stichting
mathematisch
centrum

MC

AFDELING INFORMATICA IC 1/74 APRIL

L. AMMERAAL
SYLLABUS CURSUS ALGOL 68

Voorlopige uitgave

2e boerhaavestraat 49 amsterdam

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

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

ACM - Computing Reviews - category: 1.50, 4.22

Syllabus cursus ALGOL 68

door

L. Ammeraal

SAMENVATTING

Deze syllabus is een informele inleiding tot de hogere programmeertaal ALGOL 68. Bij de behandeling van (de gereviseerde versie van) deze taal is vooral gestreefd naar leesbaarheid; enige moeilijke details zijn omzeild. Kennis van één bepaalde andere programmeertaal wordt niet vooronderstelt; wel wordt de lezer geacht bekend te zijn met de beginselen van het programmeren.

INHOUD blz.

1. INLEIDING

Deze syllabus is bedoeld als eerste kennismaking met ALGOL 68. Daarom zullen bijzonderheden die de beginner in verwarring zouden kunnen brengen zoveel mogelijk achterwege worden gelaten evenals spitsvondige opmerkingen waartoe deze taal aanleiding zou kunnen geven. Anderzijds zal worden getracht de lezer zoveel houvast te bieden dat hij in staat wordt gesteld praktisch gebruik van de taal te maken. Nadat hij op deze wijze enigszins wegwijs is geworden, is aanvullende ALGOL 68-studie sterk aan te bevelen. Een probleem hierbij vormt de nogal aanzienlijke evolutie die de taal de laatste jaren heeft ondergaan. Bij de bestudering van [1], [2] en [3] dient er daarom rekening mee te worden gehouden dat hierin de oorspronkelijke versie van de taal wordt beschreven. Het is te hopen en te verwachten dat er spoedig zowel een *Revised ALGOL 68 Report*, waarvan [4] een "blauwdruk" is, als een aantal bijgewerkte informele introducties zal verschijnen.

In deze syllabus wordt veelal de oorspronkelijke Engelse terminologie gebruikt. Als soms voor het gemak een Nederlandse uitgang gekozen wordt laat de bedoeling zich gemakkelijk raden. Men hoeft bijvoorbeeld geen groot kenner van de Engelse taal te zijn om te vermoeden dat de term "declareren" betrekking heeft op wat elders "declaration" is genoemd.

2. DENOTATIONS

Een *denotation* kan worden opgevat als een generalisatie van een "getal" of "constante". Denotations kunnen deel uitmaken van de programmatekst en vertegenwoordigen een "waarde" die direct uit hun uiterlijke verschijningsvorm volgt.

We noemen hier de volgende soorten denotations:

- *integral denotations* zijn gehele getallen, zonder plus- of minteken, geschreven als een rijtje decimale cijfers, b.v.

  4096, 4, 58, 0.

- *real denotations* zijn getallen die genoteerd worden met behulp van een decimale punt of een "times ten to the power symbol", d.i. de letter e, b.v.

  0.00123, 1.23e-3, .38, 1e0.

Opmerkingen

1. Het getal mag niet op een punt eindigen.
2. Het getal mag niet met een "times ten to the power symbol" beginnen.
3. Helemaal vooraan mag geen plus- of minteken staan.
4. Direct na het "times ten to the power symbol" mag een plus- of minteken staan.

- *boolean denotations* zijn true en false.
  Zij moeten elk als één symbool worden opgevat; daarom zijn zij hier onderstreept. Een andere gebruikelijke representatie van deze symbolen is 'TRUE' en 'FALSE'. In deze zin moet ook in het vervolg het onderstrepen van rijtjes letters worden opgevat voorzover dit althans ALGOL 68-symbolen betreft.
- *character denotations* zijn karakters zoals we die kennen van een schrijfmachine of een regeldrukker, omgeven door aanhalingstekens, b.v.
  "a", "?", " ".
  In het laatste voorbeeld dient men tussen de aanhalingstekens een spatie te lezen. Een bijzonder geval doet zich voor als het bedoelde karakter een aanhalingstekens zou moeten zijn. In dit geval schrijft men twee opeenvolgende aanhalingstekens tussen aanhalingstekens, dus " " " ". In alle andere gevallen bestaat een character denotation precies uit één karakter tussen aanhalingstekens.

Bovengenoemde vier soorten denotations onderscheiden zich door hun *mode*. Deze modes zijn respectievelijk: integral, real, boolean en character. Uit het vervolg zal blijken dat er meer modes zijn en dat het begrip "mode" niet alleen voor het onderscheid tussen denotations wordt gebruikt.

3. IDENTIFIERS, VARIABELEN EN ASSIGNATIONS

Een rijtje letters en/of cijfers, beginnend met een letter heet *tag*. Wordt zo'n tag geassocieerd met een waarde dan noemen we hem *identifier*. Een belangrijke toepassing hiervan is het gebruik van een identifier als variabele. Aan een variabele kan door middel van een *assignation* een waarde worden toegekend.

Voorbeelden

```
i := 3
x := 3.14
b := true
c := "a"
```

Hierin moet ":=" als één symbool worden opgevat; het heet *becomes symbol* ofwel *wordtteken*. Rechts van het wordtteken staan in deze voorbeelden denotations van de mode integral, real, boolean, resp. character. Links van het wordtteken staan variabelen; deze variabelen kunnen zelf ook als waarden opgevat worden. Men maakt nu onderscheid tussen deze waarden en de waarden die aan de variabelen worden toegekend en zegt dat eerstgenoemde *refereren* naar laatstgenoemde. We noemen daarom de modes van de variabelen in dit voorbeeld respectievelijk *reference to integral*, *reference to real*, *reference to boolean* en *reference to character*.

Niet elke identifier stelt een variabele voor. Zo is bijvoorbeeld pi een identifier die de vaste waarde π heeft en daarom geen variabele is. De mode van pi is "real" en niet, zoals men misschien zou verwachten, "reference to real". De identifier pi kan niet dienen als linkerlid van een assignation. Wel is

```
x := pi,
```

waarin x de mode "reference to real" heeft, een geldige *assignation*. Hier is, evenals in de eerder gegeven voorbeelden, de mode van het linkerlid gelijk aan "reference to" gevolgd door de mode van het rechterlid.

Zoals opgemerkt, is een variabele i zelf ook als een waarde op te vatten en wel met een mode die begint met "reference to"; we veronderstellen dat i de mode "reference to integral" heeft. De vraag rijst nu of we deze waarde zelf ook weer door middel van een assignation aan een variabele ii zouden kunnen toekennen, die dan refereert naar de variabele i. Dit is inderdaad het geval; we kunnen schrijven

```
ii := i,
```

waarin i de mode "reference to real" en ii de mode "reference to reference to real" heeft.

Opmerking

Als men vertrouwd is met andere programmeertalen, dan is het misschien verhelderend op te merken dat bij de assignations

```
i := 3;
ii := i
```

de denotation 3 opgevat kan worden als *constante*, i als *variabele* en ii als *pointer*.

4. DECLARATIE VAN VARIABELEN; COMMENTAAR

In de vorige paragraaf waren i, x, b en c variabelen waaraan respectievelijk waarden van de modes integral, real, boolean en character konden worden toegekend. Dit kan in een programma worden aangegeven d.m.v. de *variable declaration*:

```
int i, real x, bool b, char c.
```

Een "variable declaration" kan gecombineerd worden met het toekennen van een beginwaarde, b.v.

```
int i := 3, real x, bool b := true, char c,
```

waardoor aan i en b een beginwaarde wordt gegeven. Heeft men meer dan één variabele van dezelfde mode te "declararen", b.v.

```
int i := 3, int j, int k, int 1 := 0, int m,
```

dan kan men een afgekorte vorm gebruiken:

```
int i := 3, j,k,1 := 0, m.
```

De komma als scheider van elementen van een lijstje, zoals gebruikt in bovenstaande voorbeelden, drukt uit dat de volgorde waarin deze elementen worden afgewerkt niet dezelfde behoeft te zijn als de tekstuele volgorde. Dit is wel het geval als de puntkomma i.p.v. de komma wordt gebruikt, hetgeen overigens, zoals later zal blijken, in de laatste "variable declaration"

niet mogelijk is. In bovenstaande voorbeelden speelt de genoemde volgorde geen enkele rol, zodat het effect van

```
int i := 3, real x, bool b := true, char c
```

hetzelfde is als het effect van de drie "variable declarations"

```
int i:= 3; real x, bool b := true; char c.
```

Later zullen we situaties tegenkomen waarin we een puntkomma niet straffeloos door een komma kunnen vervangen.

Commentaar

We hebben al aardig wat "symbolen" leren kennen, b.v. "int", ";", ":=", x. Voorafgaande aan een symbool mag, zonder dat de betekenis van het programma daardoor verandert, *commentaar* worden geschreven in de vorm

```
comment ..... comment, of
co      ..... co     , of
#       ..... #.
```

Op de plaats van ..... mag men alles invullen met uitzondering van de symbolen comment, co of #. Er is één plaats waar geen commentaar is toegestaan, n.l. binnen een denotation.

5. DEREFERENCING; READ EN PRINT

De laatste assignation van het stukje programma

```
int i,j;
i := 1;
j := i
```

heeft een bijzonderheid. De mode van het linkerlid j is *niet* "reference to" gevolgd door de mode van het rechterlid i. Kortheidshalve schrijven we van nu af aan niet alleen in de programmeervoorbeelden maar ook in de toelichtende tekst ref in plaats van "reference to", int in plaats van "integral", enz. In ons voorbeeld hebbem i en j de mode ref int. De variabele j kan

refereren naar een waarde van de mode int en niet naar een waarde van de mode ref int. Het is duidelijk dat de assignation pas kan worden uitgevoerd als het rechterlid i (mode ref int) is getransformeerd in de waarde waarnaar i refereert, dat is de waarde 1 (mode int). Deze transformatie heet *dereferencing*. Dereferencing is niet in elke context mogelijk. Het rechterlid van een assignation is een voorbeeld van een context waar dereferencing is toegestaan. Vooruitlopend op de stof noemen we nog een voorbeeld van zo'n context: een "actuele parameter" bij de z.g. "call" van een "procedure". Een voorbeeld vormt de procedure "print", die gebruikt wordt om o.a. getallen af te drukken. Schrijft men

```
int i := 1; print(1); print(i),
```

dan wordt twee keer het getal 1 afgedrukt. De tweede keer treedt dereferencing op.

Dereferencing kan ook herhaald optreden.

Voorbeeld:

```
int i := 1;
ref int ii := i;
print(ii);
```

Hier zien we voor het eerst de declaratie van een soort variabele die sommigen misschien "pointer" zouden willen noemen: ii heeft de mode ref ref int en refereert dus niet naar een getal maar naar een variabele die op zijn beurt naar een getal refereert. Omdat door "print" alleen waarden kunnen worden afgedrukt waarvan de mode niet met ref begint moet twee keer dereferencing op ii worden toegepast. Het resultaat is dat het getal 1 wordt afgedrukt. Laten we nu eens veronderstellen dat op het gegeven stukje programma volgt

```
i := 2; print(ii)
```

Hoewel nu aan ii zelf geen nieuwe waarde is toegekend sinds de vorige keer dat print(ii) werd uitgevoerd, wordt de tweede keer het getal 2 afgedrukt.

Men kan zich afvragen of

```
ii := 1
```

na de gegeven declaraties een legitieme "assignation" is. Het antwoord luidt ontkennend. Het linkerlid heeft de mode ref ref int, het rechterlid int. Overgang naar de vereiste situatie waarin de mode van het linkerlid precies met één ref meer begint dan de mode van het rechterlid is niet mogelijk omdat enerzijds op het linkerlid niet "dereferencing" mag worden toegepast en anderzijds het tegenovergestelde van "dereferencing" niet bestaat en dus ook niet op het rechterlid kan worden toegepast.

Naast de genoemde uitvoerprocedure "print" is er ook een invoerprocedure "read", waarmee o.a. getallen uit b.v. een ponskaart kunnen worden gelezen. Is b.v. het getal 5 in een ponskaart aan de beurt om gelezen te worden dan is het effect van

```
read(i)
```

dat i de waarde 5 krijgt, of, preciezer gezegd, dat i gaat refereren naar 5 en dat uiteraard de kaart a.h.w. een stukje opschuift zodat hierna een eventueel volgend getal kan worden gelezen.

De vraag rijst nu of na de gegeven declaraties ook

```
read(ii)
```

is toegestaan i.p.v. read(i). Dit is inderdaad het geval, want "dereferencing" kan immers op de actuele parameter ii worden toegepast. Doordat ii in dit voorbeeld refereert naar i is het effect van read(ii) hetzelfde als van read(i).

Uiteraard zal op in- en uitvoer, tezamen *transput* genoemd, nog worden teruggekomen.

