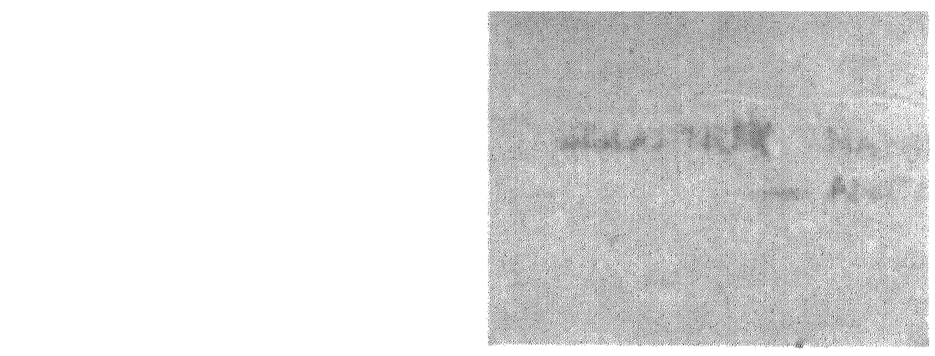
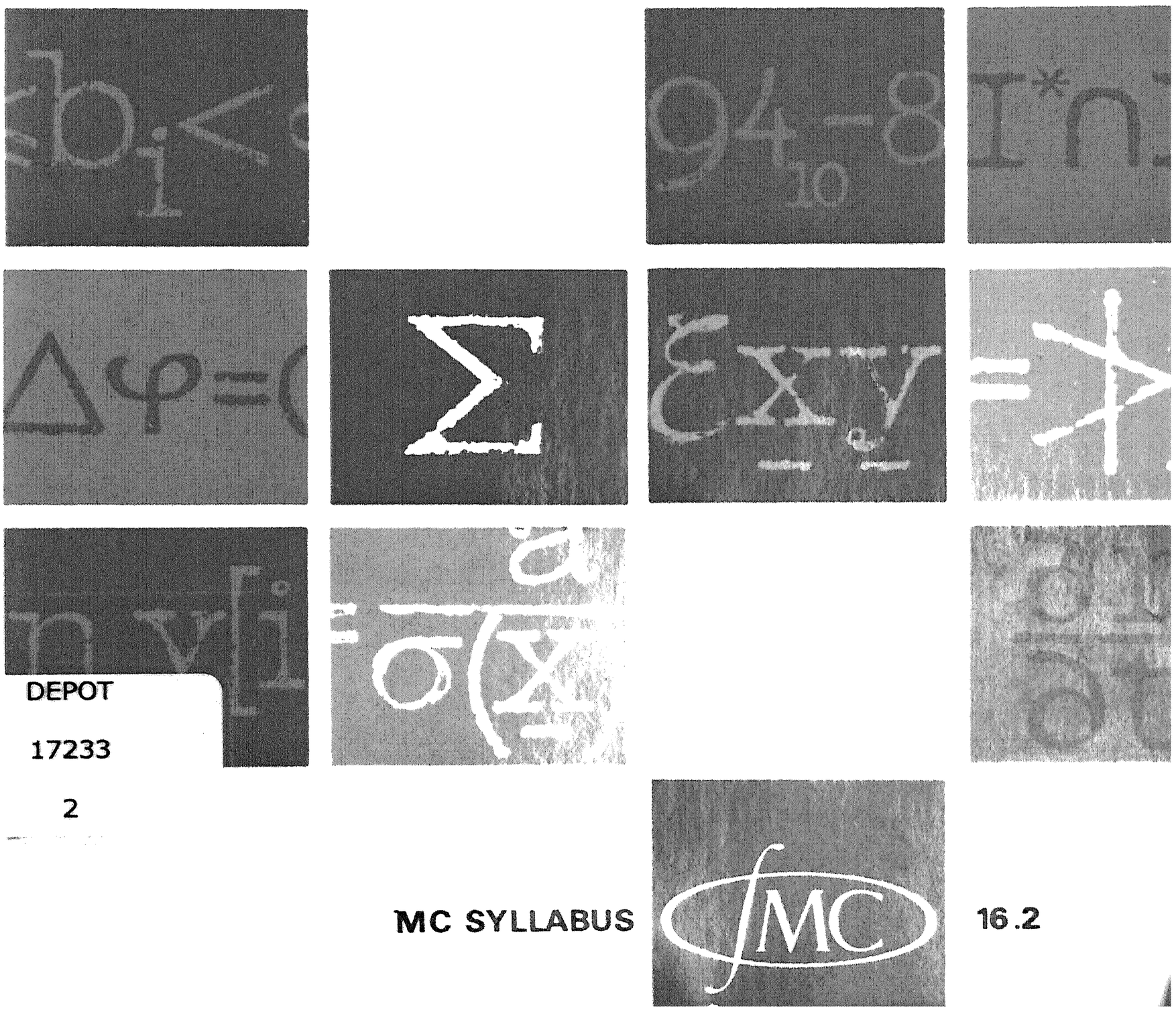


CURSUS PROGRAMMEREN

DEEL 2 DE PROGRAMMEERTAAL ALGOL 60

L. GEURTS



DEPOT

17233

MC SYLLABUS 16.2

MC SYLLABUS 16.2

L. GEURTS

CURSUS PROGRAMMEREN

DEEL 2
DE PROGRAMMEERTAAL ALGOL 60

BIBLIOTHEEK MATHEMATISCH CENTRUM
AMSTERDAM

MATHEMATISCH CENTRUM AMSTERDAM 1973

AMS(MOS) onderwerpen classificatie schema (1970): 68-01
ACM - Computing Reviews - categorie: 4.22, 1.5

Inhoud

blz.

1. Inleiding 1
   1.1. Iets over computers 1
   1.2. Het prepareren van een programma voor een computer 2
   1.3. De verwerking van een ALGOL-programma door een computer 3
2. Eerste kennismaking met ALGOL 60 5
3. Programma, compound statement en block 10
   3.1. Informele beschrijving 10
   3.2. Backus-notatie 11
4. Statements 16
   4.1. Assignment statement 16
   4.2. Go to statement, label, switch 18
   4.3. Dummy statement 20
   4.4. Conditional statement 22
   4.5. For statement 24
5. Expressions 31
   5.1. Arithmetic expression 31
   5.2. Number 35
   5.3. Variable 36
   5.4. Boolean expression 38
6. Declaration, scope 43
   6.1. *own* 47
   6.2. Declaratie van simple variables 48
   6.3. Array declaration 49
   6.4. Switch declaration 52
7. Procedures 57
   7.1. Procedure declaration 59
   7.2. Procedure statement 65
   7.3. De precieze betekenis van een procedure statement 65
   7.4. Wanneer *value*? 68
   7.5. Jensen's device 68

7.6. Function designator 73
7.7. Standaard procedures 74
8. Identifier, string, letter, digit, comment 76

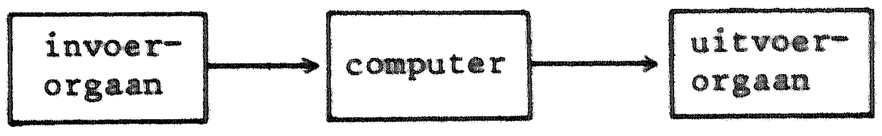
Uitwerking van de vraagstukken van deel 2 79

1. Inleiding

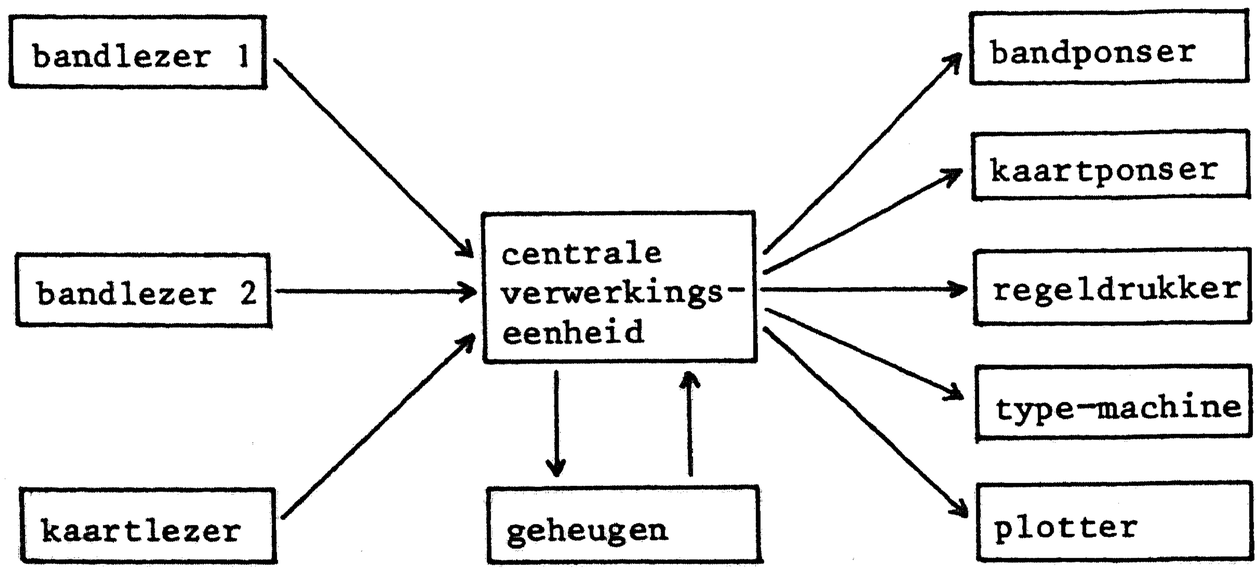
In dit deel komt het noteren van algoritmen in de programmeertaal ALGOL 60 (ALGOrithmic Language 1960) aan de orde. ALGOL 60 (of kortweg ALGOL) wordt op exakte wijze gedefinieerd in het "Revised Report on the Algorithmic Language ALGOL 60". Voor deze definitie is weer een speciaal formalisme gebruikt, de zgn. Backus-notatie, dat in 3.2 behandeld zal worden. Input en output worden door het Revised Report niet behandeld; voorzieningen op dit gebied worden aan de verschillende computerfabrikanten en rekencentra overgelaten. We zullen in deze cursus veelal de input- en outputmogelijkheden gebruiken zoals ze voor het Electrologica X8-ALGOL-systeem van het Mathematisch Centrum zijn aangegeven in de gebruikershandleiding LR 1.1. Het Revised Report is zeer geschikt als naslagwerk, maar niet als introductie tot ALGOL 60. We zullen dan ook in deze kursus een zo volledig mogelijk overzicht over ALGOL 60 geven, van tijd tot tijd verwijzend naar het Revised Report (RR) en de gebruikershandleiding (LR 1.1.).

1.1. Iets over computers

We kunnen ons een computer die ons ALGOL-programma uitvoert ruwweg voorstellen als een machine waar ons programma + invoer ingaat, en waar de output uitkomt.



Het invoerorgaan bestaat vaak uit verscheidene apparaten: ponsbandlezers en ponskaartlezers; uitvoerorganen kunnen zijn: bandponsers, type-machines, regeldrukkers (een snel soort type-apparaten), plotters (tekenapparaten).
Aan de centrale computer kunnen we twee onderdelen onderscheiden:
het geheugen en de centrale verwerkingseenheid:



Vele computers hebben enkele soorten geheugens: een kern-geheugen (het normale type), trommelgeheugen (een soort centrifuge), magnetische banden (zoiets als banden voor een band-recorder). Elk van deze media leent zich voor het opnemen en weer teruggeven van informatie.

1.2. Het prepareren van een programma voor een computer

Als we een algoritme in ALGOL 60 hebben opgeschreven, kunnen we daarmee nog niet naar de computer, want het kan niet door de bestaande invoer-apparatuur gelezen worden. We moeten het programma dus vastleggen op ponsband of op ponskaarten. Zo'n ponsband is een lange papieren strook waarin een lange reeks gaatjes-combinaties kan worden geponst; door ons programma te tikken op een flexowriter (dat is een elektrische typemachine die bij elke aanslag niet alleen een symbool afdrukt op papier maar ook nog een voor de aanslagen toets specifieke gaatjes-combinatie in een ponsband ponst), krijgen we vanzelf een voor een bandlezer leesbare vorm van ons programma.
Zo bestaan er ook ponskaartmachines, waarmee we ons programma in een serie ponskaarten kunnen vastleggen. De invoergegevens kunnen we op dezelfde manier voor de computer geschikt maken. Sommige computersystemen hebben ook terminals, dat zijn elektrische schrijfmachines die, eventueel op grote afstand, via een soort telefoonleiding met de computer verbonden zijn. Deze terminals zijn zowel invoerorgaan (men kan zijn programma erop intikken), als uitvoerorgaan (de te typen resultaten worden door de computer op de terminal getypt).

1.3. De verwerking van een ALGOL-programma door een computer

Men zou misschien geneigd zijn te denken dat de computer steeds een stukje van het te verwerken programma leest en uitvoert, totdat het programma helemaal gelezen is en de uitvoering voltooid is. Dat is wel de manier waarop de menselijk uitvoerder van algoritmen te werk gaat, maar in het algemeen is de computer-verwerking van ALGOL-programma's wat ingewikkelder georganiseerd. Toch hoeven we ons daarvan bij het opstellen van programma's niets aan te trekken, want de resultaten die de computer aflevert zullen dezelfde zijn als die de menselijke uitvoerder krijgt, als hij geen fouten maakt.
Het is uitsluitend om praktische redenen dat het met de computer anders georganiseerd is. Een van die redenen is dat ALGOL niet de "moedertaal" van zo'n computer is. Wel is er een vertaal-programma, geschreven in machine-code (dat is de moedertaal van de betreffende machine), dat een ALGOL-programma als input krijgt, en als output een programma aflevert in machine-code, dat equivalent is met het ALGOL-programma.
Ons ALGOL-programma dient dus als invoer bij een vast vertaal-programma dat alle ALGOL-programma's die aan de computer worden aangeboden vertaalt in machine-code programma's die hetzelfde effekt hebben.
De tekst van het vertaalde programma wordt (in het algemeen) in het geheugen van de computer geschreven. De invoergegevens worden ook in het geheugen geschreven, en dan begint de computer het vertaalde programma stap voor stap uit te voeren, eraan denkend dat de input-opdrachten betrekking hebben op de invoer-gegevens die nu in het geheugen staan.
Bij vele rekencentra is het nog wat ingewikkelder, en wel om de volgende reden. De meeste ALGOL-programma's die worden aangeboden duren niet lang (zeg 1 à 4 minuten), maar toch is er elke dag wel een programma van een uur of langer bij. Om er nu voor te zorgen, dat degenen die een kort programma aanbieden niet een uur op de voltooiing van een lang programma hoeven te wachten, is er voor vele reken-installaties een programma gemaakt, dat ervoor zorgt dat de aangeboden programma's niet in de volgorde waarin ze binnenkomen worden verwerkt en voltooid, maar dat de programma's als het ware in een aantal queues terecht komen, wachtend voor een "loket" tot ze aan de beurt komen. (Er is bijv. een loket waar de programma's tot 2 minuten worden geholpen, en een voor programma's van 2 tot 30 minuten, en een voor

programma's langer dan 30 minuten.) Toch is er maar één computer om de verschillende loketten te bedienen; dat kan zo: eerst wordt de klant voor loket 1 een minuut geholpen; als hij dan nog niet klaar is, wordt het betreffende programma en alle "tussenresultaten" even in de ijskast gezet, om straks weer verder te kunnen worden behandeld; dan komt de klant die bij loket 2 aan de beurt is gedurende een minuut aan bod, en wordt zonodig in ijskast 2 gezet, enz. Als alle loketten een minuut aan bod zijn geweest, komt de klant aan loket 1 weer aan de beurt.
Natuurlijk kan het bestuur-programma dat de beurten zo regelt, niet zomaar aan een binnenkomend programma zien hoelang het zal duren. Daarom is het nodig aan het begin van een programma een indicatie hierover te geven. (Voor precieze conventies op het Mathematisch Centrum, zie LR 1.1.).
Het geheel van programma's dat zo de gang van zaken bij de uitvoering van programma's regelt heet de systeem-programmatuur. De systeem-programmatuur bevat meestal ook nog een aantal standaard-programma's en stukken programma, waarvan de ALGOL-programmeur in zijn programma gebruik kan maken.
Ook zorgt de systeem-programmatuur dat fouten die bij het vertalen of verwerken in een ALGOL-programma worden gevonden aan de gebruiker worden gemeld. (Zie ook LR 1.1.7.)

N.B. De systeem-programmatuur is er alleen om het de ALGOL-programmeur gemakkelijker te maken in een aantal praktische zaken; voor het overige wordt het programmeren er niet essentieel door veranderd.

2. Eerste kennismaking met ALGOL 60

De schrijfwijze voor een aantal ALGOL-konstrukties kennen we al uit deel 1, zoals blijkt uit dit ALGOL-programma:

```
begin   comment Waar staat het grootste getal;
        integer aantal;
        aantal:= read;
        if aantal > 0 then
        begin   integer i, max, waar;
                max:= read;
                waar:= 1;
                for i:= 2 step 1 until aantal do
                begin   integer nieuw;
                        nieuw:= read;
                        if nieuw > max then
                        begin   max:= nieuw;
                                waar:= i
                        end
                end;
                printtext ('grootste staat op plaats');
                print (waar)
        end
end
```

De indeling in regels van deze tekst is niet essentieel: desnoods mag elk woord op een nieuwe regel staan, of mag het hele programma op een regel staan; de betekenis van een ALGOL-programma wordt door de lay-out niet beinvloed.
De onderstreepte Engelse woorden zoals

begin comment integer if then for step until do end

vormen samen met tekens als

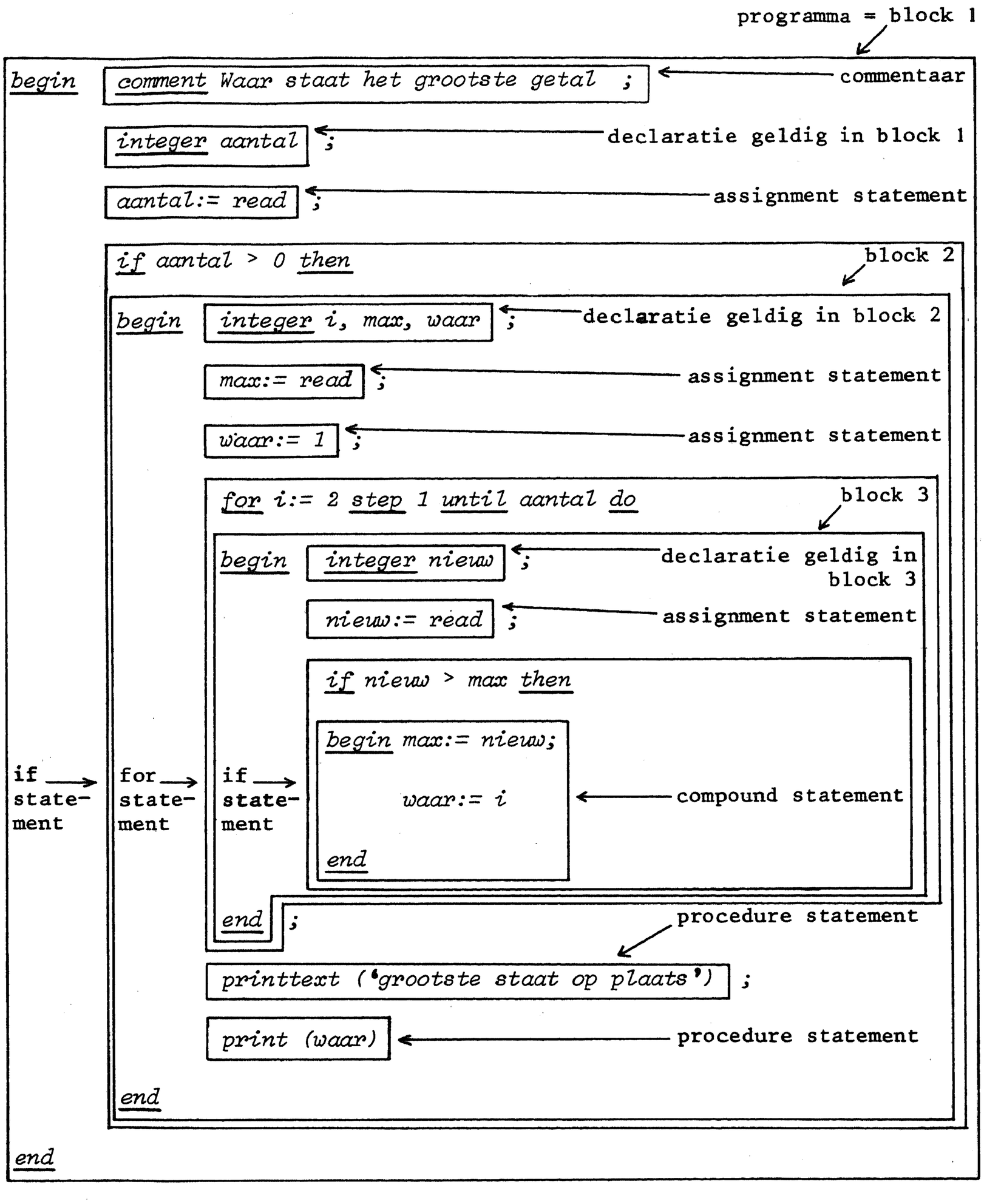
; := > , (')

het skelet van een programma. Het is een beperkte verzameling van vaste symbolen (zo'n onderstreept woord moet als een enkel symbool worden opgevat, net als :=), waartussen de tekst staat die de programmeur vrij kan kiezen.

