is van een multiprogrammerings systeem zonder hardware geheugenprotectie (waarbij meer dan één programma in het geheugen staat) is zoiets eigenlijk ontoelaatbaar. Geen enkele fout mag dan onopgemerkt blijven omdat niet alleen het eigen programma daarvan de dupe kan zijn.

De modulaire opbouw houdt in dat men de vertaler splitst in onafhankelijke stukken, de z.g. modulen (b.v. de syntactische analyse en de generatie van de vertaling).
De voordelen hiervan zijn legio. Zo hoeft men bij aanschaf van een nieuwe of uitbreiding van de oude machine niet alle modulen om te werken.
Ook kan men instructies voor een hypothetische machine laten genereren. De output van een vertaler benut n.l. slechts weinig instructies van een machine, en deze worden clichématig gebruikt. Men kan deze cliché's met weinig bits coderen en zo een compact objectprogramma verkrijgen. Een dergelijk systeem is sinds lang operabel op de X1 en biedt een ruimtebesparing van een factor 3. Uiteraard staat tegenover winst in de ene sector verlies

in de andere, in dit geval is er verlies in tijd, omdat de hypothetische machine op de X1 gesimuleerd moet worden.
Ook biedt de modulaire opbouw voordeel bij het opsporen en verbeteren van fouten.
Indien de opbouw niet modulair is kan het gebeuren dat het ene gat met het andere gestopt wordt. Zelfs kan één fout bij herstel twéé andere genereren.

De ALGOL tekst van de vertaler is getest door de MC II vertaler op de X1.
De ELAN versie is door de X8 simulator getest op de X1.
De ALGOL tekst van de vertaler is zo geschreven dat het parametermechanisme vaak ongebruikt blijft, als daar wel gebruik van gemaakt wordt, dan wordt de formele parameter meestal in de value part geplaatst en in de ELAN versie wordt voor het in werking treden van een subroutine de parameter geevalueerd en de waarde opgeslagen in een register.

I Opbouw ALGOL 60-systeem

I-1 Modulaire structuur van de vertaler

160270

In verband met het grote belang van de modulaire opbouw zullen wij nu de modulen globaal bespreken.

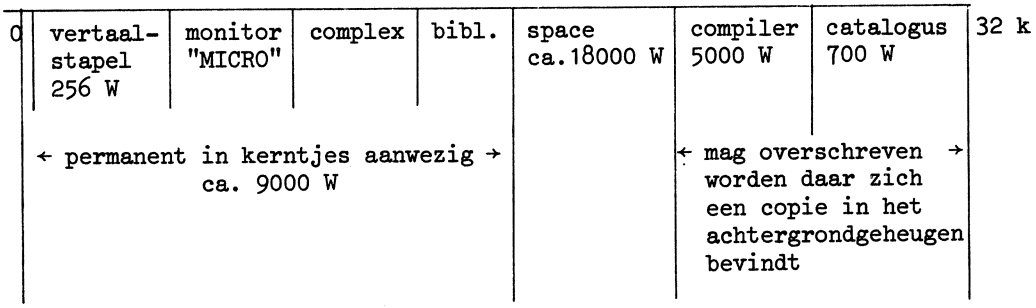
De modulen van het gehele ALGOL systeem zijn

1) de monitor
2) het running system
3) de compiler
4) de bibliotheek en de catalogus.

Zoals we reeds weten is de opzet van de modulaire opbouw dat de modulen (hier 1, 2, 3, 4) onafhankelijk van elkaar zijn. Daardoor was het (gemakkelijk) realiseerbaar dat in Eindhoven en in Utrecht de modulen 1) en 4) aan locale wensen werden aangepast en dat bij het verkrijgen van een derde geheugenkast (een kast ≈ 16 k) behalve de wijzigingen in 2) er slechts 4 instructies in 3) gewijzigd hoefden te worden.

Bespreking modulen.

De modulen zijn als volgt in het geheugen gesitueerd, bij het bedrijfssysteem "MICRO"



In de vertaalstapel vinden de vertalersubroutines (in de ELAN-versie) hun werkruimte.

ad 1) De monitor - een bedrijfssysteem waarin taal (b.v. o.a. ALGOL) sys-

temen zitten - is permanent aanwezig en verzorgt de "Input/Output" en de "scheduling". De huidige EL X8 monitor (deze bevat alleen het ALGOL-systeem omdat enerzijds het FORTRAN-systeem dat te Kiel ontwikkeld werd geen succes was en anderzijds in het M.C., waar de monitor ontwikkeld werd, een grote belangstelling voor ALGOL bestond) kijkt of er iets aangeboden wordt, wanneer er vertaald of gecontroleerd kan worden en start al dan niet tenslotte de executie waarbij het een tijdcontrole uitvoert. Ook is voor te stellen een monitor ten behoeve van time sharing. De taak is dan om van tijd tot tijd van een programma naar een ander over te schakelen zonder dat daarbij de werkgeheugens door elkaar lopen.

ad 2) Het running system (= complex) (d.w.z. de verzameling hulproutines die gebruikt worden tijdens de executiefase van een ALGOL-programma) is vanzelfsprekend vereist tijdens executiefase.

ad 3) De vertaler levert als resultaten

1) Het objectprogramma in *space*
2) De variabelen
   instruct counter
   start, dp0
   erroneous

space is de werkruimte van de vertaler en opslagruimte voor het objectprogramma.

De vertaler heeft één input parameter, de boolean *wanted*, die aangeeft of tijdens executie wel of niet een regelnummering moet worden bijgehouden. Vanwege het kortere objectprogramma kan de mogelijkheid om een programma met $\neg$ *wanted* te draaien ingeval van ruimtegebrek uitkomst brengen. Het sneller zijn (ca. $\frac{1}{2}$%) is nauwelijks interessant.

instruct counter geeft aan hoelang het objectprogramma is en waar de werkruimte begint.

start geeft het adres van de eerste bij executie uit te voeren opdracht.

dp0 vertelt iets over de administratievariabele die met de adressering te maken heeft.

erroneous vertelt of er syntactische fouten gevonden zijn of niet.

De modulen van de vertaler zijn

3-1) syntax analyzer + macrogenerator

3-2) macroprocessor

3-3) naamlijstroutines

3-4) next symbol routines

3-5) unsigned number routines.

ad 3-1) De syntax analyzer controleert en analyseert de source text en genereert de vertaling in de vorm van macro's (specificaties van handelingen die tijdens runtime verricht worden). Dit gedeelte is machine-onafhankelijk.

ad 3-2) De macroprocessor bouwt uit de gegenereerde macro's de feitelijke instructies op.

ad 3-3) Deze routines kunnen namen en gegevens omtrent namen opslaan dan wel ze weer opvragen.

Opmerking: hierbij is betrekkelijk weinig aandacht aan de efficiency besteed omdat er te weinig tijd overbleef. Men vermoedt dat een lang programma met veel namen het grootste deel van de tijd bezig is namen op te zoeken in de naamlijst. Dit kan beter. Aangezien de rekentijd in het algemeen veel langer is dan de vertaaltijd, is de verbetering van de compiler niet het meest urgent.

ad 3-4) next symbol routines bouwen uit de symbolen op de ponsband (inclusief upper en lower case, NLCR, erase, e.d.) de basic symbols van ALGOL op.

ad 3-5) Bouwen getallen op uit cijfers e.d..

I-2 De drie scans van de vertaler

De MC-vertaler scant driemaal een programma. Scannen is "lezen", waarbij slechts bij de eerste scan van de ponsband gelezen wordt. De wenselijkheid van meer dan éénmaal scannen blijkt uit het volgende voorbeeld

```
begin procedure p(a); real a;
      begin procedure q;
            begin ...
                  if a > 1 then ...
            end q;
            integer a;
            ...
      end p;
      ...
      ...
end
```

Bij eenmaal "lezen" komen er direct moeilijkheden omdat bij *if a > 1 then ...*, nog niet duidelijk is wat *a* is.
a kan bijvoorbeeld zijn

de formele parameter van *p*
een simpele locale variabele
een parameterloze funktie procedure, locaal in *p*.

De machine kan nu bijvoorbeeld verder het programma scannen totdat duidelijk is wat *a* is. Vervolgens vertaalt de machine *a* en continueert het "lezen" en vertalen. Dit proces is inefficient omdat het mogelijk is dat een (zelfde) stuk tekst meerdere malen wordt gescand met een te beperkt doel. Als tweede mogelijkheid echter zou de machine ruimte open kunnen laten voor de vertaling van *a* en gewoon met "lezen" en vertalen verder gaan. Zodra duidelijk is geworden wat *a* is, kan zijn vertaling in de open gelaten ruimte ingevuld worden. De vertaling van *a* moet dan echter van uniforme lengte zijn, onafhankelijk van de aard van *a*. Dit is duidelijk nadelig want enerzijds legt deze voorwaarde beperkingen op aan de soort instructies (de macroprocessor moet macro's van een uniform formaat afleveren) en anderzijds wordt misschien meer ruimte gebruikt dan strikt noodzakelijk is. Als derde mogelijkheid (dit doet de MC-ALGOL vertaler) kan de vertaler in zogenaamde prescans zoveel mogelijk informatie verzamelen (zie onderstaande tabel). In de laatste scan (de translation scan) vinden pas een grondiger foutdetectie en de vertaling plaats.

Overzicht van de taken van de verschillende scans.

prescan 0: opbouwen van complete naamlijst (bevat informatie over blokstructuren,identifiers e.d.)
" " constantenlijst
" " textarray (dat is het lineair opslaan van het te vertalen programma, zodat maar één keer van de band gelezen hoeft te worden).

prescan 1: a) het verzamelen van informatie over formele identifiers
b) " " " " " gebruik van labels
c) adresberekening van

C_1) { formele identifiers (1, 2 of 4 woorden)
type procedure identifiers(1,2,3 woorden)

C_2) { pseudo - for - variabelen (1 woord)
simpele locale variabelen(1 of 2 woorden)
array identifiers (1 woord)
pseudo - label - variabelen (2 woorden)

d) reserveren van plaatsruimte (in de zogenaamde statische ruimte) voor statisch geadresseerde variabelen
e) initialiseren van statisch geadresseerde pseudo - label - variabelen
f) toevoegen aan naamlijst van niet gedeclareerde identifiers (komt de identifier in de catalogus voor dan worden gegevens uit deze bibliotheek-procedure overgeheveld anders wordt foutmelding 204 gegeven).

translation scan: syntactische controle
adressering van procedure identifiers
" " switch "
" " label "
generatie van objectprogramma.

Verdere beschrijving van de taken van prescan 1.

ad a) In het bijzonder wordt gescand of ze voorkomen als linkerlid van een

assignment statement, van welk type ze zijn, bij een array wat de dimensie is en bij een procedure het aantal parameters.

ad b) In het bijzonder wordt gescand of ze op een andere manier gebruikt worden dan in *goto* <*identifier*> zonder dat blokverlating optreedt.

ad c_1) De adressen hiervan zijn altijd dynamisch geadresseerd.

ad c_2) Ten aanzien van deze groep geldt de regel als zo locaal t.o.v. een procedure (of een binnenblok daarvan) zijn, dan worden ze dynamisch geadresseerd anders statisch.
Uitzondering: *own* <*variabele*> wordt altijd statisch geadresseerd.
pseudo - for -variabelen zijn administratieve hulpvariabelen die naamloos zijn en alleen in een for - statement gebruikt worden.

Opmerking: bij de eerste en tweede scan vindt er bijna geen syntactische controle plaats. Dit is een nadeel voor programma's die niet aan de derde scan toekomen (b.v. een programma met een *end* te weinig).

ad 4) De bibliotheek, die I/O procedures, standaard functies e.d. bevat, is permanent in het kerngeheugen aanwezig. Dit is niet noodzakelijk. K.K. Koksma (MC) heeft een systeem gerealiseerd waarin de bibliotheek zich bevindt in het trommelgeheugen. Uit dit achtergrondgeheugen worden tijdens vertalen van een programma die procedures die tijdens executie nodig zijn, in het kerngeheugen opgeslagen.

I-3 Foutmeldingen

230270

We hebben gezien dat de vertaler van de X8 drie scans bezit. De vertaler heeft nog een 4e gedeelte met de naam General Purpose Procedures. Deze G.P.P. bevat bepaalde subroutines (hoofdzakelijk naamlijstroutines, lezen volgende symbool op de band e.d.) die in minstens 2 scans gebruikt worden.

Daar de translation-scan het vertaalwerk doet zal deze scan ook de meest diepgaande analyse doen. De meeste fouten worden dan ook door deze scan gedetecteerd. In G.P.P. kunnen gevonden worden de fouten met nummer 100 t/m 110 (aantal is dus 11). De fouten nr. 100 t/m 103 slaan op de fouten gevonden door *next symbol*, die met nr. 104 t/m 110 op andere fouten bijvoorbeeld na *own* geen <*type*>.

Vanuit *prescan 0* kunnen foutmeldingen 111 t/m 130 (aantal 20) verwacht worden. Dit zijn hoofdzakelijk fouten zoals dubbel declareren in hetzelfde blok. Evenzo vanuit *prescan 1* de nr's 201 t/m 204 (dat zijn er 4) en ten slotte in de vertaal scan de nr's 300 t/m 401 (102 stuks).
Soms komt een programma niet aan de 3e scan toe, dit kan gebeuren als er

1) Een sluithaak van een of andere soort is weggelaten, waardoor de vertaler na de "laatste" *end* nog meer tekst vraagt. Het bedrijfssysteem breekt de vertaling dan af. De sluithaak kan zijn behalve *end* b.v. } (dan wordt alles na { geskipt dus ook de laatste *end* enz.), of] (in een arraydeclaratie is de vertaler op zoek naar de met [corresponderende]).

2) Dubbele declaraties voor één naam in het zelfde blok. Dit is zo geconstrueerd omdat anders in de 3e scan te grote moeilijkheden zouden ontstaan.

3) Ruimte gebrek, d.w.z. "space" is vol.

Als geen van de drie bovenstaande gevallen optreedt krijgt men meer informatie over het ingelezen programma.

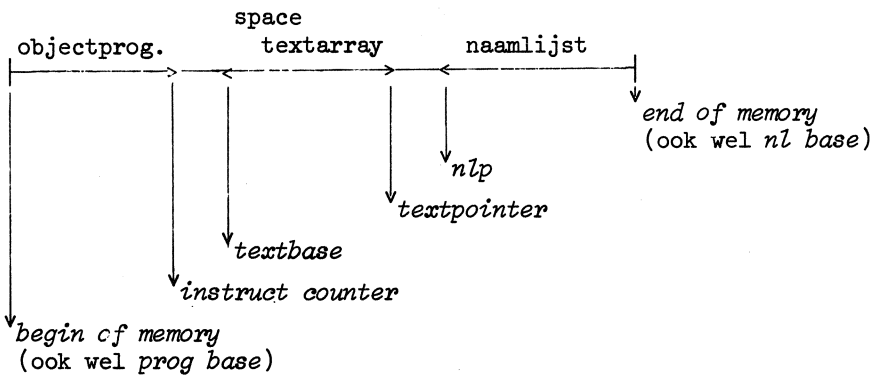
Opm.: mocht deze informatie uit een serie foutmeldingen bestaan dan is het raadzaam niet meteen de eerste fout van die lijst proberen op te sporen en te verbeteren. De fouten worden namelijk per scan die ze detecteert afgedrukt (dus niet noodzakelijk naar opklimmend regelnummer gesorteerd) en aangezien een fout ergens in het programma vaak verderop nog een of meer fout constateringen induceert, die mogelijk in een eerdere scan gemeld worden, is het dus beter eerst de fouten met een nummer hoger dan 300 te bekijken.

De translation scan is historisch gezien de oudste (dat is nog te zien want hij gaat soms voorbij aan informatie die door de 1e en 2e scan reeds verzameld is). Vervolgens heeft men uitgaande van de translation scan de *prescan 1* ontworpen, die informatie verzamelt die in de 3e scan bekend verondersteld wordt. Dit is niet veel, zo is *prescan 1* bij aritmetische expressies niet geïnteresseerd in prioriteiten doch b.v. in identifiers die dan blijkbaar van aritmetische aard moeten zijn. Evenzo is als laatste de *prescan 0* ontstaan, die zeer grof is omdat enerzijds zijn taak beschei-

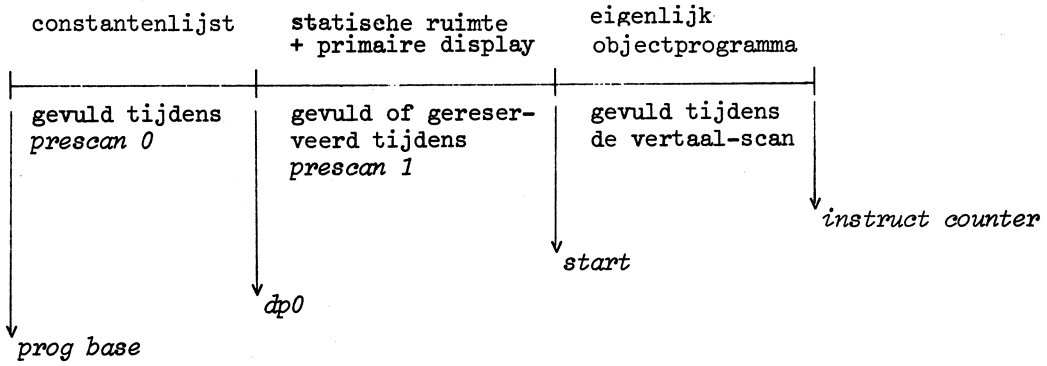
den is en anderzijds weinig ruimte beschikbaar was (men wilde de vertaler niet meer dan 5000 woorden laten bevatten, wat net lukte).

I-4 Geheugengebruik tijdens de vertaling

Ruimteverdeling tijdens vertaling.



nl is een afkorting voor naamlijst, *p* voor pointer.
De ruimte voor het objectprogramma is weer als volgt verdeeld



In de constantenlijst zitten alle constanten uit het behandelde programma die real zijn en de integers $\geq$ 32768. In de statische ruimte bevinden zich zoals reeds eerder vermeld de locaties van de statisch geadresseeerde variabelen en pseudo-variabelen. *dp0* is een afkorting van displaypointer-nul.

I-5 Overzicht van de lengte van de modulen en scans

Het volgende schema geeft een overzicht van de hoeveelheid ELAN instructies die een scan voor een gedeelte van het werk nodig heeft.

	General Purpose Procedures	*prescan 0*	*prescan 1*	*translate*	totaal
syntax analyzer + macro generator	25	226	556	2025	2832
macro processor	-	-	-	239 [1]	239
name list handling	176	223	330	246	975
next symbol routines	499 [2]	-	-	-	499
unsigned number routines	76 [3]	6	-	23	105
initialization	20	54	18	27	119
totaal	796	509	904	2560	4769

1) Exclusief *macrolist* (148 + 55 woorden) en *instructlist* (209 instructies).

2) Exclusief tabel *word delimiter* (59 woorden).
In deze tabel staan alle toegelaten onderstreepte letterstrings.
De next symbol routines controleren ook op pariteit etc.

3) Exclusief dec-bin routine.
Deze zet getallen uit 10-tallig stelsel over in het overeenkomstige getal in het 2-tallig stelsel.

De boven gedistribueerde ELAN instructies zijn verkregen door het met de hand vertalen van de ALGOL versie. Er zijn nu dus 2 versies van de vertaler n.l.

A) de ALGOL versie

B) de ELAN versie.

B) vertaalt een ALGOL programma b.v. A) in ELAN instructies. Doet men dit dan krijgt men ca. 16000 instructies.

Dit is een verlies van een factor 3 t.o.v. het handwerk.

II De vertaling van aritmetische expressies

II-1 Het vertaalschema van aritmetische expressies

Bij de semantiek van de aritmetische expressie maakt het Revised Report onderscheid tussen integer-resultaten (die exact zijn) en real-resultaten (die afwijken van het mathematisch exacte resultaat). Het resultaat van een aritmetische expressie is slechts dan van het type integer, als

1) alle er in voorkomende numbers, variables en function designators integer zijn
2) slechts de operatoren +, −, * en ÷ voorkomen, en eventueel ↑ mits de exponent daarbij niet-negatief is.

Het onderscheid tussen type real en integer is semantisch van belang o.a. bij de ÷ operator, waar geeist wordt dat beide operanden integer zijn, en bij de machtsverheffing, waarbij b.v. $(-1) \uparrow e$ slechts gedefinieerd is als e het type integer heeft.

De vertaler is echter niet in staat in alle gevallen het type van een aritmetische expressie vast te stellen. Met name hangt het type van $i \uparrow j$, met i en j beide integer, af van het teken van j, terwijl het type van een niet-value formele parameter afhangt van het type van de corresponderende actuele parameter, en dus zelfs niet bij iedere aanroep hetzelfde hoeft te zijn. Indien nu, zoals bij de meeste rekenautomaten, de representatie van rekenresultaten afhangt van het type, is het noodzakelijk tijdens de executie van het programma bij alle bewerkingen het resulterende type vast te stellen en te administreren, ten koste van veel extra werk. Het is derhalve van vitaal belang dat bij de X8 de representatie van de integers consistent is met die van de reals (de integers dus een echte subset van de reals vormen). Dit ontslaat ons van de typeregistratie en van type-transformatie (in sommige machines nodig als bij een operatie de twee operanden niet van hetzelfde type zijn). Dat het mogelijk is, dat hierdoor bij de ÷-operator een van de operanden schijnbaar exact is (representatie met binaire exponent 0), maar ontstaan is uit b.v. operaties met real operanden, zonder dat we dat tijdens de executie merken (we testen slechts op het nul-zijn van de exponent) nemen we daarbij graag op de koop toe.

Bij het splitsen van een aritmetische expressie in zijn delen onderscheiden we de volgende operatoren

1) monadische operator. Deze heeft één operand v.b. + en -
2) dyadische operator. Deze heeft twee operanden, v.b. +, -, /, ÷, *, ↑.

020370

Om een indruk te krijgen van het eigenlijke vertaalwerk beschouwen we de aritmetische expressie (= AE)

$$-x \;/\; y \;*\; z \;-\; u \;/\; (v \;*\; w) \quad .$$

Grammaticaal past een AE in de structuur

```
 <AE>::= <SAE> | <if clause> <SAE> else <AE>
<SAE>::= <T> |
         +<T> |        } hierin zijn de + en - de monadische operatoren
         -<T> |        }
         <SAE> + <T> | } worden uit eenvoudiger voorbeelden van SAE's
         <SAE> - <T>   } opgebouwd m.b.v. de dyadische operatoren + en -.
<T>::= <F> |
       <T> * <F> |
       <T> / <F> |
       <T> ÷ <F>
<F>::= <P> |
       <F> ↑ <P>
<P>::= <UN> | <VAR> | <FD> | (<AE>).
```

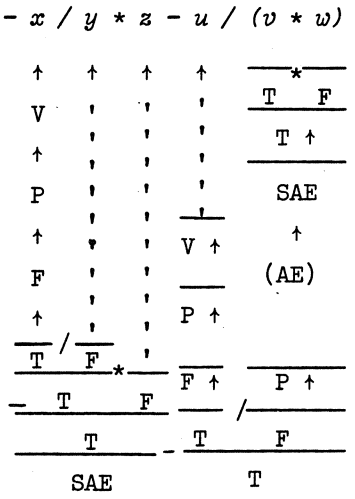
Hierin geldt SAE = simple arithmetic expression,
T = term,
F = factor,
P = primary,
UN = unsigned number,
VAR = variable,
FD = function designator.