6. ENIGE OPERATOREN

Operatoren dienen voor het vormen van *formules*. Zo is b.v.

```
a + 3 * b
```

een formule waarin de operatoren + en * voorkomen. Zoals men waarschijnlijk zal verwachten, gaat in de formule de vermenigvuldiging vóór de optelling. Om de volgorde van bewerking ook in minder eenvoudige situaties vast te leggen heeft, afgezien van z.g. monadische operatoren, waarover straks meer, elke operator zijn eigen prioriteitsnummer. Zo heeft b.v. vermenigvuldigen een hogere prioriteit dan optellen. We noemen in deze paragraaf operatoren uit de volgende tabel, waarin de prioriteit gegeven is:

operator	prioriteit
↑	8
*,/,over,mod	7
+,-	6
<,≤,≥,>	5
=,≠	4
and	3
or	2
+:=,-:=,*:=,/:=,overab,modab	1

De betekenis van een operator kan slechts gegeven worden als de mode van de "operands" waarop hij wordt toegepast gegeven is. Een aantal mogelijkheden zijn vermeld in de volgende tabel.

operator	mode 1e operand	mode 2e operand	mode resultaat	betekenis
↑	int	int	int	machtsverheffen: a↑b = a^b
	real	int	real	
*,+,-	int	int	int	vermenigvuldigen, optellen, aftrekken
	int	real	real	
	real	int	real	
	real	real	real	
/	int	int	real	delen
	int	real	real	
	real	int	real	
	real	real	real	

over	int	int	int	geheeltallige deling, 20 over 3 = 6, 20 over (-3)=-6, (-20) over 3=-6, (-20) over (-3)=6
mod	int	int	int	rest bij deling, 20 mod 3=2, 20 mod (-3)=2, (-20) mod 3=1, (-20) mod (-3)=1
<,≤,≥,>	int	int	bool	kleiner dan,
	int	real	bool	kleiner dan of gelijk aan,
	real	int	bool	groter dan of gelijk aan,
	real	real	bool	groter dan,
	char	char	bool	zie ook voorbeeld 4
=,≠	int	int	bool	
	int	real	bool	
	real	int	bool	gelijk aan,
	real	real	bool	ongelijk aan
	bool	bool	bool	
	char	char	bool	
and,or	bool	bool	bool	"en", "of"
*:=,+:=,-:=	ref int	int	ref int	
	ref real	int	ref real	
	ref real	real	ref real	operatoren gecombineerd met assignations, zie voorbeeld 5
/:=	ref real	int	ref real	
	ref real	real	ref real	
overab	ref int	int	ref int	
modab	ref int	int	ref int	

Voorbeeld 1: 3 = 6/2

Dit is een formule waarin de operatoren "=", met prioriteit 4, en "/", met prioriteit 7, voorkomen. Door dit verschil in prioriteit wordt eerst de deling uitgevoerd en wel met twee int-operanden, waarbij het resultaat volgens de tabel de mode real heeft. Een handige notatie hiervoor is (int,int) real, waarbij tussen de haakjes de modes van eerste en tweede operand staan en na het sluithaakje de mode van het resultaat. Dankzij de tweede mogelijkheid voor de "="-operator in de tabel kunnen we vervolgens

de (int,real) bool operator "=" toepassen.

Voorbeeld 2: 100/5 * 2

De operatoren "/" en "*" hebben dezelfde prioriteit (n.l. 7). Nu worden deze bewerkingen van links naar rechts uitgevoerd. Het resultaat is dus 40.0, waarbij deze schrijfwijze van het getal 40 de mode real suggereert, die inderdaad ten gevolge van de deling ontstaat.

Voorbeeld 3: 30 = 40 = 50 > 60

De uitkomst van dit nogal gezochte voorbeeld kan men vinden door de formule als volgt successievelijk te vereenvoudigen. De prioriteit van ">" is 5 en de prioriteit van "=" is 4, dus eerst ">" toepassen; dit geeft

30 = 40 = false .

Verdere vereenvoudiging, van links naar rechts wegens de gelijke prioriteit van beide "="-operatoren, geeft

false = false.

Door nu te letten op de vijfde mogelijkheid, d.i. (bool,bool) bool voor de "="-operator in de tabel zien we dat dit een geldige formule is. De uitkomst van

30 = 40 = 50 > 60

is dus de logische waarde true.

Voorbeeld 4: "a" < "b"

De operatoren =, ≠, <, ≤, ≥, > zijn volgens de tabel ook voor operands met mode char gedefinieerd. Hierbij is de betekenis van "=" en "≠" evident. De overige operatoren vereisen het bestaan van een *integral equivalent*, d.i. een uniek geheel getal voor elk karakter. De waarden hiervan worden niet in de taal maar door de implementatie (compiler) gedefinieerd. De genoemde

"relationele" operatoren nu worden op deze "integral equivalents" van de betrokken karakters toegepast in plaats van op de karakters zelf.

Voorbeeld 5:

```
i +:= 1;
kleine wijzer +:= d modab 12.
```

Het effect van deze beide formules is te beschrijven door

```
i := i+1;
kleine wijzer := kleine wijzer + d;
kleine wijzer := kleine wijzer mod 12.
```

Het verschil is evenwel dat de beide eerste vormen "formules" en de drie laatste "assignations" zijn. T.a.v. de plaats waar zij kunnen voorkomen maakt dit enig verschil uit, zoals later zal blijken.

Monadische operatoren

De tot nu toe besproken operatoren werden op twee operands toegepast; zij heten daarom *dyadische* operatoren. Er zijn ook z.g. *monadische* operatoren, die, zoals men al zal vermoeden, op één operand worden toegepast. Deze operand wordt geschreven vlak achter de operator, zoals b.v. in

```
-i.
```

Monadische operaties gaan vóór dyadische, zoals het volgende tabelletje met voorbeelden illustreert.

formule	betekenis
abs n * x	(abs n) * x
-2 * x	(-2) * x
-2 ↑ 2	(-2) ↑ 2
-2 * -3	(-2) * (-3)
- + - -3	-(+(-(-3)))

De twee laatste voorbeelden laten zien dat twee of meer operatoren op

elkaar kunnen volgen.

Evenals dit voor dyadische operatoren is gebeurd, geven we een voorlopige tabel met monadische operatoren.

operator	mode van operand	mode van resultaat	betekenis
+, -	int	int	de gebruikelijke betekenis
	real	real	
abs	int	int	de absolute waarde
	real	real	
	char	int	het "integral equivalent", zie voorbeeld 4
odd	int	bool	true als de operand oneven is; anders false
sign	int	int	+1 als de operand positief is 0 als de operand nul is
	real	int	-1 als de operand negatief is
entier	real	int	het grootste gehele getal dat niet groter dan de operand is
round	real	int	een geheel getal dat niet meer dan 0.5 verschilt van de operand
repr	int	char	het karakter waarmee het gegeven gehele getal equivalent is, zie ook abs
not	bool	bool	maakt van true false en omgekeerd

De operator abs illustreert treffend hoezeer de betekenis van een operator afhankelijk is van de mode van de operand(s). Het heeft daarom zin, het *aantal* operands en hun *modes* zozeer bij de definitie van de operand te betrekken dat b.v. "+" gezien wordt als één symbool dat verschillende operatoren representeert, waaronder die welke we hebben leren kennen en die we van elkaar onderscheiden door de notatie

```
(int,int) int
(int,real) real
(real,int) real
(real,real) real
(int) int
(real) real.
```

Uit het voorgaande kan, strikt genomen, nog niet worden opgemaakt of variabelen in plaats van denotations in formules zijn toegestaan, immers de mode van een variabele begint met ref en de meeste operands uit de gegeven tabellen hebben een mode die niet met ref begint. Daarom zij hier expliciet medegedeeld dat "dereferencing", voorzover nodig, ook op operands van toepassing is.

7. DE UNIT, SERIAL CLAUSE EN CONDITIONAL CLAUSE

We beginnen deze paragraaf met het volgende eenvoudige voorbeeld, waarvan de betekenis zich gemakkelijk laat raden.

```
if a < b
then c := 1;
     d := 2
else e := 3;
     f := 4
fi
```

Zoals de woorden "if", "then" en "else" suggereren, worden hier de assignations c := 1 en d := 2 uitgevoerd als a < b en worden de assignations e := 3 en f := 4 uitgevoerd als a ≥ b. Voor een algemenere behandeling van de z.g. "conditional clause", waarvan dit een voorbeeld is, voeren we eerst een paar nieuwe begrippen in. Allereerst keren we nog even terug tot de in paragraaf 3 besproken assignation. Over een assignation zoals b.v.

```
i := 3
```

kan iets meer worden opgemerkt dan dat hierdoor de variabele i gaat refe-

reren naar de waarde 3. Het is van belang te vermelden dat de assignation zelf ook een waarde "oplevert" en wel de waarde van het linkerlid. De mode van zo'n waarde begint dus steeds met ref. Een assignation, een denotation, een variabele, een formule, een call en nog meer begrippen die we zullen leren kennen, zijn bijzondere gevallen van een belangrijk begrip in de taal, namelijk de *unitary clause*, kortweg *unit* genoemd. Verder vermelden we hier dat het rechterlid van een assignation een "willekeurige" unit kan zijn (de aanhalingstekens om "willekeurige" hebben een zeer speciale bedoeling: er is hier een grens aan de "willekeur" en wel met betrekking tot de mode van de gekozen unit; men dient hier en in het vervolg "willekeurig" te lezen als "op de mode na willekeurig"). Uit de beide gegevens

1. een assignation is een unit,
2. het rechterlid van een assignation mag een "willekeurige" unit zijn,

volgt dat na de "variable declaration"

```
int i, j, k;
```

de volgende tekst een "assignation" is

```
i := j := k := 1,
```

immers "k := 1" is een assignation en dus een unit die na "j :=" kan komen enz.

Omdat de "unit" blijkbaar een nogal ruim begrip is, krijgt de lezer misschien de indruk dat elk op zichzelf staand brokje programmatekst een unit is. Deze indruk is onjuist. Zo is b.v. de "variable declaration"

```
int i
```

geen unit. Evenmin is het volgende stukje tekst een unit:

```
int i,j; i := 1; real x := 3.14; j := 2; print(i+j); i := j.
```

Dit stukje programmatekst is daarentegen een voorbeeld van een nieuw belangrijk begrip, de z.g. *serial clause*. Voorlopig definiëren we een serial

clause als een rijtje declaraties en units, waarvan de laatste een unit is, onderling gescheiden door puntkomma's. Een serial clause eindigt dus niet op een puntkomma. Dit "rijtje" kan ook uit één enkele unit bestaan. Hieruit volgt dat "serial clause" een nog ruimer begrip is dan "unit"; een unit is een bijzonder geval van een serial clause. Een serial clause levert zelf ook een waarde op, en wel de waarde die door de laatste unit wordt opgeleverd. Zo is

```
a := x; b := y+z; a > b
```

een voorbeeld van een serial clause die een waarde oplevert waarvan de mode bool is.

We zijn nu in staat de algemene gedaante van de *conditional clause* te geven, n.l.

```
if "serial clause met bool resultaat"
then "serial clause A"
else "serial clause B"
fi,
```

waarvoor de volgende verkorte notatie is toegestaan

```
( "serial clause met bool resultaat"
| "serial clause A"
| "serial clause B"
) .
```

Verder kan het gedeelte

```
else "serial clause B", resp.
 |   "serial clause B"
```

worden weggelaten als er geen behoefte aan is. Enige voorbeelden van de conditional clause zijn

```
if real x; read (x); x > 0 then x else -x fi
(p > q | p | q)
if x < 0 then x := 0; fout := true fi
(x+y > 100 | print(z))
```

Een conditional clause is zelf ook een bijzonder geval van een unit. Het is zelfs een unit die op alle plaatsen gebruikt kan worden, waar b.v. een variabele of een denotation is toegestaan. Dit kan uiteraard slechts als de mode van een conditional clause gedefinieerd is en aan zekere voorwaarden voldoet. Hierop zal nog worden teruggekomen. De volgende voorbeelden illustreren het gebruik van een conditional clause.

```
x := y + (p > q | p | q)
(p > q | print(p); p | print(q);q) := 0.
```

Er is vaak behoefte aan de volgende constructie

if ... then ... else if ... then ... else if ... then ... else ... fi fi fi.

Daarom is hiervoor een afgekorte vorm ingevoerd, n.l.

if ... then ... elif ... then ... elif ... then ... else ... fi

of:

(... | ... |: ... | ... |: ... | ... | ...) .

8. DE CLOSED CLAUSE, DE CASE CLAUSE EN DE LOOP CLAUSE

De behoefte kan zich voordoen een "willekeurige" serial clause te gebruiken op een plaats waar slechts een eenvoudige vorm van een unit, zoals een variabele of een denotation, is toegestaan. Een operand in een formule is een voorbeeld van zo'n plaats. Er is een bijzonder eenvoudige manier om in deze behoefte te voorzien. Deze bestaat uit het plaatsen van haakjes om de betreffende serial clause. Hierbij kan men nog kiezen tussen ronde haakjes "(" en ")" en de symbolen begin en end. De vorm die hierdoor ontstaat heeft *closed clause*. Voorbeelden van de closed clause:

```
(a+b)
begin i := 1; i + 1 end .
```

Men kan een closed clause b.v. als volgt gebruiken:

```
x := y * (real z; read(z); z+3) ↑ 2;
print((read(y); y)).
```

Een closed clause is, evenals de conditional clause overal toegestaan waar een variable of een denotation mag staan.

Hetzelfde kan gezegd worden van de nu te noemen *case clause*. Een case clause kan een bepaald type van de conditional clause vervangen, waardoor een eenvoudiger vorm ontstaat. Een voorbeeld van zo'n conditional clause is

```
if i = 1 then 3.14
elif i = 2 then 0
elif i = 3 then x
elif i = 4 then read(z); z
else -1
fi .
```

De vervangende "case clause" luidt

```
case i
in 3.14, 0, x+y, (read(z); z) out -1
esac,
```

of, in verkorte vorm

```
(i |3.14,0,x+y, (read(z);z) | -1) .
```

Na in (of de eerste vertikale streep) staan een aantal "willekeurige" units, gescheiden door komma's. Tussen case en in (of tussen "(" en "|") staat een serial clause die, eventueel door middel van dereferencing, een waarde van de mode int oplevert. Deze waarde noemen we hier, in overeenstemming met het voorbeeld, i. Als nu i tenminste gelijk aan 1 en ten hoogste gelijk aan het genoemde aantal units is, wordt uitsluitend de i-de unit gekozen. Voldoet i hieraan niet, dan wordt

het deel tussen out en esac (of tussen de tweede vertikale streep en ")") gekozen, indien dit althans aanwezig is. Dit laatste deel mag een "willekeurige" serial clause zijn. Het deel "out serial clause" (of de corresponderende verkorte vorm) mag weggelaten worden als er geen behoefte aan is. Evenals bij de conditional clause is er ook een verkorte vorm voor een samengestelde case clause, die door het volgende voorbeeld duidelijk wordt:

```
case i
in 38, 45, 87, 91
out  case j
     in 21, 67, 42
     out 15
     esac
esac
```

kan worden vervangen door

```
case i
in 38, 45, 87, 91
ouse j
in 21, 67, 42
out 15
esac.
```

In de korte notatie betekent dit dat

```
(i|38,45,87,91|(j|21,67,42|15))
```

vervangen kan worden door

```
(i|38,45,87,91|:j|21,67,42|15).
```

De tot nu toe besproken middelen bieden nog geen gelegenheid een programmadeel meer dan eens te "doorlopen". Dat hieraan behoefte is blijkt uit het volgende voorbeeld. Gevraagd wordt een programma te maken dat een rij positieve getallen leest en hun som afdrukt. De lengte van deze rij is onbekend, maar gegeven is dat de rij gevolgd wordt door een niet-positief getal.