De opdrachten, die statements worden genoemd, worden in ALGOL door puntkomma's van elkaar gescheiden.
Uit het voorbeeld zijn nog een paar dingen over ALGOL te leren:

- Een programma bestaat uit een enkele statement, want een serie statements, voorafgegaan door *begin* en gevolgd door *end* vormen een statement.
- *comment* duidt aan dat de tekst tot aan de volgende ; als commentaar voor de menselijke lezer moet worden opgevat.
- *integer i, max, waar* (dit is een declaratie) betekent dat in het block (d.w.z. een serie statements, voorafgegaan door declaraties, en omvat door *begin* en *end*) de variabelen *i*, *max* en *waar* gebruikt zullen worden voor gehele getallen.
- Voor input en output wordt gezorgd door *read*, *printtext* en *print* (behorend bij het X8-ALGOL-systeem).

Om de struktuur van het programma duidelijker te zien, bekijken we nog eens de tekst ervan, nu aangevuld met een aantal vaktermen voor de verschillende bouwstenen van een programma:


programma = block 1
begin
comment Waar staat het grootste getal ;
commentaar
integer aantal ;
declaratie geldig in block 1
aantal:= read ;
assignment statement
if aantal > 0 then
block 2
begin
integer i, max, waar ;
declaratie geldig in block 2
max:= read ;
assignment statement
waar:= 1 ;
assignment statement
for i:= 2 step 1 until aantal do
block 3
begin
integer nieuw ;
declaratie geldig in block 3
nieuw:= read ;
assignment statement
if nieuw > max then
begin max:= nieuw;
waar:= i
end
compound statement
end ;
procedure statement
printtext ('grootste staat op plaats') ;
print (waar)
procedure statement
end
end
if statement
for statement
if statement

We zien dat het gehele programma de vorm heeft van een block:

1. *begin*
2. (eventueel) commentaar
3. declaratie(s) (één in dit geval)
4. statements (twee stuks in dit geval)
5. *end*

Het commentaar wordt beëindigd door een ; en de declaraties en de statements worden onderling en van elkaar gescheiden door een ;. De declaratie *integer aantal* is geldig voor het block dat begint met het *begin* dat aan de declaratie voorafgaat, d.i. block 1, dus het gehele programma.
De statement *aantal:= read* is een assignment statement (toekenningsopdracht), herkenbaar aan :=.

De tweede statement van block 1 is een if statement. Een if statement bestaat uit een if clause, gevolgd door een statement. De if clause heeft de vorm:

1. *if*
2. een Boolean expression, d.w.z. een test die als resultaat ja (*true*) of nee (*false*) kan hebben
3. *then*

De statement die op de if clause volgt heeft hier opnieuw de vorm van een block, waarbij de declaratie *integer i, max, waar* alleen geldig is voor dit block 2. Block 2 bevat twee assignment statements en een for statement. Een for statement bestaat uit een for clause, gevolgd door een statement. De for clause heeft de vorm:

1. *for*
2. variabele
3. :=
4. arithmetic expression, d.w.z. een uitdrukking "waar een getal uitkomt"
5. *step*
6. arithmetic expression
7. *until*
8. arithmetic expression
9. *do*

De statement die op de for clause volgt heeft hier de vorm van een block,

integer *nieuw* is nu alleen geldig voor dit block 3 (de variabele *nieuw* zou dan ook niet buiten dit block gebruikt mogen worden).
Block 3 bevat een assignment statement en een if statement.
Deze if statement bevat, na de if clause *if* *nieuw* > *max* *then*, een compound statement (samengestelde statement), d.w.z. een block, maar dan zonder declaraties.
De derde en de vierde statement van block 2 zijn procedure statements, d.w.z. aanroepen van procedures, nl. van *printtext* en *print*. Deze output-procedures zijn niet gedeclareerd, omdat ze tot de standaard-procedures behoren.

3. Programma, compound statement en block (RR 4.1.)

3.1. Informele beschrijving

Een programma bestaat, afgezien van commentaar, uit een reeks statements, die in het algemeen voorafgegaan wordt door declaraties; de declaraties en de statements zijn onderling en van elkaar gescheiden door het symbool ; ; dit geheel wordt nog voorafgegaan door de "statement -haak" *begin* en afgesloten door de "statement-haak" *end*. Als er geen declaraties voorkomen heet zo'n konstruktie een compound statement, anders een block.

Zo'n block of compound statement is zelf ook weer een statement, en kan als zodanig in een ander block of compound statement optreden.

Elke statement kan beginnen met een of meer labels, van elkaar en van de rest van de statements gescheiden door het symbool : .

Declaraties zijn verplicht voor alle variabelen, arrays, procedures etc. die voorkomen. Een declaratie geldt voor het gehele block dat met het bijbehorende *begin* geopend wordt; komt echter in een binnen-block een declaratie voor met een gelijkluidende identifier (d.i. naam van b.v. een variabele), dan geldt in dat binnen-block alleen die laatste declaratie.
Het is in het algemeen verstandig declaraties pas te schrijven in het binnen-block waar dat echt noodzakelijk is. Voordelen daarvan zijn:

- de lezer van het programma kan gemakkelijk de betekenis van b.v. een variabele begrijpen, omdat hij weet in welk stuk van het programma deze kan voorkomen. Ook de programmeur zelf houdt hierdoor een beter overzicht over zijn programma.
- de programmeur hoeft zich geen zorgen erover te maken, of de naam die hij kiest al elders in het programma voor een ander doel in gebruik is; door deze naam ter plaatse in de declaratie op te nemen wordt ongewenste interferentie voorkomen.
- bij het veranderen van een deel van het programma, worden de declaraties die hierbij eventueel moeten worden veranderd, ook zoveel mogelijk in dat stuk programma aangetroffen.
- als een stuk programma in een ander programma zou kunnen worden overgenomen,

is het gemakkelijk als de bijbehorende declaraties in het stuk zelf voorkomen.
- zuinig geheugen-gebruik.

Een compound statement of een block wordt gebruikt telkens als in een constructie één statement wordt geëist waar we graag meer dan een statement zouden schrijven. Een nieuw block zullen we inwijden als we nieuwe declaraties willen doen.

3.2. Backus-notatie

De regels die gelden voor de opbouw van een ALGOL-programma kunnen op formele wijze worden weergegeven in de zgn. Backus-notatie (J.W. Backus is een van de auteurs van het Revised Report). In deze notatie komen de volgende meta-linguistische symbolen voor:

< en > gebruikt als haken die meta-linguistische variabelen aanduiden, zoals in <program>

::= in de betekenis "kan zijn"

| in de betekenis "of"

Een meta-linguistische produktie-regel zoals:

<compound tail>::= <statement> *end* | <statement> ; <compound tail>

betekent:

een compound tail kan zijn:

(1) een statement gevolgd door *end* , of

(2) een statement gevolgd door een ; gevolgd door een compound tail.

Blijkens het tweede alternatief kan een compound tail dus nog zijn:

(2.1) een statement gevolgd door een ; gevolgd door een statement gevolgd door *end* , of

(2.2) een statement gevolgd door een ; gevolgd door een statement gevolgd door ; gevolgd door een compound tail.

Samengevat:

Een compound tail is een reeks statements, onderling gescheiden door een ; en afgesloten door *end* .

Het geheel van de produktie-regels uit het Revised Report is de syntaxis (grammatika) van ALGOL. Met deze syntax kan elk korrekt ALGOL-programma geproduceerd worden, maar niet alle geproduceerde resultaten zijn korrekte

ALGOL-programma's: de restricties hiervoor worden buiten de syntax in het Revised Report genoemd.

Het produceren van een programma gaat zo:

werktekst := "<program>";
while de *werktekst* bevat nog meta-linguistische variabelen *do*
vervang in de *werktekst* elke meta-linguistische variabele door een van de alternatieven die in de produktie-regel voor deze meta-linguistische variabele genoemd worden {vervang dus b.v. <program> door b.v. <block>};
{de huidige *werktekst* is een eindproduktie van <program>}

Als we naast het stuk syntaxis dat aan het eind van dit hoofdstuk staat, nog deze produktie-regel kennen:

<dummy statement>::=

(d.w.z. de produktie van <dummy statement> is de lege tekst), dan kunnen we op de volgende manier het kortste ALGOL-programma produceren:

werktekst			toegepaste produktie-regel
<program>			
<compound statement>			2
<unlabelled compound>			5
begin <compound tail>			8
begin <statement>		*end*	11
begin <unconditional statement>		*end*	13
begin <basic statement>		*end*	16
begin <unlabelled basic statement>		*end*	19
begin <dummy statement>		*end*	23
begin		*end*	

>01> Geef een soortgelijke afleiding voor het (eveneens rijkelijk zinloze) programma

begin ; *end*

In de hieronder volgende syntaxis zijn de verschillende alternatieven op aparte regels genoemd; deze regels zijn genummerd om het refereren gemakke-

lijker te maken. Als rechts van ::= een meta-linguistische variabele staat die niet in hetzelfde stukje syntaxis wordt gedefinieerd, dan staat aan het eind van de regel een verwijzing naar de syntaxis-regel waar deze definitie wel voorkomt. In de rubriek Voorbeelden wordt voor elk van de alternatieven een voorbeeld gegeven, om de lezer een indruk te geven van wat hij zich moet voorstellen bij het betreffende alternatief, zonder dat hij hiervoor zelf een volledige afleiding hoeft te produceren met behulp van de gehele ALGOL-syntaxis.

Syntaxis

(1)	<program>::=	<block> |	
(2)		<compound statement>	
(3)	<block>::=	<unlabelled block> |	
(4)		<label> : <block>	(37)
(5)	<compound statement>::=	<unlabelled compound> |	
(6)		<label> : <compound statement>	(37)
(7)	<unlabelled block>::=	<block head> ; <compound tail>	
(8)	<unlabelled compound>::=	*begin* <compound tail>	
(9)	<block head>::=	*begin* <declaration> |	(123)
(10)		<blockhead> ; <declaration>	(123)
(11)	<compound tail>::=	<statement> *end* |	
(12)		<statement> ; <compound tail>	
(13)	<statement>::=	<unconditional statement> |	
(14)		<conditional statement> |	(43)
(15)		<for statement>	(49)
(16)	<unconditional statement>::=	<basic statement> |	
(17)		<compound statement> |	
(18)		<block>	
(19)	<basic statement>::=	<unlabelled basic statement> |	
(20)		<label> : <basic statement>	(37)
(21)	<unlabelled basic statement>::=	<assignment statement> |	(25)
(22)		<go to statement> |	(31)
(23)		<dummy statement> |	(41)
(24)		<procedure statement>	(179)

Voorbeelden

```
(1) begin   integer n;
    L:      n:= 1; go to L
    end

(2) begin L:print (read); go to L end

(3)         begin   real w;
                    w:= a; a:= z; z:= w
                end

(4) wissel: begin   real w;
                    w:= a; a:= z; z:= w
                end

(5)         begin imax:= i; max:= getal[imax] end
(6) nieuw:  begin imax:= i; max:= getal[imax] end
(7) zie (3)
(8) zie (5)
(9) begin   integer f, n

(10) begin  integer f, n; real a, z;
            integer array getal[1 : 500], S, T[0 : 3, 1 : 10];
            integer procedure KW (a); value a; real a;
            KW:= a × a

(11)        z:= w
    end

(12)        w:= a; a:= z; z:= w
    end

(13) a:= A[if b < 0 then -b else b]
(14) if b < 0 then a:= A[-b] else a:= A[b]
(15) for i:= n step -1 until 2 do f:= f × i
(16) go to L
```

```
(17) begin imax:= i; max:= getal[imax] end
(18) begin real w; w:= a; a:= z; z:= w end
(19)          n:= read
(20) L:       n:= read
(21) a:= 0
(22) go to L
(23)
(24) print (7)
```

4. Statements

4.1. Assignment statement (RR 4.2.)

We kunnen een variabele een waarde geven met behulp van de assignment statement, b.v.

A[f (1)]:= b / f (2)

Als we meer dan een variabele eenzelfde waarde willen geven, kan dat zo:

F[n]:= n:= n + 1

De uitvoering van zo'n assignment statement moeten we ons zo voorstellen:

1. Alle eventuele subscript expressions (expressies tussen vierkante haken, indices) links van het meest rechtse := teken worden uitgerekend, en wel in volgorde van links naar rechts.
2. De expressie rechts van het meest rechtse := teken wordt uitgerekend.
3. Dit resultaat wordt toegekend aan de variabelen links, met inachtneming van de in 1. uitgerekende waarden van de subscript expressions.

Als *n* de waarde *3* heeft, zullen dus na uitvoering van

F[n]:= n:= n + 1

zowel *n* als *F[3]* de waarde *4* gekregen hebben.

We concluderen dat twee opeenvolgende assignment statements van de vorm

<A> := <C> ; <B> := <C>

niet klakkeloos geschreven kunnen worden als

<A> := <B> := <C>

Bijvoorbeeld:

n:= n + 1; F[n]:= n + 1

betekent iets anders dan

n:= F[n]:= n + 1

Ook geldt niet altijd dat

<A> := <B>:= <C>

hetzelfde resultaat heeft als

<B> := <A> := <C>

Voorbeeld:

F[read]:= G[read]:= read

betekent iets anders dan

G[read]:= F[read]:= read

>02> Wat is het resultaat van deze twee dubbele assignment statements als in beide gevallen de invoergetallen zijn: 1, 2, 3?

De assignment statement kan, behalve voor het toekennen van een waarde aan een of meer variabelen, ook dienen voor het aangeven van de waarde die een type procedure zal afleveren. Een voorbeeld hiervan zien we in 2.2., voorbeeld 10. We noemen dit een assignment aan de procedure identifier (naam van de procedure, zie 7.)
De variabelen en procedure identifiers links van het meest rechtse := kunnen van deze typen zijn: *integer* (voor gehele getallen), *real* (voor reële getallen), *Boolean* (voor de logische waarden *true* en *false*).
De variabelen en procedure identifiers links moeten in elk geval van gelijk type zijn; de expressie rechts van het meest rechtse := teken moet in principe ook van ditzelfde type zijn, maar als het type links *real* is mag het rechts ook *integer* zijn, en andersom.

real b; integer a; a:= 3.14; b:= a

heeft tot gevolg dat *a* de waarde *3* krijgt (afronding naar de dichtstbijzijnde integer), en dat *b* vervolgens de (real) waarde *3* krijgt.
Fout is:

real b; integer a; a:= b:= 3.14

omdat *a* en *b* niet van gelijk type zijn.

Syntaxis

(25) <assignment statement>::= <left part list> <arithmetic expression> | (57)
(26) <left part list> <Boolean expression> (97)
(27) <left part list>::= <left part> |
(28) <left part list> <left part>
(29) <left part>::= <variable> := | (88)
(30) <procedure identifier> := (154)

Voorbeelden

(25) *A[k]:= n3:= if groter then a else z*

(26) *groter:= A[a] > A[z]*

(27) *n3:=*

(28) *A[L]:= n3:=*

(29) *n3:=*

(30) *KW:=*

4.2. Go to statement, label, switch (RR 4.3., 3.5)

Met behulp van een go to statement kunnen we ervoor zorgen dat de uitvoering van het programma bij een bepaalde statement verder gaat. Hiervoor moet deze statement voorafgegaan worden door een label en een : .
Voorbeeld van het gebruik van go to statements en labels:

```
        if g[i] < 1 ∨ g[i] > 3 then go to EIND;
        if g[i] = 2 then go to B else if g[i] = 3 then go to C;
A:      printtext (‡a‡); go to EIND;
B:      printtext (‡b‡); go to EIND;
C:      printtext (‡c‡); go to EIND;
EIND:
```

Dit stukje programma zorgt ervoor dat er een *a* wordt afgedrukt als *g[i] = 1*, een *b* als *g[i] = 2*, een *c* als *g[i] = 3*, en anders niets.
(De symbolen ‡ en ‡ zijn representaties van de string quotes ‘ en ’ uit het Revised Report; string quotes hebben de functie van aanhalingstekens.
∨ betekent of.
printtext wordt gebruikt voor het laten afdrukken van tekst, en behoort tot de output-mogelijkheden van het X8-ALGOL-systeem.)
Na *go to* hoeft niet altijd een label te staan, er mag ook een expressie staan die bij uitwerking een label oplevert: een designational expression.

Voorbeeld:

```
        go to if g[i] < 1 ∨ g[i] > 3 then EIND else
                if g[i] = 2 then B else if g[i] = 3 then C else A;
A:      printtext (‡a‡); go to EIND;
B:      printtext (‡b‡); go to EIND;
C:      printtext (‡c‡); go to EIND;
EIND:
```

De designational expression na *go to* in de eerste regel van dit voorbeeld, levert bij uitwerking een van de labels *EIND*, *A*, *B* en *C*, afhankelijk van de waarde van *g[i]*. Er is nog een soort designational expression: gebruik makend van een rij van labels kunnen we hieruit er een aanwijzen met behulp van een index (een subscript expression). Als we de rij labels *L[1]*, *L[2]*, *L[3]* hebben met waarden resp. *A*, *B* en *C*, dan betekent *go to L[2]* hetzelfde als *go to B*. Zo'n rij labels heet een switch, en we kunnen aangeven welke labels bij deze switch horen, door middel van een switch declaration.

Voorbeeld:

```
begin   switch L:= A, B, C;
        go to if g[i] < 1 ∨ g[i] > 3 then EIND else L[g[i]];
A:      printtext (‡a‡); go to EIND;
B:      printtext (‡b‡); go to EIND;
C:      printtext (‡c‡); go to EIND;
EIND:
end
```

De betekenis van deze switch declaration is: met *L[1]* zal de label *A* bedoeld worden, met *L[2]* de label *B* en met *L[3]* de label *C*; andere waarden van de subscript expression maken de designational expression zinloos, en in de meeste ALGOL-systemen fout. (Zie ook 6.4.)

Naast dezelfde soort namen als die we voor variabelen gebruiken, (zie 8.), kunnen we voor labels ook nog gehele getallen (zonder teken) gebruiken, b.v.: *7* of *007*. (Deze twee duiden dezelfde label aan.)

N.B. Er moet natuurlijk voor gezorgd worden dat de labels die in een designational expression gebruikt worden, ook geplaatst zijn in het programma, d.w.z. voorkomen met een : erachter. Ook moet de label "bereikbaar" zijn vanuit de plaats van de sprong (zie het begrip scope in 6.)

Syntaxis

(31)	<go to statement>::=	*go to* <designational expression>	
(32)	<designational expression>::=	<simple designational expression> |	
(33)		<if clause>	(48)
		<simple designational expression>	
		else <designational expression>	
(34)	<simple designational expression>::=	<label> |	
(35)		<switch designator> |	
(36)		(<designational expression>)	
(37)	<label>::=	<identifier> |	(193)
(38)		<unsigned integer>	(82)
(39)	<switch designator>::=	<switch identifier>	
		[<subscript expression>]	(96)
(40)	<switch identifier>::=	<identifier>	(193)

Voorbeelden

(31) *go to if a = 1 then (if b = 1 then A else B) else Q[a + b]*
(32) *Q[3]*
(33) *if a = 1 then L else if a = 2 then Q[3] else 007*
(34) *L*
(35) *Q[3]*
(36) *(if b = 1 then A else B)*
(37) *L*
(38) *07*
(39) *Q[if a = 1 then 2 else 3]*
(40) *Q*

4.3. Dummy statement (RR 4.4.)

Een dummy statement is een lege statement, zoals de statement in de compound

statement

begin end

of de statement na *EIND:* in

```
begin real w;
      if a > b then go to EIND;
      w:= a; a:= b; b:= w;
EIND:
end
```

of de statement die de tekst van deze procedure (procedure body) vormt:

```
procedure doe niets (i); value i; integer i;;
```

Dummy statements worden vaak gebruikt voor het plaatsen van een label (zoals de label *EIND* hierboven); ook een dummy statement als procedure body komt wel voor, b.v. in de testfase van een programma, wanneer de uitvoering van bepaalde procedures nog niet van belang is (in de definitieve versie van het programma kan dan de dummy statement vervangen worden door een procedure body die echt iets doet).