De grammaticale vormen impliceren tevens de prioriteitsregels.
Voorbeeld.

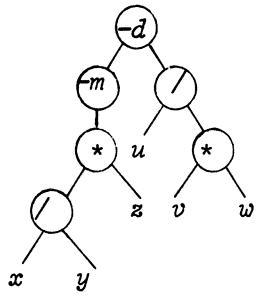
2 ↑ 3 ↑ 4
betekent $(2^3)^4 = 2^{12}$ en niet $2^{(3^4)} = 2^{81}$, want grammaticale analyse levert $(2^3)^4$.

De interpretatie van evalueren van een expressie is: uitrekenen zoals deze gegenereerd is.

In ons speciale voorbeeld kan men de AE terugbrengen tot de volgende structuur



of anders in de zogenaamde boomvorm



Hierin geldt:

-*d* is de dyadische min operatie, d.w.z. deze heeft twee operanden

-*m* is de monadische min operatie, d.w.z. deze heeft één operand.

In het volgende overzicht m.b.t. de mogelijkheden van vertaling van een SAE staat *"F = T"* of *"F = SAE"* e.d. symbolisch voor een stuk objectprogramma dat de term respectievelijk de SAE uitrekent en de waarde ervan in F aflevert.

vorm SAE	symbolisch vertaalschema	macronamen
T	*"F = T"*	
+ T	*"F = T"*	
- T	*"F = T"*	
	F = -F	*NEG*
SAE + T	*"F = SAE"*	
	MC = F	*STACK*
	"F = T"	
	F + MC [-2]	*ADD*
SAE - T	*"F = SAE"*	
	MC = F	*STACK*
	"F = T"	
	F = -F	*SUB*
	F + MC [-2]	
T * F	*"F = T"*	
	MC = F	*STACK*
	"F = Factor"	
	*F * MC [-2]*	*MUL*
T / F	*"F = T"*	
	MC = F	*STACK*
	"F = Factor"	
	MC = F	*DIV*
	F = MC [-4]	
	F / MC	
T ÷ F	*"F = T"*	
	MC = F	*STACK*
	"F = Factor"	
	SUBC(:IDI)	*IDI*
F ↑ P	*"F = Factor"*	
	MC = F	*STACK*
	"F = P"	
	SUBC(:TTP)	*TTP*

Opmerkingen.

1) macro is een verzamelnaam voor groepen bestaande uit één of meer machine-instructies die een taak moeten verrichten die in niet machine-gebonden taal omschreven is. Hierbij onderscheiden we parameterloze macros die acties specificeren (*STACK*, *ADD*, *MUL*) en macros die pas acties specificeren wanneer de macroparameter erbij gegeven is (*TRV*, *COJU*, *JU*).

2) Voor alle dyadische operatoren is het vertaalschema ruwweg
   evalueer de 1° operand (resultaat in *F*)
   STACK
   evalueer de 2° operand (resultaat in *F*)
   voer de operatie uit met de waarde aangegeven door de top van de stapel als eerste en de waarde in *F* als tweede operand.

   De macro *STACK* legt de inhoud van *F* op de stapel. De *MC* operand betekent: het adres wordt gevonden in het *B* register en achteraf wordt *B* passend (dat is hier met 2) verhoogd.
   Als neveneffect van het uitvoeren van de operatie (macros: *ADD*, *SUB*, *MUL*, *DIV*, *IDI*, *TTP*) moet *B* weer met 2 verlaagd worden zodat de nettoverandering van *B* nul is.

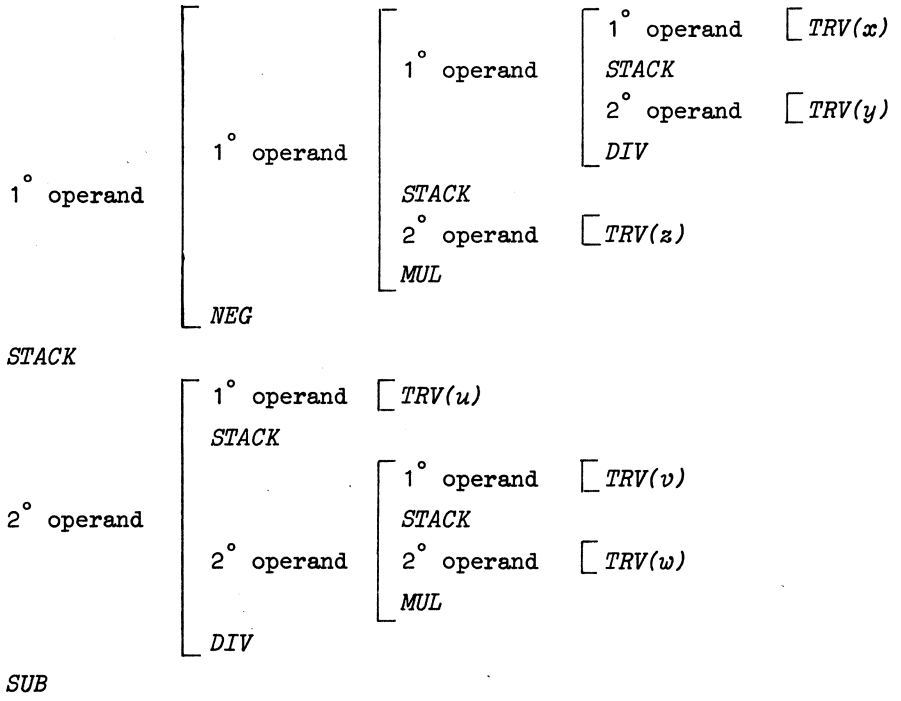
3) De macro *SUB* had ook uit de instructies *F - MC [-2]*; *F = -F* mogen bestaan.

4) De heling en machtsverheffing zijn ingewikkelder. Daartoe bedienen wij ons van de subroutines zoals *IDI* (= Integer DIvision) bij heling en *TTP* (= To The Power) bij machtsverheffing, die hun operanden op de top van de stapel en in *F* verwachten en hun resultaat in *F* afleveren.

5) Verwar F als afkorting voor Factor (in de rubriek structuur AE) niet met *F* als aanduiding van het F-register (rubriek symbolisch vertaalschema).

Dit schema gaan we nu toepassen op het eerder genoemde voorbeeld.
Laten we aannemen dat de simpele variabelen van het type real zijn en nog in het midden laten of ze statisch dan wel dynamisch geadresseerd zijn.
Met behulp van elementen uit het symbolisch vertaalschema kunnen we ons

voorbeeld herleiden tot de volgende programmastructuur



Als we dit gestrekt opschrijven dan krijgen we

```
F = x          "TRV(x)    (= Take Real Variable x)
MC = F         "STACK
F = y          "TRV(y)
MC = F         "  |
F = MC[-4]     "  | DIV
F / MC         "  |
MC = F         "STACK
F = z          "TRV(z)
F * MC[-2]     "MUL
F = -F         "NEG
MC = F         "STACK
F = u          "TRV(u)
MC = F         "STACK
F = v          "TRV(v)
MC = F         "STACK
F = w          "TRV(w)
F * MC[-2]     "MUL
MC = F         "  |
F = MC[-4]     "  | DIV
F / MC         "  |
F = -F         "  |
F + MC[-2]     "  | SUB
```

Als we de macro's *STACK*, *TRV(..)* weglaten en *MUL* vervangen door * enz. dan krijgen we

*x y / z * - u v w * / -*

De notatie heeft de naam polish reversed of postfix.
Er bestaan de volgende notaties:

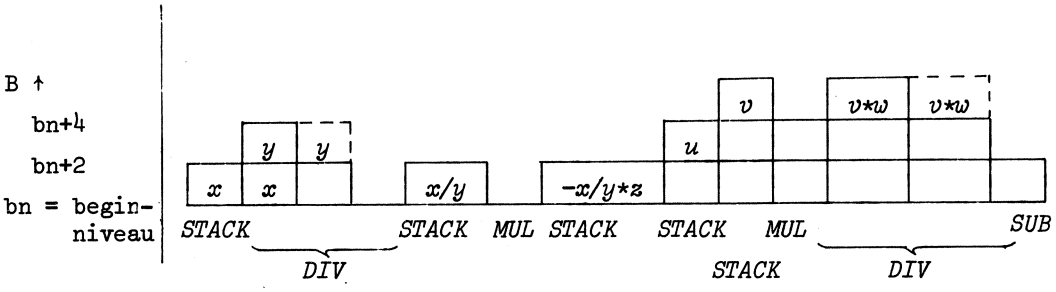
a + b	infix
ab +	postfix
+ ab	prefix.

Het essentiële bij de postfix notatie is

1°. dat overeenkomstig de machine verwerking, eerst de operanden worden opgeschreven en dan pas de bijbehorende operatie

2°. haakjes zijn overbodig.

Om een duidelijk idee te krijgen omtrent wat er tijdens de verwerking van ons voorbeeld in de stapel gebeurt geven we het volgende diagram



In het geval van: <AE>::= *if* <BE> *then* <SAE> *else* <AE>
dan krijgen we voor het corresponderende objectprogramma

```
        "C = BE"
    N,  GOTO (:LO)        "COJU (LO)
        "F = SAE"
        GOTO (:L1)        "JU (L1)
LO:     "F = AE"
L1:       .
```

II-2 De algoritme *Arithexp*

090370

Na gezien te hebben hoe de vertaling van een aritmetische expressie eruit ziet, zullen we ons bezig houden met het mechanisme dat die vertaling produceert. De algoritme *Arithexp* dat daarvoor zorgt is als volgt geplaatst in het geheel

```
begin  procedure . . . . . .
       . . . . . . . . . . . . ;
       integer procedure next symbol;   } G.P.P.
       . . . . . . . . . . . . ;
       procedure . . . . . . . ;
       procedure prescan 0;  . .
       . . . . . . . . . . . . ;
       procedure prescan 1;  . .
       . . . . . . . . . . . . ;
       procedure translate;  . .
       begin . . . . . . . . . ;
              procedure Arithexp;
              . . . . . . . . ;
              next symbol; Program;
              . . . . . . . . ;
       end translate;
       prescan 0; prescan 1; translate
end
```

N.B. *Arithexp* is locaal ten opzichte van *translate*.

De overgang van ALGOL naar instructies gaat in twee stappen

1) *Arithexp* (zie blz. 95) produceert macro's volgens het reeds geschetste algemene schema.
2) De macroprocessor "optimaliseert" nog wat en bouwt uit de macro's de feitelijke instructies.

Arithexp maakt zo als bijna alle procedures in de vertaler veelvuldig gebruik van de procedure *next symbol* en de variable *last symbol*.
last symbol en *next symbol*.

Bij veel grammaticale constructies is het nodig één symbool te ver te lezen om te zien of die constructie al dan niet compleet gelezen is. Dit is zeer duidelijk bij <unsigned integers>. Immers

<unsigned integer>::= <digit> | <unsigned integer> <digit>

m.a.w. een rijtje van minstens één cijfer.
Het is nu zonder meer duidelijk dat men "te ver" moet lezen om te kunnen zien of dat rijtje al compleet gelezen is. Om dit soort problemen op te lossen is de gewoonte ingesteld om altijd te lezen t/m het na-laatste symbool. *Next symbol* leest het eerstvolgende basic symbol (hij skipt dus spaties en andere lay-out symbolen) en bergt het, als getal gecodeerd, op in *last symbol*. Bij de bestudering van de ALGOL-tekst van de ALGOL-vertaler is kennis van deze interne representatie van de basic symbols niet nodig, dank zij de invoering van een aantal variabelen als *plus*, *minus*, *if*, *open*, *begin*, etc., die allemaal geïnitialiseerd worden op de bijbehorende interne representatie. Als we nu bijvoorbeeld willen weten of het laatst gelezen symbool *begin* (interne representatie 104) was, kunnen we volstaan met: *if last symbol = begin then .....;* wat tevens de leesbaarheid bevordert.
Als we een afspraak hebben gemaakt omtrent het systematisch "te ver" lezen, is automatisch het eerste symbool van de er op volgende grammaticale constructie al gelezen en opgeborgen in *last symbol*. De enige uitzondering met betrekking tot deze afspraak is de procedure *Program*, die het gehele programma leest t/m laatste *end* en natuurlijk niet verder (desalnietemin moet er na de laatste *end* een niet onderstreept karakter - niet noodzakelijk een basic symbol - komen omdat *next symbol* na 2 onderstreepte letters alles skipt tot de onderstreping ophoudt. Gevolg: *endelse* mag niet *end else* wel en *procedare* wordt geïnterpreteerd als *procedure)*.

Procedure *Arithexp*.

Aan het feit dat *Arithexp* met een hoofdletter geschreven wordt kan men zien dat deze procedure locaal t.o.v. een scan is (dit is een afspraak). De scan waar *Arithexp* in thuis hoort is de translation scan.
De meeste vertaal procedures hebben de volgende globale werking

a) ze verwachten het 1e symbool van de te vertalen syntactische eenheid in *last symbol;*
b) ze leveren een stuk objectprogramma af dat de vertaling bevat;
c) ze laten het 1e symbool van de volgende syntactische eenheid achter in *last symbol*.

Een van de weinige uitzonderingen is *Program* waar c) vervalt; *Arithexp*

werkt wel volgens bovenstaand schema.
We brengen ons de syntax van AE in herinnering

<AE>::= <SAE> |
if <BE> *then* <SAE> *else* <AE> .

Arithexp kijkt eerst of het 2e geval optreedt m.a.w. of *last symbol = if.* Zo nee dan roept hij *Simple arithexp* aan, zo ja dan moet er komen

```
        "C = BE"
      N, GOTO (:L0)
        "F = SAE"
        GOTO (:L1)
L0:     "F = AE"
L1:
```

N.B. bij *GOTO (:<label>)* moeten we wel bedenken dat niet volstaan kan worden met een symbolisch adres in de vorm van een identifier maar moeten we echt de getalwaarde van het adres waar we naar toe willen springen invullen. De moeilijkheid is nu dat we het adres van de 2 labels moeten aangeven en nog niet weten hoeveel ruimte *"F = SAE"* en *"F = AE"* in zullen nemen. De oplossing is gevonden in een trucje, een neveneffect van de procedure *Macro 2*, dat optreedt als de eerste parameter een *forward jumping macro* is. De hulpvariable, de integer *future 1* is de 2e parameter, *future 1* is eerst nul gemaakt. Als nu *Macro 2 (COJU, future 1)* aangeroepen wordt, wordt

1) de conditionele sprong

N, GOTO (: "adres met waarde van future 1, dus voorlopig = 0")

gegenereerd;

2) aan *future 1* wordt geassigneerd de waarde van het adres waar de sprong uit 1) staat.

Als we nu aangekomen zijn op de plaats waar naar toe gesprongen moet worden (b.v. adres a) roepen we *Substitute (future 1)* aan. De werking daarvan is dat de waarde a ingevuld wordt in de sprong opdracht op het adres vastge-

legd in *future 1*.

We beschouwen nu meer in detail het geval

<AE>::= *if* <BE> *then* <SAE> *else* <AE>

en het laatst gelezen symbool (dus opgeslagen in *last symbol*) is *if*. We lezen nu het volgende symbool (d.i. eerste symbool van BE), slaan dit op in *last symbol* en roepen *Boolexp* aan. Als *Boolexp* de BE vertaald en opgeslagen heeft en symbool *then* in *last symbol* gezet, wordt de aanroep *Macro 2 (COJU, future 1)* gedaan met boven beschreven gevolg. Er wordt vervolgens gecontroleerd of *last symbol* wel *then* bevat, zo niet dan treedt foutmelding 300 op, anders lezen we het volgende symbool (eerste symbool van SAE).
We roepen *Simple arithexp* aan, deze levert een stuk objectprogramma, dat de SAE evalueert, symbolisch hier aangeduid door *"F = SAE"* en laat het eerste niet tot de SAE behorende basic symbol achter in *last symbol*. *Arithexp* controleert of dit *else* is, zo niet geeft hij foutmelding 301, zo wel dan genereert hij

GOTO (:L1)

of beter

GOTO (:"adres nul").

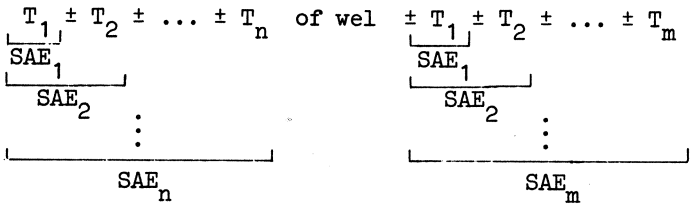
We vullen nu pas het juiste adres van *L0* in, genereren *"F = AE"* en vullen het juiste adres van *L1* in.

Bespreking *Simple arithexp*.

Herhaling <SAE>::= <T> |
+ <T> |
- <T> |
<SAE> + <T> |
<SAE> - <T> .

Simple arithexp maakt gebruik van het feit dat elke SAE, onafhankelijk van het feit, of hij van de gedaante 1 t/m 3 dan wel 4 of 5 is, altijd begint

met +<T>, -<T> of <T>.
De totale structuur van een SAE is immers



Simple arithexp onderzoekt, of het eerste karakter een + of - is. Zo ja, dan wordt het volgende symbool gelezen. Vervolgens genereert een aanroep van *Term* de vertaling van de eerste term T_1. Indien SAE met een - begint, wordt hierna de macro *NEG* ingelast. De aanroep van *Next term* genereert (recursief) de vertaling van de eventuele rest van de SAE. Indien *Next term* een + of - in *last symbol* aantreft, wordt er achtereenvolgens de macro *STACK* (die tijdens executie van het objectprogramma het resultaat tot dan toe moet redden), de vertaling van de eerstvolgende term en de macro *ADD* of *SUB* gegenereerd, waarna een nieuwe aanroep van *Next term* de (eventueel nog overgebleven) rest vertaalt.
Het proces eindigt zodra bij een aanroep van *Next term* de BE:

$$last\ symbol \neq plus \ \wedge \ last\ symbol \neq minus$$

true is m.a.w. als de SAE volledig gelezen is.

We hadden ook de procedure *Simple Arithexp* kunnen schrijven als volgt

```
procedure Simple arithexp 2;
begin  if last symbol = minus then
       begin next symbol; Macro (NEG); Term
       end else
       begin if last symbol = plus then next symbol;
             Term
       end;
   l:  if last symbol = plus then
       begin  Macro (STACK); next symbol; Term;
              Macro (ADD); goto l
       end else
       if last symbol = minus then
       begin  Macro (STACK); next symbol; Term;
              Macro (SUB); goto l
       end
end Simple arithexp 2;
```

De procedure *Next term* is nu overbodig. Dat *Simple arithexp* niet zo geschreven is heeft 2 redenen en wel

1) Men mag niet van buiten af naar een label (d.w.z. naar binnen) in een procedure springen. Met andere woorden als men de procedure *Next term* ergens anders nodig heeft dan moet men hem alsnog schrijven want *goto l* buiten het blok van *Simple arithexp 2* is verboden in ALGOL 60.
2) Springen naar een label i.p.v. gebruik van een (recursieve) procedure *Next term* maakt het moeilijker de correctheid van de vertaler te verifiëren (vergelijk hoofdstuk VII).

Op blz. 104 staat in een schema weergegeven, welke procedureaanroepen bij het eerder besproken voorbeeld

$$- x/y * z - u/(v*w)$$

tijdens de analyse van deze expressie worden gedaan. Hierin zijn de namen

van die procedures afgekort zoals op blz.104 is aangegeven en staat reeds tussen [en] de inhoud van *last symbol* op het moment van aanroep. Met een vraagteken is aangegeven een willekeurig symbool dat niet meer tot de expressie kan behoren (b.v. *do* of ;).

II-3 De vertaling van een primary

160370

In het vervolg zullen we vaak verwijzen naar [2] Het objectprogramma, gegenereerd door de X8-ALGOL 60 vertaler van het M.C.

In het vorige college is het vertalen van een aritmetische expressie door middel van het recursief gebruiken van procedures, teruggebracht tot het vertalen van een term.
De vertaling van een term wordt teruggebracht tot de vertaling van een factor en de vertaling daarvan wordt teruggebracht tot de vertaling van een primary.
(Voor structuur zie blz. 18 en voor het terugbrengen van de eigenlijke vertaling van een term zie blz. 95 met name *Term* en de verdere aanroepen van procedures vanuit deze procedure.)

Bespreking vertaling van een primary.
In het Revised Report vinden wij de volgende mogelijkheden voor een *Primary*

1. UN, dus *last symbol* bevat: <digit> $|.|_{10}$
2. VAR, " " " " : <letter>
3. FD, " " " " : <letter>
4. (AE), " " " " : (

De *procedure Primary* (zie blz. 96) moet dus deze gevallen onderscheiden. Bij het binnengaan van de procedure body zien we dat inderdaad onderscheid wordt gemaakt in de gevallen

last symbol bevat: <digit> $|.|_{10}$
last symbol bevat: <letter>
last symbol bevat: (

door middel van *if* statements met respectievelijk de BE

if last symbol = open then
if digit last symbol then
if letter last symbol then .

Als al deze BE's *false* zijn wordt foutmelding 303 gegeven en, indien *last symbol* een +, - of *if* bevat, de vertaling voortgezet door een aanroep van *Arithexp*. Hierdoor wordt bij vergeten van een haakjes paar toch een verdere syntactische analyse verkregen, bijvoorbeeld bij

x + if i > 3 then 1 else 2 + 3 i.p.v. *x + (if i > 3 then 1 else 2) + 3*

of

x ↑ -3 i.p.v. *x ↑ (-3).*

Als de BE *last symbol = open true* is dan moet voordat *Arithexp* aangeroepen wordt, eerst *next symbol* aangeroepen worden, omdat *Arithexp* het eerste symbool van de aritmetische expressie in *last symbol* verwacht.
Arithexp leest (en vertaalt) tot en met het eerste symbool, dat niet meer tot de AE behoort. Dit symbool dient een sluithaakje te zijn. Zo ja, dan wordt het volgende symbool gelezen (*Primary* moet n.l. het eerste symbool dat niet meer tot de P behoort in *last symbol* achterlaten); zo nee, dan volgt foutmelding 302.
Als de BE *digit last symbol true* is dan wordt de vertaling van de primary in twee stappen gegenereerd

1) *Unsigned number* leest het getal en bepaalt zijn waarde (opgeborgen in *value of constant*), aard (*small integer* = integer < 32768, *integer* of *real*) en in geval het een integer of real is, zijn adres in de constantenlijst (het getal is hierin dan opgeslagen tijdens de 1e scan). Dat eventuele adres wordt dan opgeborgen in *address of constant*.
2) *Arith constant* genereert de macro *TSIC* (= Take Small Integer Constant) met als macro parameter *value of constant*, of de macro *TIC* (= Take Integer Constant) of *TRC* (= Take Real Constant) met als macroparameter

address of constant (zie [2] §1.3.7).
De splitsing in twee stappen is nuttig omdat we soms (b.v. bij actuele parameters) een <unsigned number> op een andere manier vertalen, maar dan wel alle gegevens uit stap 1 nodig hebben.
Als de BE *letter last symbol* *true* is dan wordt in <type> *procedure* *Identifier* de hele identifier gelezen, in de naamlijst opgezocht en de waarde van de pointer in de naamlijst wordt aan *n* geassigneerd.
Hierna volgen aanroepen van: *Subscripted variable*, *Function designator* en *Arithname*. Als de identifier een array identifier is, dan wordt de vertaling van de variabele door *Subscripted variable* geproduceerd;
als de identifier een procedure identifier is, dan wordt de vertaling van de function designator door *Function designator* geproduceerd;
als de identifier een simpele variabele is, dan wordt de vertaling door *Arithname* geproduceerd.
In alle drie gevallen controleert *Arithname* bovendien of de identifier wel van aritmetische aard is, of, nauwkeuriger omschreven, de identifier expliciet bekend staat als niet aritmetisch van aard te zijn.
In dat geval volgt foutmelding 304.