Een programma dat hiervoor gebruikt kan worden is het volgende:

```
begin int som := 0,x;
      while read(x); x > 0
      do som +:= x
      od;
      print(som)
end .
```

Het deel vanaf while t/m od is een voorbeeld van een *loop clause*. De serial clause tussen do en od wordt uitgevoerd zolang de serial clause tussen while en do (zonodig via dereferencing) de waarde true oplevert. Zodra laatstgenoemde serial clause de waarde false oplevert, hetgeen de eerste keer al zo zou kunnen zijn, wordt de loop clause meteen beëindigd. Omdat het vaak zin heeft het herhaald doorlopen van de serial clause tussen do en od in verband te brengen met het doorlopen van een aantal opeenvolgende termen van een rekenkundige rij, is de algemene gedaante van een loop clause

for identifier from unit 1 by unit 2 to unit 3 while serial clause 1
do serial clause 2
od.

De identifier na for wordt niet vooraf gedeclareerd. Zijn mode is int en niet ref int, zodat deze identifier niet als linkerlid van een assignation binnen de loop clause kan optreden. Door unit 1, unit 2 en unit 3 moeten, eventueel na dereferencing, waarden van de mode int worden afgeleverd; door serial clause 1 moet, eventueel na dereferencing, een waarde van de mode bool worden afgeleverd.

De werking van de loop clause is als volgt. Begonnen wordt met de waarden van unit 1, unit 2 en unit 3, in een niet gedefinieerde volgorde, te berekenen. We zullen deze geheeltallige waarden resp. v_1, v_2 en v_3 noemen. Vervolgens wordt serial clause 2 uitgevoerd voor de volgende waarden van de genoemde identifier:

$$v_1,\ v_1 + v_2,\ v_1 + 2v_2,\ \ldots,\ \text{eindwaarde},$$

waarbij "eindwaarde" bepaald wordt door het nog net niet overschrijden (of, als $v_2 < 0$,"onderschrijden") van v_3, indien althans voor al deze waarden door serial clause 1 de waarde true wordt opgeleverd. Zodra door serial clause 1 de waarde false wordt opgeleverd wordt serial clause 2 niet meer uitgevoerd, waarna de loop clause beëindigd is.

Als in een loop clause "from 1" zou voorkomen, dan kan dit worden weggelaten zonder dat de werking verandert. Hetzelfde geldt voor "by 1" en "while true". Zo worden b.v. door

```
for j to 100 do print (j↑2)
```

de waarden van 1^2, 2^2, ..., 100^2 afgedrukt.

Het is ook denkbaar dat men in serial clause 2 geen behoefte heeft aan een identifier die de opgegeven waarden doorloopt. In dit geval kan ook het deel "for identifier" worden weggelaten, b.v.

```
to 3 do read(x); s +:= x od
```

zorgt ervoor dat drie getallen worden gelezen die elk bij s worden opgeteld.

Zoals al in ons eerste voorbeeld van de loop clause is getoond, is het ook mogelijk de"besturing" van de loop clause uitsluitend aan "serial clause 1" over te laten en de loop clause met while te laten beginnen.

Tenslotte zij nog vermeld dat de loop clause ook slechts de gedaante

```
do serial clause od
```

kan hebben. Zonder voorzorgen wordt in dit geval de serial clause oneindig vaak uitgevoerd. Welke voorzorgen hier bedoeld worden zal later, bij de behandeling van de "jump", duidelijk worden.

9. DE MULTIPLE VALUE, DE STRUCTURED VALUE EN DE COLLATERAL CLAUSE

Tot nu toe hebben we geen praktische middelen genoemd om met lange rijen getallen te manipuleren. Daarom kunnen we nog niet op eenvoudige wijze het volgende probleem aan. Gevraagd wordt een rij van 1000 gehele getallen in de gewone volgorde, d.w.z. van voren naar achteren, in te lezen en ver-

volgens dezelfde rij van achteren naar voren af te drukken. Het volgende programma toont het gebruik van een nieuw begrip, de z.g. *multiple value*, waarmee aan de gestelde vraag wordt beantwoord:

```
begin [1:1000] int x;
      for i to 1000 do read(x[i]) od;
      for i from 1000 by -1 to 1 do print(x[i]) od
end.
```

Hier is x een variabele die refereert naar een "multiple value". Zijn mode heet officieel "reference to row of integral", hetgeen wij verkort noteren als ref[ ]int. Een assignation waarin deze x als linkerlid optreedt is mogelijk, mits, eventueel na dereferencing, een passend rechterlid met mode "[ ] int" beschikbaar is. Naast x zelf zijn ook x[1], x[2],..., x[1000] variabelen. Hun mode is ref int. Tussen de vierkante haken, *sub symbol*, resp. *bus symbol* geheten, mogen willekeurige units staan, mits de mode, eventueel na "dereferencing", int is. Dit geldt ook voor de "grenzen" die in de declaratie van x ter weerszijden van de dubbele punt staan.

Het volgende programma doet hetzelfde als bovenstaand programma, met dien verstande dat, in plaats van 1000, de rij nu een willekeurige lengte n heeft. Het getal n moet vooraf worden ingelezen. Om illustratieve redenen is een vorm gekozen die sterker van bovenstaand programma afwijkt dan nodig is.

```
( int n;
  [1:(read(n); n)] int x,y;
  read(x); y := x;
  for i to n do print(y[n+1-i])
).
```

De unit die in de "variable declaration" op de dubbele punt volgt bestaat uit de closed clause

```
(read(n); n).
```

Hierdoor wordt, hoewel er twee variabelen, n.l. x en y, worden gedeclareerd, slechts één getal n gelezen, dat als waarde van de closed clause wordt op-

geleverd. De call "read(x)" laat zien dat ook een multiple value in zijn geheel kan worden ingelezen. In de hierop volgende assignation "y := x" wordt de multiple value, die na dereferencing van x ontstaat, in zijn geheel aan de variabele y toegekend. Hierna laat een loop clause de waarden van de multiple value in omgekeerde volgorde afdrukken: als i loopt van 1 naar n, loopt n+1-i van n naar 1. De unit, zoals hier n+1-i, die gebruikt wordt om een element van een multiple value aan te wijzen heet *subscript*. Het aantal subscripts van een multiple value behoeft niet één te zijn zoals in de gegeven voorbeelden, maar is willekeurig. In het volgende voorbeeld komen multiple values met één en met twee subscripts voor. Het programma leest een m × n-matrix A rij voor rij en daarna een vector b in en drukt de vector c = Ab af. Vooraf worden m en n ingelezen.

```
begin int m,n; read(m); read(n);
      [1:m,1:n] real a, [1:n] real b, [1:m] real c;
      read(a); read(b);
      for i to m
      do c[i] := 0;
          for j to n do c[i] +:= a[i,j] * b[j] od
      od;
      print(c)
end.
```

We hebben nu gezien hoe zowel met de multiple value in zijn geheel als met de elementen afzonderlijk kan worden gewerkt. Er is nog een ander middel om een welomschreven deel van een multiple value als één geheel d.w.z. als een nieuwe multiple value te hanteren. Laten we bijvoorbeeld eens veronderstellen dat de uit 10 rijen en 8 kolommen bestaande matrix A gedeclareerd is als

```
[1:10,1:8] real a
```

en dat we de derde kolom willen vervangen door de kolomvector b, gedeclareerd als

```
[1:10] real b.
```

Men kan dit gedaan krijgen door de assignation

```
a[ ,3] := b.
```

Het linkerlid is een nieuwe variabele die refereert naar een multiple value met één subscript met ondergrens 1 en bovengrens 10 en waarvan de elementen de mode real hebben. Dit komt goed overeen met het rechterlid zoals uit de declaratie van b blijkt en de assignation kan worden uitgevoerd. Zou men daarentegen schrijven

```
a[3, ] := b,
```

dan zou het linkerlid een bovengrens 8 en het rechterlid een bovengrens 10 impliceren. Daarom is dit bij de gegeven declaratie van b geen geldige assignation.

De voorlaatste assignation had ook geschreven kunnen worden als

```
a[1:10,3] := b, of
a[    ,3] := b[1:10], of
a[1:10,3] := b[1:10].
```

In al deze gevallen impliceert zowel het linkerlid als het rechterlid een multiple value met één subscript met ondergrens 1 en bovengrens 10.

We veronderstellen nu dat bovendien gedeclareerd is

```
[1:5] real c.
```

Willen we nu de eerste vijf elementen van de derde kolom van a vervangen door c, dan kan dit door

```
a[1:5,3] := c.
```

Het vervangen van de laatste vijf elementen van de vierde kolom van a door c gaat aldus

```
a[6:10,4] := c.
```

We zien nu iets nieuws! Opdat de geïmpliceerde onder- en bovengrenzen van

het linkerlid en het rechterlid in een geval als dit gelijk zullen zijn, hetgeen bij een assignation vereist is, heeft men de afspraak gemaakt dat de in "6:10" genoemde grenzen worden "omgenummerd vanaf 1", d.w.z. de ondergrens wordt 1 en de bovengrens 5. Dit gebeurt uiteraard zonder dat de gebruiker hier hinder van ondervindt, d.w.z. wel degelijk worden de laatste vijf elementen van de vierde kolom vervangen.

Zou gedeclareerd zijn

```
[3:7] real d,
```

dan zou "omnummering vanaf 1" bij de assignation

```
a[6:10,4] := d[3:7]
```

zowel in het linkerlid als in het rechterlid plaatsvinden, waardoor de laatste vijf elementen van de vierde kolom van a door de vijf elementen van d worden vervangen. We mogen nu evenwel niet schrijven

```
a[6:10,4] := d,
```

omdat hier "omnummering vanaf 1" wel in het linkerlid maar niet in het rechterlid zou plaatsvinden. Het kan wel eens prettig zijn, "omnummering vanaf een andere waarde dan 1" te laten plaatsvinden. Daarom is deze mogelijkheid inderdaad in de taal aanwezig; in plaats van laatstgenoemde "ongeldige assignation" kan men schrijven

```
a[6:10 at 3,4] := d.
```

Nu vindt in het linkerlid "omnummering vanaf 3" plaats, waardoor overeenstemming met het rechterlid wordt bereikt.

De technische term voor een uitdrukking van de vorm

```
× × × [...],
```

zoals a[i,j], b[j], a[3,], a[1:10,3], a[6:10 at 3,4]is *slice*. Wordt in een slice een dubbele punt gebruikt, zoals in a[1:10,3] en a[6:10 at 3,4], dan spreekt men van *trimming*. De algemene term voor hetgeen tussen sub- en bus-

symbol staat is *indexer*.

Zoals we gezien hebben, zijn de modes van de elementen van een multiple value onderling gelijk en wordt een element aangewezen door middel van een of meer gehele getallen, de z.g. *subscripts*. Er is in de taal ook een middel om een verzameling van elementen met eventueel verschillende modes als één geheel op te vatten. Men noemt zo'n verzameling een *structured value*. De afzonderlijke elementen heten *velden*. Om zo'n veld aan te wijzen, of, zoals men zegt, te *selecteren*, gebruikt men een tag (zie par. 3). Zo wordt bijvoorbeeld door de declaratie

```
struct (int i, real r, char c, [1:5] int a) s;
```

een variabele s gedeclareerd, die refereert naar een structured value met vier velden. De "field selectors" zijn de tags i, r, c en a; zij maken deel uit van de mode van de structured value, hetgeen betekent, dat de mode van s verschilt van de mode van s1 gedeclareerd door

```
struct (int i1, real r1, char c1, [1:5] int a1) s1;.
```

De mode van s noteren we als

```
ref struct (int i, real r, char c, [ ] int a).
```

Het volgende voorbeeld laat zien hoe we zowel met de gehele structured value als met de velden afzonderlijk kunnen manipuleren.

```
begin struct (int i, real r, char c, [1:5] int a) soud, snieuw;
      read (soud);
      if i of soud > 1000
      then i of snieuw := 0;
           r of snieuw := r of soud;
           c of snieuw := "a";
           a of snieuw := a of soud;
           (a of snieuw)[3] := 2 * (a of snieuw)[5]
      else snieuw := soud
      fi;
      print(snieuw)
end.
```

De uitdrukking

```
i of soud
```

is een voorbeeld van een z.g. *selection*. Het heeft de mode ref int. Evenzo zijn de modes van

```
a of snieuw        en        (a of snieuw)[3]
```

respectievelijk ref [ ] int en ref int.

Het symbool of heeft de voor de hand liggende benaming *of symbol*. Waarom in b.v.

```
(a of snieuw)[3]
```

haakjes gebruikt moeten worden zal in de volgende paragraaf worden verklaard.