Soms maakt het gebruik van een dummy statement een stukje programma iets korter. In plaats van

```
if n ≥ 1 then
begin if t[n] ≥ 1 then
      begin if r[t[n]] = 0 then s[n]:= s[n] + 1 end
end
```

kan bijvoorbeeld geschreven worden:

```
if n < 1 then else
if t[n] < 1 then else
if r[t[n]] = 0 then s[n]:= s[n] + 1
```

Een ander gebruik van een dummy statement zien we in het volgende (vgl. 4.5.):

```
for i:= read while i ≠ 0 do ;
print (read)
```

Hier staat na do een dummy statement.
Het effekt van dit stukje programma is: de invoergetallen tot en met de eerst tegengekomen 0 worden overgeslagen, het eerstvolgende getal wordt afgedrukt.

Syntaxis

(41) <dummy statement>::= <empty>
(42) <empty>::=

Voorbeelden
(41)
(42)

4.4. Conditional statement (RR 4.5.)

Als we een bepaalde statement alleen willen laten uitvoeren als aan een zekere voorwaarde is voldaan, kan dat zo worden beschreven:

if g[i] > max *then* max:= g[i]

De statement na *then* wordt nu alleen uitgevoerd als g[i] > max.
Als we meer dan één statement zouden willen laten uitvoeren als g[i] > max, dan kan dat zo:

if g[i] > max *then begin* max:= g[i]; imax:= i *end*

Als we, afhankelijk van een zekere conditie, de ene of de andere statement willen laten uitvoeren, schrijven we:

if a > b *then* max:= a *else* max:= b

Als er meer alternatieven zijn:

if a > b *then* i:= 1 *else if* a < b *then* i:= -1 *else* i:= 0

We kunnen ons de uitvoering hiervan zo voorstellen:
Als a > b dan wordt i:= 1 uitgevoerd, anders de (conditional) statement na *else*; die conditional statement wordt op dezelfde manier uitgevoerd.
Anders gezegd komt een statement van de vorm:

if A *then* <statement 1> *else if* B *then* <statement 2> *else*
if C *then* <statement 3>

neer op: werk de Boolean expressions na *if* achtereenvolgens van links naar rechts uit, totdat er een waar blijkt (*true* oplevert), en voer dan de unconditional statement na het bijbehorende *then* uit.

Als de rij nog besloten wordt door:

... *else* <statement 4>

dan wordt deze <statement 4> uitgevoerd als geen enkele van de Boolean expressions waar blijkt.

Enige regels:

1. In een conditional statement met *else* mag na *then* niet een for statement volgen; fout is dus:

   if n ≥ 1 then for i:= 1 step 1 until n do print (n) else print (0)

   Wel correct is:

   if n ≥ 1 then
   begin for i:= 1 step 1 until n do print (n) end
   else print (0)

   of:

   if n < 1 then print (0) else
   for i:= 1 step 1 until n do print (n)

   Als tussen *then* en *else* wel een for statement zou zijn toegestaan (zoals dat volgens het Report on the Algorithmic Language ALGOL 60 nog het geval was), zou de volgende, nu foute, konstruktie:

   if n > 1 then
   for i:= 1 step 1 until n do if P[i] then print (i) else print (0)

   op twee manieren geïnterpreteerd kunnen worden, nl. als:

   if n ≥ 1 then
   begin for i:= 1 step 1 until n do if P[i] then print (i) end
   else print (0)

   of als:

   if n ≥ 1 then
   for i:= 1 step 1 until n do
   begin if P[i] then print (i) else print (0) end

   De ontwerpers stonden dan ook voor de keus ofwel *then for* ... *else* te verbieden, ofwel *for* ... *do if*. Besloten is deze laatste konstruktie wel toe te laten (zie 4.5.), en *then for* ... *else* te verbieden.

2. De statement na *then* mag nooit een conditional statement zijn.

   Fout is dus

if 1 ≤ *i* ∧ *i* ≤ *n* *then* *if* P[*i*] *then* print (G[*i*])

Wel korrekt is

if 1 ≤ *i* ∧ *i* ≤ *n* *then* *begin* *if* P[*i*] *then* print (G[*i*]) *end*

of

if *i* < 1 ∨ *i* > *n* *then* *else* *if* P[*i*] *then* print (G[*i*])

> 03 > Als na *then* wel een conditional statement zou zijn toegestaan, zouden dubbelzinnige konstrukties kunnen voorkomen. Geef daar een voorbeeld van.

Syntaxis

(43)	<conditional statement>::=	<if statement> |	
(44)		<if statement> *else* <statement> |	(13)
(45)		<if clause> <for statement> |	(49)
(46)		<label> : <conditional statement>	(37)
(47)	<if statement>::=	<if clause> <unconditional statement>	(16)
(48)	<if clause>::=	if <Boolean expression> *then*	(97)

Voorbeelden

(43) *if* 1 ≤ *i* ∧ *i* ≤ *n* *then* *begin* *t*:= *t* + 1; G[*i*]:= *t* *end*

(44) *if* *a* ≥ *b* *then* print (*a*) *else* print (*b*)

(45) *if* *n* ≥ 1 *then* *for* *i*:= 1 *step* 1 *until* *n* *do* print (G[*i*])

(46) *ab*: *printmax*: *if* *a* ≥ *b* *then* print (*a*) *else* print (*b*)

(47) *if* *b* > *a* *then*
begin *if* *a* > 0 *then* *begin* *real* *w*; *w*:= *a*; *a*:= *b*; *b*:= *w* *end* *end*

(48) *if* P[*i*] ∨ *i* < 1 *then*

4.5. For statement (RR 4.6.)

Als een stuk programma een aantal keren (nul of meer) moet worden uitgevoerd is in veel gevallen deze ALGOL-konstruktie nuttig:

<programma stuk 1>;
for *i*:= 1 *step* 1 *until* *n* *do*
begin <programma stuk 2> *end*;
<programma stuk 3>

De betekenis van de regel *for i:= 1 step 1 until n do* is:

als $n \geq 1$, doe dan programma stuk 2 n keer, achtereenvolgens met $i = 1, 2, \ldots n$.

Voorbeeld:

```
begin   integer aantal, i, som;
        aantal:= read; som:= 0;
        for i:= 1 step 1 until aantal do som:= som + read;
        print (som)
end
```

Als we meer dan één statement zouden willen laten herhalen, kunnen we deze tot een statement samenvoegen:

```
begin   integer aantal, i, som;
        aantal:= read; som:= 0;
        for i:= 1 step 1 until aantal do
        begin som:= som + read; print (som) end
end
```

Voor het laten afdrukken van de drievouden van +99 tot -99:

for drievoud:= 99 step -3 until -99 do print (drievoud)

of: *for drievoud:= 99 step -3 until -100 do print (drievoud)*

Het stuk *99 step -3 until -99* heet een for list element, de variabele *drievoud* wordt de controlled variable genoemd.

Er bestaat nog een type for list element, het while element:

i:= 1; for i:= i × 2 while i < 1000 do print (i)

Dit betekent: geef steeds *i* als waarde *i* × 2, en doe *print (i)*, zolang als *i* maar kleiner dan *1000* is.

Afgedrukt wordt dus: *2 4 8 16 32 64 128 256 512*

We mogen ook meer dan een for list element achter elkaar gebruiken:

for i:= 1 step 1 until 5, 10 step 5 until 25 do print (i)

Afgedrukt wordt: *1 2 3 4 5 10 15 20 25*

En ook: *for i:= 1 step 2 until 10, i × 3 while i < 100 do print (i)*

Afgedrukt wordt dan: *1 3 5 7 9 33 99*

De betekenis van een konstruktie met een aantal for list elements, waarbij *V* de controlled variable is, kan zo worden beschreven:

```
        while er zijn nog for list elements af te handelen do
A       begin  V:= eerste waarde van het for list element dat aan de beurt is;
B              while het element is nog niet afgehandeld do
               begin voer de statement na het "do" van de for statement uit;
C                    V:= volgende waarde van het for list element
               end
        end
```

N.B. Deze gang van zaken wordt natuurlijk verstoord als in de statement na *do* een go to statement wordt uitgevoerd die naar een label buiten de for statement leidt.

Voor de beide typen for list element:

(1) P *step* Q *until* R

(2) P *while* F

hebben de stukken algoritme A, B en C de volgende betekenis:

A 1: $V:=$ P

A 2: $V:=$ P

B 1: $(V - R) \times Q \leq 0$ (oftewel V is nog niet voorbij R)

B 2: F

C 1: $V:= V + Q$

C 2: $V:=$ P

Er bestaat nog een type for list element, nl. een arithmetic expression, zoals in:

for k:= a ↑ 3 + 2 × a ↑ 2 *do begin print (k); print (k × k) end*

of in:

for i:= 1, 2, 3 *do print (A[i])*

Zo'n for list element is onmiddellijk afgehandeld nadat de statement na *do* een maal is uitgevoerd.

N.B. We moeten er wel om denken dat in een konstruktie als

for i:= 1, 2, i + 1 while i < 2 do print (i)

er sprake is van drie gescheiden for list elements, zodat de voorwaarde $i < 2$ alleen van toepassing is op het laatste, en niet op de eerste twee for list elements. Er wordt dan ook afgedrukt: *1 2*

Twee belangrijke regels:

1. De waarde van de controlled variable bij verlating van de statement na *do* wordt door het Revised Report "undefined" genoemd als de for list elements alle zijn afgehandeld, d.w.z. dat deze variabele in de ene implementatie van ALGOL 60 een andere waarde zal hebben dan in een andere implementatie. Hoewel we dus misschien verwachten dat na uitvoering van:

   for i:= 1 step 1 until 5 do print (i)

   de controlled variabele i de waarde 6 zal hebben, kunnen we er toch niet van op aan dat dat in elk geval zo is. We zullen een "undefined" konstruktie zoals:

   for i:= read while i < 0 do; print (i)

   dan ook als fout beschouwen; wel korrekt is:

   for i:= read while i < 0 do getal:= i; print (getal)

   De enige omstandigheid waaronder de controlled variable toch een gedefinieerde waarde heeft nadat de for statement is uitgevoerd, is als de statement na *do* via een go to statement wordt verlaten.

Voorbeeld:

for i:= read while true do if i ≥ 0 then go to posfnd;
posfnd: print (i);

2. Een go to statement buiten de for statement die leidt naar een label binnen de for statement heeft een ongedefinieerde betekenis (vgl. 6.). Fout is dus:

   for k:= 1 step 1 until 100 do L: G[k]:= G[k] + 1;
   :
   k:= 17; go to L;

Wel korrekt:

```
        a:= 1;
L:      for k:= a step 1 until 100 do G[k]:= G[k] + 1;
:       :
:       :
        a:= 17; go to L;
```

De statement na *do* mag elk soort statement zijn, dus ook een block, een conditional statement, een go to statement, en ook een for statement.

Voorbeeld:

```
for x:= 1 step 1 until 8 do
for y:= 1 step 1 until 8 do veld[x, y]:= 0
```

>04 > Wat wordt door elk van de volgende for-statements afgedrukt?

a. *for i:= 1 step 1 until 3 do print (i)*
b. *for i:= 1 step 1 until 1 do print (i)*
c. *for i:= 1 step 1 until 0.5 do print (i)*
d. *for i:= 1 step -1 until 1 do print (i)*
e. *for i:= 1 step 2 until 4 do print (i)*
f. *for i:= 0 step 0 until 0 do print (i)*
g. *for i:= 1, i + 1 while i ≤ 2 do print (i)*
h. *for i:= 1, i + 1 while i ≤ 1 do print (i)*
i. *for i:= 1, i + 1 while i ≤ 0 do print (i)*
j. *for i:= 1, 2, 3, 4, 5 while i ≤ 2 do print (i)*
k. *for i:= 1 while true do print (i)*
l. *for i:= 1, i × 2 while i < 95 do begin i:= i + 1; print (i) end*
m. *for i:= 1 step 1 until 3 do*
   for j:= 1 step 1 until 3 do print (10 × i + j) end
n. *for i:= 1 step i until 20 do print (i)*
o. *for i:= 3 step i until 20 do begin i:= i - 1; print (i) end*
p. *for i:= 2 step i until 20 do begin i:= i - 1; print (i) end*
q. *for i:= 1 step i until 20 do begin i:= i - 1; print (i) end*
r. *for i:= 2, -i × i while i < 100 do print (i)*
s. *for i:= 1 step i until 10, i - 3 while i ≥ 1 do print (i)*

>05> Welke van de volgende stukken programma kunnen nooit korrekt ALGOL zijn?

a. *for i:= 1 while false do print (i);*

b. *for i:= 1, 2, 3 do if G[i] > 0 then print (i);*

c. *for i:= 1 step 1 while i < 100 do print (i);*

d. *for i ≥ 1 do print (i);*

e. *for i:= if i < 0 then −i else i step i until i × i do print (i);*

f. *for i:= if a > b then b else a step 1 until if a > b then a else b do;*

g. *i:= 1; for i:= i + 1 while i < 20 do print (i); if i = 21 then i:= 20;*

h.
```
for i:= 1, 2, 3 do
begin if i > 2 then go to P;
      for j:= 1, 2, 3 do P: print (10 × i + j)
end;
```

i.
```
for i:= 1, 2, 3 do
begin for j:= 1, 2, 3 do
      begin if j > 2 then go to P; print (10 × i + j) end;
P:
end;
```

j.
```
for i:= 1, 2, 3 do for j:= 1, 2, 3 do
begin if j > 2 then go to P; print (10 × i + j); P: end;
```

k. *for i:= 1, 2, 3 do for i:= 1, 2, 3 do print (11 × i)*

l. *for i:= 1 step 1 until 3 do for i:= 1, 2, 3 do print (11 × i);*

m. *for i = 1 step 2 until 3 do print (i);*

n. *for i:= 1 step 1 until 1 do print (i); print (i)*

o. *for i:= 1 do print (i); print (i);*

p.
```
for i:= 1 step 1 until 3 do
if i = 3 then go to L else print (i); L: print (i);
```

Syntaxis

(49)	<for statement>::=	<for clause> <statement> |	(13)
(50)		<label> : <for statement>	(37)
(51)	<for clause>::=	for <variable> := <for list> do	(88)
(52)	<for list>::=	<for list element> |	
(53)		<for list> , <for list element>	
(54)	<for list element>::=	<arithmetic expression> |	(57)

(55) <arithmetic expression> *step* (57)
<arithmetic expression> *until* (57)
<arithmetic expression> | (57)
(56) <arithmetic expression> *while* (57)
<Boolean expression> (97)

Voorbeelden

(49) *for i:= 2, i × i while i < 20, i step 1 until 300 do print (P[i])*
(50) *F: for i:= 1 while t < 3 do begin t:= t + 1; go to F end*
(51) *for P[3]:= n + 7 while f > k - 1, -3, 5 step 5 until 21 do*
(52) *i × i while i < 20*
(53) *i × i while i < 20, i + 10 step 10 until 100, 200, 300*
(54) *n ↑ 3 - 5 + N[k - 1]*
(55) *-b step 3 × a until 0*
(56) *i + 1 while ¬(i < -5 ∨ i > 5)*

>06> Geef een afleiding (zoals dat in 2.2 voor *begin end* is gedaan) van <program> naar

begin <declaration>;
for <variabele>:=<arithmetic expression>
while <Boolean expression> *do*
end

>07> Kan het volgende paar ALGOL symbolen in een <statement> opeenvolgend voorkomen? Zo ja, vul de rij dan aan tot een statement.

a. *begin end*
b. *begin for*
c. *for begin*
d. *end else*
e. *else for*
f. *then for*
g. *then if*
h. *if for*
i. *begin;*
j. *end;*
k. *then;*
l. *else;*
m. *step;*
n. *do;*
o. *if;*
p. *go to;*

5. Expressions

5.1. Arithmetic expression (RR 3.3.)

Als een arithmetic expression wordt uitgewerkt komt er een getal uit. Een simple arithmetic expression is opgebouwd uit getallen, variabelen en function designators (aanroepen van procedures zoals *sign(a)*), gecombineerd door de operatoren: + − × / ÷ ↑; met behulp van ronde haakjes kunnen operanden gegroepeerd worden.
Deze variabelen en function designators moeten van integer of real type zijn. Het resultaat van optelling, aftrekking of vermenigvuldiging van twee operanden van integer type is zelf ook weer van integer type, maar als een van beide (of beide) operatoren van real type is (zijn), dan is ook het resultaat van real type. Het resultaat van de operator / is altijd real.
De operator ÷ is alleen gedefinieerd voor operanden die beide van integer type zijn. Het resultaat is dan ook integer en wordt dan gedefinieerd door:

$a \div b = sign\ (a / b) \times entier\ (abs\ (a / b))$

$sign\ (a) = -1$ als $a < 0$, 0 als $a = 0$, en $+1$ als $a > 0$

$entier\ (a)$ = de grootste integer niet groter dan a

$abs\ (a) = -a$ als $a < 0$, anders a.

Met andere woorden $a \div b$ is gelijk aan a / b in de richting van nul afgerond, dus

$7 \div 2 = 3,\ -7 \div 2 = -3,\ -7 \div (-2) = 3.$

De operator ↑ betekent tot de macht.

$a \uparrow b$ is ongedefinieerd als $a < 0$ en b real (b.v. $-2 \uparrow 3.14$)
of $a = 0$ en $b \leq 0$ (b.v. $0 \uparrow -2$)
is integer als a en b integer en $b \geq 0$ en niet $a = b = 0$
is real in de overige gevallen.

Als de exponent geheel is wordt de waarde verkregen door herhaald vermenigvuldigen:

$7^3 = 7 \times 7 \times 7$

$7^{-3} = 1 / (7 \times 7 \times 7)$

anders door middel van de formule $exp\ (b \times ln\ (a))$

waarin $exp\ (a) = e^a$

$ln\ (a) = {}^e\log\ a$

In principe wordt een arithmetic expression van links naar rechts uitgewerkt, maar er wordt rekening gehouden met de prioriteit van de operatoren:

prioriteit 1 ↑
prioriteit 2 × / ÷
prioriteit 3 + −

Dus *3 × 4 ↑ 2* betekent *3 × 16* en niet *12 ↑ 2*.
Willen we dit laatste toch, dan schrijven we *(3 × 4) ↑ 12*.
Evenzo: *2 ↑ 3 ↑ 2* betekent *8 ↑ 2* en niet *2 ↑ 9*
Dit laatste kunnen we zo schrijven: *2 ↑ (3 ↑ 2)*.
N.B. Optellen gaat dus niet voor aftrekken, en vermenigvuldigen niet voor delen (net als in het dagelijks leven).

>08> Wat is het type en de waarde van de volgende simple arithmetic expressions:

a. *3 + 5.4*
b. *3 − 3*
c. *5.4 − 5.4*
d. *0 × 5.4*
e. *6 / 3*
f. *6 ÷ 3*
g. *5.4 ÷ 3*
h. *−5 ÷ 3*
i. *2 ↑ 0*
j. *0 ↑ 2*
k. *0 ↑ 0*
l. *2 ↑ (3.14 − 3.14)*
m. *2 ↑ (1 ↑ (−1))*
n. *2 + 2 − 2 + 2*
o. *8 / 2 × 2*
p. *8 / 2 / 2*

Wat het rekenen met waarden van real type betreft, moeten we er rekening mee houden dat dit met een beperkte precisie gebeurt.

Voorbeeld:

if 3.5 − 7 / 2 = 0 then printtext (‡raak‡) else printtext (‡mis‡)

De definitie van ALGOL garandeert niet dat in dit geval de statement na *then* wordt uitgevoerd. Hoe precies er wel gerekend wordt hangt af van de implementatie van ALGOL waarmee we werken.