De procedure *Subscripted variable* wordt in alle drie gevallen aangeroepen maar doet slechts dan iets, als de Boolean procedure *Subscrvar (n)* als antwoord *true* aflevert (deze kijkt in de naamlijst naar de gegevens over de door de pointer *n* aangewezen identifier).
In dat geval wordt de vertaling in twee stappen gegenereerd
1) *Address description (n)*
2) *Evaluation of (n)*.
Opmerking: bij aanroep vanuit *Primary* hoort op de [<*subscript list*>] uiteraard geen := te volgen. De procedure *Subscripted variable* wordt echter ook aangeroepen vanuit *Assignment statement*.

De procedure *Address description* controleert of op de identifier wel een [volgt (zo nee, dan foutmelding 305), laat door *Subscript list* de indexexpressies vertalen en tellen, controleert de dimensie (met behulp van gegevens uit de naamlijst) en produceert een van de macro's *DOS* (*DOS* is geen afkorting maar staat voor: doe iets op de plaats aangegeven door de 2de parameter) of *TAK* (= Take Array Key), (de macro *TSWE* komt slechts aan bod als *Address description* aangeroepen wordt bij de vertaling van de

<designational expression>).

De procedure *Subscript list* genereert (recursief) een stuk objectprogramma, dat alle indexexpressies evalueert, en de uitkomsten alle op de laatste na op de stapel legt. Als functiewaarde wordt het aantal aangetroffen indexexpressies afgeleverd. Tevens wordt de aanwezigheid van de sluithaak gecontroleerd.

De procedure *Evaluation of (n)* genereert (bij aanroep vanuit *Arithexp* etc.) een van de macro's

TFSU (= Take Formal Subscripted of Unknown type)

TSI (= Take Subscripted Integer)

TSR (= Take Subscripted Real)

die bij executie (uit de indexwaarden op de stapel en in *F*, en uit de arraysleutel in *A*) de plaats van het betreffende arrayelement in het geheugen opzoekt en de waarde ervan in *F* overneemt.

De procedure *Function designator* doet slechts iets, als uit de gegevens in de naamlijst blijkt, dat de identifier een procedure identifier is. In de procedure *Arithname* krijgt dan, na onderzoek van het aritmetische karakter van de identifier op plaats *n*, al dan niet gevolgd door foutmelding 304, de Boolean variabele *complicated* de conditie *Formal (n)* $\vee$ *Function (n)* (ten behoeve van vertaling van for statements). Daarna wordt ingeval BE *Simple (n)* *true* is, in space ingevuld als objectprogramma

soort identifier	macro in objectprogramma
formele identifier	*DOS*
integer identifier	*TIV* (= Take Integer Variable)
real identifier	*TRV* (= Take Real Variable)

Indien de BE *Simple (n)* *false* is, is de vertaling van de primary al door een andere procedure gegenereerd.

Tot slot van de bespreking van de vertaling van een primary geven we hier nog enige opmerkingen en mogelijke verduidelijkingen van [2] §1.3.

Opmerkingen

1.3.1. Op de dynamische adresseringswijze wordt later nog nader ingegaan.

1.3.2. *DOS* is geen afkorting maar duidt de X8-instructie *DOS (Mp[q])* aan, die

1e) het dynamisch adres *Mp[q]* evalueert

2e) het gevonden adres in *S* overneemt

3e) de instructie in het geheugen, op de plaats aangewezen door dat gevonden adres, uitvoert.

APIC is een afkorting van Actual Parameter Instruction Cell.

1.3.3. *Reference vector* = *mapping vector* = *storage function* is het karakteristieke element en vertelt voor ieder toegelaten stel indices - dit toegelaten zijn (= blijven indices binnen de bounds) wordt getest aan de hand van de *reference vector* - waar de geïndiceerde variabele is opgeborgen en bevat bovendien de informatie om zo snel mogelijk bij de elementen te komen. (Dit is van belang bij meerdimensionale arrays.)

Ophalen geïndiceerde variabelen.

De instructies om een adresdescriptie te maken worden gegenereerd door de vertaler routine *Address description (n)* (zie blz. 97), die voor het opnemen van de gevraagde waarde in het betreffende register door de vertaalprocedure *Evaluation of (n)*.

De procedure *Address description* wordt ook aangeroepen in de procedure *Assignment statement* voor de vertaling van een geïndiceerd linkerlid; echter volgt dan geen *Evaluation of*.

Het is belangrijk dat in assignment statements eerst de gegevens over de linkerleden worden verzameld, voordat het rechterlid wordt uitgerekend. Dat de volgorde van handelingen consequenties heeft blijkt uit het volgende voorbeeld

```
i:= 4;
a[i]:= i:= 5;
```

wat bij executie levert:

a[4]:= 5; .

1.3.3.1. *AR* (= Address Reference vector)
TAK (= Take Array Key)
1.3.3.2. *TSR* (= Take Subscripted Real)
TSI (= Take Subscripted Integer)
TFSU (= Take Formal Subscripted of Unknown type).

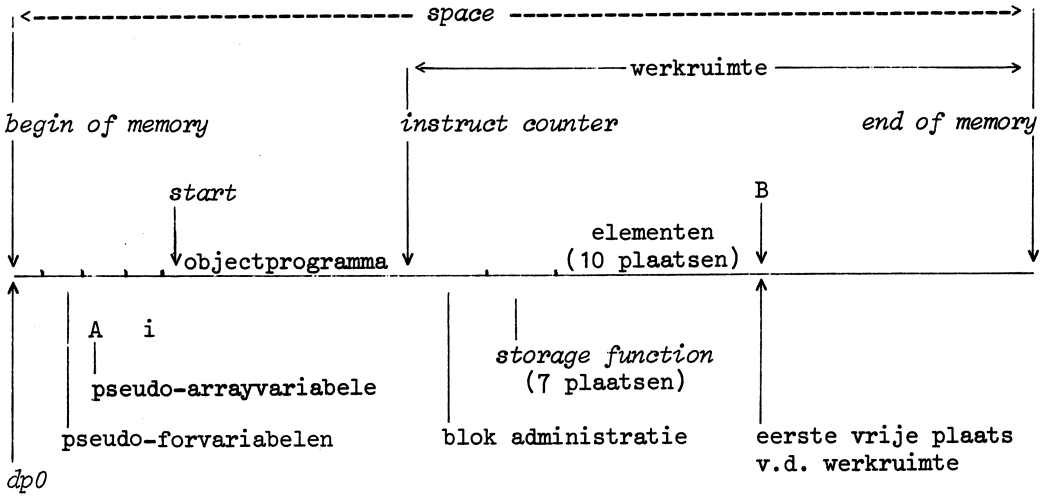
1.3.5. en 1.3.6. Hier wordt later nog op teruggekomen.

1.3.7. De deelgroep: integers in absolute waarde < 32768 worden apart behandeld omdat deze getallen in 15 bits binair zijn op te schrijven en direct in het operand gedeelte van een woord ingevuld kunnen worden.
Alle andere constanten zijn opgenomen in de constantenlijst en worden hierin geadresseerd.
TRC (= Take Real Constant)
TIC (= Take Integer Constant)
TSIC (= Take Small Integer Constant).

II-4 Voorbeelden ter toelichting van de vertaling van de array-elementen

Hoe in de praktijk de plaats gekozen wordt voor een pseudo-arrayvariabele, storage function en elementen van een array kunnen we zien als we de geheugenbezetting tijdens executie van het volgende programma bekijken.

```
begin  integer array A[1:10];
       integer i;
       for  i:= 1 step 1 until 10 do A[i]:= i;
       for  i:= 1 step 1 until 10 do PRINT (A[i])
end
```



Opmerkingen

1) de pseudo-arrayvariabele *Array Key* wijst altijd naar het laatste woord van de *storage function;*
2) er is geen constantenlijst want 1 en 10 behoren tot de Small Integer Constants; *prescan 0* vult dus niets in;
3) de ruimte van *begin of memory* tot aan *start* is ingevuld of gereserveerd tijdens *prescan 1*;
4) de *storage function* zit altijd in de stapel;
5) de pseudo-forvariabele helpt bij uitvoering van de for statements.

Om te illustreren dat *address description (n)* zowel in assignment statements als voor de evaluatie van primary's gebruikt wordt, beschouwen we het volgende voorbeeld, waarin een array element met 1 verhoogd wordt.

Ar[i]:= Ar[i] + 1;

met vertaling:

```
G  = i          "TIV(i)      ┐
A  = M[Ar]      "TAK(Ar)     │
MC = F          "STACK   1)  │  geproduceerd door  Subscripted variable
MC = A          "STAA        ┘
G  = i          "TIV(i)      ┐  geproduceerd door                    │
A  = M[Ar]      "TAK(Ar)     ┘   Address description  │ geproduceerd door
SUB (: IND)     " ┐ TSI      ┐  geproduceerd door     │  Subscripted varia-
G  = MA           ┘          ┘   Evaluation of        │ ble
MC = F          "STACK       ┐
F  = 1          "TSIC(1)     │  geproduceerd door  Next term  2)
F  + MC[-2]     "ADD         ┘
SUB 2 (: STSI) "STSI (= STore Subscripted Integer)  0)
```

0) Hierin staat *STSI* voor de runtime routine:

```
STSI:     S = F, Z      "reeds integer?
       Y, GOTO(:SSTSI)
          F + SCHOLTEN
          F - SCHOLTEN
          S = F, Z      "heeft de afronding plaats gehad met gewenst
       N, SUBC(:ERRORTABLE [5])                          resultaat?

SSTSI:    stock 1 = G   "Simplified STore Subscripted Integer
          A = MC[-1]    "haalt array sleutel van stapel
          F = MC[-2]    "haalt laatste index van stapel
          SUB(:IND)
          G = stock 1
          MA = G        "SSTI
          GOTO(LINK [2]) "13 instructies
```

(zie MR 84, blz. 56).

1) F bevat een integer; toch bergen we twee woorden op in de stapel, omdat in het algemeen een indexexpressie best een real resultaat mag opleveren.

2) Van deze drie opdrachten maakt de macroprocessor één instructie n.l.: *F + 1*. Zie blz. 41.

Het geproduceerde objectprogramma is, gezien vanuit het standpunt van efficiency, bedroevend aangezien twee maal,(onafhankelijk van elkaar) het adres van het array element wordt uitgerekend. Dit is een gevolg van het feit dat de vertaler de onderdelen van een assignment statement ook onafhankelijk van elkaar uitrekent. Dit is altijd safe en houdt de vertaler klein.

200470

We hebben ons tot nog toe bezig gehouden met de vertaling van een AE en we maken nog enkele slotopmerkingen daarover.

Het stuk vertaler dat een AE vertaalt is opgesplitst in een aantal procedures. Dit is gedaan om reden dat een onderdeel van een routine soms nodig is voor de vertaling van andere grammaticale constructies. Dat gedeelte is dan afgesplitst en geschreven als zelfstandige procedure. Zo wordt b.v. een geïndiceerde variabele in 2 stappen geevalueerd, n.l.; adresdescriptie en waardebepaling want het stuk objectprogramma dat de adresdescriptie behandelt is ook nodig voor het geval dat het linkerlid van een assignment statement een geïndiceerde variabele is (zie blz. 36 en 38). Vele procedures hebben dan ook meer dan een functie. Zie b.v. blz. 97 in de procedure *Subscripted variable*, daar komt de conditie *last symbol = colonequal* voor en aangezien er in een AE nooit := voor kan komen zal bij een aanroep van *Subscripted variable* bij de vertaler van een AE slechts het else-deel van de conditional statement met die conditie als BE in werking treden. Het then-deel van die conditional statement wordt gebruikt indien *Subscripted variable* aangeroepen wordt bij de vertaling van een assignment statement. Evenzo zal bij een aanroep van *Evaluation of* bij de vertaling van een AE hoogstens *Formal (n)* en/of *Integer (n)* *true* zijn,de andere booleans zijn zeker *false* en een deel van de procedure blijft dus ongebruikt als er een AE vertaald wordt.

II-5 De macroprocessor: genereren van instructies en optimaliseren van AE's

Een kleine optimalisering van AE's gebeurt nog door de macroprocessor. Ter inleiding de volgende voorbeelden.

We nemen als eerste voorbeeld de AE: *1 + 1* (Er zijn compilers die hier al tijdens vertaaltijd *2* van maken; de ALGOL-vertaler van de X8 doet het niet omdat de auteur van een programma een goede reden kan hebben om *1 + 1* te schrijven in plaats van *2*.) Het vertaalschema van deze AE is als volgt:

```
F  = 1        "TSIC(1)
MC = F        "STACK
F  = 1        "TSIC(1)
F  + MC[-2]   "ADD
```

Dit schema is duidelijk omslachtig, met name de derde instructie is overbodig want het register *F* bevatte al de waarde *1*. Alle informatie over de 1° operand wordt echter weggegooid als de 2° operand geëvalueerd wordt, wat de vertaler veel eenvoudiger maakt.
Toch is het nadeel evident; een beetje optimaliseren levert het schema:

```
F = 1         "TSIC(1)
F + 1         "ADDSIC(1)
```

wat bij executie een zelfde resultaat (n.l. 2) levert.

Een ander voorbeeld is de AE: $x * y$ (aanname: x en y real). Het vertaalschema luidt:

```
F  = x        "TRV(x)
MC = F        "STACK
F  = y        "TRV(y)
F * MC[-2]    "MUL    .
```

Optimaliseren levert het schema:

```
F = x         "TRV(x)
F * y         "MULRV(y)   .
```

Het optimaliseren nu, gebeurt niet door de vertaler die steeds het reeds behandelde algemene schema volgt.

Het optimaliseren gebeurt wel in de macroprocessor (ten overvloede: niet in *Arithexp* en de vandaar uit (indirect) aangeroepen procedures). Zoals boven reeds gesuggereerd genereert de Macroprocessor, bij optimalisatie, macro's die makkelijk te construeren en te begrijpen namen dragen zoals *ADDSIC* en *MULRV*.
Het optimaliseringsmechanisme treedt in werking als er een macro-tripel bestaande uit de macro *STACK*, een Satm en een Oo (in die volgorde) voorkomt. Een Satm (Simple arithmetic take macro) is één van de macro's *TRV*, *TIV*, *TSIC*, *TRC* of *TIC*. Een Oo (Optimizable operator) is één van de macro's *ADD*, *SUB*, *MUL* of *DIV*. De operatoren ÷ en ↑ zijn te ingewikkeld om voor optimalisatie in aanmerking te komen.
De vertaling van een stuk programma met een monadische min operand kan ook geoptimaliseerd worden (de monadische plus verdwijnt automatisch in Next term). Als voorbeeld (i is integer) de AE: $-i$ met vertaalschema

G = i *"TIV(i)*
F = -F *"NEG .*

Na optimalisering

G = -i *"TNIV(i)*

zie [2] §1.4.2.1. Volgens boven beschreven schema optimaliseren leidt niet altijd tot correcte resultaten (zie [2] voorbeeld §1.4.2.1.: de macro's *TSIC(2)* en *NEG* vervangen door *TNSIC(2)* levert geenszins

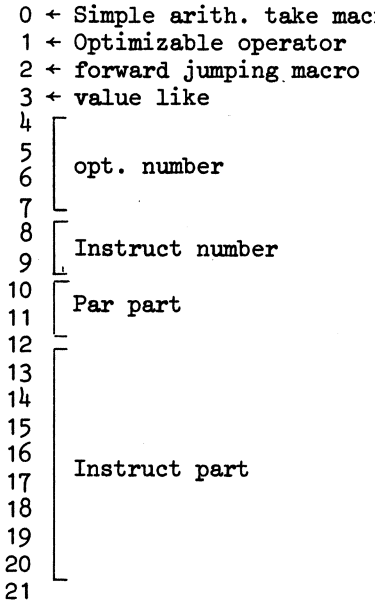
-(if BE then 1 else 2)

zoals de bedoeling was).

We bekijken nu de macroprocessor procedures (blz. 99 t/m 102) en wel speciaal de procedure *Macro 2*. We bespreken echter eerst de procedure *Produce* (blz. 100).

Een macronaam is geen string doch een integer m.a.w. een geheel getal (waarvan we de opbouw nog zullen bespreken). De integer *macro* heeft de waarde van dat getal gekregen. *Produce* onderzoekt of *macro* al dan niet ge-

lijk is aan *EMPTY* of *CODE*. Als *macro = EMPTY* dan doet *Produce* niets. Als *macro* gelijk aan *CODE* is dan wordt de parameter zonder meer opgeborgen in het objectprogramma. Dat opbergen geschiedt als volgt:
Op de plaats in het geheugen na het objectprogramma tot dan toe zetten we de parameter en geven aan dat het objectprogramma met één plaats uitgebreid is (zie blz. 14 voor de betekenis van *prog base*, *instruct counter* enz.). Vervolgens controleren we met *testpointers* o.a. of er nog wel instructies gemaakt kunnen worden; of er b.v. nog ruimte voor is.
Is *macro* ongelijk aan *CODE* en *EMPTY* dan bepalen we het aantal instructies dat de macro levert en bij welke instructie een eventuele parameter hoort.
Zoals we al vermeld hebben is een macronaam een getal. Als we dat getal binair opschrijven, wat in de machine gebeurt, dan is de bitstring zó opgebouwd, dat de bits d8 en d9 het aantal instructies aangeven. Ter verduidelijking geven we een schema van bitnummers en de eraan te hechten betekenis



Het aantal instructies wordt bepaald door *Instruct number* die met behulp van de procedure *bitstring* de bits d8 en d9 uit het getal isoleert.

Twee bits leveren 4 mogelijkheden, het aantal instructies kan 0, 1, 2 of 3 zijn. Aangezien een macro die geen instructies genereert zinloos is hadden we ook met 2 bits aan kunnen geven of een macro 1, 2, 3 of 4 instructies genereert, dit is echter niet gebeurt omdat zoals we zullen zien *Par part* dan 3 bits had moeten bevatten.
De bits d10 en d11 vormen de parameter part.
Als *Par part* nul is, is de macro parameterloos.
Als *Par part* 1, 2 of 3 is heeft de macro een macro parameter die te verwerken is in resp. instructie 1, 2 of 3. Het meest voorkomende patroon is natuurlijk *Instruct number* is 1 en *Par part* is 0 of 1.

De beperking tot 3 instructies per macro is een beperking, opgelegd door de macroprocessor die thans op het M.C. gebruikt wordt. Uiteraard zijn er macroprocessors denkbaar die meer of minder beperkingen opleggen.

Nadat we de waarde van *Instruct number* en *Par part* bepaald hebben en geassigneerd aan de integers *number* en *parnumber*, bepalen we de waarde van *Instruct part* met behulp van de procedure *Instruct part*. We vinden een getal dat 9 bits beslaat (m.a.w. $0 \leq$ *Instruct part* < 512) en dat een pointer is in de lijst van mogelijke instructies. Als parnumber > 0 (m.a.w. de macro is niet parameterloos) dan maakt *Process parameter* de parameter klaar om opgenomen te worden in het objectprogramma. De procedure *Process parameter* zoekt indien nodig het adres van de variabele op in de naamlijst. Onafhankelijk of de macro parameterloos is of niet roepen we *Process stack pointer* aan, dit is een controleprocedure. Tijdens de executie groeit de stapel n.l. voortdurend en er is dus een bewaking nodig want onbewaakt zou de stapel uit het geheugen kunnen lopen. Als we echter bij elke instructie *MC = F* zouden controleren zou dat elke keer minstens 2 instructies vergen, dit zou zeer tijdrovend en duur zijn, we doen het dus niet. We controleren slechts af en toe b.v. bij ingaan van een blok of de stapel niet te groot geworden is. *Process stack pointer* verzamelt de hiervoor benodigde gegevens. Tenslotte construeert *Produce* stuk voor stuk de instructies (haalt ze uit de instructlist en voegt eventueel de parameter toe) en bergt ze op in het objectprogramma met behulp van een aanroep van zichzelf met *CODE* als macro.

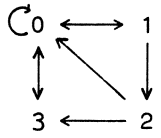
De procedure *Macro* voorziet parameterloze macro's van een loze parameter en roept *Macro 2* aan.

Macro 2 (blz. 99)

De procedure *Macro 2* behandelt macro's met een parameter, optimaliseert ze waar hij kan en stuurt ze naar *Produce*, dus naar het objectprogramma.
Macro 2 kan een aangeboden macro direct doorsturen naar het objectprogramma (via *Produce*) of die macro nog even in voorraad houden
Onder andere slaat de procedure *Load* een macro met zijn eventuele macro-parameter op in resp. *stack0* en *stack1*.
Al naar gelang er iets in voorraad is en zo ja, wat, kan *Macro 2* in 4 verschillende toestanden zijn, deze toestanden leggen we vast in de integer *state* die de waarde 0, 1, 2 of 3 kan hebben.

waarde *state*	betekenis
0	niets in voorraad
1	*STACK* " "
2	*STACK* en een Satm (in die volgorde) in voorraad
3	een Satm in voorraad.

Mogelijke overgangen van *state*



Als *state* gelijk is aan nul dan kijkt *Macro 2* of de volgende macro wellicht *STACK* is, indien dit zo is, dan bewaart hij *STACK* even (d.w.z. hij maakt *state 1*) om te zien of er misschien een Satm en een Oo op volgen (dan kan er geoptimaliseerd worden), alvorens hij via een aanroep van *Produce* er onherroepelijk een instructie van laat maken en die in het objectprogramma laat opslaan.
Als een Satm wordt aangeboden (en *state* = 0) dan wordt via een aanroep van *Load* die Satm en zijn parameter opgeslagen en *state* wordt 3.
In ieder geval levert *Produce* instructie(s) aan het objectprogramma.

Is *state* gelijk aan 1 dan volgt er een aanroep *Load (2)* m.a.w. de aangeboden macro wordt opgeslagen en *state* wordt 2. Mocht de macro geen Satm

zijn dan wordt eerst *STACK* en vervolgens de in *stack0* en *stack1* opgeslagen macro en macroparameter aangeboden aan *Produce* (de eerste macro expliciet, de tweede via *Unload*, die tevens *state* nul maakt).

Is *state* gelijk aan 2 en de aangeboden operator een Oo dan wordt *Optimize* aangeroepen (op de werking hiervan gaan we nog in). Is de macro geen Oo dan wordt *STACK* verwerkt, *state* wordt 3, en *Macro 2* roept zichzelf aan.

Is *state* gelijk aan 3 en de volgende macro *NEG* dan is optimaliseren mogelijk (zie blz. 42) en dus volgt er een aanroep van *Optimize*. Is de volgende macro niet *NEG*, dan wordt de Satm in het objectprogramma gezet door *Produce* (via *Unload*) en *state* wordt nul. Via een aanroep van zichzelf bekijkt *Macro 2* de macro na de Satm nog eens, nu dus met *state* = 0.