Zowel bij een multiple value als bij een structured value is vaak de behoefte aanwezig, de componenten waaruit hij bestaat in compacte vorm op te schrijven. Dit kan zoals aangegeven in het volgende voorbeeld:

```
[1:5] int x; x := (100,200,300,400,500);
 struct (int i, real r, char c, [1:5] int a) s;
 s := (1, 3.14, "a", (100,200,300,400,500)),
```

of, veel korter,

```
[1:5] int x := (100,200,300,400,500);
struct (int i, real r, char c, [1:5]int a) s := (1, 3.14, "a", x).
```

Een uitdrukking als

```
(100,200,300,400,500),
```

die in het algemeen bestaat uit twee of meer units gescheiden door komma's en omgeven door een paar haakjes, of door begin en end heet *collateral clause*. De volgorde waarin deze units worden doorlopen is niet gedefinieerd. Een collateral clause stelt, afhankelijk van de context, ofwel een multiple value, ofwel een structured value voor.

10. OVERZICHT VAN ENIGE GRAMMATICALE BEGRIPPEN

Begrippen als unit, serial clause en slice heeft men nodig om te kunnen definiëren welke constructies zijn toegestaan en wat hun betekenis is. Daarom geven we hier een samenvatting van deze begrippen en hun onderlinge samenhang voorzover zij behandeld zijn. Het is dus slechts een voorlopig overzicht dat geleidelijk aan nog wat zal worden uitgebreid en verfijnd. We maken hierbij gebruik van z.g. grammaticaregels of produktieregels, een notatie die na het geven van dit overzicht zal worden verklaard.

program: closed clause.
closed clause: begin symbol, serial clause, end symbol;
open symbol, serial clause, close symbol.
serial clause: "door puntkomma's gescheiden declaraties en units
eindigend op een unit".
unit: denotation; identifier; slice; call; closed clause; conditional
clause; collateral clause; case clause; loop clause;
assignation; formula; selection.

In deze produktieregels staat na de dubbele punt een nadere concretisering van het begrip vóór de dubbele punt. Hierbij moet men een komma lezen als "gevolgd door" en een puntkomma als "of". Een punt markeert het einde van een produktieregel.

De eerste regel drukt uit dat we een programma hebben leren kennen in de vorm van een closed clause.

De volgende regel toont twee alternatieven voor een closed clause; men kan de hierin voorkomende serial clause n.l. omgeven met begin en end òf met een rond open- en sluithaakje.

De derde regel is nog niet in de gewenste vorm. We merken enerzijds op dat een serial clause kan bestaan uit een enkele unit en anderzijds dat we links aan een serial clause een declaratie gevolgd door een puntkomma of een unit gevolgd door een puntkomma kunnen toevoegen. We krijgen aldus de verbeterde regel, waarin "go on symbol" "puntkomma" betekent:

serial clause: unit;
declaration, go on symbol, serial clause;
unit, go on symbol, serial clause.

In de vierde regel zijn twaalf bijzondere gevallen van het begrip "unit" gegeven. Deze twaalf soorten units zijn niet gelijkwaardig t.a.v. de plaatsen waar zij kunnen voorkomen. Zo is bijvoorbeeld als operand van een formule wel een denotation maar geen assignation toegestaan. Het heeft daarom zin de volgende hiërarchische verfijning aan te brengen:

```
unit: loop clause; assignation; tertiary.
tertiary: formula; secondary.
secondary: selection; primary.
primary: denotation; identifier; slice; call; closed clause;
        conditional clause; case clause; collateral clause.
```

Voor wie met zo'n stelsel produktieregels moeite heeft is het volgende voorbeeld misschien verhelderend:

```
dier: zoogdier; vogel; vis; ander dier.
vogel: bosvogel; watervogel; andere vogel.
watervogel: eend; zwaan; andere watervogel.
```

Even goed als men uit deze regels afleidt dat een eend een watervogel en dus een vogel en dus een dier is, volgt ook uit onze produktieregels dat b.v. een denotation een primary en dus een secondary en dus een tertiary en dus een unit is.

Aan de hand van dit overzicht kunnen we preciezere definities geven van eerder behandelde begrippen, waarbij we weer gebruik maken van produktieregels:

```
assignation: tertiary, becomes symbol, unit.
formula: monadic operator, operand;
        operand, dyadic operator, operand.
operand: formula; secondary.
selection: tag, of symbol, secondary.
slice: primary, sub symbol, indexer, bus symbol.
```

Bij al deze regels zoals ze hier zijn geformuleerd geldt impliciet een restrictie t.a.v. de mode. Zo is b.v. 5 := 5 geen assignation, ofschoon dit volgens de gegeven produktieregels wel het geval zou zijn. Een terwille van de eenvoud niet in de produktieregels verwerkte voorwaarde, waaraan niet

is voldaan, is de eis dat de mode van de tertiary vóór het wordtteken met ref moet beginnen. In deze paragraaf wordt hierop niet verder ingegaan, maar stilzwijgend aangenomen dat aan zulke voorwaarden t.a.v. de mode is voldaan.

Uit de produktieregels voor selection en slice volgt de betekenis van

a of b[i].

Zouden we deze vorm willen lezen als (a of b)[i], d.w.z. als een slice, dan is dit strijdig met de eis dat in een slice een primary vóór het sub symbol moet staan, immers, a of b is een selection en dus geen primary. Nu proberen we de gegeven vorm te lezen als a of (b[i]), d.w.z. als een selection. In een selection moet op "of" een secondary volgen. Nu is b[i] een slice en dus een primary en dus een secondary, dus deze interpretatie is inderdaad correct.

Op analoge wijze proberen we de formule

a+b↑c

eenduidig te ontleden. Nu blijkt dat door toepassing van de produktieregels voor formula en operand zowel de interpretatie (a+b)↑c als a+(b↑c) mogelijk is, zodat met gebruik van deze regels de ontleding allerminst eenduidig is. Om dit in orde te krijgen dienen de in paragraaf 6 genoemde prioriteiten in de beschouwing te worden betrokken. Hoewel het voor het programmeren niet nodig is, is het wellicht leerzaam te laten zien hoe dit met behulp van de volgende produktieregels mogelijk is.

P-formula: P-operand, P-operator, (P+1)-operand. (P = 1,2,...,9)
10-formula: monadic operator, 10-operand.
P-operand: P-formula;
(P+1)-operand. (P = 1,2,...,10)
11-operand: secondary.

Hierin is de eerste regel in feite een verkorte notatie voor negen verschillende regels die men verkrijgt door de opgegeven waarden van P in te vullen. Evenzo staat de derde regel voor tien verschillende regels.

Om nu een gegeven formule met behulp van deze regels te ontleden begint men met te noteren welke prioriteiten de dyadische operatoren in deze formule hebben, voorzover de formule althans dyadische operatoren bevat. Voor ons voorbeeld

a+b↑c

vinden we in paragraaf 6 dat + een 6-operator en ↑ een 8-operator is. Zouden we nu de gegeven vorm willen lezen als (a+b)↑c, dan moeten we de eerste regel met P = 6 toepassen op a+b, zodat a+b een 6-formule zou zijn. Vervolgens zouden we wederom de eerste regel, maar nu met P = 8, moeten toepassen voor de machtsverheffing. Dan zou dus a+b moeten fungeren als 8-operand. Dit gaat niet; het is niet mogelijk de produktieregels zo toe te passen dat een 6-formule een bijzonder geval van een 8-operand is.

Nu de interpretatie a+(b↑c). Doordat ↑ een 8-operator is, is b↑c een 8-formule en dus, volgens de derde regel een 8-operand en dus, volgens dezelfde regel met P = 7, een 7-operand. Dit is juist de gedaante waarin we b↑c nodig hebben om de eerste regel met P = 6 te kunnen toepassen, want + is een 6-operator. We dienen nu nog te verifiëren dat a een bijzonder geval van een 6-operand is. Nu is a een identifier en dus een primary en dus een secondary, en dus, volgens de vierde regel een 11-operand en dus, volgens de derde regel met P = 10, een 10-operand. Door nu nog de derde regel achtereenvolgens met P = 9,8,7 en 6 toe te passen zien we dat a inderdaad een 6-operand is. Dus is a+b↑c een 6-formule, met de betekenis a+(b↑c).

We ontleden tenslotte nog de formule

a-b-c.

We slaan weer eerst de verkeerde weg in om te zien of deze doodloopt en proberen deze formule te lezen als a-(b-c). Nu is, omdat - een 6-operator is, b-c een 6-formule en deze kan niet fungeren als 7-operand. Dus deze ontleding lukt niet. De interpretatie (a-b)-c daarentegen blijkt goed te zijn: a-b is een 6-formule en dus een 6-operand en kan daarom als één geheel links van het tweede minteken worden opgevat. Na dit minteken staat c en zo'n identifier kan, zoals eerder is gebleken, als 7-operand fungeren.

Aan het eind van deze paragraaf vatten we de gegeven produktieregels in de herziene versie samen. Verdere uitbreiding en verfijning hiervan zullen nog volgen.

```
program       : closed clause.
closed clause: begin symbol, serial clause, end symbol;
               open symbol, serial clause, close symbol.
serial clause: unit;
               declaration, go on symbol, serial clause;
               unit, go on symbol, serial clause.
unit          : loop clause;
               assignation;
               tertiary.
tertiary      : P-formula;
               secondary.
secondary     : selection;
               primary.
primary       : denotation;
               identifier;
               slice;
               call;
               closed clause;
               conditional clause;
               case clause;
               collateral clause.
assignation   : tertiary, becomes symbol, unit.
P-formula     : P-operand, P-operator, (P+1)-operand. (P = 1,2,...,9)
10-formula    : monadic operator, 10-operand.
P-operand     : P-formula;
               (P+1)-operand.                          (P = 1,2,...,10)
11-operand    : secondary.
selection     : tag, of symbol, secondary.
slice         : primary, sub symbol, indexer, bus symbol.
```

11. JUMPS; skip; COMPLETERS

De meeste programmeertalen kennen de goto-statement of sprong, hier *jump* genaamd, die beschouwd kan worden als een overblijfsel uit het machinecodetijdperk. Theoretisch kan men jumps steeds vermijden. Dit bevordert meestal de doorzichtigheid van een programma, maar er blijven gevallen denkbaar waar dit niet opgaat. Een eenvoudig voorbeeld waar, onnodig, de jump wordt gebruikt, is:

```
begin int som := 0,x;
terug: read(x);
       if x > 0 then som +:= x; goto terug fi;
       print(som)
end
```

Een programma zonder jump dat hetzelfde doet is besproken bij de introductie van de loop clause in paragraaf 8. In plaats van de jump "goto terug" had ook alleen "terug" kunnen staan: het symbool goto mag in een jump worden weggelaten.

In het volgende programma wordt de jump gebruikt om het programma te beëindigen. Ook dit programma vormt de som van een rij in te lezen positieve gehele getallen.

```
begin int som := 0,x;
      do read(x);
         if x ≤ 0 then goto klaar fi
         s +:= x
      od;
klaar: skip
end.
```

Dit programma verduidelijkt meteen een opmerking over deze simpele vorm van de loop clause, gemaakt aan het eind van paragraaf 8. Een jump is een nieuw bijzonder geval van een unit. Hetzelfde kan gezegd worden van de z.g. *skip* aan het eind van dit programma, die hier om formele redenen noodzakelijk is: een serial clause moet op een unit eindigen en "klaar:" is geen

unit maar een z.g. *label definition*, waarop nog zal worden teruggekomen. Men kan skip overal gebruiken waar om "syntactische" redenen, d.w.z. volgens de produktieregels, een unit nodig is die niet op natuurlijke wijze geleverd wordt.

De completer

Als men wil aangeven dat de uitvoering van een serial clause beëindigd moet worden, kan men gebruik maken van het symbool exit, dat *completion symbol* heet. De volgende serial clause levert de waarde true op als de in te lezen vector a het getal 8 bevat en levert anders de waarde false op:

```
        int n; read(n);
        [1:n] int a; read(a);
        for i to n
        do if a[i] = 8 then goto succes fi
        od;
        false exit
succes: true
```

Eerder is opgemerkt dat de waarde van een serial clause de waarde van de "laatste unit" van deze serial clause is. We zien nu dat hierbij de uitdrukking "laatste unit" niet in tekstuele maar in dynamische zin moet worden opgvat. Als in het voorbeeld de waarde 8 niet in a voorkomt, is de denotation false op de voorlaatste regel, dynamisch gesproken, de laatste unit van de serial clause. Om het punt na exit te kunnen bereiken moet exit steeds door een label definition, zoals hier "succes:" worden gevolgd. Het geheel, d.w.z. exit gevolgd door label definition wordt *completer* genoemd.

Door de introductie van de begrippen "label definition" en "completer" voldoet de gegeven produktieregel voor "serial clause" niet meer. We geven hiervoor in de plaats het volgende stel produktieregels, dat daarna kort wordt toegelicht.

```
serial clause: parade;
               prologue, go on symbol, parade.
prologue     : declaration;
               unit, go on symbol, prologue;
               declaration, go on symbol, prologue.
parade       : unit;
               unit, go on symbol, parade;
               label definition, parade;
               unit, completer, parade.
completer    : completion symbol, label definition.
```

Dat deze uitbreiding nogal aanzienlijk is komt vooral door een eis die nog niet is genoemd, n.l. dat bij het samenstellen van een serial clause geen label definition vóór een declaration mag komen. Het deel van de serial clause tot en met de laatste declaration noemen we *prologue*. Na de puntkomma die hierop volgt komt een z.g. *parade*. Deze eindigt op een unit en kan, slordig gezegd, aan de voorkant met label definitions, units en completers worden uitgebreid.