Een arithmetic expression kan ook nog *if*, *then* en *else* bevatten:

if <Boolean expression> *then* <simple arithmetic expression>
else <arithmetic expression>

Hieruit komt: als de <Boolean expression> *true* oplevert, dan de simple

arithmetic expression na *then*, anders de arithmetic expression na *else* (die zelf weer *if*, *then* en *else* kan bevatten, en dan op dezelfde manier uitgewerkt moet worden). We zien dat zo'n arithmetic expression die niet een simple arithmetic expression is moet eindigen met:

<simple arithmetic expression> *else* <simple arithmetic expression>

Fout is dus:

max:= if g[i] > max then g[i]

Wel correct:

max:= if g[i] > max then g[i] else max

>09> a. Geef, met behulp van de syntaxis aan het eind van deze paragraaf, een afleiding van <arithmetic expression> naar:

if <Boolean expression> *then* <simple arithmetic expression>
else <simple arithmetic expression>

b. Werk nu uit deze arithmetic expression *else* weg met behulp van de regel die in RR 3.3.3 gegeven wordt, en leidt het resultaat nu af van <arithmetic expression>.

We zien ook dat na het *then* van een arithmetic expression een simple arithmetic expression moet volgen.

Fout is dus:

max:= if a > b then if a > c then a else c else if b > c then b else c

Wel correct:

max:= if a > b then (if a > c then a else c) else if b > c then b else c

want een arithmetic expression tussen haakjes is een simple arithmetic expression.

>10> Leidt (<arithmetic expression>) af van <simple arithmetic expression>.

Syntaxis

(57) <arithmetic expression>::= <simple arithmetic expression> |
(58) <if clause>
<simple arithmetic expression>
else <arithmetic expression>
(59) <simple arithmetic expression>::= <term> |
(60) <adding operator> <term> |
(61) <simple arithmetic expression>

(48)

(62) <term>::= <adding operator> <term>

(63) | <term> <multiplying operator> <factor>

(64) <adding operator>::= + |

(65) -

(66) <factor>::= <primary> |

(67) <factor> ↑ <primary>

(68) <multiplying operator>::= × |

(69) / |

(70) ÷

(71) <primary>::= <unsigned number> | (75)

(72) <variable> | (88)

(73) <function designator> | (192)

(74) (<arithmetic expression>)

Voorbeelden

(57) *f (a) × 3 ↑ A [if a < 4 then a else 4]*

(58) *if a ≤ 4 then f (a) × 3 ↑ A[a] else f (a) × 3 ↑ A[4]*

(59) *A[n] / 3 ↑ 3*

(60) *-A[n] / 3 ↑ 3*

(61) *-A[n] / 3 ↑ 3 + (e - 1) + (if a > 0 then 5 else A[1])*

(62) *a ↑ 4*

(63) *a ↑ 4 / b × (e - 1)*

(64) *+*

(65) *-*

(66) *A[f (3.14)]*

(67) *1.2 ↑ 2 ↑ A[f (3.14)]*

(68) *×*

(69) */*

(70) *÷*

(71) *3.14*

(72) *A[f (3.14)]*

(73) *f (3.14)*

(74) *(e - 1)*

5.2. Number (RR 2.5.)

Gehele getallen worden in ALGOL geschreven als een rij cijfers, eventueel voorafgegaan door een plus- of een min-teken.
Reals kunnen van twee soorten zijn:

1. nul of meer cijfers, eventueel gevolgd door een decimal point en een of meer cijfers, dit geheel eventueel voorafgegaan door een teken. Voorbeelden:

   +0.3 +.3 -1.05 .05 -.05 1

2. het symbool $_{10}$ gevolgd door een geheel getal, en eventueel voorafgegaan door een real van type 1, of door alleen een teken. Voorbeelden:

   +1.5$_{10}$8 +3$_{10}$-7 -$_{10}$+4 -0$_{10}$-0

   De waarde van b.v. *-.3$_{10}$-7* is *-.3 × 10 ↑ (-7)*.

   (N.B. Dit betekent niet dat we ons moeten voorstellen dat de waarde van het getal *-.3$_{10}$-7* wordt berekend met behulp van een vermenigvuldiging en een machtsverheffing.)

>11> Welke van de volgende rijen symbolen zijn numbers?

a. *.0*
b. *0.*
c. *0.0*
d. *-$_{10}$0*
e. *-$_{10}$1*
f. *-$_{10}$*
g. *1.5$_{10}$1.5*
h. *-3 × 10 ↑ (-7)*
i. *3$_{10}$-2$_{10}$1*
j. *$_{10}$(1+2)*
k. *+(+5)*
l. *++5*

>12> Welke van de volgende rijen symbolen zijn expressies?

a. *5 × $_{10}$5*
b. *$_{10}$ × $_{10}$5*
c. *a × 10 ↑ a*
d. *5 × $_{10}$a*
e. *+(+5)*
f. *-3 × 10 ↑ (-7)*
g. *5 × -2*
h. *5 ↑ -2*
i. *(1)*
j. *()*

Syntaxis

(75) <unsigned number>::= <decimal number> |
(76) <exponent part> |
(77) <decimal number> <exponent part>
(78) <decimal number>::= <unsigned integer> |
(79) <decimal fraction> |
(80) <unsigned integer> <decimal fraction>
(81) <exponent part>::= $_{10}$<integer>
(82) <unsigned integer>::= <digit> | (248)
(83) <unsigned integer> <digit> (248)
(84) <decimal fraction>::= .<unsigned integer>
(85) <integer>::= <unsigned integer> |
(86) + <unsigned integer> |
(87) - <unsigned integer>

Voorbeelden

(75) *15.3*
(76) $_{10}$*-51*
(77) *15.3*$_{10}$*-51*
(78) *15*
(79) *.3*
(80) *15.3*
(81) $_{10}$*-51*
(82) *5*
(83) *51*
(84) *.3*
(85) *51*
(86) *+51*
(87) *-51*

5.3. Variable (RR 3.1.)

Er zijn twee soorten variabelen in ALGOL:
1. simple variables, die van integer, real of Boolean type kunnen zijn;

2. subscripted variables, aangegeven door de naam van een array, gevolgd door, tussen vierkante haken, een of meer subscript expressions, onderling gescheiden door komma's. Ingewikkeld voorbeeld:

A[if a > 0 then a else -a, veld[5, i + 1], huis[straat, nummer, hoog]]

Bij het vaststellen welk element van een array wordt aangeduid door een bepaalde subscripted variable, wordt, zo moeten we ons voorstellen, de waarde van elk van de subscript expressions aan een variable van integer type toegekend (zie 4.1), d.w.z. met

veld[0.7, 6.3]

wordt aangeduid de variabele

veld[1, 6]

We noemen een array met twee subscripts een twee-dimensionale array, met drie subscripts een drie-dimensionale array, etc. Het aantal dimensies mag willekeurig hoog zijn.

Syntaxis

```
(88) <variable>::=                <simple variable> |
(89)                              <subscripted variable>
(90) <simple variable>::=         <variable identifier>
(91) <subscripted variable>::=    <array identifier> [ <subscript list> ]
(92) <variable identifier>::=     <identifier>                              (193)
(93) <array identifier>::=        <identifier>                              (193)
(94) <subscript list>::=          <subscript expression> |
(95)                              <subscript list> , <subscript expression>
(96) <subscript expression>::=    <arithmetic expression>                   (57)
```

Voorbeelden

(88) *max1*
(89) *huis[straat, nr + 1, -1]*
(90) *max1*
(91) *huis[straat, nr + 1, -1]*
(92) *max1*
(93) *huis*
(94) *nr + 1*
(95) *straat, nr + 1, -1*
(96) *nr + 1*

5.4. Boolean expression (RR 3.4.)

Als een Boolean expression wordt uitgewerkt komt er een logical value uit, d.w.z. een van de waarden *true* en *false*.
(George Boole was de grondlegger van de moderne logica.)
Er zijn twee soorten operatoren die een logical value opleveren:
relational operators (zoals ≥), waarvan de operanden simple arithmetic expressions zijn, en logical operators (zoals ∨), waarvan de operanden Boolean expressions zijn.

Betekenis van de relational operators (*a* en *b* zijn van real of integer type):

a < *b*: *true* als *a* kleiner dan *b*, anders *false*
a ≤ *b*: *true* als *a* kleiner dan *b* of *a* gelijk aan *b*, anders *false*
a = *b*: *true* als *a* gelijk aan *b*, anders *false*
a ≥ *b*: *true* als *a* groter dan *b* of *a* gelijk aan *b*, anders *false*
a > *b*: *true* als *a* groter dan *b*, anders *false*
a ≠ *b*: *true* als *a* niet gelijk aan *b*, anders *false*

Betekenis van de logical operators (*p* en *q* van type Boolean):

p	*false*	*false*	*true*	*true*	
q	*false*	*true*	*false*	*true*	
¬*p* :	*true*	*true*	*false*	*false*	(spreek uit: niet b, of not b)
p ∧ *q*:	*false*	*false*	*false*	*true*	(spreek uit: p en q)
p ∨ *q*:	*false*	*true*	*true*	*true*	(spreek uit: p of q)
p ⊃ *q*:	*true*	*true*	*false*	*true*	(spreek uit: p impliceert q)
p ≡ *q*:	*true*	*false*	*false*	*true*	(spreek uit: p identiek q)

Dus: ¬*p* is het logisch tegengestelde van *p*
p ∧ *q* is alleen *true* als zowel *p* als *q* *true* is
p ∨ *q* is *true* als minstens een van beide operanden *true* is
p ⊃ *q* is alleen *false* als *p* *true* is en *q* *false*
p ≡ *q* is *true* als *p* dezelfde logische waarde heeft als *q*

Net als arithmetic expressions worden Boolean expressions in principe van links naar rechts uitgewerkt, maar ook hier gelden prioriteitsregels:

prioriteit 1 de arithmetische operatoren (volgens hun eigen prioriteiten)
prioriteit 2 relational operators: $< \leq = \geq > \neq$
prioriteit 3 $\neg$
prioriteit 4 $\wedge$
prioriteit 5 $\vee$
prioriteit 6 $\supset$
prioriteit 7 $\equiv$

Als we een andere volgorde willen dan kunnen we dit met haakjes aangeven:

$p \vee q \wedge r$ betekent $p \vee (q \wedge r)$

Willen we het anders, dan schrijven we:

$(p \vee q) \wedge r$

Als we willen nagaan of a positief is maar niet tussen 20 en 30:

if $a > 0 \wedge (a < 20 \vee a > 30)$ *then* ...

of: *if* $a > 0 \wedge \neg(a \geq 20 \wedge a \leq 30)$ *then* ...

of: *if* $a > 0 \wedge a < 20 \vee a > 30$ *then* ...

of: *if* $a \leq 30 \supset a < 20 \wedge a > 0$ *then* ...

Als we de betekenis van een bepaalde Boolean expression willen vaststellen, is het vaak nuttig dat zo te doen: stel de waarde van de Boolean expression vast voor elke mogelijke combinatie van waarden van de afzonderlijke Boolean expressions. Voorbeeld:

$a \leq 30 \supset a < 20 \wedge a > 0$

Er zijn 4 mogelijke combinaties van waarden van de Boolean expressions

$a \leq 30$, $a < 20$ en $a > 0$

nl. die welke overeenkomen met waarden van a:

$a \leq 0$, $0 < a < 20$, $20 \leq a \leq 30$, $a > 30$

Als $a \leq 0$ (bv. -10) dan komt er uit de Boolean expression:

$a \leq 30 \supset a < 20 \wedge a > 0$

true $\supset$ *true* $\wedge$ *false*

true $\supset$ *false*

false

Als $0 < a < 20$ (bv. 10), dan krijgen we:

$a \leq 30 \supset a < 20 \wedge a > 0$

true $\supset$ *true* $\wedge$ *true*

true $\supset$ *true*

true

Als $20 \leq a < 30$ (bv. 25), dan:

$$a \leq 30 \supset a < 20 \wedge a > 0$$
$$\underline{true} \supset \underline{false} \wedge \underline{true}$$
$$\underline{true} \supset \underline{false}$$
$$\underline{false}$$

En als $a > 30$ (bv. 50):

$$a \leq 30 \supset a < 20 \wedge a > 0$$
$$\underline{false} \supset \underline{false} \wedge \underline{true}$$
$$\underline{false} \supset \underline{false}$$
$$\underline{true}$$

Uit deze analyse blijkt dat de Boolean expression

$$a \leq 30 \supset a < 20 \wedge a > 0$$

$\underline{true}$ oplevert als $0 < a < 20$ of $a > 30$

>13> Ga met behulp van deze methode de equivalentie na van:

a. $\neg a \vee b$ en $\neg(a \wedge \neg b)$ en $a \supset b$
b. $(a \wedge b) \wedge c$ en $a \wedge (b \wedge c)$
c. $a \wedge (b \vee c)$ en $a \wedge b \vee a \wedge c$

>14> Ga na dat uit de volgende Boolean expressions altijd $\underline{true}$ komt:

a. $\underline{false} \supset a$
b. $a \vee \underline{true}$
c. $a \supset \underline{true}$
d. $a \vee \neg a$
e. $\neg(a \wedge \neg a)$
f. $a \leq 0 \vee a > 0$
g. $a > 0 \supset a \geq 0$
h. $a \wedge b \vee a \wedge \neg b \vee \neg a \wedge b \vee \neg a \wedge \neg b$
i. $(a \supset b) \wedge (b \supset a) \equiv (a \equiv b)$
j. $(a \supset b) \wedge a \supset b$

>15> Als we de beschikking zouden hebben over maar één van de logische operatoren van ALGOL, welke zou dat dan moeten zijn, opdat we toch een Boolean expression kunnen opschrijven equivalent met:

$a > 0 \wedge (a < 20 \vee a > 30)$

Hoe ziet deze Boolean expression er dan uit?

>16 > Welke van de volgende rijen symbolen zijn af te leiden van <Boolean expression>:

a. *false*

b. ¬*true*

c. ¬¬*false*

d. *if true then false else true*

e. *if if true then false else true then false else true*

f. *(true)* ∨ *((false))*

g. *if false then if true then false else true else false*

h. *if true then false*

Syntaxis

(97) <Boolean expression>::= <simple Boolean> |

(98) <if clause> <simple Boolean> *else* <Boolean expression> (48)

(99) <simple Boolean>::= <implication> |

(100) <simple Boolean> ≡ <implication>

(101) <implication>::= <Boolean term> |

(102) <implication> ⊃ <Boolean term>

(103) <Boolean term>::= <Boolean factor> |

(104) <Boolean term> ∨ <Boolean factor>

(105) <Boolean factor>::= <Boolean secondary> |

(106) <Boolean factor> ∧ <Boolean secondary>

(107) <Boolean secondary>::= <Boolean primary> |

(108) ¬<Boolean primary>

(109) <Boolean primary>::= <logical value> |

(110) <variable> | (88)

(111) <function designator> | (192)

(112) <relation> |

(113) (<Boolean expression>)

(114) <logical value>::= *true* |

(115) *false*

(116) <relation>::= <simple arithmetic expression> (59)
<relational operator>
<simple arithmetic expression> (59)

(117) <relational operator>::= < |
(118) ≤ |
(119) = |
(120) ≥ |
(121) > |
(122) ≠

Voorbeelden

(97) ¬(*p* ⊃ *q*) ∧ *a* > *b* ∨ *p* ⊃ *p* ∨ *q* ≡ *verbonden (1, knoop[1])* ∨ *p* ≡ *true*
(98) *if a* > *b then p else if a* = *b then p* ⊃ *q else p* ≡ *q*
(99) *a* > *b*
(100) ¬(*p* ⊃ *q*) ∧ *a* > *b* ∨ *p* ⊃ *p* ∨ *q* ≡ *verbonden (1, knoop[1])* ∨ *p* ≡ *true*
(101) ¬(*p* ⊃ *q*) ∧ *a* > *b* ∨ *p*
(102) ¬(*p* ⊃ *q*) ∧ *a* > *b* ∨ *p* ⊃ *p* ∨ *q* ⊃ *true* ⊃ *verbonden (1, knoop[1])* ∨ *p*
(103) ¬(*p* ⊃ *q*) ∧ *a* > *b*
(104) ¬(*p* ⊃ *q*) ∧ *a* > *b* ∨ *p* ∨ *verbonden (1, knoop[1])* ∧ *true*
(105) ¬(*p* ⊃ *q*)
(106) ¬(*p* ⊃ *q*) ∧ *a* > *b* ∧ *p* ∧ *verbonden (1, knoop[1])* ∧ *true*
(107) *a* ≠ *b*
(108) ¬*a* ≠ *b*
(109) *true*
(110) *n*[3]
(111) *verbonden (knoop*[*i*]*, knoop*[*j*]*)*
(112) *a* ≠ *b*
(113) *(p* ∨ *q)*
(114) *true*
(115) *false*
(116) *A*[*if P*[*i*] *then i else* −*i*] > *f*
(117) <
(118) ≤
(119) =
(120) ≥
(121) >
(122) ≠

6. Declaration, scope (RR 5., 2.7, 4.1.3.)

De namen van de volgende soorten grootheden moeten als ze in een programma voorkomen, gedeclareerd zijn:

simple variables
arrays
procedures
switches

Deze declaratie is nodig opdat enige essentiële zaken bekend zijn. De declaraties staan aan het begin van het block waarbij zij horen.
Een declaratie is geldig voor het gehele block, met al zijn binnenblocks, met dien verstande, dat een declaratie van een bepaalde identifier in een binnenblock voorrang heeft boven de declaratie van een gelijkluidende identifier in een omvattend block. Verder mag in de aanhef van een block een identifier niet tweemaal gedeclareerd worden. Hierdoor geeft een declaratie uitsluitsel over het soort grootheid dat door een identifier wordt aangeduid.

Voorbeeld:

```
(1)     begin   integer i, a;
(2)             i:= 3; a:= 1;
(3)             begin   integer i;
(4)                     i:= a;
(5)                     print (i)
(6)             end;
(7)             print (i)
(8)     end
```

De identifier *i* die in regel 2 en regel 7 voorkomt duidt dezelfde grootheid aan als gedeclareerd in regel 1, terwijl de identifier *i* in regel 4 en 5 de grootheid aanduidt die in regel 3 gedeclareerd wordt.
De verzameling statements en expressions waarin een bepaalde identifier een en dezelfde grootheid aanduidt, noemen we de scope van die grootheid.

Van de grootheid *i* uit regel 1 is de scope: regel 1, 2, 7, 8
Van de grootheid *i* uit regel 3 is de scope: regel 3, 4, 5, 6
Van de grootheid *a* uit regel 1 is de scope: regel 1 t/m 8
Het is duidelijk dat de scope van een grootheid nooit kan reiken buiten het block waarbij hij hoort.

Er bestaat nog een vijfde soort grootheden: labels. Deze behoeven niet gedeclareerd te worden, maar moeten, als ze ergens in het programma voorkomen, geplaatst zijn, d.w.z. voorkomen gevolgd door een : en een statement. Zo'n plaatsing kan beschouwd worden als een declaratie, en deze declaratie moet dan geacht worden gedaan te zijn aan het begin van het kleinste block dat de gelabelde statement omvat. Dit houdt ook in dat in dat block geen declaratie mag voorkomen van een grootheid met een gelijkluidende identifier, en dat er ook geen gelijknamige labels bij hetzelfde block mogen horen. T.a.v. de scope betekent dit, dat nooit van buiten een block gesprongen kan worden naar een label binnen dat block.
Voorbeeld:

```
(1)     begin   integer i, a;
(2)             i:= 1;
(3)             begin   integer i;
(4)                     i:= 2;
(5)             a:      i:= i + 1;
(6)                     if i < 5 then go to a; print (i)
(7)             end;
(8)             print (i)
(9)     end
```

De identifier *a* in regel 5 duidt een label aan, en deze kan dus beschouwd worden als gedeclareerd in het block regel 3 t/m 7.
De scope van de label *a* uit regel 5 is: regel 3 t/m 7
De scope van de integer *a* uit regel 1 is: regel 1, 2, 8, 9
Nog een voorbeeld:

```
(1)     begin   integer i;
(2)             i:= 1;
(3)     a:      begin   integer a;
(4)                     i:= a:= i + 1;
(5)                     print (a)
(6)             end;
(7)             if i < 5 then go to a;
(8)             print (i)
(9)     end
```

De eerste *a* in regel 3 is een label, die beschouwd kan worden als gedeclareerd in de aanhef van het block regel 1 t/m 9, want dit is het kleinste block dat de gelabelde statement (nl. regel 3 t/m 6) omvat. Omdat in het binnenblock weer een declaratie van de identifier *a* voorkomt, is de scope van de label *a*: regel 1, 2, 7, 8, 9 en de scope van de integer *a*: regel 3, 4, 5, 6.

Een voorbeeld van een fout programma:

```
begin  integer i, a;
       i:= 1; a:= 2;
a:     begin  integer i;
              i:= a; print (i)
       end
end
```

In het buitenste block worden twee *a*'s gedeclareerd: een echt (als integer), een door als label voor een van de statements van dit buitenste block geplaatst te zijn.