Optimize

Deze procedure, die op 2 macro's werkt (n.l. de opgeslagen Satm in *stack0* en de Oo in *macro*) zoekt in een lijst de geoptimaliseerde macro. Stel dat *Macro 2* aangeroepen wordt met *state* = 2 d.w.z. *STACK* en b.v. *TSIC (2)* in voorraad en de volgende macro is een Oo (b.v. *ADD*) dan zoekt *Optimize* de geoptimaliseerde macro op in de volgende tabel

Satm / Oo	TRV	TIV	TRC	TIC	TSIC
NEG	TNRV	TNIV	TNRC	TNIC	TNSIC
ADD	ADDRV	ADDIV	ADDRC	ADDIC	ADDSIC
SUB	SUBRV	SUBIV	SUBRC	SUBIC	SUBSIC
MUL	MULRV	MULIV	MULRC	MULIC	MULSIC
DIV	DIVRV	DIVIV	DIVRC	DIVIC	DIVSIC
EQU	EQURV	EQUIV	EQURC	EQUIC	EQUSIC
UQU	UQURV	UQUIV	UQURC	UQUIC	UQUSIC
LES	LESRV	LESIV	LESRC	LESIC	LESSIC
MST	MSTRV	MSTIV	MSTRC	MSTIC	MSTSIC
MOR	MORRV	MORIV	MORRC	MORIC	MORSIC
LST	LSTRV	LSTIV	LSTRC	LSTIC	LSTSIC

De rijen worden gegeven door het Opt. number van de Oo (genummerd van 1

t/m 11) en de kolommen door het opt. number van de Satm (genummerd van 0 t/m 4). In het voorbeeld:

```
"STACK
"TSIC (2)
"ADD
```

bepaald *Optimize* de opt. numbers van *TSIC* (= *macro*) en *ADD* (= *stack0*) en met behulp van deze gegevens zoekt hij *ADDSIC* op in de bovenstaande tabel (in feite is in plaats van een *integer array tabel* [*1:11, 0:4*] een gelineariseerd *integer array tabel* [*5:59*] gedeclareerd).
Via *Unload* wordt *Produce* aangeroepen die de instructies in het objectprogramma zet.

270470

We hebben gezien hoe een tripel *STACK, simple arithmetic take macro* (= Satm) en *Optimizable operator*(= Oo) door de macroprocessor vervangen wordt door één macro.
Men kan zich echter afvragen: gaat dit optimaliseren niet een keer fout, m.a.w. vervangt de macroprocessor b.v. de opeenvolgende macro's TSIC en NEG altijd door TNSIC, ook waar dit niet toelaatbaar is?
In het volgende reeds eerder genoemde voorbeeld (zie blz. 42 en [2]) komen bij de vertaling van de ALGOL tekst met onze huidige kennis de macro's *TSIC* en *NEG* na elkaar voor, zonder dat je wilt dat de macroprocessor deze macro's vervangt door *TNSIC*.

```
ALGOL                          ELAN
- (if BE then 1 else 2)          "C = BE"
                              N,  GOTO(:L0)
                                  F = 1
                                  GOTO(:L1)
                              L0: F = 2      "TSIC
                              L1: F = -F     "NEG
```

Om nu in detail te zien hoe de vertaling in feite gebeurt gaan we nog even

terug naar de vertaling van een AE en wel een primary. (Zie blz. 32.)
De SAE - *(if BE then 1 else 2)* heeft tot gevolg dat tijdens het vertalen de volgende lijst procedures wordt aangeroepen

effect van *Arithexp*, aangeroepen vanuit *Primary*

next symbol	"*if* opruimen
Boolexp	"vertaal de BE en genereert stuk objectprogramma
Macro 2 (COJU, future 1)	"levert macro *COJU (future 1)* af in de vertaling
next symbol	"*then* opruimen
SAE → Macro 2 (TSIC, 1)	"levert uiteindelijk macro *TSIC(1)*
Macro 2 (JU, future 2)	"levert macro *JU (future 2)*
Substitute(future 1) → Macro (EMPTY)	
next symbol	"*else* opruimen
AE → Macro 2 (TSIC, 2)	"levert macro *TSIC(2)*
Substitute(future 2) → Macro (EMPTY)	"levert macro *EMPTY*
	"Pas nu wordt de macro *NEG* vanuit procedure *SAE* ingevuld

(zie blz. 95 e.v.).

Men ziet dus dat *Substitute* een macro *EMPTY* genereert en deze macro staat tussen *TSIC* en *NEG*.

De macroprocessor herkent deze opeenvolging niet als een optimaliseerbare groep en is dan genoodzaakt deze macro's weg te schrijven naar het objectprogramma.
Om te zien dat de procedure *Macro (EMPTY)* wordt aangeroepen in *Substitute* beschouwen we *Substitute* en de heading van *Subst 2*.

```
procedure Substitute (address); integer address;
begin Subst 2 (Order counter, address) end Substitute;
procedure Subst 2 (address 1, address 2);
value address 1, address 2;
integer address 1, address 2;
begin ....
      ....
end Subst 2;
```

Het subtiele zit in de *value* specificatie van *address 1* . Daardoor wordt de waarde van de actuele parameter (= *Order counter*) eenmalig geevalueerd en geassigneerd aan een (nieuwe) locale variabele. Daar echter *order counter* een procedure is met in z'n body de aanroep van *Macro (EMPTY)* wordt dus tevens de macro *EMPTY* gegenereerd. Deze macro levert geen bijdrage tot het objectprogramma (zie de tekst van *Produce*, blz. 100).

Opmerkingen

1) Deze handelwijze om de macroprocessor te laten leeglopen door middel van het aanbieden van een macro (die geen effect heeft op het verdere objectprogramma) waar de macroprocessor niets mee kan doen, wordt vaker toegepast, n.l. in al die gevallen waarin naar een adres uit het objectprogramma verwezen moet kunnen worden (in het voorbeeld dus door de macro's *COJU* en *JU*).
2) In de moduul, "macro generator", komt nergens *instruct counter* voor.

Bij de behandeling van de vertaling van *if* ... *then* ... *else* constructies is reeds genoemd dat een aanroep van *Macro 2*, met de macro's *JU* en *COJU*, aan de macroparameter het adres in het objectprogramma waar de betreffende instructie was opgeborgen, zou toekennen. Dit gebeurt nu in de laatste statement van de procedure *Macro 2*.
De eis dat de metaparameter ≤ 0 is hangt samen met het feit dat b.v. *Macro (JU)* op 2 manieren aangeroepen kan worden en wel

1°) adres waar naartoe gesprongen moet worden is kant en klaar (≡ macroparameter ≤ 0)
2°) adres waar naartoe gesprongen moet worden is af te leiden uit de pointer in de naamlijst (≡ macroparameter > 0).

III De vertaling van Boolean expressies

III-1 Het vertaalschema van primary's

We zullen hier slechts op ingaan in de vorm van opmerkingen omdat de vertaling analoog is aan die van AE's. (Zie voor samenhang: [2], §1.5 t/m §1.6.2.)

Er zijn verschillende representaties van *true* en *false* in de X8

1) Het resultaat van de evaluatie van een BE komt in het éénbitsregister *C* en wel staat *Y* (*C* = 0) voor *true*
   en *N* (*C* = 1) voor *false*.
2) De simpele, locale of globale variabele.
   Deze beslaat één X8 woord en wel
   als X8 woord $\geq$ +0 dan boolean variable = *true* (d26 = 0),
   als X8 woord $\leq$ -0 dan boolean variable = *false* (d26 = 1).
   (Zie [2], §1.5.1.)
3) De locale of globale subscripted variabele. Een element van een geïndiceerde variabele wordt gerepresenteerd door één bit (0 = *true*, 1 = *false*) d.w.z. door ieder X8 woord kunnen dus 27 Boolean-arrayelementen gerepresenteerd worden.
   (Zie [2], §1.5.3.)

De vertaling van Boolean primaries lichten we toe met als voorbeeld het vertaalschema van de relaties (zie [2], §1.5.7.).
We onderscheiden de volgende relaties

relatie	bijbehorende macro
=	*EQU* (≡ EQUal)
≠	*UQU* (≡ UneQUal)
<	*LES* (≡ LESs)
$\leq$	*MST* (≡ at MoST)
>	*MOR* (≡ MORe)
$\geq$	*LST* (≡ at LeaST).

Het vertaalschema is ook hier

1e evalueer eerste operand (= SAE) met resultaat in *F*;

2e stapel F;

3e evalueer tweede operand (= SAE) met resultaat in *F*;

4e voer operatie uit op top van de stapel en *F* met resultaat in *C* (dit levert dus een macro afhankelijk van de relatie) en verlaag de stapel met 2.

(Zie voor de vertaling in ELAN van de relaties [2], §1.5.7.1.)

Opmerkingen

1) De macro *LES* (behorende bij <) bestaat uit de instructies

F - MC[-2], P
Y, F - O, P.

Hiervan is de eerste voldoende in bijna alle gevallen. Immers: als *AE1 > AE2*, komt uit *AE2 - AE1* zeker een negatief resultaat, als *AE1 < AE2*, zeker een positief resultaat, terwijl als *AE1 = AE2*, uit *AE2 - AE1*, vanwege de *-0* preferentie van de aritmetiek, bijna steeds *-0* komt (dus conditie *N*). De enige uitzondering hierop is, als *AE2* bij evaluatie *+0* op levert en *AE1* bij evaluatie *-0* geeft *((+0) - (-0) = (+0))*. In dat geval zorgt de tweede instructie (die alle waarden van *F* onveranderd laat behalve nu juist *+0*) voor de juiste uitkomst in *C*.

2) De macro *MOR* bestaat uit 3 instructies. De eerste twee zijn dezelfde als de macro *MST*; deze leveren de conditie precies verkeerd (voor *MOR* tenminste) af. De omkering van de conditie gebeurt door de instructie *S = -T, P.*

(Als we in ELAN schrijven *S = -T* dan komt in de machine het equivalent van *S = -M[62]*. De hardware levert i.p.v. *M[62]* echter af 27 bits met in de laatste 18 bits *OT* en in het tekenbit *C*, dus een getal $\geq$ *+0* als *C = Y* en $\leq$ *-0* als *C = N*. Door het extra min-teken in de *P* vraag wordt dus de conditie omgekeerd.)

3) Bij *LST* wordt de conditie-zettende variant *E* gebruikt. Elke conditie-zettende variant vult het teken van het bij de instructie gevormde resultaat in het eenbitsregister *LT* in. De *E*-variant vult de conditie

C in op grond van het gelijk of ongelijk zijn van de oude en de nieuwe waarde van *LT*.
De macro *LST* (behorend bij $\geq$) bestaat uit de instructies

```
   F - MC[-2], Z
N, S = -1, E .
```

Als *AE1 = AE2*, wordt de tweede opdracht geskipt omdat de eerste de conditie *Y* maakt (tevens het goede antwoord!)
Als *AE1 > AE2*, wordt door de eerste opdracht *C = N* en *LT = 1* (d.w.z. -), door de tweede opdracht *C = Y* en *LT = 1* (omdat in *S* het negatieve getal *-1* gevormd is).
Als *AE1 < AE2*, wordt door de eerste opdracht *C = N* en *LT = 0* (d.w.z. +), door de tweede opdracht *C = N* en *LT = 1*.
In dit geval zou ook als opdrachtenreeks genomen kunnen worden

```
   F - MC[-2], Z
N, F = -F, P
```

het nadeel is dat de tweede opdracht bij uitvoering 7,5 μ sec duurt, terwijl *N, S = -1, E* 2,5 μ sec in beslag neemt.

III-2 Optimaliseren van relaties

Aangezien het vertaalschema geheel analoog is aan dat van een aritmetische operatie, kan een analoge optimalisering toegepast worden.
De macroprocessor optimaliseert alleen wanneer de AE2 een Satm is (d.w.z. een van de macro's: *TRV, TIV, TRC, TIC* of *TSIC* aflevert).
Zo wordt in geval dat de Satm de macro *TIV*, dat is het tripel

```
"STACK
"Satm
"MOR
```

vervangen door de macro: *MORIV* (zie tabel op blz. 46), die bestaat uit

```
        G - M[n], P
Y, F - O, P.
```

In het geval de Satm de macro *TIC* is wordt dit tripel *STACK*, *TIC*, *MOR* vervangen door de macro *MORIC* met als enige opdracht *G - M[n], P*; de opdracht *Y, F - O, P* is overbodig omdat in de constantenlijst nooit een *-O* staat (een minteken voor een getal wordt verwerkt als operator). Men ziet dus dat in dit bizondere geval het tripel dat staat voor 5 opdrachten vervangen is door een macro die slechts uit één opdracht bestaat.

III-3 De vertaling van Boolean operaties

040570

We behandelen na de vertaling van de Boolean primary's de vertaling van de Boolean operaties (zie [2], §1.6). Deze operaties zijn monadisch (¬) of dyadisch (∧, ∨, ⊐ en ≡).

Het algemene vertaalschema van de monadische operatie ¬ is:

1° evalueer de operand (resultaat in *C*);

2° keer de conditie om, dit laatste gebeurt door de macro *NON* (zie [2], §1.6.1).

Het algemene vertaalschema van de dyadische operatie is:

1° evalueer de 1° operand (resultaat in *C*);

2° leg het resultaat op de stapel;

3° evalueer de 2° operand (resultaat in *C*);

4° voer de operatie uit op *C* en het bovenste woord van de stapel.

Punt 2 wordt uitgevoerd door de macro *STAB* (STAck Boolean), die door de macroprocessor omgezet wordt in de instructies

```
 S = T
MC = S   ;
```

STAB zet dus een woord op de stapel waarvan het tekenbit de conditie vertegenwoordigt.

Zie [2], §1.6.1 voor het symbolisch vertaalschema van de boolean operaties. We kijken als voorbeeld naar de operatie ∧ en de bijbehorende macro *AND*. *AND* kan niet volstaan met het slechts dan uitvoeren van het 4° bovengenoem-

de punt als de 2° operand *true* oplevert (er moet n.l. ook nog ontstapeld worden). Vanwege dat ontstapelen is er nog de instructie

```
N, B - 1
```

toegevoegd.
AND in de vorm

```
Y, S = MC[-1], P
N, B - 1
```

zou niet voldoen; in het geval dat BE1 is *false* en BE2 is *true* zou er twee keer (en dus teveel) ontstapeld worden.

We kijken nog even naar het voorbeeld $x = y \wedge i > 1$ (zie [2], §1.6.2), dit geeft de volgende macro's

```
TRV(x)
EQURV(y)
STAB
TIV(i)
MORSIC(1)
AND      .
```

III-4 Moeilijkheden tijdens vertalen van Boolean primary's

In Boolean primary's kunnen zowel stukken Boolean expressie als stukken aritmetische expressie voorkomen (n.l. in relaties). Bij het begin van het vertalen van een Boolean primary is derhalve niet altijd bekend, wat voor type expressie er zal blijken te staan. Zo is b.v. van

if (<*expressie*>)

niet duidelijk of <*expressie*> een AE of BE is, er kan b.v. op volgen

> 1 *then*
of
∧ (<*boolean expressie*>) *then*

(in het eerste geval was <*expressie*> dus een AE en in het tweede een BE). We kunnen dus niet klakkeloos de procedure *Boolexp* aanroepen na *if(*. Vanwege deze moeilijkheid beschikt de vertaler over de procedure *Arboolexp* die de volgende vorm heeft

```
procedure Arboolexp (type); integer type;
begin ...
end Arboolexp;
```

Arboolexp kent aan zijn parameter *type* één uit 3 mogelijke waarden toe en wel

```
type:= ar   (= 4)
    := bo   (= 2)
    := arbo (= 8)
```

Hiermee brengt *Arboolexp* verslag uit over hetgeen hij tijdens de vertaling van die expressie ontdekt heeft over het type van die expressie.
De betekenis van de eerste twee waarden is zonder meer duidelijk, de derde waarde wordt toegekend b.v. als *prescan 1* er niet in geslaagd is het type van een ongespecificeerde formele te bepalen. Als voorbeeld (*f1* en *f2* ongespecificeerde formelen) geven we

```
(((if true then f1 else ((f2))))).
```

We geven nu nog de tekst van enige procedures die voor de vertaling van Boolean primary's zorg dragen en waarin met name geillustreerd wordt op welke wijze de vertaler verder gebruik maakt van het door een aanroep van *Arboolexp* gevonden type.

```
procedure Boolprim;
begin integer type, n;
   if last symbol = open
   then begin next symbol; Arboolexp (type);
          if last symbol = close
          then next symbol
          else ERRORMESSAGE (309);
          if type = ar
          then Rest of relation
          else if type = arbo
               then begin if arithoperator last symbol
                          then Rest of relation
                          else Relation
                    end
        end
   else if letter last symbol
        then ...
end Boolprim;

procedure Rest of relation;
begin Rest of arithexp;
   if ¬ Relation then ERRORMESSAGE (311)
end Rest of relation;

boolean procedure Relation;
begin integer relmacro;
      if relat operator last symbol
      then begin  relmacro:= Relatmacro;
                  Macro (STACK); next symbol;
                  Simple arithexp; Macro (relmacro);
                  Relation:= true
           end
      else Relation:= false
end Relation;
```

```
procedure Rest of arithexp;
begin Next primary;
      Next factor;
      Next term
end Rest of arithexp;
```

Vindt *Boolprim* in *last symbol* (het eerste basic symbol van de boolean primary) een openingshaak, dan wordt, na de obligate aanroep van *next symbol* *Arboolexp* aangeroepen. *Arboolexp* leest de <expressie>, vertaalt die en vergaart kennis over het type.
Mocht de sluithaak weggelaten zijn dan volgt er een foutmelding, staat) er wel dan lezen we het volgende symbool.

Wat er hierna gedaan wordt hangt sterk af van de waarde van *type* (dus van de kennis, door *Arboolexp* over de expressie tussen haakjes verkregen).

Indien *type = bo* is kennelijk een complete Boolean primary verwerkt en zijn we klaar (indien nu een relationele operator volgt is er iets fout en volgt een foutmelding, zij het niet meer vanuit *Boolprim*).

Indien *type = ar* dan wordt er geeist, dat er na de eerste operand (arithmetisch) een relationele operator gevolgd door een arithmetische operand komen. Dit doet de procedure *Rest of relation.*

In het geval *type = arbo* moeten we voorzichtiger zijn. Slechts indien er op de) een aritmetische operator volgt, mogen we de aanwezigheid - verder op - van een relationele operator en een 2° operand vereisen. Anders mogen we slechts vragen (en niet eisen) of er een relationele operator staat en in dat geval de vertaling van een relatie voltooien *). Let dus op het verschil tussen de procedures *Relation* en *Rest of relation.* Ter toelichting geven we een schema van wat er gebeurt in alle mogelijke combinaties van het door *Arboolexp* gevonden type en het op het sluithaakje volgende basic symbol.

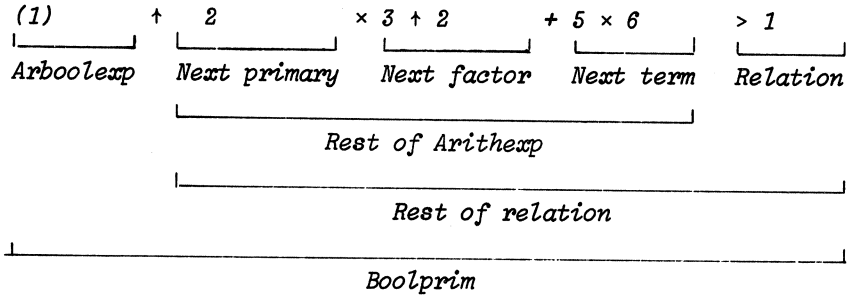
---

*) Indien na de) noch een aritmetische noch een relationele operator volgt, moeten we aannemen dat de expressie tussen haken de complete Boolean primary is (en dus van type *bo* alhoewel we dat niet als zodanig herkend hebben).

last symbol	*arithoperator*	*relatoperator*	anders
v.b. / *type*	*+ (...) > 1*	*> 1*	*v(...)*
ar	*Rest of arithexp*, *Relation* 1)	*Relation* 2)	foutmelding 311
bo	- 3)	- 3)	-
arbo	*Rest of arithexp*, *Relation* 1)	*Relation* 4)	- 5)

1) beide via *Rest of relation.*
2) via *Rest of relation.* De aanroep van *Rest of arithexp* gebeurt wel, maar heeft geen effect.
3) een foutmelding volgt te zijner tijd, maar niet vanuit *Boolprim.*
4) niet via *Rest of relation.*
5) de aanroep van *Relation* in dit geval heeft geen effect.

In het voorbeeld



worden de diverse onderdelen vertaald door de eronder aangegeven vertaalprocedures.
Merk op dat in *Rest of relation* de waarde van *Relation* gevraagd wordt, tijdens het bepalen daarvan wordt als neveneffect een stuk vertaalarbeid verricht (indien nodig).
In *Rest of arithexp* blijkt weer het nut van de splitsing van *Arithexp* in allerlei procedures (zie blz. 31). We kunnen nu volstaan met een zeer korte procedure die een stuk AE prioriteit na prioriteit behandelt.

IV De vertaling van assignment statements

IV-1 Het algemene vertaalschema van assignment statements

110570

De algemene structuur van een assignment statement kan symbolisch worden weergegeven met

L1:= L2:= ...:= Ln:= E.

Het Revised Report (zie 4.2.3) schrijft voor dat dit proces moet gebeuren in de volgende stappen

1) Alle subscript expressies van de variabelen in het linker gedeelte van een assignment statement worden achtereenvolgens en wel van links naar rechts geevalueerd.
2) De expressie van de statement wordt geevalueerd.
3) De waarde van de expressie wordt geassigneerd aan alle variabelen in het linker gedeelte van de statement (ingeval van geindiceerde variabelen wordt geassigneerd aan de geindiceerde variabele met index ontwikkeld in 1).

Voorbeeld: *A[read]:= read,* bij getallenband *3,5* kent aan *A[3]* de waarde *5* toe.
Men zou zich kunnen afvragen of de indicering, hieronder verstaan we het herleiden van een *address description* (zie [2], 1.3.3.1 en 4.1.5) tot de plaats van het element in het geheugen, niet al in stap 1 gedaan had kunnen worden.
Dit zou iets efficienter zijn en minder geheugenruimte vragen dan de aanstonds te behandelen methode. Dat dit niet gebeurt hangt samen met een (niet-gerealiseerde) opzet voor de implementatie van own dynamic array's in de z.g. statische interpretatie (we gaan hier maar niet verder op in).

We zullen vervolgens de assignment bekijken aan

1) een locale of globale simpele variabele
2) een simpele geindiceerde variabele
3) een formele identifier.

IV-2 De assignment aan een lokale of globale simpele variabele
(Zie ook [2], §4.1.2).

Omdat de preparatie leeg is, krijgt men het volgende schematische objectprogramma

<evaluatie van E>
<assignment aan de simpele variabele>.