12. RANGES EN SCOPES

Het declareren van een identifier en het optreden van een identifier in een label definition zullen we beide de *definitie* van deze identifier noemen. Elk ander gebruik van deze identifier heet een *toepassing* ervan. Het is nu nodig bij een definitie van een identifier vast te leggen waar in het programma gebruik van deze definitie kan worden gemaakt, d.w.z. waar deze identifier kan worden toegepast. Hiertoe zullen we eerst het begrip *range* invoeren. Een serial clause die ontstaat door de buitenste haken van een closed clause te verwijderen en waarvan een declaratie of een label definition een van de samenstellende delen is, is een voorbeeld van een "range". Betreft nu zo'n declaratie of label definition de definitie van b.v. de identifier i, dan kunnen toepassingen van deze i overal in de genoemde range voorkomen, met uitzondering van kleinere ranges die erin bevat zijn en waarin bovendien een nieuwe definitie van i voorkomt.

Voorbeeld

```
 1  begin
 2     int i; i := 1; print(i);
 3     begin
 4        real i := 3.14;
 5        print(i)
 6     end;
 7     begin int j := 2;
 8           print(i)
 9     end;
10     print(i)
11  end.
```

In dit voorbeeld behoren de regelnummers niet tot het programma. De op regel 2 gedefinieerde identifier i kan worden toegepast binnen de range gevormd door de regels 2 t/m 10, met uitzondering van de uit de regels 4 en 5 bestaande kleinere range omdat daarin i opnieuw wordt gedefinieerd. Op regel 5 heeft de binnenste range voorrang boven de buitenste, zodat 3.14 wordt afgedrukt. Op regel 8 bevinden we ons weliswaar in een kleinere range maar hier is i niet opnieuw gedefinieerd, zodat dit een toepassing van de "buitenste" i is. Door de regels 2, 5, 8 en 10, worden respectievelijk de waarden

1, 3.14, 1, 1

afgedrukt.

Behalve voor een identifier is ook voor een waarde een gebied in het programma gedefinieerd waar deze waarde kan worden gebruikt. Dit gebied heet de *scope* van die waarde. Heeft een waarde een eenvoudige mode, zonder ref, b.v. int, []bool, struct(real r, char c), dan is de scope van die waarde het hele programma, zodat in dit geval het begrip "scope" slechts een triviale betekenis heeft. Zo is b.v. het programma

```
begin int i;
      begin int j := 2;
            i := j
      end;
      print(i)
end
```

correct; de waarde 2 kan niet alleen in de binnenste range, maar ook daarbuiten worden gebruikt, zoals hier gebeurd is bij het afdrukken. Het wordt anders als de mode van een waarde met ref begint; de technische term voor zo'n waarde is *name*. Afgezien van een uitzondering die in de volgende paragraaf wordt behandeld, is de scope van een name de kleinste range waarin hij ontstaat. Daarom is

```
begin ref int ii;
      begin int j := 2;
            ii := j
      end;
      print(ii)
end
```

fout. Nadat de binnenste range verlaten is, kan j, opgevat als "name", d.w.z. als waarde van de mode ref int, niet meer worden gebruikt. Door print(ii) zou op (ii) twee keer dereferencing moeten worden toegepast. Bij de eerste keer zou de waarde van j (opgevat als name) moeten ontstaan en deze is er niet meer. Om deze moeilijkheid te voorkomen wordt verboden dat bij een assignation de scope van de waarde die moet worden toegekend aan het linkerlid kleiner is dan de scope van het linkerlid. Men lette op een belangrijk verschil tussen de beide laatste programma's. In het eerste treedt bij i := j dereferencing op; niet de waarde j zelf maar de waarde 2, waarvan de scope het gehele programma is, wordt aan i toegekend. In het tweede programma treedt bij ii := j geen dereferencing op; de waarde j zelf, met een kleinere scope dan ii zou aan ii moeten worden toegekend, hetgeen, zoals gezegd, verboden is.

13. GENERATORS; DE IDENTITY DECLARATION

Er is een middel om een waarde met een mode die met ref begint, d.w.z. een "name",te laten ontstaan, zonder dat daarbij meteen een variabele in het geding hoeft te zijn. Dit middel is een z.g. *generator*, die, evenals de selection een bijzonder geval is van een secondary.

Voorbeelden van generators zijn

a) heap real
b) loc real
c) real
d) loc [1:100] struct (int i,[1:5] bool b)

Een generator bestaat uit een z.g. *actual declarer*, die ook bij een variable declaration gebruikt wordt, al of niet voorafgegaan door heap of loc. Voorbeeld c is slechts een verkorte schrijfwijze van voorbeeld b, d.w.z. loc kan worden weggelaten zonder dat de betekenis verandert. Een generator die met heap begint, kortweg *heap generator* genoemd, genereert een waarde waarvan de scope het gehele programma is. De andere soort generator, *local generator* geheten, genereert een waarde waarvan de scope de kleinste omvattende range is. De mode van de gegenereerde waarde is steeds ref gevolgd door de mode die gegeven wordt door de "actual declarer". Door de genoemde beperkte scope van een local generator is het volgende voorbeeld fout.

```
ref int ii;
(real x; ii := loc int).
```

In de assignation is de scope van de door loc int gegenereerde "name" kleiner dan de scope van ii, hetgeen strijdig is met de restrictie genoemd in de vorige paragraaf. Vervanging van loc door heap maakt dit voorbeeld correct.

Een voorbeeld van een plaats waar een generator kan worden gebruikt is de z.g. *identity declaration*. Dit is een nieuw type declaratie; het bevat een gelijkteken, zoals de volgende voorbeelden laten zien.

e) ref int i = loc int
f) real pi2 = pi↑2.

In deze voorbeelden worden de identifiers i en pi2 gedefinieerd. Anders dan bij een variable declaration wordt bij een identity declaration de mode van de identifier zelf opgegeven en niet de mode van waarden die eraan kunnen worden toegekend. Zo heeft in voorbeeld e de identifier i de mode ref int. In feite is voorbeeld e equivalent met de variable declaration

```
int i.
```

In voorbeeld f daarentegen wordt een identifier gedefinieerd die geen variabele is. De mode van pi2 is real; pi2 is te beschouwen als een constante en kan niet als linkerlid van een assignation optreden. Het is duidelijk dat men bij het door elkaar gebruiken van "identity declarations" en "variable declarations" ervoor moet waken geen vergissingen t.a.v. de mode te maken. Zo wordt b.v. door

```
real x = 3.14;
real y := 3.14
```

de mode van x real en de mode van y ref real.

In deze paragraaf is reeds de term "actual declarer" genoemd, die in een generator en in een variable declaration wordt gebruikt. In tegenstelling hiermee wordt in een identity declaration links van het gelijkteken een z.g. *formal declarer* gebruikt. Een verschil tussen beide soorten declarers treedt op bij de multiple value; hier worden bij de formal declarer de grenzen weggelaten. Zo is b.v. [1:5] int een actual declarer en [ ] int een formal declarer.

Voorbeelden:

g) [1:5] int a := (10,20,30,40,50)
h) [ ] int a = (10,20,30,40,50)
i) ref [ ] int a = loc [1:5] int
j) ref [ ] int a = loc [1:5] int := (10,20,30,40,50).

Voorbeeld g geeft een variable declaration weer zoals we er al meer hebben

gezien. In de voorbeelden h, i en j worden identity declarations getoond. Links van het gelijkteken zijn de grenzen weggelaten; zij worden bepaald door hetgeen rechts van het gelijkteken staat. In voorbeeld i komt links een formal declarer en rechts een actual declarer voor. Voorbeeld j toont iets nieuws. Om dit te verklaren dient eerst te worden medegedeeld dat de algemene gedaante van een identity declaration is

```
"formal declarer" identifier 1 = unit 1,
                  identifier 2 = unit 2,
                      :              :
                  identifier n = unit n.
```

Omdat rechts van het gelijkteken een "willekeurige" unit staat mag dit best een assignation zijn. De assignation

```
loc [1:5] int := (10,20,30,40,50)
```

is ook correct omdat hier het linkerlid een generator en dus een secondary en dus een tertiary is, met een mode die met ref begint en verder overeenkomt met het rechterlid. Zoals in paragraaf 7 vermeld is, is de waarde van een assignation gelijk aan de (nieuwe) waarde van het linkerlid, dus in ons voorbeeld van de mode ref [ ] int hetgeen overeenkomt met de mode links van het gelijkteken. Overigens zou dereferencing voor de unit rechts van het gelijkteken toegestaan zijn, evenals dit toegestaan is voor het rechterlid van een assignation.

14. ROUTINES

Een belangrijk middel om herhaling van identieke stukken programma's te vermijden en om programma's overzichtelijk in te richten is de *procedure*. Aan een procedure zijn twee aspecten verbonden. In de eerste plaats moet een declaratie van de procedure gegeven worden, waarin vermeld wordt welke acties de procedure voorschrijft en in de tweede plaats moet er een middel zijn om deze acties op gang te brengen. Bij wijze van voorbeeld veronderstellen we dat op een aantal plaatsen in een programma een formule van de vorm $x^2 + y^2 + z^2$ moet worden uitgerekend. Men kan dan de volgende proce-

dure declareren:

```
proc f = (real x,y,z) real: x↑2 + y↑2 + z↑2
```

Door deze declaratie wordt het berekenen van $x^2 + y^2 + z^2$ alleen voorgeschreven, maar nog niet uitgevoerd. Hebben we nu in het vervolg van het programma de waarde van b.v. $a^2 + b^2 + c^2$ nodig, dan wordt deze waarde verkregen door de *call*

```
f(a,b,c).
```

Een call is een bijzonder geval van een primary en is al in de paragrafen 5 en 10 terloops ter sprake gekomen. In de call staan tussen haakjes de z.g. *actual parameters*, die "willekeurige" units kunnen zijn. In bovenstaande proceduredeclaratie zijn x, y en z z.g. *formal parameters*. Het verband dat ten gevolge van een call gelegd wordt tussen formele en actuele parameters laat zich, enigszins gesimplificeerd, weergeven met behulp van een denkbeeldige "identity declaration", waarin links van de gelijktekens formele en rechts actuele parameters staan. In ons voorbeeld krijgen we

```
real x = a, y = b, z = c.
```

Laten we nu op deze "identity declaration" de berekening van de formule x↑2 + y↑2 + z↑2 volgen dan is uiteraard het effect dat $a^2 + b^2 + c^2$ berekend wordt.

In dit voorbeeld volgt de formule x↑2 + y↑2 + z↑2 op de dubbele punt. In het algemeen kan dit een "willekeurige" unit zijn. Het gedeelte van de proceduredeclaratie na het gelijkteken, inclusief deze unit, heet *routinetext*. Zo'n routinetext representeert een waarde, *routine* genaamd, met een mode die we in ons voorbeeld noteren door proc (real,real,real) real.
Een routinetext is weer een nieuw voorbeeld van een unit. Men kan een routine, evenals elke andere "waarde", b.v. door middel van een assignation aan een variabele toekennen, b.v.

```
proc (real,real,real) real var;
var := (real x,y,z) real: x↑2 + y↑2 + z↑2;
```

of, bij de gegeven declaratie van f,

```
var := f.
```

Ook zou men in plaats van de gegeven declaratie van f de identity declaration

```
proc (real,real,real) real f = (real x,y,z) real: x↑2 + y↑2 + z↑2
```

hebben kunnen schrijven, al betekent dit meer schrijfwerk.

Bij ons eerste voorbeeld van een procedure was het afleveren van een waarde essentieel. Het is ook mogelijk dat we niet in het afleveren van precies één waarde geïnteresseerd zijn, maar alleen in het "uitvoeren van een stukje programma". In zo'n geval wordt in de proceduredeclaratie vlak vóór de dubbele punt de declarer void gebruikt.

Voorbeeld.

```
proc verwissel = (ref int x,y) void:
(int w := x; x := y; y := w);
```

De bedoeling hiervan is de waarden van twee variabelen te laten verwisselen. Behalve de reeds gesignaleerde declarer void bevat deze procedure een ander belangrijk verschil met het eerste voorbeeld. Hierin dienden de parameters uitsluitend om waarden aan de procedure "mee te geven"; de procedure "verwissel" daarentegen geeft bovendien de verwisselde waarden terug. Dit laatste is slechts mogelijk omdat hier de formele parameters een mode hebben die met ref begint. Ook nu kan weer een fictieve "identity declaration" dienst doen om het effect van een call te beschrijven. Zo wordt het effect van de call in

```
int u := 1, v := 2; verwissel (u,v)
```

beschreven door

```
ref int x = u, y = v;
int w := x; x := y; y := w
```

hetgeen tot gevolg heeft dat aan u de waarde 2 en aan v de waarde 1 wordt toegekend. Door de identity declaration ref int x = u wordt een toekenning aan x equivalent met een toekenning aan u. Het is alsof u en x verschillende naambordjes boven dezelfde brievenbus zijn.

De algemene vorm van een *call* wordt gegeven door de produktieregels

```
call                    : primary, open symbol, actual parameter list,
                        : close symbol.
actual parameter list: unit;
                         unit, comma symbol, actual parameter list.
```

Interessant is dat vóór het open symbol een "willekeurige" primary mag staan. Dit biedt b.v. de mogelijkheid een keuze tussen een aantal procedures te maken zonder telkens de actuele parameters te noemen, b.v.

```
proc som (real x,y, int n) real: (x+y)↑n;
proc verschil (real x,y, int n) real: (x-y)↑n;
c := (a<b | som | verschil) (a,b,5).
```

Doordat in de produktieregel voor de call expliciet haakjes en actuele parameters genoemd worden, krijgt men misschien de indruk dat elke procedure parameters heeft. Zoals het volgende voorbeeld laat zien is dit niet het geval.

```
proc lees = real: (real x; read(x); x);
real y;
y := 3 * lees↑3 + 2 * lees↑2 + lees.
```

Hier is "lees" een procedure zonder parameters die een waarde van de mode real aflevert. Toch heeft de identifier "lees" niet de mode "real", maar de mode proc real. In de formule achter "y :=" komt "lees" voor in connectie met operatoren die niet op operands van de mode proc real, maar wel op operands van de mode real kunnen worden toegepast. Het is daarom nodig een object van de mode proc real om te zetten in een object van de mode real. Deze overgang heet *deproceduring*. Deproceduring vindt plaats door een procedure zonder parameters te activeren.