We kunnen stellen dat een identifier moet voorkomen binnen de scope van precies één grootheid die onder dezelfde naam gedeclareerd is (of als label geplaatst). Welke grootheid dat in een bepaald geval is, kunnen we zo nagaan: kijk in het kleinste block dat de identifier omvat of de identifier daar gedeclareerd is (of als label geplaatst): zo ja, dan is de grootheid gevonden; zo nee, herhaal dan het proces voor het (volgende) omvattende block, net zo lang totdat er een block met een declaratie (of geplaatste label) met die naam gevonden is. Vinden we er twee in een block, of vinden we er nooit een, dat is het programma fout.

>17> Welke van de volgende programma's zijn korrekt, en wat drukken ze af?

a.
```
begin integer i, a;
      i:= 1; a:= 2;
      begin  integer i;
             a:= i:= 3;
             print (i)
      end;
      print (a)
end
```

b.
```
begin integer i;
      a:     i:= 1;
             begin  integer i;
                    i:= 2;
             a:     i:= i + 1;
                    if i = 3 then go to a
             end;
             if i = 4 then go to a; print (i)
end
```

c.
```
begin   integer i, a;
        i:= 1; a:= 2;
        begin   integer i;
        a:      i:= a;
                print (i)
        end;
        print (i)
end
```

d.
```
begin   integer a, b;
        a:= b:= 1;
        begin   real b;
                a:= b:= a + 1;
                print (a)
        end;
        a:= b:= a + 1;
        print (a)
end
```

e.
```
begin   integer i;
        i:= 1;
        begin   integer a;
                a:= i;
                print (a)
        end;
        a:= a + 1;
        print (a)
end
```

f.
```
begin   integer i;
        i:= 1;
        begin   integer a;
                a:= i;
        b:      a:= i:= a + 1;
                if a < 5 then go to b;
                print (a)
        end;
        if i = 5 then go to b;
        print (i)
end
```

De declaraties van een block mogen in willekeurige volgorde tussen *begin* en de eerste statement staan.

Voorbeeld:

```
begin   real r; integer i, a; real f;
        procedure dummy; ;
        Boolean array A, F[1 : 5], B[1 : 2, 1 : 10], C[1 : 2, 1 : 10];
        Boolean array D[1 : 5];
        switch S:= L1, L3, L2; Boolean b;
        :
```

Syntaxis

(123) <declaration>::= <type declaration> | (127)
(124) <array declaration> | (135)
(125) <switch declaration> | (146)
(126) <procedure declaration> (149)

Voorbeelden

```
(123) own Boolean first, av
(124) real array coord[1 : n, 1 : 3], x, y, z[1 : max (a, b)]
(125) switch S:= L, if a < 3 then T[a] else alarm
(126) Boolean procedure groter (a, b, n); value n; array a, b; integer n;
      begin   integer i;
              groter:= false;
              for i:= 1 step 1 until n do if a[i] > b[i] then go to nee;
      ja:     groter:= true;
      nee:
      end
```

6.1. *own* (RR 5.)

Wanneer een block een maal is uitgevoerd en verlaten, en het wordt dan, b.v. via een go to statement, opnieuw binnengegaan, dan hebben de variabelen die in dat block gedeclareerd worden, een ongedefinieerde waarde, totdat ze opnieuw een waarde krijgen.
Voorbeeld:

```
(1)      begin   integer i;
(2)              i:= 3;
(3)      b:      begin   integer a;
(4)                      if i = 3 then a:= 4;
(5)                      a:= a + 1; print (a); i:= a
(6)              end;
(7)              if i = 5 then go to b
(8)      end
```

Als regel 7 bereikt wordt heeft i de waarde 5, dus wordt er teruggesprongen naar label b in regel 3; de assignment in regel 4 wordt dan niet uitgevoerd, omdat $i \neq 3$. Als dan regel 5 aan de beurt komt, wordt gebruik gemaakt van de waarde van a, die echter, sinds het block waarin a gedeclareerd wordt voor de laatste keer werd binnengegaan, geen waarde heeft gekregen. De waarde van a is dus ongedefinieerd.

Als we willen dat bepaalde variabelen in een block bij opnieuw binnengaan van dat block dezelfde waarde hebben als ze hadden toen het block de laatste keer werd verlaten, dan kunnen we dat aangeven door voor de declaratie van deze variabelen *own* te schrijven.
Voorbeeld:

```
begin   integer i, a;
        i:= 3; a:= 2;
b:      begin   own integer a;
                if i = 3 then a:= 4;
                a:= a + 1; print (a); i:= a
        end;
        a:= a + 1;
        if a < 5 then go to b;
        print (i)
end
```

>18> Wat wordt door dit programma afgedrukt?

>>>> Kan een block ooit anders dan via het eerste *begin* worden binnengegaan?

Hoewel ook own arrays mogelijk zijn, zullen we de precieze betekenis daarvan hier niet behandelen; in zeer veel implementaties van ALGOL is een nuttig gebruik van own arrays, n.l. met grenzen die bij binnenkomst in een block niet steeds dezelfde zijn, verboden.

6.2. Declaratie van simple variables (RR 5.1.)

Integer gedeclareerde variabelen kunnen alleen gehele getallen als waarde aannemen, Boolean variabelen alleen *true* en *false*, en real variabelen alleen getallen.

Syntaxis

(127) <type declaration>::= <local or own type> <type list>
(128) <local or own type>::= <type> |
(129) *own* <type>

(130) <type list>::= <simple variable> | (90)
(131) <simple variable> <type list> (90)
(132) <type>::= *real* |
(133) *integer* |
(134) *Boolean*

Voorbeelden

(127) *Boolean first, av*
(128) *real*
(129) *own integer*
(130) *first*
(131) *first, av*
(132) *real*
(133) *integer*
(134) *Boolean*

6.3. Array declaration (RR 5.2.)

In een array declaration wordt niet alleen het type van een rij variabelen genoemd(*integer*, *real* of *Boolean*), ook worden de grenzen genoemd waarbinnen de index (of indices) moet(en) liggen. Deze grenzen behoeven niet in de vorm van getallen te worden opgegeven, het mogen allerlei arithmetic expressions zijn:

```
begin   integer n;
        n:= read;
        begin   Boolean array check[1 : 3, 1 : n - 1];
                :
        end
end
```

We moeten er wel voor zorgen dat de grootheden die in deze arithmetic expressions optreden in het omvattende block bekend zijn. Voor variabelen die in deze grenzen optreden is dit duidelijk: ze zouden geen waarde kunnen hebben (tenzij het om own variabelen gaat); maar ook funktie-procedures die in de grenzen voorkomen, mogen niet pas in datzelfde block gedeclareerd worden (zie 7).

Fout is dus:

```
begin   integer procedure f (n); value n; integer n; f:= n ↑ 2;
        real array s[1 : f (4)];
        .
        .
        .
end
```

Wel korrekt:

```
begin   integer procedure f (n); value n; integer n; f:= n ↑ 2;
        begin   real array s[1 : f (4)];
                .
                .
                .
        end
end
```

Dit houdt in dat in het buitenste block van het programma alleen getallen in de grenzen kunnen voorkomen (en nog zgn. standard functions, zoals *abs* en *sign*, die niet gedeclareerd behoeven te worden; maar RR 5.2.4.2. lijkt dat te verbieden; fout is dan: *begin integer array s[1: read]* als begin van een programma).

Enige regels:

1. **De bovengrens moet groter of gelijk zijn aan de bijbehorende ondergrens.**
   Fout is dus: *integer array f[1 : 3, 3 : 1]*
   Wel korrekt: *integer array g[1 : 3, 1 : 1]*
2. **Wat betreft grenzen die niet integer zijn, geldt dat ze, net als subscript expressions, a.h.w. worden toegekend aan integer variabelen, zodat ze worden afgerond naar het dichtstbijzijnde gehele getal (d.w.z. als *r* een real grens is, dan komt dit neer op *entier (r +.5)*).**
   Voorbeeld: *real array f[-1.3 : 2.7]*
   betekent hetzelfde als *real array f[-1 : 3]*
3. **Elke keer dat het block wordt binnengegaan, worden de grenzen eenmaal berekend.**
   Voorbeeld:

```
for i:= 1 step 1 until 100 do
begin   real array getal[1 : read, 1 : i];
        destasineer (getal, read, i - 1)
end
```

Hier wordt 200 keer een getal gelzen (en wellicht nog vaker, als in de procedure *destasineer* nog getallen worden gelzen). Ook is de tweede bovengrens van de array getal steeds 1 hoger dan de vorige keer.

4. Het RR geeft geen uitsluitsel over de volgorde waarin de grenzen worden berekend.
   Het effekt van:

   begin integer array g[1 : read, 1 : read]

   is dan ook niet gedefinieerd.
5. Als in een array declaration niet *integer*, *real* of *Boolean* voor *array* staat dan betekent dit hetzelfde als wanneer er *real* voor zou staan.
   Dus: *begin array p[1 : 10]*
   betekent hetzelfde als: *begin real array p[1 : 10]*

Syntaxis

(135) <array declaration>::= *array* <array list> |
(136) <local or own type> *array* <array list> (128)
(137) <array list>::= <array segment> |
(138) <array list> , <array segment>
(139) <array segment>::= <array identifier> [<bound pair list>] | (93)
(140) <array identifier> , <array segment> (93)
(141) <bound pair list>::= <bound pair> |
(142) <bound pair list> , <bound pair>
(143) <bound pair>::= <lower bound> : <upper bound>
(144) <lower bound>::= <arithmetic expression> (57)
(145) <upper bound>::= <arithmetic expression> (57)

Voorbeelden

(135) *array p[1 : 10]*
(136) *Boolean array verbonden [1 : aantal knopen, 1 : aantal knopen]*
(137) *Andriessen [1 : 128]*
(138) *Andriessen [1 : 128], Struycken[1 : 50, 1 : 50],*
Dekkers[1 : 10, 1 : 10, 0 : 4]
(139) *Andriessen[1 : 128]*
(140) *Barbaud, Xenakis, Zinovieff[1 : 1000]*

```
(141) 1 : 10
(142) 1 : 10, 1 : 10, 0 : 4
(143) 1 : 10
(144) 0
(145) if n > 100 then 100 else n
```

6.4. Switch declaration (RR 5.3.)

Door de switch declaration

```
switch seizoen:= lente, zomer, herfst, winter
```

worden aan de rij labels *lente*, *zomer*, *herfst*, *winter* rangnummers gegeven, zodat b.v.:

```
go to seizoen[3]
```

betekent:

```
go to herfst
```

De elementen van een switch behoeven niet labels te zijn, het mogen allerlei designational expressions zijn:

```
switch integer:= integer[s], teken[s],
                 if s = 5 then fractie[s] else fout,
                 tien[s], integer[s], einde
```

We zien dat ook elementen van dezelfde switch in de designational expressions mogen voorkomen. We moeten ons niet voorstellen dat deze switch declaration zo wordt uitgevoerd, dat eens en voor al wordt bepaald welke labels er uit de designational expressions komen (dat dus de waarde van *s* wordt bepaald, b.v. *6*, dus *integer*[*1*] = *integer*[*6*] = *einde*, etc.).
De switch declaration betekent wel: als *go to integer*[*1*] voorkomt, dan wordt de waarde van *s* bepaald en als deze dan *6* blijkt, moet naar *einde* gesprongen worden, etc.

Het kan voorkomen dat in een switch declaration grootheden voorkomen, waarvan de scope zich niet uitstrekt over de plaats waar de switch gebruikt wordt, zoals de integer variabele *i* in de switch declaration van dit programma:

```
(1)      begin integer i;
(2)  A:      i:= 2;
(3)          begin   switch S:= if i = 2 then A else B,
                                if i = 2 then B else A;
(4)                  go to S[i];
(5)          B:      begin   integer i;
(6)                          i:= 1;
(7)                          go to S[i]
(8)                  end;
(9)          A:
(10)         end
(11) end
```

Als we willen vaststellen welke grootheid wordt aangeduid door de identifier *i* in regel 7, kijken we naar het kleinste omvattende block, nl. regel 5 t/m 8, en zien dat daar een grootheid met die naam gedeclareerd wordt: die is het dus. Deze grootheid heeft in regel 6 de waarde *1* gekregen, dus *go to S[i]* in regel 7 komt neer op *go to S[1]*, oftewel

go to if i = 2 then A else B

Maar welke *i* is hier nu bedoeld? Om dat vast te stellen kijken we naar het kleinste omvattende block, nl. regel 3 t/m 10, daar is geen *i* gedeclareerd, dus kijken we naar het block dat dit block weer omvat, nl. regel 1 t/m 11, en daar is inderdaad een grootheid *i* gedeclareerd: die is het dus. Deze integer *i* heeft als scope regel 1 t/m 4 en regel 9 t/m 11. Deze *i* heeft in regel 2 de waarde *2* gekregen, dus de go to statement die we aan het bekijken zijn, komt neer op:

go to if 2 = 2 then A else B

d.w.z. *go to A*

>19> Welke label *A* wordt nu bedoeld?

We hebben dus gezien dat de vraag welke grootheid een identifier in een switch declaration aanduidt, beantwoord kan worden zonder dat we daartoe een stuk van het programma behoeven uit te voeren: we kunnen het zien aan de plaats waar de switch declaration staat.

Om de waarde vast te stellen die de grootheid heeft op het moment dat de switch in een go to statement wordt gebruikt, moeten we vaak wel een stuk programma uitvoeren: het gaat om de waarde die de grootheid op dat moment (waarop de go to statement moet worden uitgevoerd) heeft, niet om de waarde die hij heeft op het "moment" van de declaration. We zien dat dit verschilt van de gang van zaken bij array declarations, waar de waarde van boven- en ondergrenzen wel wordt vastgesteld op het moment van declaratie; daar moeten de grootheden die in deze grenzen voorkomen dan ook een waarde hebben, en mogen dus niet pas in hetzelfde block gedeclareerd worden (zie 2.12.3.).
(Dit verschil tussen de behandeling van switch declarations en array declarations vindt zijn verklaring in het feit dat de grenzen in een array declaration ook echt op het ogenblik van declaratie berekend moeten kunnen worden i.v.m. reservering van de vereiste geheugenruimte, terwijl van een switch declaration alleen iets wordt uitgewerkt zodra er naar een element van de switch wordt gesprongen; het aantal elementen kan onmiddellijk aan de switch declaration worden afgelezen.)

Enige opmerkingen:

1. In de praktijk worden switches bijna uitsluitend gebruikt om afhankelijk van een integer waarde een bepaald tekstje te laten afdrukken, zoals in het voorbeeld in 2.4.
2. Als de subscript expression van een switch declaration niet van type integer is, dan wordt deze expressie (net als bij subscripts en grenzen van een array) a.h.w. aan een integer variabele toegekend, zodat afronding naar het dichtstbijzijnde gehele getal plaats vindt.
3. De integer waarde die uit een subscript expression komt, moet een van de getallen 1 t/m n zijn, waarin n het aantal elementen in de switch declaration is.
   Als dit niet het geval is, moet volgens RR 4.3.5. de gehele go to statement equivalent zijn met een dummy statement. Het volgende voorbeeld maakt duidelijk dat waarschijnlijk geen enkele implementatie hiervoor zorgt:

```
begin switch S := A, B;
   A: go to S[read];
   B:
end
```

Het is duidelijk dat een getal al gelezen is als wordt opgemerkt dat het verwijst naar een te hoog of te laag switch element, en daarmee is het effekt minstens al dat er een getal is gelezen.
We zullen een sprong naar een te hoog of te laag switch element dan ook als fout beschouwen.

Syntaxis

(146) <switch declaration>::= *switch* <switch identifier> := <switch list> (40)

(147) <switch list>::= <designational expression> | (32)

(148) <switch list> , <designational expression> (32)

Voorbeelden

(146) *switch kleinzoon:= Beatrix[i], Irene[i], Margriet[i]*

(147) *Willem Alexander*

(148) *Willem Alexander, Johan Friso, Constantijn*

Opgaven voor programma's zonder procedures.

Behalve procedures zijn nu alle mogelijkheden van ALGOL 60 behandeld.
Schrijf programma's voor de volgende opgaven. Gebruik
voor input van getallen: *a:= read;*
voor afdrukken van getallen: *print (a);*
voor een nieuwe regel (new line, carriage return): *nlcr;*
voor afdrukken van tekst: *printtext (‡resultaat:‡);*
Neem aan dat er 140 symbolen op een regel passen, en dat elk getal 21 posities beslaat.

>20> Een perfekt getal is een positief geheel getal dat gelijk is aan de som van zijn delers (inclusief 1, exclusief het getal zelf), b.v.
28 = 1 + 2 + 4 + 7 + 14.
Bepaal de perfekte getallen tot en met 1000.

>21> Ontbind een serie invoergetallen in factoren.

>22> Geef alle manieren om een geheel positief invoergetal te schrijven als som van de kwadraten van twee niet-negatieve gehele getallen.

>23> Gegeven zijn de koördinaten die de stand van een zwarte koning, een witte koning en een witte toren op een schaakbord bepalen. Ga na of de stand reglementair is, en of de zwarte koning schaak staat.

>24> Gegeven is een serie getallen a_i, voorafgegaan door hun aantal n.
Het gemiddelde m is gedefinieerd door:

$$m = \frac{1}{n}\sum a_i$$

en de spreiding s door:

$$s^2 = \frac{1}{n}\sum (a_i - m)^2$$

Bereken gemiddelde en spreiding van de serie getallen.

7. Procedures (RR 3.2., 4.7., 5.4.)

Procedures stellen ons in staat ingewikkelde handelingen, die we eenmaal voor een algemeen geval hebben beschreven, door middel van een korte uitdrukking voor een speciaal geval te laten uitvoeren. Aan het gebruik van procedures in een programma kunnen twee kanten worden onderscheiden:

1. De beschrijving van de handeling die door een procedure identifier wordt aangeduid (procedure declaration); deze beschrijving geschiedt in termen van formele parameters.
2. Het toepassen van de procedure:
   voor functie procedures door een aanroep in een expressie;
   voor procedures door een aanroep als procedure statement.
   Bij dit gebruiken van de procedures worden actuele parameters opgegeven die óf voor de formele parameters moeten worden gesubstitueerd (call by name), óf waarvan de waarde aan de formele parameters moet worden toegekend (call by value).

Voorbeeld van call by value

```
procedure druk kwadraat (n); value n; real n;
print (n × n)
```

Deze procedure kan geactiveerd worden door een aanroep als:

```
druk kwadraat (13)
```

We moeten ons voorstellen dat deze aanroep zo wordt uitgevoerd:

```
n:= 13;
print (n × n)
```

en niet zo:

```
print (13 × 13)
```

Dit onderscheid kan van groot belang zijn. Als de aanroep luidt:

```
druk kwadraat (read)
```

dan is het belangrijk te weten dat dit betekent:

```
n:= read; print (n × n)
```

en niet:

```
print (read × read)
```

Ook in andere gevallen is het onderscheid van belang:

druk kwadraat (f ↑ (s ÷ 2) × sin (t))

betekent:

n:= f ↑ (s ÷ 2) × sin (t);
print (n × n)

en niet:

print (f ↑ (s ÷ 2) × sin (t) × f ↑ (s ÷ 2) × sin (t))

Beide interpretaties leveren weliswaar hetzelfde resultaat, maar de tweede is veel minder efficiënt dan de eerste.

We moeten ons dus voorstellen dat *value* <parameter 1>, <parameter 2> betekent: ken de waarde van de overeenkomstige actuele parameters toe aan de formele parameters.

Voorbeeld van call by name

In gevallen waar we de toekenning van de waarde van de actuele parameter aan de formele parameter niet wensen, laten we de betreffende formele parameter weg in de value list.

Voorbeeld:

procedure wissel (a, b); integer a, b;
begin integer w; w:= a; a:= b; b:= w end

De aanroep:

wissel (p, q)

betekent nu:

begin integer w; w:= p; p:= q; q:= w end

en niet:

a:= p; b:= q;
begin integer w; w:= a; a:= b; b:= w end

Deze laatste interpretatie zou ook niet het gewenste effekt hebben.

N.B. De procedure declaration beschrijft wel een bepaalde handeling, maar dat wil niet zeggen dat deze handeling ook wordt uitgevoerd. Alleen een aanroep van de procedure geeft het sein tot uitvoering van de procedure.

7.1. Procedure declaration

Een procedure declaration stelt niet alleen vast dat een bepaalde identifier naar een procedure verwijst, maar geeft ook de precieze tekst van die procedure.