De evaluatie van E hebben we al besproken ingeval E aritmetisch of boolean is (zie blz. 28 e.v.). Afhankelijk van het type van de simpele variabele n worden bijpassende macro's in het objectprogramma ingevuld.
Voor een overzicht hebben wij de volgende tabel geconstrueerd

type simpele var.	macro in objectprogramma	instructie waaruit de macro bestaat (stat. adressering)
real	*STR(x)* (= STore Real)	*M[x] = F*
integer	*STI(i)* (= STore Integer)	*S = F, Z* "In *S* wordt bit- "patroon van *M[57]* "overgenomen d.w.z. "de kop van *F*.
		N,SUBC(:RND) "Als de kop van *F* "ongelijk nul is, "wordt de subrou- "tine *RND* aange- "roepen die of de "afgeronde waarde "van *E* in *G* zet of "foutmelding 510 "geeft.
		M[i] = G "De afgeronde waar- "de wordt wegge- "schreven naar ge- "heugenplaats *i*.
boolean	*STB(b)* (= STore Boolean)	*S = T* "*M[62]* wordt in *S* "overgenomen (be- "vat 18 bits van "*OT* en de bits van "de eenbitsregister "dus ook de conditie "*C*, en wel in het "tekenbit!)
		M[b] = S

Opmerkingen

0) De adressen 57 t/m 62 spelen in de X8 een speciale rol.

1) Bij dynamische adressering (= de variabele is gedeclareerd in procedures of binnenblokken daarvan) wordt in de assignment, de waarde van de expressie weggeschreven naar de geheugenplaats aangewezen door de uitwerking van het dynamische adres: *Mp*[*q*] (= *M*[*M*[*63*]+*p*]+*q*).

2) Als het type van de simpele variabele integer is dan zien we dat er afronding plaats vindt van de expressie E bij assignering. Deze afronding is overbodig in het geval de expressie E reeds een keer expliciet is afgerond. Dan worden voor volgende assigneringen (dezelfde statement!) i.p.v. de macro's *STI* de macro's *SSTI(i)* (= Simplified STore Integer) gegenereerd (zie [2], §4.1.2). Deze "optimalisatie" is duidelijk te zien in het volgende voorbeeld van een meervoudige assignering aan integer variabelen.

Voorbeeld

ALGOL	ELAN	
i:= j:= 0	*F = 0*	*"TSIC(0)*
	S = F, Z	*"STI(j)*
	N, SUBC(:RND)	
	M[j] = G	
	M[i] = G	*"SSTI(i)*

IV-3 De assignment aan een geïndiceerde variabele
(Zie ook [2], §4.1.5).

Hier is de preparatie niet leeg. In het geval de variabele waaraan geassigneerd wordt een element van een real, statisch geadresseerd array is (d.w.z. *Ar*[*AE1, ..., AEn*]) dan bestaat de preparatie uit de opdrachten

```
F  = AE1    "Evalueer de eerste index
MC = F      "STACK
- - -
F  = AEn    "Evalueer de laatste index
A  = M[Ar]  "TAK(Ar)
MC = F      "STACK; stapel laatste index
MC = A      "STAA; stapel de array key.
```

Voor de evaluatie van de expressie *E* verwijzen we nu ook weer naar de voorgaande colleges waar dit behandeld is (zie blz. 28 e.v.).

De assignmentoperatie wordt bepaald door het type van de geindiceerde variabele. (Zie voor overzicht van de macro's die hiervoor zorg dragen, [2], §4.1.5.) In het genoemde overzicht staat *SUB2* voor de subroutinesprong die de link (d.w.z. d17 t/m d0 = opgehoogde *OT*, d26 = 0, d21 = *OF*, d20 = *LT*, d18 = *C*) in geheugenplaats 10 achterlaat.
Dat de link is opgeborgen in geheugenplaats 10 i.p.v. 9 of 8 is gedaan omdat men een eenvoudige algemene wijze van opbergen van de link wenste. Geheugenplaatsen 8 en 9 zijn niet te gebruiken omdat de aanroep van respectievelijk *IND* en *INDB* deze geheugenplaatsen gebruikt.
De link wordt niet op de stapel gelegd omdat de indices van het array er onder liggen. Dit aantal indices is afhankelijk van de dimensie van het array. Bij assignering wordt de stapelwijzer met een dimensie-afhankelijk getal verlaagd, waardoor een erboven liggende link slechts met kennis van dimensie terug te vinden zou zijn.

IV-4 De assignment aan een formele identifier en het parametermechanisme

Het R.R. noemt dit geval niet eens, maar herleidt het via een algemene regel naar de gevallen 1 en 2:
Men dient namelijk de statement waarmee de procedure wordt aangeroepen met een aantal actuele parameters te vervangen door een copie van de procedure body met overal de formele parameter(s) vervangen door de bijbehorende actuele parameter(s) en vervolgens dit blok uit te voeren.
In termen van onderstaand programma wordt
regel 7 vervangen door regel 3 t/m 6 met voor *f* de real variabele *x* gesub-

stitueerd
regel 8 vervangen door regel 3 t/m 6 met voor f de integer variabele i gesubstitueerd en
regel 9 vervangen door regel 3 t/m 6 met voor f, $Ar[1]$ gesubstitueerd.

```
1     begin real x; integer i; array Ar[1:1];
2           procedure P(f); real f;
3           begin ...
4                 f:= 0;
5                 ...
6           end;
7           P(x);
8           P(i);
9           P(Ar[1]);
10    end
```

In het programma wat dan ontstaan is komen slechts assignments aan simpele of subscripted variabelen voor, waarvan de vertaling eerder besproken is. In de X8 gebeurt deze handelswijze niet omdat het inefficient is. Er wordt in plaats van bovenbeschreven methode gebruik gemaakt van APIC's (= Actual Parameter Instruction Cell's). Bij binnenkomst in de procedure worden er op de stapel een aantal woorden APIC gevormd, die een paar instructies (in het meest gecompliceerde geval 3) bevatten.

CELL	taakomschrijving
APIC 0	evalueer de waarde van de actuele parameter
APIC 1	(bevat geen instructie)
APIC 2	prepareer assignment
APIC 3	assigneer aan de actuele parameter.

De evaluatie van een formele parameter f(zie [2], §1.3.2) vindt plaats door de instructie

DOS(Mp[q]).

De instructie *DOS(Mp[q])* heeft het volgende tot effect

1) Het adres van de geheugenplaats met dynamisch adres (p,q) wordt geevalueerd: *Mp[q] = M[M[63]+p]+q.*
2) Dit adres wordt in *S* genomen.
3) De inhoud van de door dit adres aangegeven geheugenplaats wordt in *OR* genomen.
4) *OR* wordt als instructie uitgevoerd (dit kan alles zijn zelfs *DOS*).

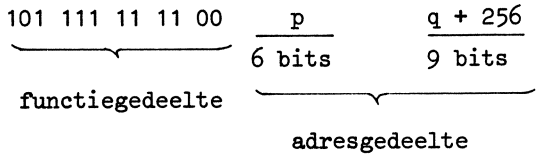
In de assignment statement *f:= AE1* krijgen we achtereenvolgens

DOS(f[2])	*"APIC 2 wordt uitgevoerd.*
"F = AE1"	*"rechterlid wordt geevalueerd.*
DOS(f[3])	*"APIC 3 wordt uitgevoerd.*

Voor het voorbeeld in [2], §4.1.6 geldt dus

f:= f+1 ⟷	*DOS(f[2])*	*"preparatie d.m.v. APIC 2*
	DOS(f)	*"evaluatie van formele parameter in rechterlid assignment statement d.m.v. APIC 0*
	F + 1	*"*
	DOS(f[3])	*"assigneer d.m.v. APIC 3.*

(In de machine correspondeert met de instructie *DOS(Mp[q])* het bitpatroon



zo is b.v. *DOS(M2[3])* 101 111 11 11 00 000 010 100 000 011.)

V De vertaling van for statements

Wij zullen hier slechts aandacht schenken aan een simpele vorm van de for statement aan de hand van het voorbeeld

for i:= 1, 2 do S;

(zie voor symbolisch vertaalschema [2], §4.8.1).
Bij de vertaling van ieder blok worden geheugenplaatsen gereserveerd voor pseudo-forvariabelen. In zo'n geheugenplaats staat gecodeerd met welk for list element uit de for list de machine verder moet gaan, als deze klaar is met de statement achter *do*.
In het bovenstaande eenvoudige geval van de for statement luidt de vertaling

```
        F=: L1              "TSIC(L1)
        for var = G         "SSTI (for var)
        F = 1               "TSIC(1)
        S = F, Z            "   ⎫
     N, SUBC(: RND)         "   ⎬ STI(i)
        M[i] = G            "   ⎭
        GOTO(: ACTION)      "JU(ACTION)
    L1: F=: L2              "TSIC(L2)
        for var = G         "SSTI (for var)
        F = 2               "TSIC(2)
        S = F, Z            "   ⎫
     N, SUBC(: RND)         "   ⎬ STI(i)
        M[i] = G            "   ⎭
        GOTO(:ACTION)       "JU(ACTION)
    L2: GOTO(:FINISH)       "JU(FINISH)
ACTION: <vertaling van statement achter do>
        GOTO(for var)       "IJU (for var)
FINISH:
```

Opm. De adressen *L1, L2* in de macro's *TSIC* en *FINISH* in de macro *JU*, worden daarin ingevuld door hetzelfde future mechanisme zoals we reeds ontmoet

hebben bij de conditionele expressie (zie blz. 28). Het adres *ACTION* in de macro's *JU* wordt door een analoog mechanisme ingevuld.
Doordat bij het future mechanisme in de opdracht de waarde van *future* wordt ingevuld en vervolgens in *future* het adres van die opdracht, worden bij herhaald gebruik alle plaatsen waar straks nog het juiste adres van *ACTION* moet worden ingevuld aan elkaar gekoppeld en bevat *future* het beginadres van die keten. Door *Substitute* wordt de hele keten afgelopen en bijgewerkt totdat een adres gelijk aan 0 wordt gevonden.

VI De vertaling van blokken

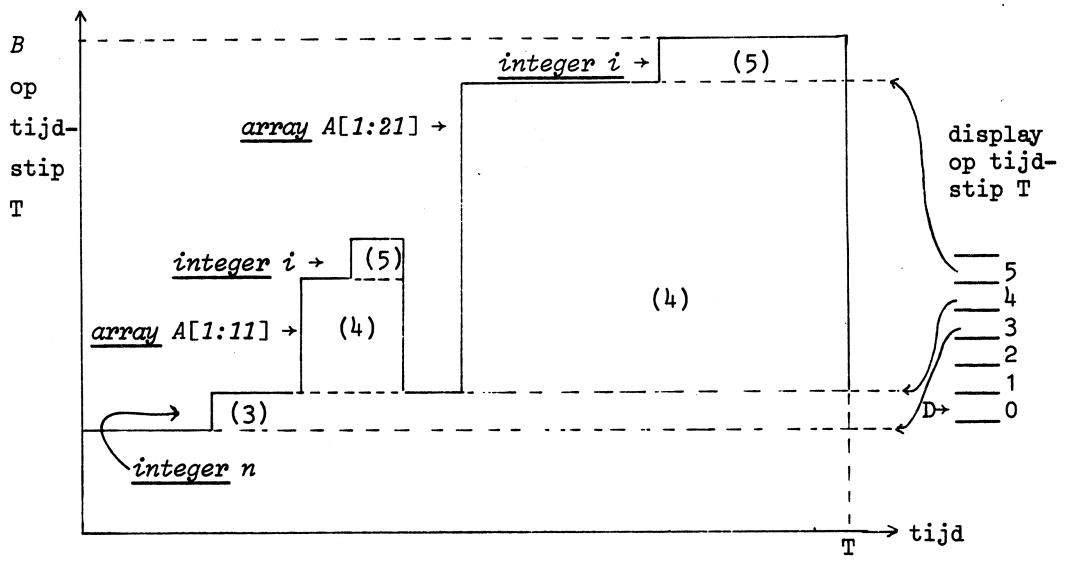
VI-1 Blokcellen in de stapel

250570

Variabelen hoeven geen betekenis te hebben in het totale programma. Volgens het R.R. (2.7) hebben ze slechts betekenis in de zogenaamde scope, dat is het kleinste blok dat de declaratie van de betreffende variabele omvat. Daarbuiten zijn ze niet alleen onbekend, maar mogen zelfs vergeten worden (d.w.z. undefined worden, zie R.R. hoofdstuk 5).
ALGOL-60 heeft een geneste structuur van blokken, dus ook een geneste structuur van scope's. De feiten dat variabelen beperkt zijn tot de scope's en dat die scope's in elkaar genest zijn maken het mogelijk variabelen die in parallelle blokken gedeclareerd worden het zelfde stuk geheugen (doch niet gelijktijdig) te laten beslaan. De machine kan toch niet in twee parallelle blokken tegelijk met de executie bezig zijn.
Dank zij de geneste blokstructuur en de geneste scope's kunnen we er een zogenaamd dynamisch geheugenbeheer op na houden en wel door middel van een stapel. Bij de blokingang wordt een blokcel aan de stapel toegevoegd waarin de locale grootheden hun plaats krijgen; bij blokverlating wordt dat stuk stapel weer vrijgegeven. Bij een dergelijk dynamisch geheugenbeheer moeten de adressen van variabelen dynamisch zijn, omdat in het algemeen niet van te voren bekend is waar de blokcel in de stapel zal komen te liggen en dit ook van de blokbinnenkomst tot blokbinnenkomst kan verschillen.
Als voorbeeld (aanname: alle variabelen zijn dynamisch geadresseerd, dus het voorbeeld is een binnenblok van een procedurebody)

```
begin blok 3: begin integer n;
                    n:= 1;
                l: n:= n + 10;
begin blok 4:       begin real array A[1:n];
                          begin integer i;
                                if n > 100 then goto end;
einde blok 5:             end;
einde blok 4:       end;
                    goto l;
              end:
einde blok 3:end;
```

De veranderingen in de stapel geschieden als volgt



((3),(4) of (5)) geeft het bloknummer aan.
Het register *B* geeft de eerste vrije plaats in de stapel aan.
Duidelijk is in dit schema te zien dat de integer *i* bij verschillende slagen van de go to loop steeds op een andere plaats in de stapel komt, maar toch moet de vertaler aan *i* een adres toewijzen. Conclusie: statische adressering is hier onmogelijk.

In de volgende tabel is nog aangegeven hoeveel woorden steeds de blokcellen in bovenstaand voorbeeld in beslag nemen en waarvoor die woorden gebruikt worden. De grootheden *wp* en *local space* worden verderop nog toegelicht.

	relatief adres	aantal woorden	gebruik	lengte blokcel	local space
blok 3	0	1	*wp*	4	4
	1	1	*n*		
	2 en 3	2	*end*		
blok 4	0	1	*wp*	2n + 9	2
	1	1	*array key A*		
	2 t/m 8	7	*storage function A*		
	9 t/m 2n+8	2n	elementen van *A*		
blok 5	0	1	*wp*	2	2
	1	1	*i*		

Aangezien er alleen bij binnenkomst van een blok nieuwe locale grootheden ontstaan is de aangewezen methode om aan een variabele een adres met 2 componenten toe te voegen (hier p en q genoemd) en wel zo dat p het bloknummer (de display level genoemd) aangeeft en q de plaats van de variabele t.o.v. het begin van de blokcel. De waarde van het bloknummer is één hoger dan die van het omvattende blok. In het voorbeeld is de adressering van de integer *i* het dynamisch adres (5,1) dat bij elke slag van de loop een ander statisch adres oplevert. Om een variabele te kunnen vinden, moeten we het begin van de blokcel weten te vinden en er q bij optellen.
Al die beginnen (genaamd blokcel pointers, afgekort tot *pp*)zitten in een lijst (genaamd *display*). Er zit dus ergens in het geheugen een lijstje (van voldoende lengte), de machine vindt het begin adres (de *display pointer*) of (*D*) in geheugenplaats drie en zestig (m.a.w. *M[63] = D*). Zo levert de ELAN instruktie *F = M3[5]* de volgende akties M[M[M[63] + 3] + 5]

$2\frac{1}{2}$ µ sec

$2\frac{1}{2}$ µ sec

5 µ sec

De laatste aktie duurt 5 μ sec, dat is $2\frac{1}{2}$ μ sec voor het ophalen van de kop en $2\frac{1}{2}$ μ sec voor de staart.
De totale tijdsduur is $12\frac{1}{2}$ μ sec (de tijd die nodig is om een statisch geadresseerd woordenpaar in *F* te halen is $7\frac{1}{2}$ μ sec d.i. $2\frac{1}{2}$ + (2 * $2\frac{1}{2}$) μ sec). Deze lange duur wordt veroorzaakt door het feit dat de display pointer in een gewone geheugenplaats is opgeborgen en niet in een register. Dit is een historisch gegroeide designfout.

VI-2 De vertaling van blokingang en blokverlating

We kijken nu wat er gebeurt als we een eenvoudig blok (geen procedure body) binnengaan (zie [2], §5.2).

A =: MC	"*TBL (display level);* waardoor *A* de nieuwe *blokcel pointer* bevat. Dit is iets korter dan de instructie *A = B*, want dan wordt het hele woord overgenomen nu daarentegen de achterste 18 bits.
F =: MD[display level] *B + [local space]*	"*F* bevat nu de plaats (het adres) in de *display* waar de *pp* moet worden ingevuld. De macro waaruit deze 2 instructies gevormd worden is *ENTRB (local space)*. *B* wordt opgehoogd voor de reservering van de dynamisch geadresseerde grootheden. *local space* bevat altijd tenminste 1 woord ten behoeve van de *wp* (zie verderop) en bevat nooit de ruimte nodig voor de *storage function* en de elementen van een local array. De q uit een dynamisch adres voldoet steeds aan $1 \leq q \leq$ [*local space*] - 1. De haken om *local space* duiden aan, dat hier in feite een getal staat, dat door de vertaler is uitgerekend voor dat blok.

SUB 2 (: ENTRB)

De subroutine *ENTRB* voert 3 taken uit en wel
1) hij vult de *pp* in *display* in;
2) hij vult *wp* in op de plaats nul van de *blokcel*;

3) hij test op het overkoken van de stapel, dit testen bij een blokingang is al aangekondigd op blz. 44.

De blokcel bevat altijd minstens 1 woord (zie punt 2) en wel *wp* (working space pointer) die de waarde van de eerste vrije plaats van de stapel op het moment dat aan de statements van het blok begonnen wordt. Tijdens executie van elk statement van een blok kan de stapel groeien (bij voorbeeld bij een procedure aanroep) maar na afloop hoort de stapel weer z'n oude waarde aan te nemen, het rustniveau, aangegeven door *wp*. Terwille van sprongen naar een label van een blok vanuit eventuele binnenblokken of procedures, waarbij de stapel afgelaagd moet worden tot het rustniveau, wordt dat niveau in *wp* vastgelegd. Voor de eenvoud wordt de *wp* in elk blok bijgehouden.

Voor *wp* nemen we voorlopig de nieuwe waarde van *B* (zie [2], §5.8) na de eerste opdracht van de macro *ENTRB*. Slechts door arraydeclaraties kan de stapel verder nog verhoogd worden. In dat geval wordt na afloop van de behandeling van alle declaraties ingelast de macro *SWP* (Set Working space Pointer), die de instructie *MO[512 * display level] = B* vormt.
(N.B. *MO[512 * p] = Mp[0]*).

Zie voor de taken die door *ENTRB* uitgevoerd worden blz. 109 "enige subroutines uit het complex" de subroutines *ENTRB* en *ENTRIS 4*.

De door *ENTRIS 4* aangeroepen routine *CONSIDER* geeft in zijn eenvoudigste vorm foutmelding 609 als *B* te groot is en geeft in dat geval de besturing terug aan het bedrijfssysteem.

Tenslotte merken we nog op dat *SUBC(: ENTRB)* als 4^e instructie niet zou voldoen. De link zou op de stapel gelegd worden (in plaats van op geheugenplaats 10). Maar indien *B* buiten het toegestane gebied wijst is dat schrijven aldaar ontoelaatbaar (als we kijken of iets neer geschreven kan worden mogen we niet eerst iets neerschrijven).

Bespreking voorbeeld [2], §5.11.

We nemen eerst aan dat alle variabelen statisch geadresseerd zijn en *display level* = 3.

Merk op dat eerst de beide arraygrenzen op de stapel worden gelegd.

Bij het 2^e vertaalschema, waarbij we dynamische adressering van alle variabelen veronderstellen (en waar display level = 3), in [2], §5.11 is de 3^e instructie: *B + 10*, dit cijfer 10 is opgebouwd als volgt : 1+1+1+1+2+2+2.

i, *Ar* en *Eps* nemen elk een plaats in en wel *M3[1]*, *M3[2]* en *M3[3]*, *AA*, *BB* en *CC* nemen elk 2 plaatsen in en wel die beginnen met resp. *M3[4]*, *M3[6]* en *M3[8]*. De label *DD* is superlocaal en er wordt dus geen variabele aan toe gekend (zie [2], §5.2).
Bij een array declaratie van de vorm *array jan, piet, klaas [1:10]* (voor een meer algemener voorbeeld zie [2], §5.5) worden de onder- en bovengrenzen maar eenmaal uitgerekend en op de stapel gelegd. De routine *RAD* construeert dan toch 3 referencevectoren en reserveert 3 segmenten voor de elementen. *RAD* krijgt in *F* het aantal te declareren arrays mee en in A het adres van de arraysleutel van het eerste array uit het lijstje (de andere sleutels hebben gewoon de er op volgende geheugenplaatsen toegewezen gekregen).

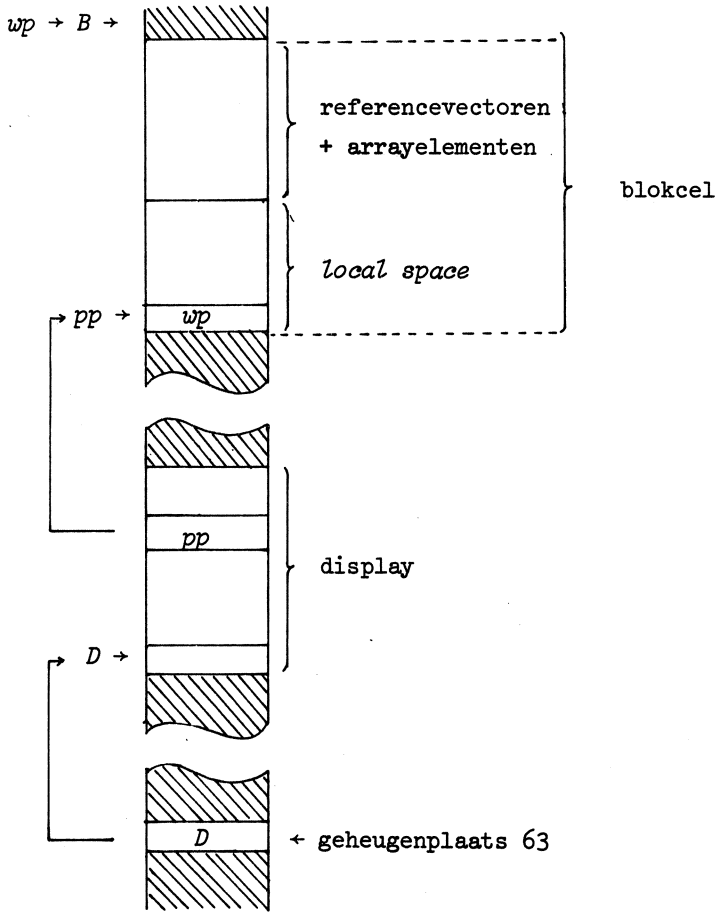
Op de vertaling van de declaraties van een blok en op de eventuele daarop volgende macro *SWP* volgt de vertaling van de statements van het blok. De vertaling van het blok wordt afgesloten door de macro *EXITB:*

B = MD[display level] "EXITB(display level)

waardoor de tijdens de blokingang en de verwerking van array declaraties gereserveerde geheugenplaatsen weer worden vrij gegeven. (*B* wordt namelijk gezet op *pp* en wijst dan dus naar het nulde woord van de daardoor vrijgegeven blokcel.)
Van een blok bevat de blokcel:

1) *local space* voor o.a. dynamisch geadresseerde array-sleutels (1 woord per array), gereserveerd door de macro *ENTRB*;
2) plaatsruimte voor reference vectoren en elementen van arrays. Deze ruimte wordt in de stapel gereserveerd door de macro's *RAD*, enz.