Het declareren van operatoren

In paragraaf 6 zijn een aantal operatoren genoemd. We noemen deze *standaardoperatoren*. Als zich de behoefte voordoet, te beschikken over een operator die niet onder de standaardoperatoren voorkomt, dan kan men deze operator zelf declareren en wel op een wijze die veel gelijkenis met een proceduredeclaratie vertoont. We nemen als voorbeeld een test op "gelijkheid" van twee reële getallen, waarbij we de bedenkelijke afspraak maken dat twee getallen als "gelijk" worden beschouwd als hun verschil in absolute waarde kleiner is dan 0.0001. Men een proceduredeclaratie zou dit als volgt kunnen

```
proc gelijk = (real x,y) bool: abs (x-y) < .0001;
```

waarop b.v.

```
if gelijk (a,1.5) then print(1.5) fi
```

zou kunnen volgen. Zou men nu liever over de operator "gelijk" beschikken, die, zoals elke dyadische operator, tussen twee operands in geplaatst wordt en aldus een formule vormt, dan zou men in plaats van de gegeven proceduredeclaratie de volgende "operatordeclaratie" kunnen schrijven

```
op gelijk = (real x,y) bool: abs (x-y) < .0001.
```

Voor deze dyadische operator moet ook een "prioriteitsdeclaratie" verstrekt worden. Dit kan b.v. door

```
prio gelijk = 4;
```

Hierna zouden we kunnen schrijven

```
if a gelijk 1.5 then print(1.5) fi.
```

In plaats van de onderstreepte operator gelijk hadden we ook het gewone gelijkteken kunnen gebruiken:

```
op = = (real x,y) bool: abs (x-y) < .0001.
```

In dit geval heeft deze operator vanzelf de prioriteit 4 zoals voor het gelijkteken gebruikelijk is (zie ook paragraaf 6). Door deze declaratie krijgt het gelijkteken een afwijkende betekenis binnen de kleinste range waarin deze declaratie voorkomt, voorzover het althans toegepast wordt op twee operands die beide de mode real hebben. We zien hieraan dat bij het opzoeken van de declaratie van een operator, zijn representatie (b.v. "=" of gelijk) alleen niet voldoende houvast biedt, maar dat de modes van de operands erbij betrokken moeten worden. Dit is ook al in paragraaf 6 ter sprake gekomen.

We kunnen zelf ook monadische operatoren declareren, b.v.

```
op inverse = (int i) real: (abs i < 1e-20 | 1e20 | 1/i)
```

Na deze declaratie krijgen r1 en r2 in

```
real r1 := inverse 100, real r2 := inverse 0
```

de waarden 0.01, resp. 10^{20}.

Voor monadische operatoren wordt geen prioriteitsdeclaratie gegeven; zij hebben steeds de hoogste prioriteit.

15. DE "MODE DECLARATION"; complex, bits, bytes

Met de reeds besproken middelen kunnen we al willekeurig veel verschillende modes kiezen. Waar in de voorbeelden eenvoudshalve een mode als b.v. real gekozen is, had ook een mode als b.v.

```
[ ] struct (ref [ ] proc (int, struct (int a,b)) real p, [ ] bool q)
```

gebruikt kunnen worden. Nu is er een middel, de z.g. *mode declaration*, waarmee minder eenvoudige modes verkort kunnen worden genoteerd. Bovendien zal aan het eind van deze paragraaf blijken dat dit middel meer is dan een afkortingsmechanisme. Een voorbeeld van een mode declaration is

```
mode matrix = [1:3, 1:3] real, vector = [1:3] real.
```

Hierna kunnen we b.v. de "matrices" a en b en de "vectoren" u en v decla-

reren d.m.v. de variable declaration

```
matrix a,b, vector u,v.
```

We hebben hiermee voor eigen gebruik in ons programma a.h.w. de twee nieuwe modes matrix en vector aan de taal toegevoegd. Voor objecten van deze modes kunnen wij zelf operatoren definiëren. Allereerst declareren we bij wijze van voorbeeld een operator "*" om het inwendig product $p_1q_1 + p_2q_2 + p_3q_3$ van de vectoren p en q te berekenen:

```
op * = (vector p,q) real:
(real s:=0; for i to 3 do s+:=p[i] * q[i] od; s).
```

Een plus- en een maaloperator voor twee "matrices" x en y kunnen wij nu declareren als

```
op += (matrix x,y) matrix:
(matrix z;
 for i to 3
 do for j to 3 do z[i,j] := x[i,j] + y[i,j] od
 od;
 z);

op * = (matrix x,y) matrix:
 (matrix z;
 for i to 3
 do for j to 3 do z[i,j] := x[i, ] * y[ ,j] od
 od;
 z);
```

Men lette vooral op de vierde regel van de laatste declaratie, waar het inwendig produkt van de i-de rij van x en de j-de kolom van y wordt berekend.

Behalve bij het programmeren, kunnen mode- en operatordeclaraties ook worden gebruikt om op een compacte en precieze manier de betekenis van standaardmodes en -operatoren duidelijk te maken. Zo hadden we b.v. in paragraaf 6, als daar deze nieuwe begrippen al bekend waren geweest, de operator or kunnen definiëren als

```
op or = (bool a,b) bool : if a then true else b fi.
```

Nu deze begrippen inmiddels wel bekend zijn, kunnen we ervan gebruik maken om enige nieuwe standaardmodes en -operatoren te definiëren. In de eerste plaats vallen hieronder de volgende in de taal aanwezige faciliteiten voor het rekenen met complexe getallen.

```
mode compl = struct (real re,im);
op i = (real a,b) compl: (a,b);
op re = (compl a) real: re of a;
op im = (compl a) real: im of a;
op abs = (compl a) real: sqrt(re a↑2 + im a↑2);
op conj = (compl a) compl; re a i -im a;
op = = (compl a,b) bool: re a = re b and im a = im b;
op ≠ = (compl a,b) bool: not (a = b);
op + = (compl a,b) compl: (re a + re b) i(im a + im b);
op + = (compl a) compl: a;
op * = (compl a,b) compl;
(re a * re b - im a * im b) i (re a * im b + im a * re b);

prio i = 9.
```

Bovenstaande lijst van operatoren is niet volledig: op analoge wijze worden de operatoren -, /, ↑ gedefinieerd voor complexe operands. Bovendien wordt de operator i gedefinieerd voor het geval één van de operands de mode int heeft en worden x, -, *, /, =, ≠ ook gedefinieerd voor het geval één van beide operands de mode int of real heeft; bij ↑ wordt geëist dat de tweede operand de mode int heeft. Het gebruik van "sqrt" bij de declaratie van abs loopt vooruit op de behandeling van de standaardfuncties in paragraaf 19.

Twee andere standaardmodes zijn bits en bytes. Zij zijn sterk machine-georiënteerd. De mode bits wordt gedefinieerd door

```
mode bits = struct ([1:bitswidth] bool F),
```

waarin F, in tegenstelling tot re en im bij compl, slechts een symbolische

notatie voor een selector is. De bedoeling hiervan is de gebruiker, bij gebruik van bits, om machinetechnische redenen te dwingen de operator elem, formeel gedefinieerd als

op elem = (int i, bits b) bool: F of b,

te gebruiken. De bovengrens bitswidth is een geheel getal, b.v. 48, dat verband houdt met de constructie van de machine; bitswidth is een standaardidentifier van de mode int die men niet zelf hoeft te definiëren. Dyadische operatoren die men op operands van de mode bits kan toepassen en die een bool resultaat opleveren zijn =, ≠, ≤, ≥. Hun betekenis zal worden verklaard aan de hand van de speciale "denotation" die voor objecten van de mode bits is ingevoerd. Een voorbeeld hiervan staat rechts van het gelijkteken in

bits b = 2r1011

Is nu bitswidth b.v. gelijk aan 6, dan is b hierdoor een structured value met als enige veld een multiple value met de elementen

false, false, true, false, true, true .

Men ziet dat 1 overeenkomt met true en 0 met false en dat, om aan "bitswidth" waarden te komen, zo nodig links wordt aangevuld met false. De 2 vlak voor r, de z.g. *radix*, is de basis van het talstelsel waarin het deel achter de r wordt geschreven. I.p.v. 2 kan men ook 4, 8 of 16 als radix kiezen. Na de r volgt een rijtje cijfers die kleiner zijn dan de radix. Bij een radix 16 fungeren de letters a t/m f als de "cijfers" 10 t/m 15, d.w.z. a = 10, b = 11, c = 12, d = 13, e = 14, f = 15. De volgende formules, die alle true opleveren, verklaren de gebruikte operatoren.

2r1101 = 2r1101
not(2r0010 ≤ 2r1001)
2r1001 ≤ 2r1101
2r1101 ≠ 2r1001
16rf3a7 ≥ 16rf387
16rf3a7 = 2r1111 0011 1010 0111.

Twee operatoren met mode proc (bits, int) bits zijn shl en shr. Zij "schuiven" naar links, resp. rechts, zodat b.v. geldt 2r1101 shl 4 = 2r1101000 (mits bitswidth ≥ 8) en 8r705 shr 3 = 8r70. Twee operatoren met mode proc (bits, bits) bits zijn and en or. Hun betekenis blijkt uit

2r1100 and 2r1001 = 2r1000, en
2r1100 or 2r1001 = 2r1101.

Tenslotte zijn er drie monadische operatoren:
not, met mode proc (bits) bits, verwisselt true en false,
dus not 2r1011 = 2r0100;
abs, met mode proc (bits) int, converteert van bits naar int,
b.v. abs 2r10111 = 23;
bin, met mode proc (int) bits, converteert van int naar bits,
b.v. bin 23 = 2r10111.

Een verband zoals tussen de modes bool en bits bestaat ook tussen de mode char en de nu te noemen mode bytes. Formeel kan men de mode bytes declareren als

mode bytes = struct ([1:byteswidth] char F).

Ook nu is er een operator elem nodig, gedefinieerd door

op elem = (int i, bytes b) char : (F of b)[i].

De dyadische operatoren <, ≤, =, ≠, ≥, > voor bytes hebben een betekenis analoog aan die welke behandeld zal worden bij een algemenere vorm van een rij karakters, de z.g. string.

Er is een klasse van modes, die men wel eens oneindige of recursieve modes noemt, die alleen met behulp van een mode declaration te noteren zijn. Een voorbeeld is

mode knoop = struct (int i, ref knoop links, rechts)

De mode knoop wordt hier gedefinieerd in termen waarin hijzelf voorkomt. We

zien hieraan dat een mode declaration meer kan zijn dan een middel om een mode verkort te noteren. Op een belangrijke toepassing van zulke modes, de z.g. *list processing*, zal in deze syllabus niet verder worden ingegaan.

16. STRINGS; UNITED MODES; DE CONFORMITY CLAUSE

Tot nu toe hebben we ons gehouden aan de regel dat de mode van een variabele zich uitsluitend door het voorvoegsel ref onderscheidt van de mode van een waarde die aan de variabele kan worden toegekend. Op deze regel bestaan twee uitzonderingen. De eerste betreft z.g. *flexibele bovengrenzen van multiple values*. Men kan b.v. als volgt een variabele a declareren

```
flex [1:0] int a;
```

Door deze declaratie refereert a aanvankelijk naar een "multiple value" waarvan de bovengrens 0 is, maar kan a ook refereren naar een multiple value met een andere bovengrens. Zo is de bovengrens na b.v.

```
a := (10,20,30,40,50)
```

gelijk aan 5. De mode van a heet officiëel "reference to flexible row of integral", hetgeen we afkorten tot ref flex [ ] int, en de mode van een waarde waarnaar a kan refereren heet gewoon "row of integral", afgekort tot [ ] int.

Een belangrijke toepassing is de standaard mode string, formeel gedefinieerd door

```
mode string = flex [1:0] char.
```

Declareert men

```
string s;
```

dan zou men kunnen schrijven

```
s := ("a","b","c","d","e").
```

Wegens het belang van deze toepassing is hiervoor een eenvoudiger middel ingevoerd, de z.g. *string denotation*, met behulp waarvan men in plaats van deze assignation mag schrijven

s := "abcde".

Een "string denotation" bestaat uit twee aanhalingstekens waartussen een aantal karakters staan. Zou dit aantal één zijn, dan stond er de in paragraaf 2 genoemde character denotation. Daarom sluit men dit geval bij string denotation uit; schrijft men

string s := "a",

dan heet "a" officieel *character denotation*. In de volgende paragraaf zal blijken dat het onderscheid tussen beide soorten denotations in een geval als dit slechts een formaliteit is.

Voor strings zijn een aantal standaardoperatoren beschikbaar. In de eerste plaats zijn er de operatoren lwb en upb die voor willekeurige multiple values zijn gedefinieerd, maar hier bijzonder goed van pas komen. Hebben we b.v. een multiple value a met minstens drie subscripts, dan leveren b.v. de formules

3 lwb a, 1 upb a

respectievelijk de ondergrens van de derde subscript en de bovengrens van de eerste subscript van a op. De prioriteit van deze operatoren is 8. Ook hun monadische versies zijn beschikbaar: men kan een eventuele 1 die aan lwb of upb voorafgaat weglaten. Verder zijn er voor strings in de eerste plaats de "relationele" operatoren $<$, $\leq$, $=$, $\neq$, $>$, $\geq$. Voor de strings x en y geldt x = y als x en y even lang zijn en karakter voor karakter gelijk of beide "leeg". Hierbij verstaan we onder de *lengte* van de string het aantal karakters waaruit hij bestaat; een lege string heeft een lengte nul. Uiteraard is $x \neq y$ als not (x = y) geldt. Voor de betekenis van de overige relationele operatoren moet men denken aan de lexicografische ordening zoals die b.v. in een telefoonboek wordt gebruikt; hierbij wordt gebruik gemaakt van de betekenis die, volgens paragraaf 6, operatoren van dit type al hebben voor karakters. Verder zijn er de operatoren +, *, +:=, +=: om uit

strings en/of karakters nieuwe strings te vormen. Hun betekenis wordt duidelijk uit het volgende voorbeeld

```
string s := "ab", t;
t := "cd" + "ef"; co t = "cdef" co
t := 3 * "ab";    co t = "ababab" co
s +:= "cd";       co s = "abcd" co
"gh" +=: s;       co s = "ghabcd" co
t := "pq" + 3 * s[2:4] + t[1];
                  co t = "pqhabhabhaba" co.
```

Bij al de genoemde stringoperatoren mag een operand van de mode char de plaats van een operand van de mode string innemen.