Voorbeeld:

```
procedure maak nul (A, a, z); value a, z; integer a, z; array A;
begin   integer i;
        for i:= a step 1 until z do A[i]:= 0
end
```

Na *procedure* komt eerst de heading van de procedure:

1. de procedure identifier
2. tussen haakjes de formele parameters, van elkaar gescheiden door een komma
3. een ;
4. value part: *value* gevolgd door een of meer parameters, van elkaar gescheiden door een komma en besloten met een ;
5. specification part (waarin informatie wordt gegeven over het type van een aantal formele parameters) afgesloten door een ;

(Als er geen value parameters zijn, ontbreekt het onder 4. genoemde, en als er geen specificaties gegeven worden, ontbreekt het onder 5. genoemde. Als er helemaal geen parameters zijn ontbreekt ook het onder 2. genoemde.)
Na de procedure heading volgt een statement, de procedure body, die de handeling beschrijft.

We laten de verschillende onderdelen van een procedure declaration nog eens de revue passeren.

- Een function procedure begint met *integer procedure*, *real procedure* of *Boolean procedure*, afhankelijk van het type waarde dat er uit moet komen. Andere procedures beginnen met *procedure* .
- De formele parameters zijn identifiers, dus nooit bv. *A[1 : 10]* of *A[i]* of *f (a)*, wel *A* of *f*.
- In plaats van een komma tussen twee formele parameters, kan ook een "vette komma" geschreven worden: *)* <tekstje> *:(*

  Voorbeeld:

```
sommeer array (A) van: (lower) tot en met: (upper) resultaat: (som);
```

- In de value part worden de value-parameters opgesomd (de daarin niet genoemde formele parameters zijn name-parameters).
- In de specification part kunnen de volgende specifiers gebruikt worden voor formele parameters die staan voor de volgende soorten actuele parameters:

actual parameter	formal parameter		
	specifier		mag
	partiële	volledige specificatie	value
real expression		*real*, *integer*	ja
integer expression		*real*, *integer*	ja
Boolean expression		*Boolean*	ja
string		*string*	nee
designational expression		*label*	ja
switch identifier		*switch*	nee
real array identifier	*array*	(*real*) *array*, *integer array*	ja
integer array identifier	*array*	(*real*) *array*, *integer array*	ja
Boolean array identifier	*array*	*Boolean array*	ja
real procedure identifier	*procedure*	*real procedure*, *integer procedure*	nee
integer procedure identifier	*procedure*	*real procedure*, *integer procedure*	nee
Boolean procedure identifier	*procedure*	*Boolean procedure*	nee
niet-functie-procedure		*procedure*	nee

Van alle parameters die in de value part genoemd worden moet een volledige specificatie gegeven worden, van de name-parameters mag volstaan worden met partiële specificatie, en mag deze zelfs achterwege blijven.

- De procedure body zal in het algemeen de vorm van een block hebben, maar het volgende kan ook:

```
Boolean procedure triangle (a, b, c); value a, b, c; real a, b, c;
triangle:= a + b > c ∧ b + c > a ∧ c + a > b
```

of

```
procedure kwnrm (...) ...
if ... then ... else
begin   integer ...
          :
end
```

- In de body van een functie-procedure moet ervoor gezorgd worden dat tenminste een maal een assignment aan de procedure identifier plaats vindt, opdat inderdaad een bepaalde waarde wordt afgeleverd; links van het := teken staat dan alleen de procedure identifier.

Voorbeeld:

```
Boolean procedure in orde (array, lower, upper, i, expr);
                           value lower, upper, i, expr;
                           integer array array;
                           integer lower, upper, i, expr;
begin   in orde:= false;
        if lower ≤ i ∧ i ≤ upper then
        begin if array[i] = expr then in orde:= true end
```

>25> Wat is, in gewoon Nederlands gezegd, het resultaat van een aanroep van procedure *in orde* ?

Een assignment aan de procedure identifier moet niet verward worden met een assignment aan een variabele, waarvan de waarde verderop in de procedure body weer gebruikt kan worden.

Fout is dus:

```
real procedure som (array, lower, upper); value lower, upper;
                   integer lower, upper; real array array;
begin   integer i;
        som:= 0;
        for i:= lower step 1 until upper do som:= som + array[i]
end
```

Het lijkt hier de bedoeling dat in de statement *som:= som + array[i]* steeds een waarde bij de oude waarde van *som* wordt opgeteld. Aangezien *som* geen variabele is, werkt dit niet. Elk voorkomen van de procedure identifier anders dan onmiddellijk links van een := teken, duidt op een (recursieve) aanroep van de procedure (in dit geval een foute aanroep, daar de actuele parameters ontbreken).

>26> Corrigeer de real procedure *som*.

Een functie-procedure hoeft niet als enig resultaat te hebben dat een waarde wordt afgeleverd.

Voorbeeld:

```
real procedure groot (a, b); real a, b;
begin   if b > a then
        begin   real w;
                w:= a; a:= b; b:= w
        end
        groot:= a
end
```

Het gevolg van een aanroep van deze procedure is niet alleen dat *groot* als waarde de grootste van de waarden van de twee actuele parameters krijgt, maar ook dat die twee waarden eventueel worden omgewisseld.

>27> Zoek de fouten in de procedure declarations in het volgende programma:

```
(1)  begin   integer i; real r;
(2)          integer procedure f (i); value i;
(3)          f:= if i = 0 then 1 else f (i - 1) × i;
(4)          procedure correct; if i > 0 then i:= i + 1;
(5)          Boolean array procedure negatie (A, n); value n; Boolean array A;
(6)          begin   integer i;
(7)                  for i:= 1 step 1 until n do negatie[i]:=¬A[i]
(8)          end;
(9)          real procedure kwadraat (a); real a; value a; kwadraat:= a × a;
(10)         real procedure som (augendum) plus (addendum);
(11)                         value addendum; real augendum ) plus : (addendum;
(12)         begin   real optelling;
(13)                 optelling:= augendum + addendum
(14)         end;
(15)         procedure een (A); real array A[1 : 100];
(16)         begin   integer i;
(17)                 for i:= 1 step 1 until 100 do A[i]:= 1
(18)         end
```

Syntaxis

(149) <procedure declaration>::= *procedure* <procedure heading>
<procedure body> |
(150) <type> *procedure* <procedure heading>(132)
<procedure body>
(151) <procedure heading>::= <procedure identifier>
<formal parameter part> ;
<value part> <specification part>
(152) <procedure body>::= <statement> | (13)
(153) <code>
(154) <procedure identifier>::= <identifier> (193)
(155) <formal parameter part>::= <empty> | (42)
(156) (<formal parameter list>)
(157) <value part>::= *value* <identifier list> ; |
(158) <empty> (42)
(159) <specification part>::= <empty> | (42)
(160) <specifier> <identifier list> ; |
(161) <specification part> <specifier>
<identifier list> ;
(162) <formal parameter list>::= <formal parameter> |
(163) <formal parameter list>
<parameter delimiter> <formal parameter>
(164) <identifier list>::= <identifier> | (193)
(165) <identifier list> , <identifier> (193)
(166) <specifier>::= *string* |
(167) <type> | (132)
(168) *array* |
(169) <type> *array* | (132)
(170) *label* |
(171) *switch* |
(172) *procedure* |
(173) <type> *procedure*
(174) <formal parameter>::= <identifier> (193)
(175) <parameter delimeter>::= , |
(176)) <letter string> : (
(177) <letter string>::= <letter> | (196)
(178) <letter string> <letter> (196)

Voorbeelden

```
(149) procedure check triangle (a, b, c, correct); value a, b, c;
                                real a, b, c; Boolean correct;
      correct:= a + b > c ∧ b + c > a ∧ c + a > b
(150) Boolean procedure triangle (a, b, c); value a, b, c; real a, b, c;
      triangle:= a + b > c ∧ b + c > a ∧ c + a > b
(151) som (i, a, z, f); value a, z; integer i, a, z; real f;
(152) begin   real s; s:= 0;
              for i:= a step 1 until z do s:= s + f;
              som:= s
      end
(153) aBrAcAdAbRa
(154) som
(156) (array, lower, upper) resultaat: (som)
(157) value lower, upper;
(158)
(159)
(160) label alarm, einde;
(161) label alarm, einde; string diagnose; integer i;
(162) lower
(163) lower, upper) resultaat: (som
(164) alarm
(165) alarm, einde
(166) string
(167) Boolean
(168) array
(169) real array
(170) label
(171) switch
(172) procedure
(173) integer procedure
(174) lower
(175) ,
(176) ) resultaat: (
(177) t
(178) ultaat
```

7.2. Procedure statement

Een procedure statement is een statement die bestaat uit alleen een aanroep van een procedure, b.v.

maak nul (frekwentie, 0, 9)

Ook een aanroep van een function procedure kan als statement fungeren; de real procedure *groot* uit 7.1. kunnen we zo aanroepen:

groot (m, n)

met als gevolg dat de waarden van *m* en *n* op volgorde worden gezet.
De waarde van *groot* wordt dan "weggegooid".

>28> Wat is het resultaat van:

print (groot (p,q) + groot (q,r))

en van:

groot (p,q); groot (q,r); print (p + q)

Een procedure statement moet corresponderen met een procedure declaration:

- de naam moet hetzelfde zijn
- het aantal parameters moet gelijk zijn
- de soort van elke parameter moet in overeenstemming zijn met de specificaties (zie tabel in 7.1.)
- de vette komma's hoeven niet gelijk te zijn: als in de procedure declaration staat: *som (array) van: (lower) tot: (upper)*
  mogen we in de procedure statement

i.p.v.	*som (A) van: (1) tot: (n)*
best schrijven:	*som (A, 1, n)*
of zelfs	*som (A) tot: (1) van: (n)*

  waarbij de betekenis precies gelijk is.

7.3. De precieze betekenis van een procedure statement.

Een programma waarin een procedure statement voorkomt houdt dezelfde betekenis na de volgende herschrijf-transformatie:

stap 1: Bepaal van welke procedure de procedure statement een aanroep is;
stap 2: Schrijf de body van deze procedure voor de procedure statement in de plaats;

stap 3: Schrijf hierboven

begin <declarations van de formal value parameters,
volgens de specificaties>;
<formal value parameter 1>:= <actual parameter 1>;
:
:

(Dit voor alle formal parameters die in de value list staan);

stap 4: Vervang alle formele name-parameters door de tekst van de corresponderende actuele parameter;

stap 5: Schrijf hieronder *end* (horend bij het *begin* van stap 3)

Voorbeeld:

```
begin   integer i; real array F, G[1 : 10];
        procedure verwissel de arrays (A) en: (B) tot: (n);
                value n; integer n; real array A, B;
        begin   integer i;
                for i:= 1 step 1 until n do
                begin   real w;
                        w:= A[i]; A[i]:= B[i]; B[i]:= w
                end
        end;
        for i:= 1 step 1 until 9 do
        begin F[i]:= i; G[i]:= i × i end;
        verwissel de arrays (F, G, 7);
        for i:= 7, 8, 9 do print (F[i]); print (G[i]) end
end
```

Herschreven:

```
begin   integer i; real array F, G[1 : 10];
        procedure verwissel de arrays (A) en: (B) tot: (n);
                value n; integer n; real array A, B;
        begin   integer i;
                for i:= 1 step 1 until n do
                begin   real w;
                        w:= A[i]; A[i]:= B[i]; B[i]:= w
                end
        end;
```

```
for i:= 1 step 1 until 9 do
begin F[i]:= i; G[i]:= i × i end;
begin   integer n;
        n:= 7;
        begin   integer i;
                for i:= 1 step 1 until n do
                begin   real w;
                        w:= F[i]; F[i]:= G[i]; G[i]:= w
                end
        end
end;
for i:= 7, 8, 9  do begin print (F[i]); print (G[i]) end
end
```

Nu het programma herschreven is volgens het gegeven herschrijfreglement komt er geen aanroep van de procedure *verwissel de arrays* meer in voor.

>29> Herschrijf de aanroep

verwissel de arrays (A, C, A[0])

volgens het herschrijfreglement.

>30> Gegeven is deze procedure declaration:

```
procedure verhoog (variabele) met: (waarde);
                value waarde; integer variabele, waarde;
variabele:= variabele + waarde;
```

Herschrijf nu de aanroep

verhoog (i,1)

Herschrijf deze zelfde aanroep, voor het geval dat ook de parameter *variabele* in de value part is opgenomen. Wat is het verschil in het effekt?

7.4. Wanneer *value*?

Als vuistregel geldt:
een parameter moet in de value part worden opgenomen als alleen de waarde van de bijbehorende actuele parameter van belang is; als alleen of mede de vorm waarin de actuele parameter gegeven wordt, van belang is, dan moet de parameter niet in de value part worden opgenomen.

>31> Als een parameter alleen door de procedure gebruikt wordt om een resultaat aan toe te kennen (output-parameter), moet deze parameter dan in de value part worden opgenomen?

>32> Als een parameter alleen dienst doet als input-parameter, d.w.z. dat door middel van deze parameter een waarde aan de procedure wordt opgegeven, moet deze parameter dan in de value part worden opgenomen?

7.5. Jensen's device

Een speciaal gebruik van name-parameters wordt naar degene die het bedacht heeft Jensen's device genoemd.
Eenvoudig voorbeeld:

```
real procedure somf (i, a, z, f); value a, z; integer i, a, z; real f;
begin   real s;
        s:= 0;
        for i:= a step 1 until z do s:= s + f;
        somf:= s
end
```

Om enig inzicht te krijgen in de werking van deze procedure, nemen we de aanroep:

```
kwadratensom:= somf (n, 1, 100, n × n)
```

en herschrijven deze:

```
begin   integer a, z;
        a:= 1; z:= 100;
        begin   real s;
                s:= 0;
                for n:= a step 1 until z do s:= s + n × n;
                kwadratensom:= s
        end
end
```

Bij de uitvoering van deze uitgeschreven aanroep zien we dat *kwadratensom* als waarde krijgt de som van de kwadraten van de gehele getallen 1 t/m 100.
Het is mogelijk met behulp van *somf* elke willekeurige uitdrukking die afhangt van een zekere variabele, te sommeren voor een aantal opeenvolgende gehele waarden van die variabele.
Voorbeeld:

```
print (somf (k, -10, +10, R[k]))
```

Deze aanroep van *somf* betekent hetzelfde als

```
R[-10] + R[-9] + ... + R[9] + R[10]
```

De kracht van Jensen's device is gelegen in de combinatie van drie factoren:

1. aan de kant van de declaratie zijn de formele parameter *i* (de lopende variabele) en de formele parameter *f* (de te sommeren grootheid) niet in de value part opgenomen;
2. aan de kant van de aanroep is voor de actuele parameter die met *f* korrespondeert een uitdrukking gekozen waarin de actuele parameter die met *i* korrespondeert voorkomt;
3. in de procedure declaratie wordt *f* steeds gebruikt nadat *i* met *1* verhoogd is.

>33> Wat komt er, in gewoon Nederlands gezegd, uit:

```
somf (k, 1, 10, A[k + k])
```

Nog een voorbeeld van Jensen's device:

```
begin   integer p; real a;
        procedure vermenigvuldig in (produkt) de funktie: (f) voor:
                                        (i) van: (a) tot: (z);
                value a, z; integer i, a, z; real f, produkt;
        begin   real p;
                p:= 1;
                for i:= a step 1 until z do p:= p × f;
                produkt:= p
        end;
        vermenigvuldig in (a, p + p × p) voor:
                                (p) lopend van: (1) tot: (4);
        print (a)
end;
```

Herschreven:

```
begin   integer p; real a;
        procedure vermenigvuldig in (produkt) de functie: (f) voor:
                                    (i) van: (a) tot: (z);
          value a, z; integer i, a, z; real f, produkt;
begin   real p;
        p:= 1;
        for i:= a step 1 until z do p:= p × f;
        produkt:= p
end;
begin   integer a', z;
        a':= 1; z:= 4
        begin   real p';
                p':= 1;
                for p:= a' step 1 until z do
                p':= p' × (p + p × p);
                a:= p'
        end
        end;
        print (a)
end
```

N.B. Op enkele punten is iets afgeweken van het letterlijke transformatie-reglement:

1. Als door substitutie identifiers in de procedure body belanden die daar al voorkwamen (hetzij als formele parameters, hetzij als in de procedure body gedeclareerde grootheden (of geplaatste labels)), dan worden die laatste identifiers stelselmatig "omgedoopt" (b.v. door een accent erachter).
2. Als de substitutie van een actuele name-parameter in een expressie de betekenis van die expressie geweld aangedaan wordt, worden om die te substitueren actuele parameters haakjes geplaatst. Bijvoorbeeld in $p' := p' \times f$.

Als hier voor f gesubstitueerd wordt: $p + p \times p$
dan ontstaat de uitdrukking: $p' := p' \times p + p \times p$
en dat betekent iets anders dan het bedoelde
$p' := p' \times (p + p \times p)$.

Wat oppervlakkiger uitgedrukt: het natuurlijke gaat voor het letterlijke. Wat preciezer uitgedrukt: als de actuele parameter corresponderend met een value formele parameter geen variabele is, maar wel een expressie, dan wordt deze bij substitutie tussen haakjes geplaatst.

>34> Als in het boven gegeven programma de aanroep van de procedure luidt:

vermenigvuldig in (z[1], z[a] + z[a + 1], a, 1, 3)

(waarbij *real array* z[1 : 4] gedeclareerd is en een waarde heeft gekregen), herschrijf dan deze aanroep volgens het gegeven reglement. Denk om conflicten tussen identifiers, en om het natuurlijke dat boven het letterlijke gaat.

>35> Herschrijf ook deze aanroep:

vermenigvuldig in (a, i, i, 1, 10)

Voer de herschreven statement uit.

>36> Het inprodukt van twee rijen getallen $a_1,\ldots,a_n$ en $b_1,\ldots,b_n$ is gedefinieerd als $a_1 \times b_1 + a_2 \times b_2 + \ldots + a_n \times b_n$. Schrijf een functie-procedure die het inprodukt van twee rijen getallen uitrekent. Denk erom dat deze procedure zo moet kunnen worden aangeroepen:

print (inprodukt (n, 0, 5, 2 × n, 2 × n + 1))

en zo:

a:= inprodukt (k, 1, 10, A[k], B[k - 1])

>37> Herschrijf de aanroep *print (inprodukt (n, 0, 5, 2 × n, 2 × n + 1))* volgens het herschrijfreglement, en voer het zo verkregen programma uit om te bepalen welk getal er wordt afgedrukt.

>38> Schrijf een functie-procedure *schuifinprod* die het inprodukt berekent van twee arrays, maar verschoven over een zeker stuk, b.v.

A[2] × B[5] + A[3] × B[6] + ... + A[11] × B[14]

Laat de body van deze procedure bestaan uit een aanroep van de procedure *inprodukt*.

>39> Schrijf een functie-procedure *inprod* die hetzelfde doet als *inprodukt*, maar waarvan de body bestaat uit een aanroep van *somf*.

Enige opmerkingen:

1. Een aanroep van een procedure is natuurlijk alleen korrekt, als de her-

schrijf-transformatie een korrekt stuk ALGOL 60 oplevert.

2. Als een formele parameter van het type *real array*, *integer array* of *Boolean array* in de value list voorkomt, betekent dit, dat in het extra geïntroduceerde block een array van hetzelfde type en met dezelfde grenzen wordt gedeclareerd; aan elk element van deze array wordt dan de waarde toegekend van de actuele array.
3. Als een formele parameter van het type label in de value list voorkomt, betekent dit dat in het extra block een grootheid van het type *label* wordt "gedeclareerd", waarna dan de actuele designational expression wordt "toegekend".
   Dit is de enige uitzondering op de regel dat de herschrijf-transformatie tot korrekt ALGOL 60 moet leiden.
   (Dit is de zoveelste bevestiging van onze langzaam groeiende verdenking dat de auteurs van het RR ons zand in de ogen willen strooien door te suggereren dat er, naast de types *real*, *integer* en *Boolean*, geen type *label* zou bestaan. Eerdere bevestigingen vonden we in de scope van labels, in het feit dat labels gewoon meedoen in het declaratie-concert wat betreft dubbele declaraties en de bepaling van de grootheid die door een identifier wordt aangeduid, en in de konstruktie van designational expressions.
   Ook het feit dat switches een procedure-achtig mombakkes voorkregen, heeft ons niet belet het label-array-karakter van deze misbaksels te onderkennen. Toch zullen we ons groot houden, en de halfslachtige notaties en beperkingen die het RR ons oplegt aanhouden.)