Door *ENTRB* wordt op plaats nul van de blokcel de voorlopige *wp* ingevoerd. Als er op *ENTRB* array declaraties volgen, wordt de definitieve waarde van *wp* door de macro *SWP* ingevuld.
We hebben dan dus het volgende stapelbeeld


wp → B →
referencevectoren
+ arrayelementen
local space
blokcel
pp →
wp
pp
display
D →
D
← geheugenplaats 63

VII De vertaling van procedures

VII-1 Blokcel en display van een procedure

010670

De vertaling van procedures is ingewikkelder dan vertalen van blokken omdat o.a. procedures zichzelf recursief kunnen aanroepen. De moeilijkheid die dan optreedt als we de techniek toepassen die we gebruiken bij het vertalen van blokken willen we toelichten met behulp van de volgende inefficiente procedure

```
integer procedure fac (n); value n; integer n;
fac:= if n > 1 then n * fac (n-1) else 1;
```

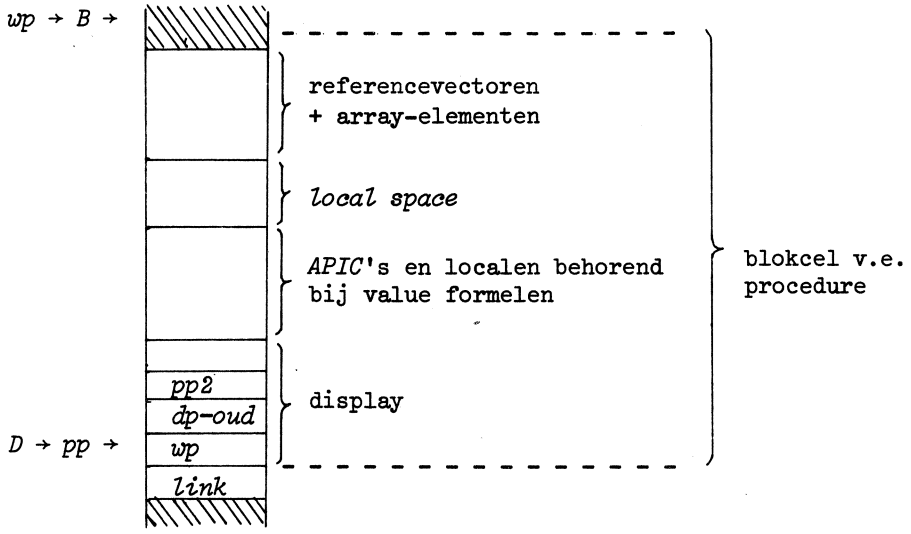
Aan de body van deze procedure wordt door de vertaler een blok toegekend, het zogenaamde body-block (zie [2], §6.1), met een display level, dat 1 hoger is dan die van het blok waarin de proceduredeclaratie staat. Tijdens de executie van de function designator *fac (4)* worden op de stapel 4 blokcellen gelegd (voor de aanroepen van *fac (n)* met $n = 4, 3, 2$ en 1).
De overeenkomstige locale variabelen (in het voorbeeld respectievelijk n en *fac*) in al die incarnaties hebben verschillende geheugenplaatsen, maar hetzelfde, hem door de vertaler toegekende dynamische adres $Mp[q]$ (waarin p voor de display level van de procedure staat, en q voor het relatieve adres t.o.v. het begin van de blokcel).
Dit betekent, dat bij elke incarnatie ofwel de meest recente *pp* in een vaste display moet worden ingevuld (zoals bij het ALGOL-systeem van de X1 gebeurt) ofwel voor elke incarnatie van een procedure een geheel nieuwe display moet worden geconstrueerd (zoals bij de X8 gebeurt).
De blokcel van een procedure bestaat uit

1) display. De lengte hiervan is [display level + 1] + zoveel extra plaatsen als maximaal nodig is voor binnenblokken, binnen-binnenblokken etc. van de procedure (de *pp* van deze blokken moeten n.l. allemaal in die display passen aangezien gewone blokken geen eigen display hebben). Plaats 0 en 1 van de display worden voor andere doeleinden gebruikt.
2) voor elke formele parameter hetzij een *APIC*, hetzij 1 of 2 woorden voor

een waarde, indien die parameter in de value lijst voorkomt (zie verderop).

3) *local space* (met o.a. arraysleutels van 1 woord).

4) referencevectoren en arrayelementen.



Bij genoemde punten kunnen we de volgende aantekeningen maken

1) De display level van een procedure is groter of gelijk 2.
Dit is het gevolg van de wijze waarop een display level aan een programma wordt toegekend: n.l. het hele programma heeft display level 0 (het hoeft n.l. geen ongelabeld blok te zijn); het echte 1e ongelabelde blok heeft dan display level 1. Pas in dit laatste blok mag een procedure gedeclareerd worden. Dus display level $\geq$ 2. (zie [2], §5.2.)

2) variabelen en pseudo variabelen die locaal zijn in een blok met display level 0 of 1 zijn steeds statisch geadresseerd. Bij dynamische adressering gebruiken we dus nooit display level 0 en 1. De plaatsen 0 en 1 van alle displays van procedures zijn derhalve geschikt om andere gegevens te bewaren, n.l. de wp van de blokcel (op plaats 0, net als bij een gewoon blok), en het adres van de direct voor de aanroep geldende display (op plaats 1).

3) Voor een procedure met display level n ($n \geq 2$) wordt de display als

volgt gevuld

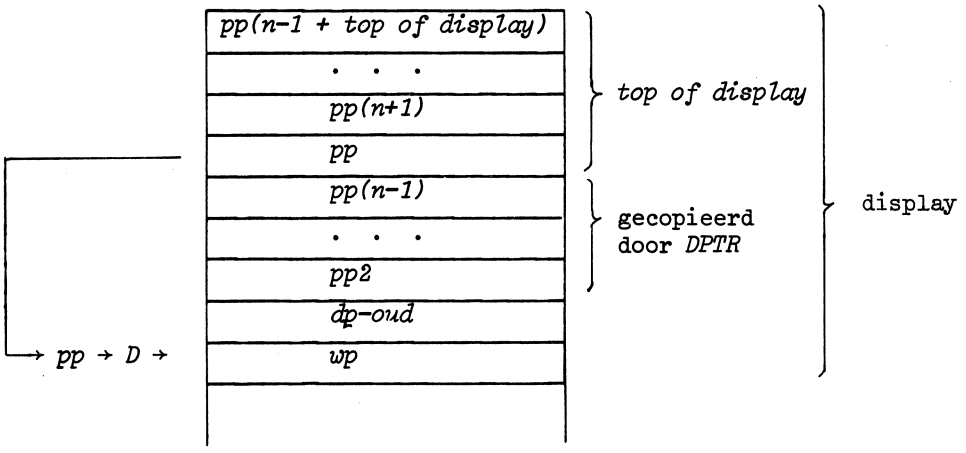
plaats 0: *wp* van de procedure

plaats 1: *dp* die heerste direct voor de aanroep van de procedure (*dp* = display pointer).

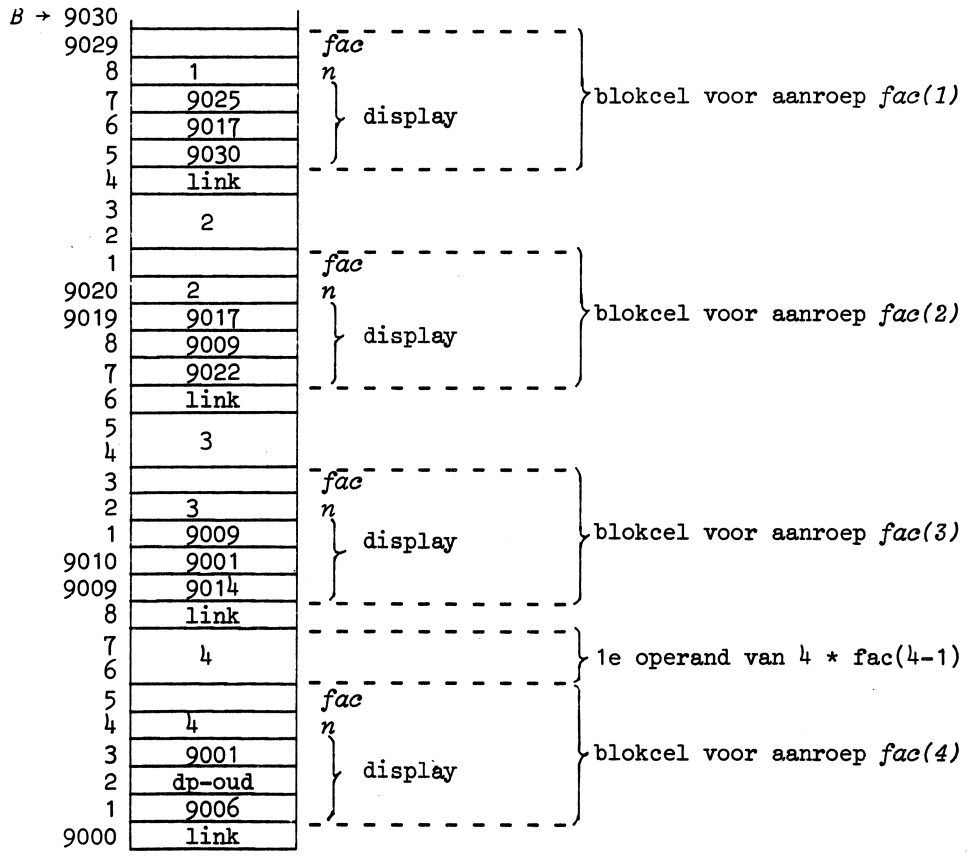
plaats 2 t/m plaats n-1: copieën van plaats 2 t/m n-1 van een oudere display (bij een normale aanroep van een procedure is dat de display die gold direct voor de aanroep van de procedure; bij een procedure die geactiveerd wordt via een formele procedure de display die gold bij de aanroep van (oudere) procedure waarbij eerstgenoemde procedure als actuele parameter werd meegegeven). Dit copieëren gebeurt door de macro *DPTR*.

plaats n: *pp* van de procedure zelf (merk op dat dus een van de woorden van een display naar het begin van die display wijst). Dit gebeurt door de macro *ENTRPB*.

plaats n+1 t/m plaats *n-1 + top of display:* deze worden slechts gereserveerd (te samen met plaats n door de macro *INCRB*) en te zijner tijd bij blokingang van binnenblokken ingevuld door de macro *ENTRB* uit de vertaling van die binnenblokken.

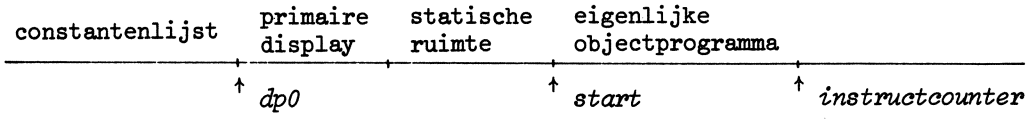


4) Bij een aanroep *fac (4)* van eerder genoemde procedure *fac* ziet de stapel er als volgt uit op het moment van maximale stapelhoogte (in de veronderstelling dat *fac* gedeclareerd staat in het buitenste blok en de *B* bij aanroep van *fac (4)* op 9000 stond):



Het adres van de locale variabele *n* (die correspondeert met de value parameter) is *M2[3]*, het adres van de locale variabele *fac* (bestemd voor de functie waarde) is *M2[4]*. Deze laatste geheugenplaats krijgt pas een waarde bij de assignatie van de waarde van het rechterlid aan het linkerlid van de assignment statement en is daarom hier nog niet ingevuld. Alleen *fac* behoort tot de local space.

5) Als geen enkele procedure geactiveerd is, wordt de zgn. primaire display gebruikt. Deze bevindt zich direct voor de statische ruimte. Het begin adres heet *dp0*. Deze primaire display speelt slechts een rol bij goto-statements. Het array *Space* ziet er als volgt uit



VII-2 De vertaling van procedure declaraties: De macro's *DPTR* en *INCRB*

De vertaling begint met de volgende 4 instructies

```
Pr:     S = D                          "Neem in S de waarde uit M[63] over.
Pr[1]: A = - [display level -2], Z "
        SUB (: DPTR)                   "DPTR (= DisPlay TRansport) met als
                                       "taken:
                                       "1) In A nieuwe waarde van pp nemen
                                       "(dit is juist de waarde van B bij aan-
                                       "roep van DPTR; het zal tevens de nieuwe
                                       "waarde van D worden),
                                       "2) De heersende D redden door deze over
                                       "te nemen naar plaats 1 van de nieuwe te
                                       "construeren display.
                                       "3) Er is dan meteen een plaats gereser-
                                       "veerd voor de wp n.l. op plaats 0 van
                                       "de nieuwe te construeren display.
                                       "4) Het copieren in de nieuwe display
                                       "van pp2, pp3, ..., pp[display level -1].
        B + [top of display]           "INCRB (top of display) (= INCRement B).
                                       "Hierdoor wordt in de display plaats ge-
                                       "reserveerd voor pp[display level],
                                       "d.w.z. de pp van de procedure blokcel
                                       "zelf + pp's van eventuele binnenblokken.
```

Voor procedures die gedeclareerd worden in het buitenste blok van een programma, en dus display level = 2 hebben wordt de 2e instructie van de vertaling van de declaratie: *A = -0, Z*. Door deze instructie wordt de conditie *Y* gezet. Hierop reageert *DPTR* door het overslaan van het gedeelte dat voor het display copieren zorgt. Het copiëren van *pp2* t/m *pp(n-1)* is voor n = 2 namelijk leeg.

DPTR bestaat uit de instructies:

```
DPTR:    G = D              "haal M[63] naar de staart van F en vul
         MC = F             "de kop met copieën van het tekenbit
                            "stapel nu 2 woorden waarvan: 1e woord
                            "copieën van het tekenbit van G en 2e
                            "woord de inhoud van G is. Hiermee is
                            "heersende D gered (2e woord) en 'n
                            "plaats gereserveerd voor de wp.
      Y, GOTO(: END)        "Als [display level -2] nul is hoeven we
                            "niet te copieren en nog slechts A de
                            "goede waarde te geven.
         stock = A          "A wordt gered
         GOTO(: TEST)
DOUBLE:  S + 2              "  ⎫ copieën van
         F = MS             "  ⎪ pp2, pp3, ..., pp[display level -1]
         MC = F             "  ⎬ worden telkens met 2 tegelijk gesta-
TEST:    A+2, P             "  ⎪ peld. (Zgn. double transport)
      N, GOTO(: DOUBLE)     "  ⎭
         A-2, Z
      N, A = MS[2]          "  ⎫ ingeval display level oneven is, moet
      N, MC = A             "  ⎬ nog 1 woord getranporteerd worden
                            "  ⎭ (single transport)
         A = stock          "herstel A op [display level -2]
END:     A + : MC[-2]       "vorm pp in A
         GOTO(LINK)
```

Bij de opdracht achter *END* bevat *B* het adres van *display*[*display level*]. Het adres: *MC*[-2] wijst dus naar *pp + display level -2*. Dit wordt bij *A* opgeteld die *-(display level -2)* bevat. Het resultaat in *A* is dus juist pp.

VII-3 Het parametermechanisme, *APD* en *APIC*

In de output van de vertaler heeft de declaratie van een procedure *P* de vorm van een subroutine (voorafgegaan door een spronginstructie over de

subroutine heen, zodat hij bij binnenkomst in het blok niet wordt uitgevoerd)

```
        GOTO(: over de procedure)  ┐
    P:  -                          │
        -                          ├ declaratie P.
        -                          │
        GOTO(: EXIT PB)            ┘
over de procedure:
```

De vertaling van de procedure statement *P(...)* wordt gevormd door een aanroep van de subroutine, gevolgd door enige codewoorden, voor elke actuele parameter één

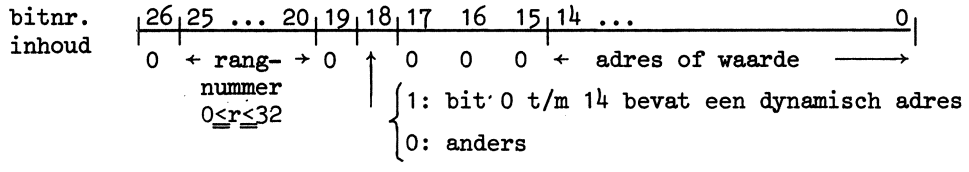
```
SUBC(: P)        "aanroep
-                "  ┐
-                   ├ codewoorden of APD's
-                   ┘ (= Actual Parameter Descriptors)
```

Een aanroep van de subroutine heeft tot gevolg dat er een blokcel voor *P* op de stapel komt. Deze bevat achtereenvolgens (vgl. de figuur op blz. 75)

1) nieuwe display van blokcel pointers
2) voor elke formele parameter (die niet in de value list voorkomt) een cel van 2 (of 4) machine woorden, de *APIC*.
   APIC[0] bevat een X8-instructie
   APIC[1] bevat o.a. een pointer, die wijst naar de display die heerste op het moment van aanroep.
3) ruimte voor locale variabelen.

Iedere *APIC* wordt geconstrueerd aan de hand van de overeenkomstige *APD*, zoals die wordt aangetroffen onder de aanroep van de subroutine.

De structuur van de *APD* is



In het geval dat de actuele parameter een kleine positieve integer k is krijgen we

APD	*APIC*		
*(7 * d20 + k)*	*0*	*F = [k]*	"neem getal ≤ 15 bits in *F*
	1	*(7 * d20 + [dp])*	

Opmerking naar aanleiding van vragen.

1) Variabelen met display level 0 kunnen nooit voorkomen, volgens het R.R. is een programma of een compound statement of een blok met of zonder label. De vertaler denkt zich om dit blok of compound statement met of zonder label nog een blok met display level 0, t.o.v. waarvan eventuele labels buiten de blokken van het programma locaal zijn.
   Een variabele heeft derhalve nooit display level 0.
2) De procedure *EXIT* breekt de stapel niet netjes af. Door *EXIT* wordt het programma abrupt afgebroken en aan het monitorsysteem doorgegeven dat een nieuw programma verwerkt kan worden. Bij springen vanuit een procedure naar een label buiten de procedure moet de stapel wel afgebroken worden. In totaal moeten bij een blok-verlatende sprong de volgende grootheden veranderd worden
   1) (vertelt waar je bezig bent)
   2) (Stapelwijzer, vertelt waar je werkruimte zich bevindt)
   3) (display pointer, vertelt wat je dynamische adressen betekenen).
   Hierbij wordt gebruik gemaakt van de inhoud van de pseudo-label variabele, die de nieuwe waarde voor *D*, de nieuwe waarde voor *OT*, en het display level bevat. Via de nieuwe *D* en het display level is de *pp* van het blok waarin de label locaal is te vinden; de blokcel bevat op plaats 0 de *wp*, de nieuwe waarde voor *B*.

080670

In het voorafgaande hebben we gezien, dat de vertaling van een procedure begint met de instructies

Pr: *S = D*
A = -[display level -2], Z
SUB(: DPTR)
B + [top of display]

Door deze instructies is een nieuwe display geconstrueerd, deels door copieren van een oude display, deels door reserveren van plaats (ten minste 1 plaats voor de *pp* van de eigen blokcel). Ook is al de oude *dp* gered en plaats gereserveerd voor de *wp*.
Ook hebben we gezien, dat bij de vertaling van een procedure statement de subroutineaanroep naar de vertaling van de declaratie gevolgd wordt door codewoorden, *APD*'s genaamd, voor elke actuele parameter een *APD*.

VII-4 De vertaling van de formele parameterlijst: De macro's CEN, CLPN, CRV en CIV

Voor elke formele parameter komt er in het objectprogramma een subroutineaanroep te staan.
We bespreken eerst parameters die niet in de value lijst voorkomen (parameter called by name: zie [2], §6.3.2).
Er zijn nu 2 mogelijkheden en wel

a) De onderhavige parameter komt ergens in de procedure body voor links van := (m.a.w. de parameter is linkerlid van een assignatie of controlled variable in een for statement): de macro *CLPN* (Call Left Part by Name) wordt nu gebruikt;
b) anders: we gebruiken *CEN* (= Call Expression by Name).
CEN legt een *APIC* van 2 woorden op de stapel, *CLPN* een *APIC* van 4 woorden (de twee extra woorden dienen om de assignatie tot stand te brengen).

We hebben al gezien (blz. 63) hoe de *APIC 2* diende voor de preparatie van de assignment en de *APIC 3* voor de effectuering daarvan.
Voor een overzicht wat een actual parameter allemaal kan zijn zie RR 3.2.1, we vinden daar de vijf mogelijkheden en wel

<string>, <expression>, <array identifier>, <switch identifier>, <procedure identifier>.

Vergelijken we dit met de "Opbouw van *APD* en *APIC*" dan blijkt dat er daar onderscheid gemaakt wordt tussen 33 verschillende soorten actual parameters. Dit komt omdat enerzijds onderscheid gemaakt wordt tussen expressies en identifiers van verschillende type (real, integer, Boolean en designational) anderzijds tussen variabelen en niet-variabelen.

Opmerking: in het R.R. worden variabelen niet afzonderlijk genoemd als actuele parameter (ze zijn daar een bijzonder geval van <expression>). Toch spelen ze in het parameter mechanisme een speciale rol, zie R.R. 4.7.5.2, waar geeist wordt dat een formele parameter die linkerlid is van een assignment statement in een procedure body en niet in de value lijst voorkomt alleen kan corresponderen met een actuele parameter die een variabele is (terloops wordt nog opgemerkt dat een variabele als een speciaal geval van een expressie beschouwd wordt).
Omdat het wel of niet variabele zijn in de codering tot uitdrukking komt, is *CLPN* in staat te controleren of de actuele parameter wel een variabele is (exacter: er wordt gekeken of er van de actuele parameter expliciet bekend is dat hij geen variabele is, in dat geval wordt er een foutmelding gegeven). Dit is een van de weinige gevallen dat er iets betreffende de correspondentie actuele/formele parameters gecontroleerd wordt tijdens runtime.
De *APD*'s voor variabelen hebben de codenummers 0 t/m 3 (de simpele variabelen), 16 t/m 19 (geïndiceerde variabelen van bekend type) en tenslotte nog nr. 32 (geïndiceerde variabelen van onbekend type).
Indien een actuele parameter een simpele real, boolean, integer of string variabele is (nr. 0, 1, 2, 3) wordt er bij de vertaling van de aanroep van de procedure volstaan met de *APD*.
Roepen we b.v. *Pr (x, A[i], 1)* aan (aanname: x is real, A is real array) dan ziet de vertaling er in principe als volgt uit

```
GOTO(: L0)
     ...                    "  }
ISR: ...                    "  } Impliciete Sub Routine voor A[i]
     ...                    "  }
 L0: SUBC(: Pr)             "aanroep van de procedure, de link
                             wordt op de stapel gelegd
    ( 0 * d20 + adres x)    "APD van x
    (16 * d20 +: ISR)       "APD van A[i]
    ( 7 * d20 + 1)          "APD van 1
```

ISR heet impliciet omdat hij door de vertaler gemaakt wordt, zonder in de gegeven tekst een expliciet analogon te hebben, zoals b.v. een subroutine die gecreëerd is naar aanleiding van een procedure declaratie die samenhangt met die declaratie. De *ISR* is in dit geval gemaakt omdat de parameter *A*[*i*] te ingewikkeld is om in één woord te worden gekarakteriseerd. Is er geen *ISR*, dan is er natuurlijk ook geen sprong over de impliciete subroutine.