De andere uitzondering op de regel die aan het begin van deze paragraaf werd genoemd, vormen de z.g. *united modes*. We kunnen b.v. declareren

```
union (int, bool) ib;
```

De variabele ib kan hierdoor zowel naar een object met mode int als naar een object met mode bool refereren. Zo zijn b.v. de assignations

```
ib := 3    en    ib := true
```

beide mogelijk. De variabele ib heeft de mode "reference to union of integral and boolean", hetgeen we afkorten tot ref union (int, bool). {Deze mode kan overigens, zonder dat de duidelijkheid er op vooruitgaat, ook gespecificeerd worden door b.v.

```
ref union (bool, int),
ref union (bool, int, bool),
ref union (bool, union (int, bool)).}
```

Evenals flex is union blijkbaar iets dat bij de variabele zelf behoort en niet bij de waarde waarnaar de variabele refereert. Er is een speciaal middel om te onderzoeken wat de mode is van het object waarnaar zo'n variabele refereert en om de waarde van dit object te kunnen gebruiken. Dit middel is de z.g. *comformity clause*, een nieuw bijzonder geval van een unit. Laten we

veronderstellen dat b.v. gedeclareerd is

```
union (int, bool, real, char) ibrc
```

en dat op zeker moment moet worden onderzocht wat de mode is van het object waarnaar ibrc ten gevolge van een eerder uitgevoerde assignation refereert. Is deze mode int, dan willen we in "unit 1" de waarde van het object als identifier i (met mode int) kunnen gebruiken, is de mode bool dan willen we in "unit 2", deze waarde als identifier b (mode bool) beschikbaar krijgen. In de andere gevallen (hier real of char) willen we "serial clause" laten uitvoeren. Dit kan met de "conformity clause"

```
case ibrc
in (int i) : "unit 1",
   (bool b) : "unit 2"
out "serial clause"
esac.
```

Een kortere notatie is

```
(ibrc | (int i) : "unit 1",
        (bool b) : "unit 2"
      | "serial clause").
```

De identifiers i en b na int, resp. bool kunnen als er geen behoefte aan is, worden weggelaten, evenals het gedeelte out "serial clause". In plaats van int en bool in dit voorbeeld kan men willekeurige "formal declarers" die niet met union beginnen, gebruiken. Verder is men niet beperkt tot twee alternatieven tussen in en out, zoals in dit voorbeeld het geval is.

De declarer string en een declarer die begint met union kunnen ook gebruikt worden voor identifiers die geen variabelen zijn. Deze schijnbaar overbodige faciliteiten kunnen als volgt nuttig worden gebruikt. In de eerste plaats is het prettig een identity declaration als b.v.

```
string s = "abcdefg"
```

te mogen opschrijven, ook al is er hier geen sprake van flexibiliteit omdat

de waarde van s niet meer veranderd kan worden. Verder zullen vaak formele parameters van procedures een mode hebben die door string of door een declarer beginnend met union wordt gespecificeerd. Deze laatste mogelijkheid stelt ons in staat b.v. een uitvoerprocedure als print te gebruiken, waaraan waarden als actuele parameter worden meegegeven die een tamelijk willekeurige mode hebben.

Naast "echte" modes kan men bij unions ook de "oneigenlijke" mode void gebruiken. Men kan b.v. schrijven:

```
union (int, real, void) u := empty.
```

Hier is empty de z.g. *void-denotation;* deze is te beschouwen als een "waarde" van de oneigenlijke mode void.

17. COERCIONS; DE CAST

We hebben gezien dat het kan voorkomen dat op een plaats in ons programma een waarde van een zekere mode vereist wordt, terwijl een waarde van een andere mode in eerste instantie beschikbaar is. De mode die beschikbaar is zullen we hier *a-priori-mode* noemen en de mode die vereist wordt *a-posteriori-mode*. Afhankelijk van de genoemde plaats in ons programma, die we *syntactische positie* noemen, is overgang van de a-priori-mode naar de a-posteriori-mode al dan niet mogelijk. Deze overgang heet *coercion*. Een al eerder besproken, belangrijk voorbeeld van coercion is "dereferencing". Evenzo *deproceduring*, de "aanroep" van een procedure zonder parameters, waarbij de a-priori-mode b.v. proc real en de a-posteriori-mode real kunnen zijn. Een ander type coercion doet zich voor in de volgende voorbeelden

```
real r := 0;
complex z := 3.14;
complex z1 := 0;
[1:bitswidth] bool := 2r10101;
string s := bytes; # waarin bytes een object van de
                     mode bytes is #.
```

Dit type coercion, waarbij wordt overgegaan van een meer bijzondere mode naar een meer algemene mode, heet *widening*. Speciale vermelding verdient het bovenstaande voorbeeld complex z1 := 0. Hier vindt tweemaal achtereen "widening" plaats: eerst van int naar real, daarna van real naar complex.

Een soort coercion die enigszins op widening lijkt is *rowing*, waarvan we het volgende voorbeeld geven

```
string s = "a".
```

Hier moet a.h.w. "row of" aan de a-priori-mode worden toegevoegd; rechts van het gelijkteken staat immers een "character denotation".

Bij de behandeling van "united modes" is het voorbeeld

```
union (int, bool) ib;
ib := 3
```

gebruikt. Hier vindt overgang van de mode int naar de mode union (int, bool) plaats. Deze coercion heet *uniting*.

Een op het eerste gezicht triviale vorm van coercion is *voiding*, dat is het verloren laten gaan van een waarde zoals dat b.v. in de serial clause

```
1; 2; 3; 4; 5
```

gebeurt voor de waarden 1, 2, 3 en 4. (Alleen 5 wordt afgeleverd als waarde van deze serial clause.) Toch heeft dit begrip betekenis, zoals wellicht uit het volgende voorbeeld duidelijk wordt.

```
int i := 0;
proc p = void : i +:= 1;
proc q;
q := p;
p; q; print(i).
```

De vraag is welke waarde van i wordt afgedrukt. De mode van q is ref proc void; de mode van p is proc void. Daarom vindt bij de assignation

q := p geen enkele coercion plaats, zodat i hierbij niet verandert. Op de hierop volgende regel verwachten we bij "p;" wèl deproceduring, d.w.z. het verhogen van i met 1. Dit nu wordt, formeel gesproken, bereikt door de z.g. "voiding" van p en door te eisen dat vóór "voiding" in deze syntactische positie zo mogelijk "deproceduring" wordt toegepast. Hierop volgt "q;" dat zelfs achtereenvolgens "dereferencing","deproceduring" en "voiding" tot gevolg heeft. Het resultaat is dus het afdrukken van de waarde 2.

Samenvattend kennen we dus de coercions: dereferencing, deproceduring, uniting, widening, rowing en voiding. Aan het begin van deze paragraaf is opgemerkt dat de mogelijkheid om een coercion te laten plaats vinden ook afhankelijk is van de syntactische positie. De volgende syntactische posities zijn belangrijke voorbeelden van het geval dat alle "coercions" die nodig zijn ook inderdaad worden uitgevoerd:

- het rechterlid van een assignation (b.v. i in x := i)
- een actuele parameter van een procedure (b.v. i in p(i))
- een subscript van een multiple value (b.v. i in a[i])
- een unit in een identity declaration (b.v. 5 in real x = 5).

Er is ook een belangrijke syntactische positie waar alleen dereferencing. deproceduring en uniting zijn toegestaan, namelijk een operand in een formule.

Een nog strengere beperking geldt voor het linkerlid van een assignation; hier is alleen deproceduring toegestaan.

Er is een speciaal middel om in een gegeven syntactische positie elk van de genoemde "coercions" mogelijk te maken. Dit is een nieuw soort unit, de z.g. *cast*. Een cast bestaat uit een formal declarer die de a-posteriori-mode specificeert, gevolgd door tussen haakjes een unit die een waarde van een a-priori-mode levert. Voorbeelden van casts zijn real(i) en ref int (ii); hoe casts kunnen worden gebruikt blijkt uit

```
op k = (real x) real: (x↑3)/(x↑2 + 1);
op k = (int i) real : k real (i);
print(k 5);

int i; ref int ii := i; ref int (ii) := 7.
```

De monadische operator k voor een operand van de mode int wordt in dit voorbeeld gedefinieerd in termen van een reeds gedefinieerde gelijknamige operator voor een operand van de mode real. In de laatste regel heeft i de mode ref int en ii de mode ref ref int. Daarom is ii := 5 niet toegestaan: dereferencing is in het linkerlid van een assignation niet zonder meer mogelijk. Pas door de cast ref int (ii) wordt overgegaan van de mode ref ref int naar ref int, waarna de toekenning van 5 (aan i!) kan plaatsvinden.

18. BALANCING; DE IDENTITY RELATION; nil

In paragraaf 7 is de conditional clause behandeld, waarvan

```
if i < j then 1 else 3.14 fi
```

een voorbeeld is. Men kan zich nu afvragen of b.v. de monadische formule

```
entier (i < j | 1 | 3.14)
```

wel correct is. Immers, als i < j zou zijn en de waarde 1 met mode int afgeleverd zou worden, dan zou de operator entier moeten worden toegepast op een operand van de mode int, waarvoor hij niet is gedefinieerd. Daarom worden in een geval als dit de twee of meer alternatieven a.h.w. tegen elkaar afgewogen om de mode van het resultaat te bepalen. Hier heeft 1 de mode int en 3.14 de mode real. Er is een coercion (n.l. widening) om van int naar real over te gaan en niet omgekeerd. Daarom is hier de mode van de conditional clause real, ongeacht de uitkomst van de test i < j. Dit tegen elkaar afwegen van verschillende modes om tot een gemeenschappelijke mode te komen heet *balancing*. Kan niet door balancing de vereiste gemeenschappelijke mode gevonden worden dan is de constructie fout, zelfs al zouden er zich geen praktische problemen voordoen, zoals in

```
int i := if true then 1 else "a" fi.
```

Daarentegen is

```
int i; ref int ii := i, int a,j; read(a);
(a > 0 | j | ii) := 0
```

wel correct. Hier betekent balancing dat op ii dereferencing wordt toegepast om van de mode ref ref int over te gaan op de mode ref int, waardoor overeenstemming met de mode van j wordt bereikt. Balancing kan ook optreden in een case-clause, een collateral clause en een conformity clause.

Er is nog een nieuwe soort unit te behandelen waarbij eveneens balancing kan optreden, n.l. de z.g. *identity relation*. Dit is een middel dat de taal biedt om te kunnen vaststellen of twee "names" identiek zijn, m.a.w. of twee variabelen permanent naar hetzelfde object refereren. Een identity relation bestaat uit twee tertiaries, gescheiden door het "is-symbol" :=: of het "is-not-symbol" :≠:, die beide, eventueel na deproceduring een mode hebben die met ref begint. Dereferencing, ook meer dan één keer, mag slechts op één van beide tertiaries worden toegepast. Als een identity relation

```
tertiary1 :=: tertiary2
```

de waarde true oplevert, dan levert

```
tertiary1 :≠: tertiary2
```

de waarde false op en omgekeerd.

In het volgende voorbeeld leveren de identity relations alle de waarde true op.

```
int i := 1, j := 1; ref int l = i
ref int ii := i, kk := i;
i :≠: j;
i :=: 1;
ii :≠: kk;
i :=: i;
ref int (ii) :=: ref int (kk);
ii :=: i;
```

In de laatste identity relation treedt balancing op. Op de regel die hier-

aan voorafgaat staan twee casts die beide i opleveren.

Soms ontstaat de behoefte expliciet te kunnen aangeven dat een variabele waarvan de mode met ref ref begint naar geen andere variabele refereert. Hiervoor bestaat een speciale tertiary, de z.g. *nihil*, die geschreven wordt als nil. Men gebruikt deze vooral om het eind van een keten aan te geven, zoals in het volgende voorbeeld.

```
mode cel = struct (int i, ref cel volgende);
ref cel start := loc cel := (1,nil);
```

Aan de keten, die nu nog slechts uit één cel bestaat kan vooraan een cel worden toegevoegd door b.v.

```
start := loc cel := (2,start).
```

Hierna leveren de volgende identity relations de waarde true op

```
volgende of start :≠: nil
volgende of volgende of start :=: nil.
```

19. DE STANDARD PRELUDE; long EN short; TRANSPUT; COLLATERAL ACTIONS

Een programma dat wij schrijven heet officieel een *particular-program*. Wij moeten het ingebed denken in één of meer omvattende ranges waarin tal van nuttige procedures, operatoren, identifiers en modes gedeclareerd zijn. Voorafgaande aan ons programma moeten wij ons de z.g. *standard prelude* voorstellen. In het Revised Report [4] is deze in zijn geheel opgenomen. Zelfs als men onverhoopt niet aan een grondige studie van het Revised Report toekomt, kan het raadplegen van deze "standard prelude" sterk worden aanbevolen. Tot de onderwerpen hieruit die nog genoemd moeten worden, behoren in de eerste plaats de mathematische functies

```
sqrt, exp, ln, cos, arccos, sin, arcsin, tan, arctan.
```

Van deze procedures, waarvan de mode proc (real) real is, wordt de betekenis bekend verondersteld. Een procedure van de mode proc real, d.w.z. zon-

der parameters, is random, die een pseudo-aselecte trekking uit een eenheidsinterval [0,1) aflevert.