Syntaxis

(179) <procedure statement>::= <procedure identifier> (154)
<actual parameter part>

(180) <actual parameter part>::= <empty> | (42)
(181) (<actual parameter list>)

(182) <actual parameter list>::= <actual parameter> |
(183) <actual parameter list>
<parameter delimiter> (175)
<actual parameter>

```
(184) <actual parameter>::=          <string> |                    (258)
(185)                                <expression> |
(186)                                <array identifier> |          (93)
(187)                                <switch identifier> |         (40)
(188)                                <procedure identifier>        (154)
(189) <expression>::=                <arithmetic expression> |     (57)
(190)                                <Boolean expression> |        (97)
(191)                                <designational expression>    (32)
```

Voorbeelden

```
(179) check triangle (3, 4, 5, driehoek)
(180)
(181) (3, 4, 5, driehoek)
(182) driehoek
(183) 3, 4) hypotenusa: (5, driehoek
(184) 'het argument is negatief:'
(185) if i < 10 then opnieuw else klaar
(186) frekwentie
(187) Irene
(188) triangle
(189) p ↑ 3
(190) if if p then a = b else false then q else q ⊃ r
(191) if i < 10 then opnieuw else klaar
```

7.6. Function designator

Een function designator is een aanroep van een functie-procedure in een expressie, b.v.

```
if triangle (3, 4, 5) then ...
```

Hierin is *triangle (3, 4, 5)* een function designator.
Voor function designators gelden dezelfde regels als voor procedure statements. Verder stelt een function designator een waarde voor van real, integer of Boolean type.

Syntaxis

(192) <function designator>::= <procedure identifier> (154)
<actual parameter part> (180)

Voorbeelden

(192) *groot (A[lower], A[upper])*

7.7. Standaard procedures

ALGOL 60 kent een aantal procedures die niet gedeclareerd behoeven te worden. Het zijn functie-procedures met een parameter die zowel van real als van integer type mag zijn:

abs (E) als E < 0 dan -E anders E
sign (E) als E < 0 dan -1, als E = 0 dan 0, anders +1
sqrt (E) vierkantswortel uit E (square root)
sin (E) sinus van E (E in radialen)
cos (E) cosinus van E (E in radialen)
arctan (E) boogtangens van E (in radialen)
ln (E) natuurlijke logaritme van E
exp (E) e^E (e is grondtal van de natuurlijke logaritme)
entier (E) het grootste gehele getal dat niet groter is dan E.

Het resultaat van al deze functies is van type real, alleen *sign (E)* en *entier (E)* zijn van type integer (en in de meeste implementaties zal *abs (E)* van hetzelfde type zijn als E).
Een implementatie van ALGOL beschikt in het algemeen over nog meer standaard procedures, b.v. voor input en output. (Het X8-ALGOL-systeem kent o.a. de functie-procedure *read*, en de procedures *print* en *printtext*. Zie LR 1.1.)

N.B. Het feit dat al deze procedures ter beschikking staan impliceert niet dat het verboden is de namen van deze procedures voor andere grootheden te gebruiken. Als we dat doen, moeten we wel bedenken, dat binnen de scope van die (gelijknamige) grootheden geen gebruik gemaakt kan worden van de betreffende standaard procedure.

>40> Twee verschillende gehele positieve getallen zijn bevriend als ze de som van elkaars delers (inclusief 1, exclusief het getal zelf) zijn (vgl. >20>).
Bepaal de paren bevriende getallen waarbij de kleinste van de twee kleiner of gelijk is aan 1000.

>41> Gegeven zijn de koördinaten die de stand van een zwarte koning, een witte koning en een witte toren op een schaakbord bepalen. Ga na of de zwarte koning schaak staat, pat of mat, en hoeveel zetten de zwarte koning nog heeft (neem aan dat de stand reglementair is).

8. Identifier, string, letter, digit, comment (RR 2.4.,2.6.,2.1.,2.2.2., 2.3.)

Identifiers kunnen bestaan uit letters en cijfers; ze moeten beginnen met een letter. Voor de duidelijkheid mogen ze spaties bevatten, maar deze hebben geen betekenis, dus

check triangle

betekent hetzelfde als

checktriangle.

Een string is een rij ALGOL-symbolen waarin de string quotes ‘ en ’ alleen genest voorkomen. Strings kunnen in een ALGOL-programma voorkomen als actual parameter. Letters kunnen zowel kleine als hoofdletters zijn. Commentaar voor de menselijke lezer kan volgens deze conventies in een programma voorkomen (de konstruktie links is equivalent met die rechts):

; *comment* <een rij ALGOL-symbolen waarin ; niet voorkomt>; ;

begin comment <een rij ALGOL-symbolen waarin ; niet voorkomt>; *begin*

end <een rij ALGOL-symbolen waarin *end* en ; en *else* niet voorkomen> *end*

Voorbeeld:

```
if kolom > 8 then
begin   comment Nu moet de stand van dames nr. 1 t/m 8
                afgedrukt worden;
        integer k;
        nlcr; comment Elke bordstand op een aparte regel;
        for k:= 1 step 1 until 8 do print (dame[k])
end van de actie als kolom > 8
else ...
```

In de volgende konstruktie:

```
if p > 0 then
begin p:= p - 1; q:= q + 1 end begin
comment else p:= p + 1;
```

komt zowel de tweede van de gegeven commentaar-strukturen (*begin comment* ...) als de derde (*end* ...) voor, maar beide overlappen

elkaar.

Als we eerst *begin comment else p:= p + 1* door *;* vervangen, dan blijft over:

```
if p > 0 then
begin p:= p - 1; q:= q + 1 end;
```

Hierin komt geen commentaar meer voor.
Als eerst *end begin comment* vervangen wordt door *end*, dan blijft staan:

```
if p > 0 then
begin p:= p - 1; q:= q + 1 end
else p:= p + 1
```

We zien dat het verschil maakt welke vervanging we eerst uitvoeren.
Het Revised Report bepaalt dat eerst de commentaar-struktuur vervangen moet worden die het eerst in de tekst begint, in dit geval dus:
end begin comment

Syntaxis

(193) <identifier>::= <letter> |
(194) <identifier> <letter> |
(195) <identifier> <digit>
(196) <letter>::= *a|b|c|d|e|f|g|h|i|j|k|l|m|*
n|o|p|q|r|s|t|u|v|w|x|y|z|
A|B|C|D|E|F|G|H|I|J|K|L|M|
N|O|P|Q|R|S|T|U|V|W|X|Y|Z

(248) <digit>::= *0|1|2|3|4|5|6|7|8|9*

(258) <string>::= ‘ <open string> ’

(259) <open string>::= <proper string> |

(260) ‘ <open string> ’ |

(261) <open string> <open string>

(262) <proper string>::= <any sequence of basic symbols not containing ‘ or ’ > |

(263) <empty> (42)

Voorbeelden

(193) *n*

(194) *nummer*

(195) *nummer 1*

(196) *a*

(248) *0*

(258) ‘*het maximum is*’

(259) *het maximum is*

(260) ‘*het maximum is*’

(261) ‘‘*het maximum*’ *is*’ ‘*het grootste*’

(262) *het maximum is*

(263)

Uitwerking van de vraagstukken van deel 2

<01<

```
                                    <program>
                                <compound  statement>                                  2
                                <unlabelled compound>                                  5
      begin                       <compound tail>                                      8
      begin          <statement>          ;                  <compound tail>          12
      begin          <statement>          ;            <statement>          end       11
      begin   <unconditional statement>   ;   <unconditional statement>    end       13
      begin      <basic statement>        ;       <basic statement>        end       16
      begin <unlabelled basic statement>  ; <unlabelled basic statement>   end       19
      begin       <dummy statement>       ;       <dummy statement>        end       23
      begin                                                                end
```

<02< Als de invoergetallen zijn: 1, 2, 3
dan betekent *F[read]:= G[read]:= read*
hetzelfde als *F[1]:= G[2]:= 3*
Maar *G[read]:= F[read]:= read*
betekent *G[1]:= F[2]:= 3*

<03< De (foute) konstruktie

if g[i] ≠ 0 then if i < n then i:= i + 1 else i:= i - 1

kan op twee manieren worden geïnterpreteerd, nl. als

```
if g[i] ≠ 0 then
begin if i < n then i:= i + 1 end
else i:= i - 1
```

of als:

```
if g[i] ≠ 0 then
begin if i < n then i:= i + 1 else i:= i - 1 end
```

< 04 < a. 1 2 3

b. 1

c.

d. 1

e. 1 3

f. 0 0 0 0 0 0 0 0 0 0 0...

g. 1 2

h. 1

i. 1

j. 1 2 3 4

k. 1 1 1 1 1 1 1 1 1 1 1...

l. 2 5 11 23 47 95

m. 11 12 13 21 22 23 31 32 33

n. 1 2 4 8 16

o. 2 3 5 9 17

p. 1 1 1 1 1 1 1 1 1 1 1...

q. 0 -1

r. 2 -4 -16 -256...

s. 1 2 4 8 13 10 7 4 1

< 05< Fout zijn:

c. *1 step 1 while i ≤ 100* is geen for list element

d. *i ≥ 1* is geen for list element

g. na afloop van de for statement wordt van de controlled variable gebruik gemaakt in de Boolean expression *i = 21*

h. de go to statement *go to P* leidt van buiten de for statement *for j:=...* naar een label binnen die for statement.

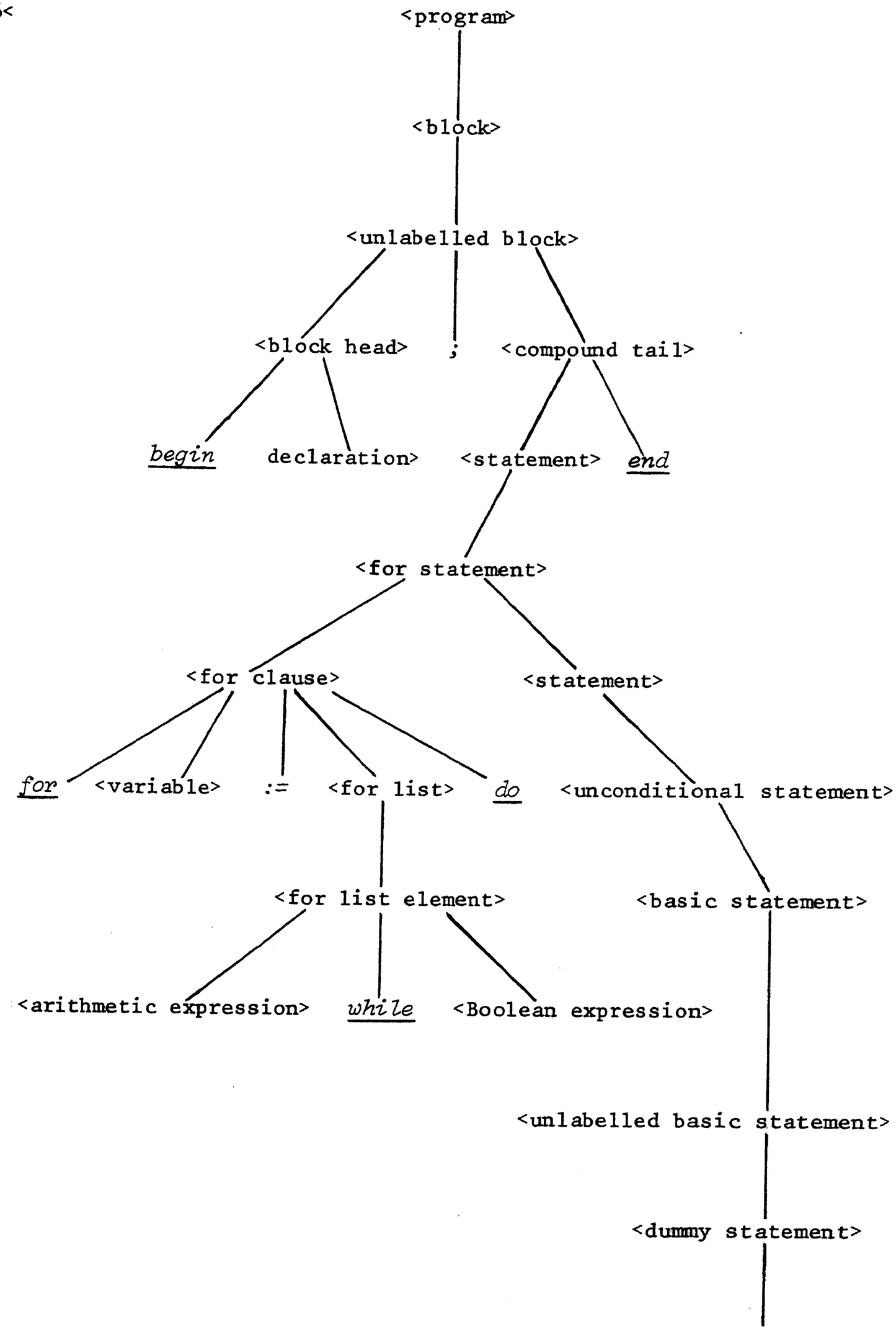
l. nadat de eerste 3 getallen zijn afgedrukt wordt bij het ophogen van de controlled variable *i* (de eerste) gebruik gemaakt van de waarde van de controlled variable *i* (de tweede), die op dat moment een ongedefinieerde waarde heeft.

m. *for i = 1* is geen korrekt begin van een for statement; *for i:= 1* wel.

n. de laatste opdracht *print (i)* maakt gebruik van een ongedefinieerde waarde

o. idem.

<06<


<program>
<block>
<unlabelled block>
<block head>
;
<compound tail>
begin
declaration>
<statement>
end
<for statement>
<for clause>
<statement>
for
<variable>
:=
<for list>
do
<unconditional statement>
<for list element>
<basic statement>
<arithmetic expression>
while
<Boolean expression>
<unlabelled basic statement>
<dummy statement>

<07< a. korrekt: is een statement

b. korrekt: *begin for i:= 1 do end*

c. fout

d. korrekt: *if a > 0 then begin end else*

e. korrekt: *if a > 0 then else for a:= 1 do*

f. korrekt: *if a > 0 then for a:= 1 do*

g. fout

h. fout

i. korrekt: *begin ; end*

j. korrekt: *begin begin end ; a:= 1 end*

k. korrekt: *begin if a > 0 then ; a:= 1 end*

l. korrekt: *begin if a > 0 then else ; a:= 1 end*

m. fout

n. korrekt: *begin for i:= 1 do ; end*

o. fout

p. fout

<08<

a.	real	8.4	i.	integer	1
b.	integer	0	j.	integer	0
c.	real	0	k.	undefined	
d.	real	0	l.	real	1
e.	real	2	m.	real	2
f.	integer	2	n.	integer	4
g.	undefined		o.	real	8
h.	integer	-1	p.	real	2

<09< a. <arithmetic expression>
<if clause> <simple arithmetic expression> *else* <arithmetic expression>
if <Boolean expression> *then* <simple arithmetic expression>
else <simple arithmetic expression>

b. Volgens RR 3.3.3. is dit equivalent met:
if <Boolean expression> *then* <simple arithmetic expression> *else*
if true then <simple arithmetic expression>
Dit valt echter volgens de syntaxis niet van <arithmetic expression> af te leiden; elke poging moet stranden op het feit dat een arithmetic expression die met *if* begint ook op *else* <arithmetic expression> moet eindigen.

<10< <simple arithmetic expression>
<term>
<factor>
<primary>
(<arithmetic expression>*)*

<11< Geen number zijn:

b. een getal eindigt nooit op een punt;
f. na $_{10}$ moet een exponent volgen;
g. de exponent na $_{10}$ moet geheel zijn;
h. dit is wel een arithmetic expression, maar niet een number;
i. de exponent moet geheel zijn, en mag dan ook niet $_{10}$ bevatten;
j. de exponent moet een number zijn;
k. dit is wel een expression, maar niet een number;
l. geen twee tekens achter elkaar.

<12< Fout zijn:

b. na $_{10}$ moet een exponent volgen;
d. de exponent na $_{10}$ moet een number zijn;
g. moet zijn: $5 \times (-2)$;
h. moet zijn: $5 \uparrow (-2)$;
j. er moet een expression tussen de haakjes staan.

<13< a.

a	*false*	*false*	*true*	*true*	
b	*false*	*true*	*false*	*true*	
$\neg a \vee b$	*true*	*true*	*false*	*true*	identiek
$\neg(a \wedge \neg b)$	*true*	*true*	*false*	*true*	identiek
$a \supset b$	*true*	*true*	*false*	*true*	identiek

b.

a	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*true*
b	*false*	*false*	*true*	*true*	*false*	*false*	*true*	*true*
c	*false*	*true*	*false*	*true*	*false*	*true*	*false*	*true*
$(a \wedge b) \wedge c$	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*true*
$a \wedge (b \wedge c)$	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*true*

identiek

c.	*a*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	
	b	*false*	*false*	*true*	*true*	*false*	*false*	*true*	*true*	
	c	*false*	*true*	*false*	*true*	*false*	*true*	*false*	*true*	
	$a \wedge (b \vee c)$	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	identiek
	$a \wedge b \vee a \wedge c$	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	identiek

<14< a. *false* ⊃ *false* en *false* ⊃ *true* zijn beide waar

b. *false* ∨ *true* en *true* ∨ *true* zijn beide waar

c. *false* ⊃ *true* en *true* ⊃ *true* zijn beide waar

d. *false* ∨ ¬*false* en *true* ∨ ¬*true* zijn beide waar

e. ¬(*false* ∧ ¬*false*) en ¬(*true* ∧ ¬*true*) zijn beide waar

f. $a \leq 0$ is equivalent met $\neg a > 0$

dus $a \leq 0 \vee a > 0$ is equivalent met $\neg a > 0 \vee a > 0$

Dit levert altijd *true* volgens d.

g. $a > 0 \supset a \geq 0$

Voor $a < 0$: *false* ⊃ *false* is waar

Voor $a = 0$: *false* ⊃ *true* is waar

Voor $a > 0$: *true* ⊃ *true* is waar

h.

a	*false*	*false*	*true*	*true*
b	*false*	*true*	*false*	*true*
$a \wedge b$	*false*	*false*	*false*	*true*
$a \wedge \neg b$	*false*	*false*	*true*	*false*
$\neg a \wedge b$	*false*	*true*	*false*	*false*
$\neg a \wedge \neg b$	*true*	*false*	*false*	*false*
$a \wedge b \vee a \wedge \neg b \vee \neg a \wedge b \vee \neg a \wedge \neg b$	*true*	*true*	*true*	*true*

i. $(a \supset b) \wedge (b \supset a) \equiv (a \equiv b)$

Voor *a* en *b* *false*:

(*false* ⊃ *false*) ∧ (*false* ⊃ *false*) ≡ (*false* ≡ *false*)

true ∧ *true* ≡ *true*

true ≡ *true*

true

Voor *a* *false* en *b* *true*:

(*false* ⊃ *true*) ∧ (*true* ⊃ *false*) ≡ (*false* ≡ *true*)

true ∧ *false* ≡ *false*

false ≡ *false*

true

Voor *a* *true* en *b* *false*:

(true ⊃ *false)* ∧ *(false* ⊃ *true)* ≡ *(true* ≡ *false)*

false ∧ *true* ≡ *false*

false ≡ *false*

true

Voor *a* en *b* *true*:

(true ⊃ *true)* ∧ *(true* ⊃ *true)* ≡ *(true* ≡ *true)*

true ∧ *true* ≡ *true*

true ≡ *true*

true

Dus wat ook de waarden van *a* en *b* zijn, de Boolean expression levert altijd *true* op.

j. *(a* ⊃ *b)* ∧ *a* ⊃ *b*

Voor *a* en *b* *false*:

(false ⊃ *false)* ∧ *false* ⊃ *false*

true ∧ *false* ⊃ *false*

false ⊃ *false*

true

Voor *a* *false* en *b* *true*:

(false ⊃ *true)* ∧ *false* ⊃ *true*

true ∧ *false* ⊃ *true*

false ⊃ *true*

true

Voor *a* *true* en *b* *false*:

(true ⊃ *false)* ∧ *true* ⊃ *false*

false ∧ *true* ⊃ *false*

false ⊃ *false*

true

Voor *a* en *b* *true*:

(true ⊃ *true)* ∧ *true* ⊃ *true*

true ∧ *true* ⊃ *true*

true ⊃ *true*

true

<15< Dit voorbeeld kan zo geschreven worden:

if $a > 0 \supset a \geq 20 \supset a > 30$ *then* ...