In het overzicht van *APD*'s en *APIC*'s zien we, dat op grond van de *APD* voor *x* tijdens executie door *CEN* een *APIC* geconstrueerd wordt die bestaat uit

a) een haalopdracht *F = M*[*adres x*]

b) een copie van de codering en de display pointer *(0 * d20 +* [*dp*]*)*.

Elke *APIC* bevat als *APIC1* zo'n copie van de codering + een displaypointer. Deze laatste is in het algemeen dezelfde als is opgeslagen als geredde oude dp op plaats 1 van de blokcel van een procedure (uitzondering: het geval van de doorgegeven formele). Deze displaypointer wordt slechts gebruikt binnen impliciete subroutine's (om de betekenis van dynamische adressen van het moment van aanroepen te doen herleven).

Als er geen impliciete subroutine is, is dus feitelijk de aanwezigheid van die dp in *APIC1* overbodig. Het zou bij de constructie van de *APIC* echter extra tijd vergen om dat uit te zoeken en het is sneller om de *APIC1* steeds op dezelfde manier samen te stellen.

In genoemd overzicht staat bovendien bij *APD*'s met codenummers 0 t/m 3, 16 t/m 19 en 32 ook gegeven de inhoud van *APIC2* en *APIC3*, zoals die door *CLPN* extra (boven *APIC0* en *APIC1* als door *CEN* geproduceerd) gemaakt worden. Deze twee instructies verzorgen respectievelijk de preparatie van een

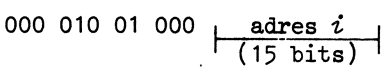
assignment en de eigenlijke assignatie aan de actuele parameter (zie blz.). De instructie in *APIC2* is in geval van codering 0 t/m 3, waarbij preparatie van assignments overbodig is, steeds zo gemaakt, dat ze

1) zo kort mogelijk duurt ($2\frac{1}{2}$ µs) en "onschuldig" is
2) het adres van de variabele bevat.

Het punt 2) is voor een real variabele overbodig. omdat daar de assignatie door de instructie in *APIC3*, n.l. *M[adres van x] = F*, direct gebeurt. Maar bij een integer variabele *i* mag in *APIC3* niet staan: *M[adres van i] = G*; omdat er mogelijk nog afgerond moet worden.
Er staat dus een subroutine aanroep naar *STAI* van het complex. De subroutine *STAI* (STore Actual Integer) zorgt voor afronden en assignatie, maar moet dan wel het adres van *i* kunnen achterhalen en vindt dat in *APIC2*.
Die routine *STAI* ziet er uit als volgt

```
STAI: U, S = F, Z      "S is de kop van F, is dat reeds integer?
      N, SUBC(: RND)   "zo nee, rond af
         S = MS[-1]    "neem adres van de variabele i in S
         MS = G        "berg het afgeronde resultaat in i op
         GOTOR(MC[-1])
```

Toelichting:
de assignering aan de formele parameter wordt uitgevoerd door een *DOS* instructie (zie blz. 64) en wel *DOS (Mp[q+3])*, als *Mp[q]* het adres van de *APIC* is. *DOS* heeft een neveneffect: hij neemt het adres waar de instructie stond in *S*. Het adres waar het resultaat van de afronding naar toe moet is nu gemakkelijk te vinden, omdat in *S* het adres van *APIC3* staat.
Dit gebeurt in de laatste drie instructies van *STAI*.
Opm.: in *S* zit na de instructie *S = MS[-1]* de inhoud van *APIC2*, die het bitpatroon



heeft.
NB bij dynamische adressering zijn behalve het eerste bit (dat een nul moet zijn) alleen de achterste 18 bits van belang, de 8 tussenliggende bits (waar nog enkele enen tussen zitten) zijn hier niet van belang.

De taak van *CEN* behelst: uit de *APD* (op grond van het rangnummer) de bijbehorende *APIC* (2 woorden) te construeren. Daartoe moet de *APD* gevonden worden.
De *APD* is te vinden onder de aanroep van de onderhavige procedure, het adres van die aanroep bevindt zich 1 plaats onder *pp* (die zich na *DPTR* in *A* bevindt).
Alle routines voor de behandeling van formele parameters moeten de link met 1 ophogen en *A* invariant laten, het tweede, opdat bij de behandeling van de volgende parameter de link gevonden kan worden, het eerste, opdat bij de behandeling van de volgende parameter de volgende *APD* geraadpleegd wordt en de link uiteindelijk wijst naar de 1° instructie na de *APD*'s.
Toelichting op de instructies van *CEN*.
CEN voldoet aan bovengenoemde eisen (zie b.v. de 2° instructie die de link met 1 ophoogt). De instructie

PLUSS (MA[-1])

heeft als effect dat de som van de operand (hier de inhoud van een dynamisch adres) en de inhoud van *S* gezet wordt in *S* en in de operand. We hebben nu met één instructie 2 dingen bereikt, n.l.

1) de link is met één opgehoogd en
2) in S staat bijna het adres van de *APD* (we zitten er één naast) die dan ook in de 3° instructie eenvoudig gehaald kan worden.

Na de 3° instructie

S = MS[-1]

hebben we de *APD*, deze bestaat uit 2 delen, n.l.

a) het rangnummer + het bit dat de adresseringswijze aangeeft en
b) een adres of een waarde.

De delen a) en b) scheiden we in de 4° en 5° instructie.
In de instructie

F=: MS

komt het adres (de achterste 18 bits van *S*) in *F*.

De instructie

S'x' - 327676

isoleert de code door *S* logisch met het getal -327676, of binair geschreven 111 111 111 111 000 000 000 000 000, te vermenigvuldigen.
De operatie *'x'* kijkt bit voor bit of er bij beide operanden een 1 staat, dan levert hij op die plaats een 1 anders een 0. Conclusie: in *S* komt

| copie van de oude kop van *S* | 15 nullen |

Door nu bij *S* de oude (heersende) *dp* op te tellen verkrijgen we *APIC1*, die we boven op de stapel zetten (6° en 7° instructie).
De instructie

RUS (15)

doet alle bitjes 15 plaatsen opschuiven en dan vragen we naar het 3° bit van achteren. Dit bit, oorspronkelijk het 18° bit van achteren, is het bit dat bepaalt of de adressering dynamisch of statisch is. De desbetreffende instructie

U, S'x'8, Z

vraagt of dat bit nul is (adres is statisch) en de volgende instructie

Y, JUMP(2)

laat dan 2 instructies overslaan. Is het adres dynamisch dan herleiden we het tot een statisch adres in de instructies

G + MASK 15
DO (G)

waardoor bij *G* (waarin het dynamisch adres geisoleerd was) de opdracht

F=: MO[-256]

(met 15 nullen als adres) opgeteld wordt en de zo verkregen opdracht als

instructie wordt uitgevoerd. Vervolgens schuiven we nogmaals de bits 5 plaatsen op door de instructie

RUS (5) ,

we hebben nu juist 20 bits opgeschoven, zodat achter in *S* precies het rangnummer staat. We kijken of rangnummer < 16 is door de instructie

U, S'x'48, Z

(bedenk dat 48 binair 110000 is en dus de conditie wordt *Yes* als het 5° en 6° bit van achteren nul zijn).
Alle actuele parameters die geen aanleiding geven tot een impliciete subroutine doch volledig beschreven worden door een *APD* hebben n.l. een rangnummer < 16 en in dat geval hangt de instructie voor *APICO* sterk van het rangnummer af. Is het rangnummer $\geq$ 16 dan bevat *APICO* een aanroep van *ISR*, het rangnummer is dan verder niet meer van belang en we vervangen het daarom door de vaste waarde -1.
De selectie van de *APICO*-instructie in afhankelijkheid van het rangnummer (dat in *S* staat) geschiedt door de instructies

S +: MASK 0
G + MS .

Merk op dat in het geval dat het rangnummer $\geq$ 16 is we wijzen naar *MASK[-1]* en dat daar inderdaad het bitpatroon van een subroutine aanroep staat.
(Het geval rangnummer = 15 krijgt een hele aparte behandeling, die staat na de label *FORMAL*.)
Na het vormen van de *APICO*-instructie uit de geselecteerde instructie en het adres (of de waarde) uit de *APD*, volgt

MC[-1] = G

die het resultaat naar de stapel wegschrijft.
Hadden we bijvoorbeeld de 3° parameter uit het voorbeeld *Pr(x,A[i],1)* onder handen, dan was het codenummer *7 * d20* en hadden we na de laatst besproken stapeling *F = 1* gestapeld, verkregen door *F = 0* bij *1* op te tellen (in de

instructie *G + MS*).
Zoals *CEN* uit een *APD* een *APIC* van 2 woorden construeert, construeert *CLPN* (na controle of het rangnummer wel 0 t/m 3, 16 t/m 19 of 32 is) een *APIC* van 4 woorden. De eerste 2 hiervan worden door *CLPN* met behulp van *CEN* gemaakt, de andere 2 worden er vervolgens bij gemaakt op grond van het rangnummer uit (de door *CEN* gemaakte) *APIC1* en eventueel het adres uit *APIC0*.

Voor parameters uit de value lijst worden subroutines aangeroepen, die geen *APIC* op de stapel achterlaten, maar die de waarde van de actuele parameter éénmaal uitrekenen, en die waarde op de stapel leggen (en zo tevens een of twee geheugenplaatsen reserveren voor de locale variabele, als hoedanig de formele parameter uit de value lijst verder fungeert).
Als voorbeeld nemen we het geval, dat een formele parameter uit de value lijst als real gespecificeerd staat. Deze parameter wordt dan behandeld door een aanroep van de subroutine *CRV*. Deze dient

1) de link van de procedure met 1 te verhogen
2) *A* invariant te laten
3) de actuele parameter te evalueren
4) deze als real, dus in 2 woorden, op de stapel te zetten, dus in de blokcel op te nemen.

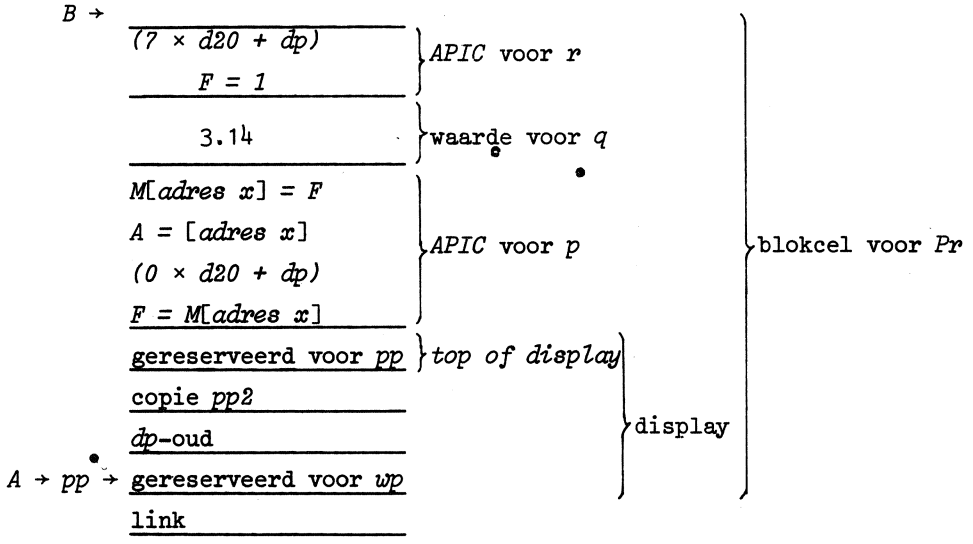
Toelichting op *CRN*: het ophogen van de link gebeurt door een aanroep van *CEN*, die tevens een *APIC* van 2 woorden maakt.
De evaluatie van de actuele parameter geschiedt door *APIC0* met een *DOS* uit te voeren. Aangezien hierdoor de waarde van *A* aangetast kan worden (als de actuele parameter ingewikkeld is, b.v. een arrayelement), wordt vooraf *A* gered en achteraf *A* hersteld. Tenslotte wordt de stapelruimte, in beslag genomen door de *APIC*,vrij gegeven (door 2 stapel verlagende instructies) en *F* weggeschreven over de link van *CRV* en de plaats waar *A* gered was heen.
Als de specificatie integer i.p.v. real is, wordt i.p.v. *CRV* de subroutine *CIV* aangeroepen met dezelfde taken 1), 2) en 3) als *CRV*, maar die i.p.v. 4) de geëvalueerde waarde zonodig tot een integer moet afronden en als 1 woord op de stapel moet zetten.
Als de heading van de procedure *Pr* luidt

procedure Pr(p,q,r); value q; real p, q, r;

, het displaylevel (van *Pr*) = 3 en top of display = 1 en als in de body van *Pr* aan *p* geassigneerd wordt dan luidt de aanhef van de vertaler van de declaratie van *Pr*

```
Pr:  S = D
     A = -1, Z
     SUB(: DPTR)
     B + 1
     SUB1(: CLPN)
     SUBC(: CRV)
     SUB(: CEN)
```

Een aanroep *Pr(x,A[i],1)* leidt, indien direct voorafgaande aan de aanroep *A[i]* de waarde 3.14 bevat, tot het volgende stapelbeeld op het moment na de behandeling van alle formele parameters



VII-5 De rest van de vertaling van een procedure declaratie: de macro's *TDL, ENTRPB, EXITP*

Als eenmaal afgerekend is met de correspondentie tussen de formele en (de bijbehorende) actuele parameter, kunnen we verder gaan met de vertaling van de procedure.

Er volgen de instructies

```
F =: MA[display level]     "TDL(display level)
B + [local space]          "ENTRPB(local space)
SUB2(: ENTRPB)
```

Deze vormen het analogon van de instructies voor de standaard blokingang (zie blz. 70)

```
A =: MC
F =: MD[display level]
B + [local space]
SUB2(: ENTRB)
```

Bij de ingang van een normaal blok moet de *pp* nog in *A* gezet worden (door *A =: MC*), bij een procedure staat deze reeds in *A* (sinds *DPTR*). In *F* moet het adres genomen worden, waar de *pp* in de *display* gezet moet worden; bij een normaal blok wijst *D* naar het begin van de *display*, bij een procedure *A* (en niet *D*!).
De taken van *ENTRPB* (ENTRance Procedure Body) zijn dezelfde als die van *ENTRB* (n.l. *pp* invullen in display, *wp* invullen op plaats nul van de blokcel, testen op overkoken van de stapel) plus extra het invullen van *pp* in *D* (hetgeen de nieuwe display van kracht maakt). Zie de tekst van *ENTRPB*.

Hierop volgen de vertaling van de declaratie en van de statements van de body. Tenslotte wordt de vertaling van een procedure declaratie voltooid met de procedureverlating.
Zie [2], §6.10 voor de procedureverlating. Er moet bij de procedureverlating

1) een stuk stapel vrijgegeven worden
2) de oude display weer van kracht gemaakt worden
3) teruggekeerd worden via de link, wat gedaan wordt door *EXITP*.

Voor het uitvoeren van taak 2) is een register nodig, we gebruiken daarvoor het *S*-register. Bedenk dat dat het enige register is dat altijd beschikbaar is, want het *F*-register kan een functiewaarde bevatten en *A* zou de waarde van een string procedure kunnen bevatten, de opzet om een type *string* in te voeren is echter in de X8 nooit voltooid.

We verlaten *EXITP* niet met *GOTOR* want dan worden alle kleine register gevuld vanuit de link met de oorspronkelijke waarden, m.a.w. het resultaat van een boolean procedure zou verdwijnen. Zie de tekst van *EXITP*.

VIII Garanties voor de eindigheid van het vertaalproces

Tenslotte nog een laatste onderwerp; en wel de vraag of het mogelijk is een tekst op te stellen die als hij aangeboden wordt aan de vertaler deze in een oneindig lang durende loop zou brengen (met de ALGOL-vertaler van X1 kan dit namelijk). Een andere vraag is: is er een bewijs te leveren dat zoiets niet kan?
Allereerst stellen we aan de in te voeren tekst de beperking dat diens lengte eindig moet zijn. De tekst *begin begin* ..... is dus niet toegelaten.
Indien de vertaler zou bestaan uit een collectie procedures die elk voor zich géén goto statements bevatten en ook geen for statements maar slechts assignment statements en procedure statements, al dan niet conditioneel, zou de eis dat alvorens een procedure aanroep plaats vindt er eerst een symbool van de tekst gelezen moet worden voldoende zijn om de eindigheid van het vertaalproces (als de tekst eindig is) te garanderen.
Deze eis is voor de vertaler iets te streng. De vertaler voldoet wel op alle essentieële punten aan het volgende stelsel eisen.
Voor elke procedure geldt een van de volgende n eigenschappen (met n eindig)

0) de afloop van het in de procedure beschreven proces kan gegarandeerd worden omdat
   a) er geen procedureaanroepen in voorkomen
   b) slechts aanroepen van procedures in voorkomen waarvan eigenschap 0 al bewezen is.

i) (0 < i < n) in het proces dat door de procedure beschreven wordt, wordt iedere aanroep van een procedure, waarvoor eigenschap j geldt ($j \geq i$) vooraf gegaan door het consumeren van tenminste één symbool van de tekst (input string) terwijl tenminste één aanroep van een procedure met eigenschap i-1 niet vooraf gegaan wordt door zo'n consumptie.

We bekijken een (fictief) voorbeeld van een procedure uit de vertaler:

```
procedure compound tail;
begin statement; ...;
      if last symbol = semi colon
      then begin next symbol;
                 compound tail;
           end;
      ...
end compound tail,
```

indien nu voor *statement* eigenschap k geldt dan geldt voor *compound tail* eigenschap k+1. Men voelt gemakkelijk aan, en men kan ook formeel bewijzen dat als *statement* een eindig proces beschrijft dan doet *compound tail* dat ook. Hiermee is aangegeven hoe met volledige inductie het bewijs geleverd zou kunnen worden dat de vertaler nooit in een oneindige cyclus kan raken. N.B. het is van groot belang om over dit soort zaken (gecompliceerde programma's) dit genre uitspraken te kunnen doen. Men ziet ook van welk vitaal belang de modulaire opbouw hierbij is!

Dat de vertaler een stuk ALGOL in een equivalent ELAN-programma vertaalt kan (misschien) bewezen worden ook weer dank zij de modulaire opbouw. Verschillende stukken worden n.l. onafhankelijk vertaalt en voor deze onderdelen apart kan men bewijzen dat de semantische betekenis onaangetast blijft.

Hiermee wordt dit college beeindigd. De behandeling van de vertaling van ALGOL-programma's en van de opbouw van de vertaler is nog lang niet compleet. Het is echter de hoop, dat toch uit de verf is gekomen, hoe het mogelijk is, met rekenautomaten ALGOL-programma's te verwerken; en dat dit niet geheimzinnig of moeilijk is, alleen maar gecompliceerd door de veelheid van op zichzelf belangrijke details.