De identifier pi met mode real is, evenals de hierin te noemen identifiers, voor de gebruiker beschikbaar omdat hij in de standard prelude is gedefinieerd. Zijn waarde is een benadering van het getal π met een precisie die door de constructie van de machine wordt bepaald. Men kan een indruk van deze precisie krijgen d.m.v. de identifier small real, die de mode real heeft en waarvan de waarde het kleinste reële getal is, zodanig dat de machine door de beide formules 1 + small real > 1 en 1 - small real < 1 true laat opleveren. Omdat binnen de machine waarden van de mode real niet onbeperkt groot kunnen zijn, is er de identifier max real, die de grootst mogelijk reële waarde heeft. Evenzo heeft de identifier maxint (met mode int) de grootst mogelijke waarde van de mode int

De declarers int, real, compl, bits en bytes kunnen een of meer malen door long of short worden voorafgegaan, waardoor nieuwe modes ontstaan, waarbij de precisie of het waardebereik (door long) vergroot of (door short) verkleind wordt. Het aantal keren long dat men voorafgaande aan int kan gebruiken plus 1, dus b.v. 2 als alleen int en long int mogelijk is, is "op te vragen" d.m.v. de identifier intlengths. Een analoge betekenis hebben de identifiers intshorths, reallengths, realshorths, bitslengths, bitsshorths, byteslengths en bytesshorths. Om ook bij laatstgenoemde modes de precisie of het waardebereik te kunnen opvragen kan men de genoemde identifiers small real, max real en max int laten voorafgaan door een passend aantal malen long of short.

In- en uitvoer worden tezamen *transput* genoemd. In [4] wordt transput, in het kader van de standard prelude, zeer uitvoerig behandeld. In hoeverre deze faciliteiten reeds voor praktisch gebruik beschikbaar zijn hangt af van de te gebruiken compiler. In deze syllabus wordt slechts een klein deel van de transputfaciliteiten, die de taal biedt behandeld. Transput vindt plaats d.m.v. z.g. *files*. In de standard prelude komt de mode declaration van de mode file voor. Deze maakt het mogelijk dat in de standard prelude ook de variable declaration

```
file stand in, stand out, stand back
```

voorkomt. Bij veel transput procedures moet een variabele die de betreffende

file aangeeft als actuele parameter worden opgegeven. Nu hebben we al kennis gemaakt met de transputprocedures read en print waarbij dit niet het geval was. Toch maken deze procedures gebruik van de eerste twee genoemde "standard files":

b.v. read(x) is equivalent met get(stand in, x),
print(x) is equivalent met put(stand out, x).

De procedures get en put zijn generalisaties van de procedures read en print. Laatstgenoemde maken impliciet gebruik van de files stand in en stand out; get en put zijn daarentegen ook voor andere files te gebruiken. Een soortgelijk verband is er tussen de volgende procedures, die gebruikt worden om te lezen van en te schrijven naar een "achtergrondgeheugen":

b.v. readbin(x) is equivalent met getbin(stand back, x),
writebin(x) is equivalent met putbin(stand back, x).

Wil men andere files dan stand in, stand out en stand back gebruiken, dan dient men deze zelf te declareren en te "openen", b.v.

```
file file1;
open (file1, "idf1", chan1).
```

De modes van de parameters van open zijn respectievelijk ref file, string, channel. De opgegeven string is een identificatie waaraan de file te herkennen is. Voor de derde parameter moet men een identifier invullen die men niet zelf declareert, maar die voorkomt op een lijstje dat door de compilerbouwer wordt verstrekt; hierop wordt in deze syllabus niet verder ingegaan. In bovenstaand voorbeeld is ervan uitgegaan dat er al een file met identificatie "idf1" aanwezig is. Is dit niet het geval, dan kan men hem laten ontstaan door b.v. de call

```
establish(file1, "idf1", chan1, p, l, c).
```

Hierin hebben p, l en c welgedefinieerde waarden van de mode int; door middel hiervan geeft men de maximale omvang van de over te dragen informatie op: p is het aantal "pages", l het aantal "lines" per "page" en c het

aantal characters per "line".

Het tegenovergestelde hiervan, d.w.z. het laten verdwijnen van een file gaat b.v. als volgt

```
scratch (file1).
```

De indeling in pages, lines en characters verklaart de volgende calls:

```
newpage (file1),
newline (file1),
space (file1).
```

Door newpage en newline wordt op een nieuwe "page", resp. "line" overgegaan; space geeft één spatie.

Al eerder is opgemerkt dat aan print en read parameters van velerlei mode mogen worden meegegeven; hiertoe behoren ook de identifiers newpage, newline en space, dus b.v.

```
print((x, newline, "abc"))
```

is equivalent met

```
print(x); newline (stand out); print ("abc") .
```

De dubbele haken in de eerste call zijn hier nodig: de collateral clause "(x, newline, "abc")"treedt op als één actuele parameter van print en wordt dus geschreven tussen de haakjes in "print( )".

Om getallen in een willekeurig formaat te kunnen afdrukken, of althans als string beschikbaar te hebben, zijn er de conversieroutines whole, fixed en float. In de standard prelude wordt hun betekenis gedefinieerd d.m.v. proceduredeclaraties die we hier slechts symbolisch met een ruwe aanduiding van hun werking weergeven; steeds geldt dat de te vormen string aan de linkerkant zonodig met spaties wordt aangevuld en dat de string uit b.v. enkel sterren (*) bestaat als het aantal opgegeven posities te klein is:

```
mode number = union (real, int);

proc whole = (number v, int width) string;
¢ levert, zonodig afgerond op een geheel getal,
v af in de vorm van width karakters ¢

proc fixed = (number v, int width, after) string;
¢ levert, zonodig afgerond, v af als width karakters
met after cijfers achter de decimale punt ¢

proc float = (number v, int width, after, exp) string:
¢ levert v af als width karakters in floating-point-repre-
sentatie, met after cijfers in de mantisse na de punt en met
exp posities na het "times ten to the power symbol" ¢.
```

Voorbeeld:

```
print(fixed(pi,8,4))
```

levert 3.1416, waarbij twee spaties aan de 3 voorafgaan.

Gebruikers voor wie deze representatiefaciliteiten voor getallen onvoldoende zijn kunnen gebruik maken van "formatted transput", waarbij gebruik wordt gemaakt van z.g. "format texts". Hierop wordt in deze syllabus niet ingegaan; een informele behandeling hiervan is te vinden in [2].

Parallel actions

In de standard prelude zijn een aantal operatoren genoemd die nauw samenhangen met de z.g. *parallel clause*, die, syntactisch gesproken, bestaat uit een collateral clause, voorafgegaan door het symbool par; de algemene gedaante is dus

```
par (unit 1, unit 2, ..., unit n), of
par begin unit 1, unit 2, ..., unit n end.
```

De hierin genoemde units worden parallel, d.w.z. onafhankelijk van elkaar uitgevoerd, tenzij gebruik wordt gemaakt van z.g. *semaphores*; dit zijn ob-

jecten van de mode sema, welke mode in de standard prelude gedeclareerd is als

```
mode sema = struct (ref int F).
```

Hierin is de selector F, evenals bij bits en bytes, niet direct te gebruiken; men kan de betreffende geheeltallige waarde te pakken krijgen door de operator level, gedeclareerd als

```
op level = (sema s) int: F of s.
```

Deze operator levert een geheel getal bij een gegeven sema. Het omgekeerde doet een operator die ook level heet en gedefinieerd wordt door

```
op level = (int i) sema:
   (sema s; F of s := heap int := i: s).
```

Verder zijn in de standard prelude de operatoren down en up, beide van de mode proc (sema) void, gedefinieerd. De bedoeling van up is de genoemde geheeltallige waarde met 1 te verhogen en de bedoeling van down is deze waarde met 1 te verlagen. Hier zit evenwel meer aan vast, hetgeen aan de hand van het volgende voorbeeld wordt verklaard.

```
sema voorraad = level 0;

par
begin do "produceer één exemplaar en breng dit
          naar het magazijn";
          up voorraad
      od,
      do down voorraad;
         "haal een exemplaar uit het magazijn
          en geef het uit"
      od
end.
```

Hier staat tussen de aanhalingstekens geen string denotation maar een sym-

bolische aanduiding van een stukje programma. De sema voorraad heeft een niet-negatieve geheeltallige waarde die gelijk is aan het aantal exemplaren van een artikel in een magazijn. Het magazijn is hier onbeperkt groot verondersteld en bevat slechts artikelen van één soort. Het is duidelijk dat het brengen naar en het halen uit het magazijn dan processen zijn, die onderling onafhankelijk zijn met één restrictie, n.l. dat er geen negatieve voorraad bestaat. Daarom betekent "down voorraad" in het geval dat de geheeltallige waarde nul is: wacht eerst tot deze waarde door de uitvoering van "up voorraad" groter dan nul is geworden.

Voor verdere toepassingen van semaphores wordt verwezen naar [5].

20. EEN BIJGEWERKT GRAMMATICAAL OVERZICHT MET VERWIJZINGEN

Ter vervanging van het overzicht aan het eind van paragraaf 10 worden de volgende produktieregels gegeven. Om het opzoeken te vergemakkelijken zijn tussen accoladen verwijzingen naar paragrafen van deze syllabus vermeld.

```
program        : closed clause {8}.
closed clause  : begin symbol {10}, serial clause {7,11}, end symbol {10};
                 open symbol {10}, serial clause, close symbol {10}.
serial clause  : parade {11};
                 prologue {11}, go on symbol {10}, parade.
prologue       : declaration {4,13,14,15};
                 unit {7}, go on symbol, prologue;
                 declaration, go on symbol, prologue.
parade         : unit;
                 unit, go on symbol, parade;
                 label definition {11}, parade;
                 unit, completer {11}, parade.
completer      : completion symbol {11}, label definition.
unit           : assignation {3};
                 identity relation {18};
                 routine text {14};
                 jump {11};
                 skip {11};
                 tertiary.
```

```
tertiary      : P-formula {6,10};
                nihil {18};
                secondary.
secondary     : generator {13};
                selection {9};
                primary.
primary       : slice {9};
                call {14};
                cast {17};
                denotation {2};
                format text {19};
                identifier {3};
                closed clause {8};
                collateral clause {9};
                parallel clause {19};
                conditional clause {7};
                case clause {8};
                conformity clause {16};
                loop clause {8}.
P-formula     : P-operand, P-operator, (P+1)-operand. (P = 1,...,9)
10-formula    : monadic operator, 10 operand.
P-operand     : P-formula;
                (P+1)-operand.                        (P = 1,...,10)
11-operand    : secondary.
selection     : tag, of symbol {9}, secondary.
slice         : primary, sub symbol {9}, indexer {9}, bus symbol {9}.
denotation    : integral denotation {2};
                real denotation {2};
                boolean denotation {2};
                character denotation {2};
                void denotation {16};
                bits denotation {15};
                string denotation {16}.
```

declaration : variable declaration {4,9};
identity declaration {13};
procedure declaration {14};
operator declaration {14};
priority declaration {14};
mode declaration {15}.

Het is mogelijk deze verzameling produktieregels zover uit te breiden dat al de genoemde begrippen door achtereenvolgende toepassing van een aantal produktieregels worden uitgedrukt in *symbolen*, de kleinste bouwstenen waaruit wij ons programma opgebouwd denken. Verder is het mogelijk tal van extra voorwaarden, die gewoonlijk apart, op niet-grammaticale wijze, worden genoemd, in de produktieregels te verwerken. Hierbij gebruikt men een techniek, die in zeer bescheiden mate ook in bovenstaande produktieregels is toegepast en wel in de regels voor P-formula en P-operand. Hierin is P een z.g. *metabegrip*; door voor P één geschikt getal te kiezen en dit getal konsekwent in een regel te substitueren ontstaat pas een "gebruiksklare" produktieregel. De gegeven regel waarin P nog voorkomt hadden we, in overeenstemming met [1] en [4], dan ook beter "hyperregel" in plaats van "produktieregel" kunnen noemen. In [1] en [4] worden aparte produktieregels voor metabegrippen gegeven; de metabegrippen worden geschreven met hoofdletters en stellen geen getallen maar rijtjes kleine letters, z.g. *protonotions* voor. Deze beschrijvingswijze, *Van Wijngaardengrammatica* genoemd, biedt de mogelijkheid op syntactische wijze talen te definiëren waarvoor een z.g. contextvrije grammatica zonder metabegrippen ontoereikend is.

LITERATUUR

[1] A. van Wijngaarden (Editor), B.J. Mailloux, J.E.L. Peck and C.H.A. Koster, *Report on the Algorithmic Language ALGOL 68*, Numerische Mathematik, 14 (1969) 79-218.

[2] C.H. Lindsey and S.G. van der Meulen, *Informal introduction to ALGOL 68*, North Holland Publishing Company, Amsterdam, 1971.

[3] J.E.L. Peck, *An ALGOL 68 companion*, Dept. of Computer Science, University of British Colombia, Vancouver B.C., Canada, 1972.

[4] A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, M. Sintzoff, C.H. Lindsey, L.G.L.T. Meertens and R.G. Fisker, *Revised Report on the Algorithmic Language ALGOL 68*, Lineprinter Version, March 1974.

[5] E.W. Dijkstra, *Cooperating Sequential Process*, in: *Programming Languages*, Genuys, F. (ed.), Academic Press, London etc., 1968.

[6] P.M. Woodward and S.G. Bond, *ALGOL 68-R Users Guide*, London, Her Majesty's Stationary Office, 1972.

[7] L. Ammeraal, *An interpreter for simple ALGOL 68 programs*, Mathematisch Centrum Rapport IW 7, Amsterdam, 1973.