(Dat we aan de operator $\supset$ zeker voldoende hebben, kunnen we zo inzien:
$\neg p$ kan geschreven worden als $p \supset$ *false*
$p \wedge q$ kan geschreven worden als $p \supset q \supset \neg(q \supset p)$
$p \vee q$ kan geschreven worden als $(\neg p \supset q) \wedge (\neg q \supset p)$
$p \equiv q$ kan geschreven worden als $(p \supset q) \wedge (q \supset p)$
Met behulp van deze regels kunnen we dus uit elke Boolean expression successievelijk de logische operatoren verwijderen, totdat alleen $\supset$ overblijft.) Met wat meer kunst en vliegwerk kan het voorbeeld ook zo geschreven worden:

if $abs(a - 10) < 10 \vee a > 30$ *then* ...

waarin ook slechts één logische operator voorkomt. In het algemeen is de operator $\vee$ echter niet voldoende.

<16< Niet afleidbaar van <Boolean expression> zijn:

c. wel: $\neg(\neg$*false*$)$
g. wel: *if false then (if true then false else true) else false*
h. wel: *if true then false else false*

<17< a. korrekt; drukt af: *3 3*
b. korrekt; drukt af: *1*
c. fout: in regel 4 wordt *a* als label geplaatst, en als arithmetic expression gebruikt.
d. fout: in regel 4 zijn de left parts van de assigment statement van verschillend type.
e. fout: in regel 7 en 8 komt een *a* voor die in dat block niet bekend is.
f. fout: in regel 9 wordt gesprongen naar een label *a* die in dat block niet bekend is.

<18< Afgedrukt wordt: *5 6 7*

<19< Om te bepalen welke *A* in regel 3 bedoeld is, moeten we kijken of bij het kleinste block dat deze *A* omvat een grootheid *A* hoort. Dat is zo, nl. de label *A* op regel 9. Die label *A* is dus bedoeld.

<20< Eerste niveau:

```
   begin   integer i;
           for i:= 1 step 1 until 1000 do
A          if som delers van i = i then output
   end
```

Tweede niveau:

```
A: begin   integer s, d;
           s:= 0;
           for d:= 1 step 1 until i ÷ 2 do
           if i ÷ d × d = i then s:= s + d;
           if s = i then begin nlcr; print (i) end
   end
```

Resultaat:

```
   begin   integer i;
           for i:= 1 step 1 until 1000 do
           begin   integer s, d;
                   s:= 0;
                   for d:= 1 step 1 until i ÷ 2 do
                   if i ÷ d × d = i then s:= s + d;
                   if s = i then begin nlcr; print (i) end
           end
   end
```

<21< Eerste niveau:

```
   begin   integer i;
           for i:= read while i > 0 do
           if i > 1 then
A          druk de ontbinding van i af
   end
```

Tweede niveau:

```
A: begin   integer d;
           nlcr; print (i); printtext (‡ = ‡);
           for d:= 2, 3, d + 2 while d ≤ sqrt (i) do
B          deel alle factoren d uit i en druk ze af;
           if i > 1 then print (i)
   end
```

Derde niveau:

```
B: begin   integer macht, dmacht, dm;
           macht:= 0; dm:= 1;
           for dm:= dm × d while i ÷ dm × dm = i do
           begin dmacht:= dm; macht:= macht + 1 end;
           if macht > 0 then
           begin   nlcr; print (d);
                   if macht > 1 then
                   begin printtext (‡ ↑ ‡); print (macht) end;
                   printtext (‡ × ‡); i:= i ÷ dmacht
           end
   end
```

Resultaat (met kleine wijziging):

```
begin   integer i;
        for i:= read while i > 0 do
        if i > 1 then
        begin   integer d, g;
                nlcr; print (i); printtext (ǂ = ǂ); g:= sqrt (i);
                for d:= 2, 3, d + 2 while d ≤ g do
                begin   integer macht, dmacht, dm;
                        macht:= 0; dm:= 1;
                        for dm:= dm × d while i ÷ dm × dm = i do
                        begin dmacht:= dm; macht:= macht + 1 end;
                        if macht > 0 then
                        begin   nlcr; print (d);
                                if macht > 1 then
                                begin   printtext (ǂ ↑ ǂ);
                                        print (macht)
                                end;
                                printtext (ǂ × ǂ); i:= i ÷ dmacht;
                                g:= sqrt (i)
                        end
                end;
                if i > 1 then print (i)
        end
end
```

<22< Eerste niveau:

```
begin   integer i;
        i:= read;
        if i > 0 then
        begin   integer x, y;
                print (i); printtext (ǂ = ǂ);
A               zoek de paren (x,y) zodat x × x + y × y = i en x ≥ y
        end
end
```

Tweede niveau:

```
A1:for x:= entier (sqrt (i ÷ 2)) step 1 until entier (sqrt (i)) do
   begin   y:= sqrt (i - x × x);
           if x × x + y × y = i then output;
   end
```

Tweede niveau op andere manier:
het is niet nodig bij elke x de bijbehorende y te bepalen door worteltrekken. Het is mogelijk alleen de paren (x,y) te bekijken die vlak langs de rand van de cirkel $x^2 + y^2 = i$ liggen, en wel over de helft van het cirkelkwart in het kwadrant van positieve x en positieve y.

```
A2:x:= entier (sqrt (i)); y:= sqrt (i - x × x);
       for x:= x while x ≥ y do
       begin   integer s; s:= x × x + y × y;
               if s > i then x:= x - 1 else
               if s < i then y:= y + 1 else
                   begin output; x:= x - 1; y:= y + 2 end
       end
```

Derde niveau:

output:

```
nlcr; print (x); printtext (↑ 2 +); print (y); printtext (↑ 2)
```

Resultaat 1:

```
begin   integer i; i:= read;
        if i > 0 then
        begin   integer x, y, g;
                print (i); printtext (ǂ = ǂ); g:= entier (sqrt (i));
                for x:= entier (sqrt (i ÷ 2)) step 1 until g do
                begin   y:= sqrt (i - x × x);
                        if x × x + y × y = i then
                        begin   nlcr; print (x);
                                printtext (ǂ ↑ 2 + ǂ);
                                print (y); printtext (ǂ ↑ 2 ǂ)
                        end
                end
        end
end
```

Resultaat 2:

```
begin   integer i; i:= read;
        if i > 0 then
        begin   integer x, y;
                print (i); printtext (ǂ = ǂ);
                x:= entier (sqrt (i)); y:= sqrt (i - x × x);
                for x:= x while x ≥ y do
                begin   integer s; s:= x × x + y × y;
                        if s > i then x:= x - 1 else
                        if s < i then y:= y + 1 else
                        begin   nlcr; print (x);
                                printtext (ǂ ↑ 2 + ǂ);
                                print (y); printtext (ǂ ↑ 2 ǂ);
                                x:= x - 1; y:= y + 2
                        end
                end
        end
end
```

<23< Eerste niveau:

```
        begin   integer array x, y[1 : 3];
                integer z, k, t, i; z:= 1; k:= 2; t:= 3;
                for i:= z, k, t do
                begin x[i]:= read; y[i]:= read end;
A               bepaal of stukken buiten het bord staan, of er twee stuk-
                ken op hetzelfde veld staan, of de koningen tegen elkaar
                staan; zo niet, ga na of de toren de zwarte koning be-
                strijkt
    end
```

Tweede niveau:

```
A: for i:= z, k, t do
   if x[i] < 0 ∨ x[i] > 8 ∨ y[i] < 0 ∨ y[i] > 8 then
   begin nlcr; printtext (‡stuk buiten bord‡); go to klaar end;
   for a:= 1, 2 do for b:= a + 1 step 1 until 3 do
   if x[a] = x[b] ∧ y[a] = y[b] then
   begin nlcr; printtext (‡stukken op gelijk veld‡); go to klaar end;
   for a:= x[z] - 1, x[z], x[z] + 1 do
   for b:= y[z] - 1, y[z], y[z] + 1 do
   if a = x[k] ∧ b = y[k] then
   begin nlcr; printtext (‡koningen staan tegen elkaar‡); go to klaar end;
   schaak:= false;
   if x[z] = x[t] then
   begin   if x[k] = x[z] then
           begin   comment de witte koning kan ertussen staan;
                   if y[k] < y[z] ≡ y[k] < y[t] then schaak:= true
           end
   end else
   if y[z] = y[t] then
   begin   if y[k] = y[z] then
           begin if x[k] < x[z] ≡ x[k] < x[t] then schaak:= true end
   end;
   if schaak then printtext (‡zwarte koning staat schaak‡);
klaar:
```

Resultaat:

```
begin   integer array x, y[1 : 3]; Boolean schaak;
integer z, k, t, i, a, b; z:= 1; k:= 2; t:= 3;
for i:= z, k, t do
begin x[i]:= read; y[i]:= read end;
for i:= z, k, t do
if x[i] < 0 ∨ x[i] > 8 ∨ y[i] < 0 ∨ y[i] > 8 then
begin printtext (‡stuk staat buiten het bord‡); go to klaar end;
for d:= 1, 2 do for b:= a + 1 step 1 until 3 do
if x[a] = x[b] ∧ y[a] = y[b] then
begin printtext (‡stukken op gelijk veld‡); go to klaar end;
for a:= x[z] - 1, x[z], x[z] + 1 do
for b:= y[z] - 1, y[z], y[z] + 1 do
if a = x[k] ∧ b = y[k] then
begin printtext (‡koningen staan tegen elkaar‡); go to klaar end;
schaak:= false;
if x[z] = x[t] then
begin   if x[k] = x[z] then
        begin   comment de witte koning kan ertussen staan;
                if y[k] < y[z] ≡ y[k] < y[t] then schaak:= true
        end
end else
if y[z] = y[t] then
begin   if y[k] = y[z] then
        begin if x[k] < x[z] ≡ x[k] < x[t] then schaak:= true end
end;
if schaak then printtext (‡zwarte koning staat schaak‡);
klaar:
end
```

<24< Eerste methode:

```
begin   integer n; n:= read;
        begin   real array a[1 : n];
                integer i;
                real m, s;
                m:= 0;
                for i:= 1 step 1 until n do
                begin a[i]:= read; m:= m + a[i] end;
                m:= m / n;
                printtext (‡gemiddelde = †); print (m);
                s:= 0;
                for i:= 1 step 1 until n do s:= s + (a[i] - m) ↑ 2;
                nlcr; printtext (‡spreiding = †);
                print (sqrt (s / n))
        end
end
```

Tweede methode:

Het opbergen van de getallen is alleen nodig voor de berekening van de spreiding. Die kan echter ook zo berekend worden:

$$s^2 = \frac{1}{n}\Sigma(a_i - m)^2 = \frac{1}{n}\Sigma(a_i^2 - 2a_i m + m^2) =$$

$$\frac{1}{n}\Sigma a_i^2 - \frac{1}{n}\Sigma 2a_i m + \frac{1}{n}\Sigma m^2 =$$

$$\frac{1}{n}\Sigma a_i^2 - \frac{2}{n}m\Sigma a_i + \frac{1}{n}nm^2 =$$

$$\frac{1}{n}\Sigma a_i^2 - 2m^2 + m^2 = \frac{1}{n}\Sigma a_i^2 - m^2$$

Als we dus de som van de kwadraten van de getallen bijhouden, is het opbergen in de array niet nodig.

```
begin   integer n, i; real a, s, s2; s:= s2:= 0;
        for i:= 1 step 1 until n do
        begin a:= read; s:= s + a; s2:= s2 + a × a end;
        printtext (‹gemiddelde = ›); print (s / n); nlcr;
        printtext (‹spreiding = ›);
        print (sqrt (s2 / n - (s / n) ↑ 2))
end
```

<25< *in orde* bepaalt of de array *array* een *i*-de element heeft, en zo ja of de waarde van dat element gelijk is aan de waarde van *expr*

<26< De procedure body kan luiden:

```
begin   real s; integer i;
        s:= 0;
        for i:= lower step 1 until upper do s:= s + array[i];
        som:= s
end
```

<27<
(2) Specificatie van *i* ontbreekt; is verplicht wegens *value i*.

(5) Het type van een procedure kan niet *Boolean array* zijn, wel *Boolean*.

(7) Nu *negatie* niet van het type *array* kan zijn, mag ook geen subscript expression tussen vierkante haken volgen.

(9) *value a* moet voorafgaan aan *real a*.

(10) Tussen *plus* en *(* moet een *:* staan.

(11) In de specificatie mag geen "vette komma" voorkomen.

(12-14) *som* moet een waarde krijgen.

(15) In de specificatie mogen alleen identifiers voorkomen, dus niet [*1 : 100*].

<28< a. Het Revised Report geeft niet aan in welke volgorde de operanden van een expressie worden uitgewerkt!
Als we aannemen dat bij de uitvoering van

print (groot (p, q) + groot (q, r))

eerst *groot (p, q)* wordt uitgevoerd en dan pas *groot (q, r)* dan is het effekt van de opdracht, dat de waarden van de drie

variabelen na afloop zo gerangschikt staan, dat *r* de kleinste waarde bevat. De som van de beide andere waarden wordt afgedrukt.

b. Als eerst *groot (q, r)* wordt uitgevoerd en dan *groot (p, q)* dan is het effekt, dat de grootste van de drie waarden in *p* komt. Afgedrukt wordt dan ofwel het dubbele van deze waarde, ofwel de som van deze waarde en de middelste waarde (nl. als *p* anvankelijk al de grootste waarde bevatte).

c. Het effekt van *groot (p, q); groot (q, r); print (p + q)* is gelijk aan dat van a.

```
<29<    begin  integer n;
               n:= A[0];
               begin  integer i;
                      for i:= 1 step 1 until n do
                      begin  real w;
                             w:= A[i]; A[i]:= C[i]; C[i]:= w
                      end
               end
        end

<30<    begin  integer waarde;
               waarde:= 1;
               i:= i + waarde
        end
```

Als *variabele* ook in de value part staat:

```
        begin  integer variabele, waarde;
               variabele:= i; waarde:= i;
               variabele:= variabele + waarde
        end
```

In het eerste geval wordt de waarde van *i* veranderd, in het tweede geval niet.

<31< De parameter moet dan niet in de value part worden opgenomen, want zijn aanvankelijke waarde is van geen belang.

<32< De parameter moet wel in de value part worden opgenomen, want alleen zijn aanvankelijke waarde is van belang.

<33< De som van de eerste 10 elementen op de hoofddiagonaal van array *A*.

<34<
```
begin  integer a', z';
       a':= 1; z':= 3;
       begin  real p;
              p:= 1;
              for a:= a' step 1 until z' do
              p:= p × (z[a] + z[a + 1]);
              z[1]:= p
       end
end
```

<35<
```
begin  integer a', z;
       a':= 1; z:= 10;
       begin  real p;
              p:= 1;
              for i:= a' step 1 until z do p:= p × i;
              a:= p
       end
end
```

Resultaat is dat de variabele *a* de waarde van 10! krijgt: 3628800.

<36<
```
real procedure inprodukt (i, a, z, f, g); value a, z;
                          integer i, a, z; real f, g;
begin  real s;
       s:= 0;
       for i:= a step 1 until z do s:= s + f × g;
       inprodukt:= s
end
```

```
<37<    begin   integer a, z;
                a:= 0; z:= 5;
                begin   real s;
                        for n:= a step 1 until z do
                        s:= s + (2 × n) × (2 × n + 1);
                        inprodukt:= s
                end;
                print (inprodukt)
        end
```

Afgedrukt wordt: 250

```
<38<    real procedure schuifinprod (i, a, z, A, B, d); value a, z, d;
                                integer i, a, z, d; array A, B;
        schuifinprod:= inprodukt (i, a, z, A[i], B[i + d])

<39<    real procedure inprod (i, a, z, f, g); value a, z;
                        integer i, a, z; real f, g;
        inprod:= somf (i, a, z, f × g)
```

<40< Eerste niveau:

```
        begin   integer i;
                for i:= 1 step 1 until 1000 do
                begin   integer j;
                        j:= som delers van i;
A                       if j ≤ i then else
B                       if som delers van j = i then output
                end
        end
```

Tweede niveau:

In A en B komt dezelfde handeling voor:

```
integer procedure som delers van (k); value k; integer k;
begin   integer s, d;
        s:= 0;
        for d:= 1 step 1 until k ÷ 2 do
        if k ÷ d × d = k then s:= s + d;
        som delers van:= s
end
```

A en B kunnen nu zo geschreven worden:

```
A: j:= som delers van (i);
B: if som delers van (j) = i then
   begin nlcr; print (i); print (j) end
```

Resultaat:

```
begin   integer i;
        integer procedure som delers van (k); value k; integer k;
        begin   integer s, d;
                s:= 0;
                for d:= 1 step 1 until k ÷ 2 do
                if k ÷ d × d = k then s:= s + d;
                som delers van:= s
        end;
        for i:= 1 step 1 until 1000 do
        begin   integer j;
                j:= som delers van (i);
                if j ≤ i then else
                if som delers van (j) = i then
                begin nlcr; print (i); print (j) end
        end
end
```

<41< Eerste niveau:

A *begin* initialisatie;
B lees stand stukken;
C ga na of de witte toren de zwarte koning bestrijkt;
D ga voor alle zetten voor de zwarte koning na of hij door witte koning of toren bestreken wordt;
E trek konklusies
end

```
A: integer array x, y[1 : 3];
   integer z, k, t, i; z:= 1; k:= 2; t:= 3;
B: for i:= z, k, t do
   begin x[i]:= read; y[i]:= read end;
```

In C en D komt meermalen de kwestie van het schaak staan van zwart voor:

```
Boolean procedure zwart schaak (xz, yz, xk, yk, xt, yt);
                  value xz, yz, xk, yk, xt, yt;
                  integer xz, yz, xk, yk, xt, yt;
begin   Boolean schaak;
        schaak:= false;
        if xz = xt then
        begin   if xk = xz then
                begin   comment de witte koning kan ertussen staan;
                        if sign (yk - yz) = sign (yk - yt) then
                        schaak:= true
                end
        end else
        if yz = yt then
        begin   if yk = yz then
                begin   if sign (xk - xz) = sign (xk - xt) then
                        schaak:= true
                end;
                zwart schaak:= schaak
    end
C:  schaak:= zwart schaak (x[z], y[z], x[k], y[k], x[t], y[t]);
D:  zwarte zetten:= 0;
    for xz:= x[z] - 1, x[z], x[z] + 1 do
    for yz:= y[z] - 1, y[z], y[z] + 1 do
    if xz ≠ x[z] ∨ yz ≠ y[z] then
    begin   if ¬(zwart schaak (xz, yz, x[k], x[t], y[t]) ∨
            abs (x[z] - x[k]) ≤ 1 ∧ abs (y[z] - y[k]) ≤ 1) then
            zwarte zetten:= zwarte zetten + 1
    end;
E:  if zwarte zetten = 0 then
    begin if schaak then printtext (‡mat‡) else printtext (‡pat‡) end
    else
    begin   if schaak then
            begin printtext (‡schaak‡); nlcr end;
            print (zwarte zetten); printtext (‡mogelijke zetten‡)
    end
```

Resultaat:

```
begin   integer array x, y[1 : 3];
        integer zwarte zetten, xz, yz; Boolean schaak;
        Boolean procedure zwart schaak (xz, yz, xk, yk, xt, yt);
value xz, yz, xk, yk, xt, yt; integer xz, yz, xk, yk, xt, yt;
begin   Boolean schaak; schaak:= false;
        if xz = xt then
        begin   if xk = xz then
                begin   if sign (yk - yz) = sign (yk - yt) then
                                schaak:= true
                end
        end else
        if yz = yt then
        begin   if yk = yz then
                begin   if sign (xk - xz) = sign (xk - xt) then
                                schaak:= true
                end
        end;
        zwart schaak:= schaak
end;
for i:= z, k, t do begin x[i]:= read; y[i]:= read end;
schaak:= zwart schaak (x[z], y[z], x[k], y[k], x[t], y[t]);
zwarte zetten:= 0;
for xz:= x[z] - 1, x[z], x[z] + 1 do
for yz:= y[z] - 1, y[z], y[z] + 1 do
if xz ≠ x[z] ∨ yz ≠ y[z] then
begin   if ¬(zwart schaak (xz, yz, x[k], y[k], x[t], y[t]) ∨
                abs (x[z] - x[k]) ≤ 1 ∧ abs (y[z] - y[k]) ≤ 1) then
        zwarte zetten:= zwarte zetten + 1
end;
if zwarte zetten = 0 then
begin if schaak then printtext (‡mat‡) else printtext (‡pat‡) end
else
        begin   if schaak then
                begin printtext (‡schaak‡); nlcr end;
                print (zwarte zetten); printtext (‡mogelijke zetten‡)
        end
end
```