APPENDIX

```
    procedure  Arithexp;
    begin   Integer future1, future2;
            If last symbol = if
            then begin  future1:= future2:= 0;
                        next symbol; Boolexp; Macro2 (COJU, future1);
                        if last symbol ≠ then then ERRORMESSAGE (300)
                                           else next symbol;
                        Simple arithexp;
                        if last symbol = else
                        then begin  Macro2 (JU,future2);
                                    Substitute (future1);
                                    next symbol; Arithexp;
                                    Substitute (future2)
                             end
                        else ERRORMESSAGE (301)
                 end
            else Simple arithexp
    end  Arithexp;

    procedure  Simple arithexp;
    begin   if last symbol = minus  then begin  next symbol; Term;
                                                Macro (NEG)
                                         end
                                    else begin  if last symbol = plus
                                                then next symbol;
                                                Term
                                         end;
            Next term
    end  Simple arithexp;

    procedure  Next term;
    begin   if last symbol = plus  then begin  Macro (STACK);
                                               next symbol; Term;
                                               Macro (ADD); Next term
                                        end
            else
            if last symbol = minus then begin  Macro (STACK);
                                               next symbol; Term;
                                               Macro (SUB); Next term
                                        end
    end  Next term;

    procedure  Term;  begin  Factor; Next factor end  Term;
```

```
procedure  Next factor;
begin   if last symbol = mul then begin  Macro (STACK);
                                          next symbol; Factor;
                                          Macro (MUL); Next factor
                                  end
        else
        if last symbol = div then begin  Macro (STACK);
                                          next symbol; Factor;
                                          Macro (DIV); Next factor
                                  end
        else
        if last symbol = idi then begin  Macro (STACK);
                                          next symbol; Factor;
                                          Macro (IDI); Next factor
                                  end
end  Next factor;

procedure  Factor;  begin  Primary; Next primary end  Factor;

procedure  Next primary;
begin  if last symbol = ttp then begin  Macro (STACK);
                                        next symbol; Primary;
                                        Macro (TTP); Next primary
                                end
end  Next primary;

procedure  Primary;
begin   integer  n;
        if last symbol = open then begin  next symbol; Arithexp;
                                          if last symbol = close
                                          then next symbol
                                          else ERRORMESSAGE (302)
                                  end
        else
        if digit last symbol  then begin  Unsigned number;
                                          Arithconstant
                                  end
        else
        if letter last symbol then begin  n:= Identifier;
                                          Subscripted variable (n);
                                          Function designator (n);
                                          Arithname (n)
                                  end
        else
        begin  ERRORMESSAGE (303);
               if last symbol = if ∨ last symbol = plus ∨
                  last symbol = minus
               then Arithexp
        end
end  Primary;
```

```
procedure  Arithname (n); integer n;
begin   if Nonarithmetic (n) then ERRORMESSAGE (304);
        complicated:= Formal (n) V Function (n);
        if Simple (n)
        then begin if Formal (n) then Macro2 (DOS, n) else
                   if Integer(n) then Macro2 (TIV, n)
                                 else Macro2 (TRV, n)
             end
end  Arithname;

procedure  Subscripted variable (n); integer n;
begin   if Subscrvar (n) then begin  Address description (n);
                                      if last symbol = colonequal
                                      then begin Macro (STACK);
                                                 Macro (STAA)
                                           end
                                      else Evaluation of (n)
                                 end
end  Subscripted variable;

procedure  Address description (n); integer n;
begin   if last symbol = sub
        then begin  next symbol; dimension:= Subscript list;
                    Check dimension (n);
                    if Formal (n)       then Macro2 (DOS, n) else
                    if Designational(n) then Macro2 (TSWE, n)
                                        else Macro2 (TAK, n)
             end
        else ERRORMESSAGE (305)
end  Address description;

procedure  Evaluation of (n); integer n;
begin   if  Designational(n)
        then begin  if Formal (n) then Macro (TFSL)
                    else  Macro (TSL)
             end
        else
        if  Boolean (n)      then Macro (TSB)  else
        if  String (n)       then Macro (TSST) else
        if  Formal (n)       then Macro (TFSU) else
        if  Integer (n)      then Macro (TSI)
        else Macro (TSR)
end  Evaluation of;
```

```
integer procedure  Subscript list;
begin   Arithexp;
        if last symbol = comma
        then begin  Macro (STACK); next symbol;
                    Subscript list:= Subscript list + 1
             end
        else begin  if last symbol = bus
                    then next symbol
                    else ERRORMESSAGE (306);
                    Subscript list:= 1
             end
end  Subscript list;
```

```
integer procedure  Order counter;
begin   Macro (EMPTY); Order counter:= instruct counter
end  Order counter;


procedure  Macro (macro number); integer  macro number;
begin   Macro2 (macro number, parameter) end  Macro;


procedure  Macro2 (macro number, metaparameter);
integer  macro number, metaparameter;
begin   macro:= if macro number < 512 then macro list[macro number]
                                      else macro number;
        parameter:= metaparameter;
        if state = 0
        then begin if macro = STACK then state:= 1
                   else
                   if Simple arithmetic take macro then Load (3)
                   else
                   Produce (macro, parameter)
             end
        else
        if state = 1
        then begin Load (2);
                   if ¬ Simple arithmetic take macro
                   then begin Produce (STACK, parameter); Unload end
             end
        else
        if state = 2
        then begin if Optimizable operator then Optimize
                   else
                   begin Produce (STACK, parameter); state:= 3;
                         Macro2 (macro, parameter)
                   end
             end
        else
        if state = 3
        then begin if macro = NEG then Optimize
                   else
                   begin Unload; Macro2 (macro, parameter) end
             end;
        if Forward jumping macro ∧ metaparameter ≤ 0
        then Assign (metaparameter)
end  Macro2;
```

```
procedure  Load (state i); integer  state i;
begin   stack0:= macro; stack1:= parameter; state:= state i end  Load;


procedure  Unload;
begin   Produce (stack0, stack1); state:= 0 end  Unload;


procedure  Optimize;
begin   stack0:= tabel[5 × Opt number (macro) + Opt number (stack0)];
        Unload
end  Optimize;


procedure  Assign (metaparameter); integer  metaparameter;
begin   metaparameter:= - (instruct counter - 1) end  Assign;


procedure  Produce (macro, parameter); integer  macro, parameter;
begin   integer  number, par number, entry, count;
        if macro = EMPTY then
        else
        if macro = CODE
        then begin space[prog base + instruct counter]:= parameter;
                   instruct counter:= instruct counter + 1;
                   test pointers
             end
        else begin number:= Instruct number (macro);
                   par number:= Par part (macro);
                   entry:= Instruct part (macro) - 1;
                   if par number > 0
                   then Process parameter (macro, parameter);
                   Process stack pointer (macro);
                   for count:= 1 step 1 until number do
                   Produce (CODE, instruct list[entry + count] +
                                  (if  count = par number
                                   then parameter else 0))
             end
end  Produce;


procedure  Process stack pointer (macro); integer  macro;
begin   if ¬ in code body
        then
        begin integer  reaction;
```

```
                reaction:= B reaction (macro);
                if reaction < 9
                then begin b:= b + reaction - 4;
                           if b > max depth then max depth:= b
                     end
                else
                if reaction = 10 then b:= 0
                else
                if reaction = 11 then b:= b - 2 × (dimension - 1)
                else
                if reaction = 12
                then begin if ecount = 0
                           then
                           begin ret level:= b;
                                 ret max depth:= max depth;
                                 b:= 0; max depth:= max depth isr
                           end;
                           ecount:= ecount + 1
                     end
                else
                if reaction = 13
                then begin if macro = EXITSV
                           then
                           begin if b > max depth isr
                                 then max depth isr:= b;
                                 b:= b - 2 × (dimension - 1)
                           end;
                           if ecount = 1
                           then
                           begin if max depth > max depth isr
                                 then max depth isr:= max depth;
                                 b:= ret level;
                                 max depth:= ret max depth
                           end;
                           if ecount > 0 then ecount:= ecount - 1
                     end
                else
                if reaction = 14
                then begin b:= display level + top of display;
                           if b > max display length
                           then max display length:= b;
                           ret max depth:= max depth
                     end
                else
                if reaction = 15
                then begin if b > max proc level
                           then max proc level:= b;
                           b:= 0; max depth:= ret max depth
                     end
           end
end  Process stack pointer;
```

```
procedure  Process parameter (macro, parameter);
integer  macro, parameter;
begin   if Value like (macro)
        then
        begin if macro = TBC
              then parameter:= if parameter = true then 0 else 1
              else
              if macro = SWP then parameter:= d9 × parameter
              else
              if macro ≠ EXITSV then parameter:= abs (parameter)
        end
        else
        begin if macro = JU ∨ macro = SUBJ ∨ macro = NIL ∨ macro = LAST
              then begin if parameter < 0
                         then parameter:= - parameter
                         else parameter:= Program address (parameter)
                   end
              else parameter:= Address (parameter) +
                               (if Dynamic (parameter)
                                then (if macro = TLV ∨ macro = TAA
                                      then function digit
                                      else if macro = STST
                                           then function letter
                                           else c variant)
                                else 0)
        end
end  Process parameter;

Boolean procedure  Simple arithmetic take macro;
begin   Simple arithmetic take macro:= bit string (d1, d0, macro) = 1
end  Simple arithmetic take macro;

Boolean procedure  Optimizable operator;
begin   Optimizable operator:= bit string (d2, d1, macro) = 1
end  Optimizable operator;

Boolean procedure  Forward jumping macro;
begin   Forward jumping macro:= bit string (d3, d2, macro) = 1
end  Forward jumping macro;

Boolean procedure  Value like (macro); integer  macro;
begin   Value like:= bit string (d4, d3, macro) = 1 end  Value like;

integer procedure  Opt number (macro); integer  macro;
begin   Opt number:= bit string (d8, d4, macro) end  Opt number;

integer procedure  Instruct number (macro); integer  macro;
begin   Instruct number:= bit string (d10, d8, macro)
end  Instruct number;

integer procedure  Par part (macro); integer  macro;
begin   Par part:= bit string (d12, d10, macro) end  Par part;
```

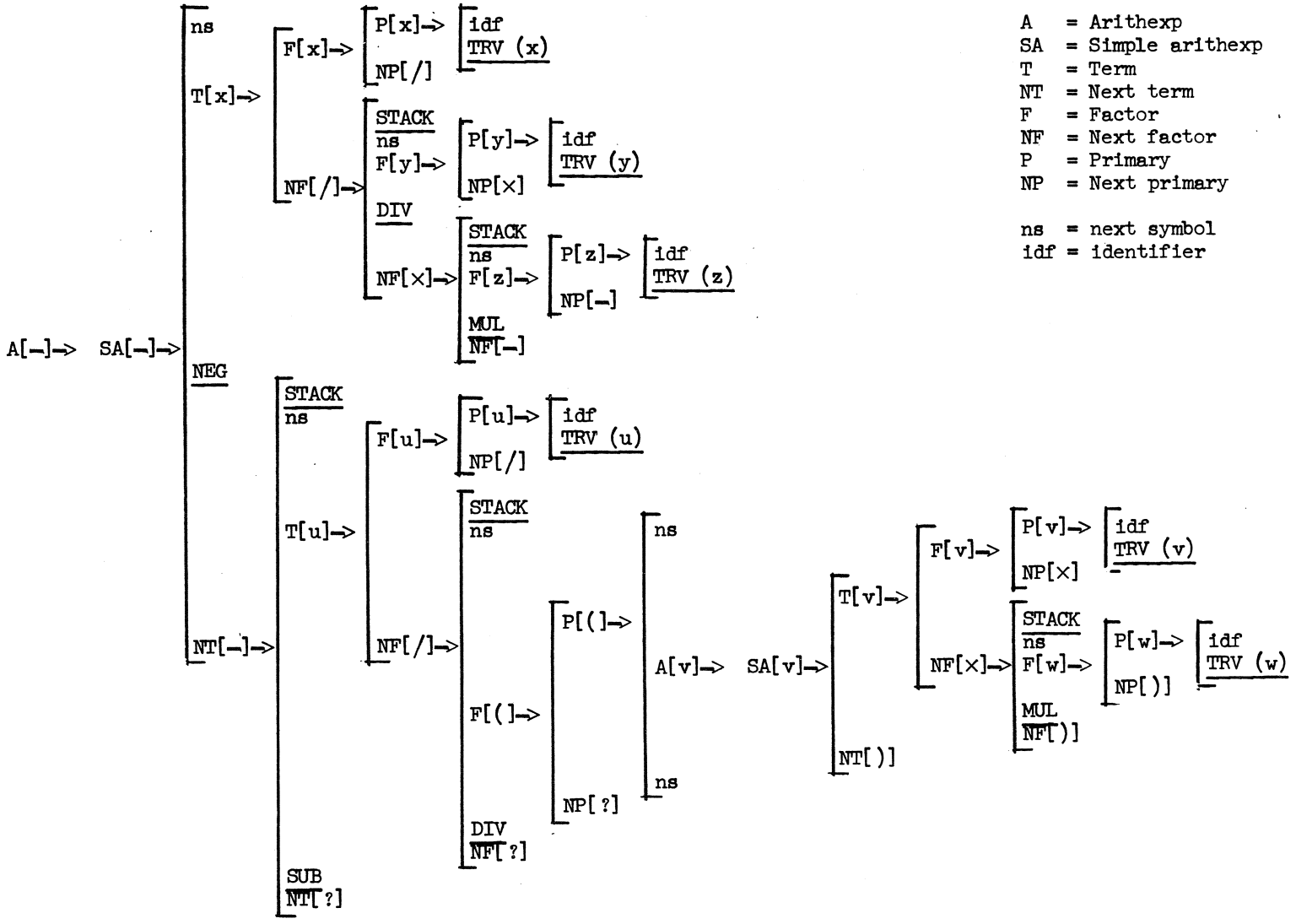
```
integer procedure  Instruct part (macro); integer  macro;
begin   Instruct part:= bit string (d21 , d12, macro) end  Instruct part;

integer procedure  B reaction (macro); integer  macro;
begin   B reaction:= macro : d21 end  B reaction;
```

Aanroepen van de procedures in het territorium van Arithexp bij de vertaling van de expressie:

$$- x / y \times z - u / (v \times w)$$

waarbij x, y, z, u, v, w geacht worden de identifiers van simpele locale of globale variabelen te zijn.


A = Arithexp
SA = Simple arithexp
T = Term
NT = Next term
F = Factor
NF = Next factor
P = Primary
NP = Next primary
ns = next symbol
idf = identifier
A[–]→ SA[–]→
ns
T[x]→
F[x]→
P[x]→
idf
TRV (x)
NP[/]
NF[/]→
STACK
ns
F[y]→
P[y]→
idf
TRV (y)
NP[×]
DIV
NF[×]→
STACK
ns
F[z]→
P[z]→
idf
TRV (z)
NP[–]
MUL
NF[–]
NEG
NT[–]→
STACK
ns
T[u]→
F[u]→
P[u]→
idf
TRV (u)
NP[/]
NF[/]→
STACK
ns
F[(]→
P[(]→
ns
A[v]→
SA[v]→
T[v]→
F[v]→
P[v]→
idf
TRV (v)
NP[×]
NF[×]→
STACK
ns
F[w]→
P[w]→
idf
TRV (w)
NP[)]
MUL
NF[)]
NT[)]
NP[?]
ns
DIV
NF[?]
SUB
NT[?]

simple arithmetic take macro's (met bijbehorend opt number):

TRV (0), TIV (1), TRC (2), TIC (3), TSIC (4)

optimizable operators (met bijbehorend opt number):

ADD (2), SUB (3), MUL (4), DIV (5),

EQU (6), UQU (7), LES (8), MST (9), MOR (10), LST (11)

opt number van NEG: 1

forward jumping macro's:

JU, COJU, YCOJU

name like macro's (= non-value like macro's):

TRV, TIV, TBV, TSTV, TAK, TSWE,

STR, STI, SSTI, STB, STST,

DOS, DOS2, DOS3,

JU (alleen als zijn parameter > 0), IJU, IJU1,

SUBJ (alleen als zijn parameter > 0), ISUBJ,

NIL, LAST,

TAA,

TNRV, TNIV,

ADDRV, ADDIV, SUBRV, SUBIV, MULRV, MULIV, DIVRV, DIVIV,

EQURV, EQUIV, UQURV, UQUIV, LESRV, LESIV,

MSTRV, MSTIV, MORRV, MORIV, LSTRV, LSTIV,

SLNC, RLNC

Opbouw van APD en APIC

actuele parameter	APD	APIC
real variable identifier (x)	(0 × d20 + adres x)	0 F = M[adres x] 1 (0 × d20 + [dp]) (2 A = [adres x]) (3 M[adres x] = F)
integer variable identifier (i)	(1 × d20 + adres i)	0 G = M[adres i] 1 (1 × d20 + [dp]) (2 A = [adres i]) (3 SUBC (:STAI))
Boolean variable identifier (b)	(2 × d20 + adres b)	0 S = M[adres b], P 1 (2 × d20 + [dp]) (2 A = [adres b]) (3 SUBC (:STABO))
string variable identifier (st)	(3 × d20 + adres st)	0 A = M[adres st] 1 (3 × d20 + [dp]) (2 A = [adres st]) (3 SUB2 (:STAST))
floating point constante (fp)	(4 × d20 + adres fp)	0 F = M[adres fp] 1 (4 × d20 + [dp])
integer constante (n)	(5 × d20 + adres n)	0 G = M[adres n] 1 (5 × d20 + [dp])
logische waarde (lw)	(6 × d20 + (lw|0|1))	0 S = (lw|0|1), P 1 (6 × d20 + [dp])
kleine positieve integer (k)	(7 × d20 + k)	0 F = [k] 1 (7 × d20 + [dp])
real array identifier (ar)	(8 × d20 + adres ar)	0 A = M[adres ar] 1 (8 × d20 + [dp])
integer array identifier (ai)	(9 × d20 + adres ai)	0 A = M[adres ai] 1 (9 × d20 + [dp])
Boolean array identifier (ab)	(10 × d20 + adres ab)	0 A = M[adres ab] 1 (10 × d20 + [dp])
string array identifier (ast)	(11 × d20 + adres ast)	0 A = M[adres ast] 1 (11 × d20 + [dp])

Opbouw van APD en APIC (vervolg 1)

actuele parameter	APD		APIC
switch identifier (sw)	(12 × d20 + adres sw)	0	A = [adres sw]
		1	(12 × d20 + [dp])
kleine negatieve integer (– k)	(13 × d20 + k)	0	F = – [k]
		1	(13 × d20 + [dp])
label identifier (lb)	(14 × d20 + adres lb)	0	A = [adres lb]
		1	(14 × d20 + [dp])
formele identifier (f)	(15 × d20 + adres f)	0	[copie van APIC0]
		1	[copie van APIC1]
		(2	[bijbehorende APIC2])
		(3	[bijbehorende APIC3])
element van real array	(16 × d20 + adres ISR)	0	SUBC (:ISR)
		1	(16 × d20 + [dp])
		(2	SUBC (:ISR[2]))
		(3	SUB3 (:STASR))
element van integer array	(17 × d20 + adres ISR)	0	SUBC (:ISR)
		1	(17 × d20 + [dp])
		(2	SUBC (:ISR[2]))
		(3	SUB3 (:STASI))
element van Boolean array	(18 × d20 + adres ISR)	0	SUBC (:ISR)
		1	(18 × d20 + [dp])
		(2	SUBC (:ISR[2]))
		(3	SUB3 (:STASB))
element van string array	(19 × d20 + adres ISR)	0	SUBC (:ISR)
		1	(19 × d20 + [dp])
		(2	SUBC (:ISR[2]))
		(3	SUB3 (:STASST))
niet-assigneerbare expressie van arithmetische type	(20 × d20 + adres ISR)	0	SUBC (:ISR)
		1	(20 × d20 + [dp])
niet-assigneerbare expressie van Boolean type	(22 × d20 + adres ISR)	0	SUBC (:ISR)
		1	(22 × d20 + [dp])
niet-assigneerbare expressie van string type	(23 × d20 + adres ISR)	0	SUBC (:ISR)
		1	(23 × d20 + [dp])

Opbouw van APD en APIC (vervolg 2)

actuele parameter	APD		APIC
real procedure identifier	(24 × d20 + adres ISR)	0 1	SUBC (:ISR) (24 × d20 + [dp])
integer procedure identifier	(25 × d20 + adres ISR)	0 1	SUBC (:ISR) (25 × d20 + [dp])
Boolean procedure identifier	(26 × d20 + adres ISR)	0 1	SUBC (:ISR) (26 × d20 + [dp])
string procedure identifier	(27 × d20 + adres ISR)	0 1	SUBC (:ISR) (27 × d20 + [dp])
designational expressie	(28 × d20 + adres ISR)	0 1	SUBC (:ISR) (28 × d20 + [dp])
integer constante of designational expressie	(29 × d20 + adres ISR)	0 1	SUBC (:ISR) (29 × d20 + [dp])
procedure identifier	(30 × d20 + adres ISR)	0 1	SUBC (:ISR) (30 × d20 + [dp])
niet assigneerbare expressie van onbekend type	(31 × d20 + adres ISR)	0 1	SUBC (:ISR) (31 × d20 + [dp])
mogelijk-assigneerbare expressie van onbekend type	(32 × d20 + adres ISR)	0 1 (2 (3	SUBC (:ISR) (32 × d20 + [dp]) SUBC (:ISR[2])) SUB3 (:STASU))

Enige subroutines uit het complex

```
ENTRIS:          A =   D
                 MC =  A              "save dp
                 S = MS[1]
                 D =   S              " new value of dp = APIC[1]
ENTRIS4:      U, B - bcheck, P        " dangerous growth of stack ?
              N, GOTO (LINK[2])
                 GOTO (:CONSIDER)     " 7 instructions

EXITIS:          S = MC[-1]
                 D =   S              " reset dp
                 GOTO (MC[-1])        " 3 instructions

ENTRPB:          D =   A              " new value of dp
ENTRB:           MA =  B              " wp
                 MC =  A              " pp
                 GOTO (:ENTRIS4)      " 4 instructions

DPTR:            'BEGIN'  DOUBLE, TEST, END
                 G =   D
                 MC =  F              " dp[call] in linkdata
              Y, GOTO (:END)          " no transport in case length = 0
                 stock = A
                 GOTO (:TEST)
DOUBLE:          S +   2
                 F =  MS
                 MC =  F              " double transport
TEST:            A +   2, P
              N, GOTO (:DOUBLE)
                 A -   2, Z
              N, A = MS[2]
              N, MC =   A             " single transport
                 A = stock
END:             A + :MC[-2]          " construct pp in A
                 GOTO (LINK)          " 16 instructions
                 'END' DPTR

EXITP:           B = :MD              " pp
                 S = M[B+1]
                 D =   S              " reset dp
                 GOTO (MC[-1])        " 4 instructions
```

Enige subroutines uit het complex (vervolg 1)

```
CRV:        MC = A              " save pp
            SUB (:CEN)
            DOS (M[B-2])        " execute APIC[0]
            A = MC[-3]          " reset pp in A
            S = MC[-3]          " take link in S
            M[B-2] = F          " overwrite pp/link by real value
            GOTO (:MS)          " 7 instructions

CEN:        'BEGIN' FORMAL, MASK0, MASK15

            S = 1
            PLUSS (MA[-1])      " increase link by 1
            S = MS[-1]          " take APD
            F = :MS             " isolate address
            S 'X' - 32767       " isolate code
            S + :MD             " code + (cleared) dp
            MC[1] = S           " together make APIC[1]
            RUS (15)
          U, S 'X' 8, Z         " static address ?
          Y, JUMP (2)
            G + MASK15
            DO (G)              " reduce dynamic address to static c
            RUS (5)             " isolate ordinal number
          U, S 'X' 48, Z        " actual parameter simple ?
          N, S = - 1            " else select SUBC (:M[0])
          U, S - 15, Z          " passed on formal ?
          Y, GOTO (:FORMAL)
            S + :MASK0
            G + MS              " select function part of APIC[0]
            MC[-1] = G          " APIC[0]
            GOTO (LINK)
FORMAL:     S = MG[1]           " APIC[1]
            G = MG              " and APIC[0]
            M[B] = S            " copied
            MC[-1] = G
            GOTO (LINK)
            SUBC (:M[0])        " complicated actual parameters
```

Enige subroutines uit het complex (vervolg 2)

```
MASK0:          F = M[0]            " real variable
                G = M[0]            " integer variable
                S = M[0], P         " Boolean variable
                A = M[0]            " string variable
                F = M[0]            " real constant
                G = M[0]            " integer constant
                S =   0, Z          " logical value
                F =   0             " small integer constant
                A = M[0]            " real array identifier
                A = M[0]            " integer array identifier
                A = M[0]            " Boolean array identifier
                A = M[0]            " string array identifier
                A = :M[0]           " switch identifier
                F = - 0             " negative small integer constant
                A = :M[0]           " label
MASK15:         F = :M0[-256]       " 43 instructions

                'END' CEN
```

Literatuur college "Programmeren voor rekenautomaten" 1969-1970

hardware:

NASHELSKY, L., "Digital Computer Theory",
Wiley, 1967.

programmeren:

KNUTH, D.E., "The Art of Computer Programming", Vols. I, II,
Addison-Wesley, 1967/1969.

FOSTER, J.M., "List Processing",
Elsevier, 1968.

standaard-functies:

HASTINGS, C. et al., "Approximations for Digital Computers",
Princeton University Press, 1955.

LYUSTERNIK, L.A. et al., "Handbook for Computing Elementary Functions",
Pergamon Press, 1965.

FIKE, G.T., "Computer Evaluation of Mathematical Functions",
Prentice-Hall, 1968.

ALGOL 60:

NAUR, P. (ed), "Revised Report on the Algorithmic Language ALGOL 60",
Regnecentralen, 1962.

RUTISHAUSER, H., "Description of Programming in ALGOL 60",
Handbook for Automatic Computing, Vol. I/part a,
Springer-Verlag, 1967.

HIGMAN, B., "What Everybody should know about ALGOL",
in Collected Algorithms from CACM, en
in The Computer Journal 6 (1963) 50.

HIGMAN, B., "A Comparative Study of Programming Languages",
Elsevier, 1967.

compilers:

RANDELL, B. en RUSSELL, L.J., "ALGOL 60 Implementation",
Academic Press, London, 1964.

GRAU, A.A. et al., "Translation of ALGOL 60",
Handbook for Automatic Computing, Vol. I/part b,
Springer-Verlag, 1967.

BOLLIET, L., "Compiler Writing Techniques",
in GENUYS, F. (ed), "Programming Languages",
Academic Press, 1968.

HOPGOOD, F.R.A., "Compiling Techniques",
Elsevier, 1969.

ROSEN, S. (ed), "Programming Systems and Languages",
McGraw-Hill, 1967.

KRUSEMAN ARETZ, F.E.J., "Het objectprogramma gegenereerd door de X8-ALGOL
60-vertaler van het MC",
Mathematisch Centrum, 1971.