Technische Hogeschool Eindhoven

Onderafdeling der Wiskunde

Inleiding tot de Informatica

Dictaatnr. 2.269 Prijs f 3,50

TECHNISCHE HOGESCHOOL EINDHOVEN
Afdeling Algemene Wetenschappen
Onderafdeling der Wiskunde

INLEIDING tot de INFORMATICA

Auteur Onbekend

70er jaren

Inhoudsbeschrijving
INLEIDING tot de INFORMATICA
Auteur Onbekend
70er jaren

JdG, 31 Juli 2005

0. Inleiding programmastructuren

Computersystemen zijn uit de hedendaagse maatschappij niet meer weg te denken, daar zij met grote snelheid en betrouwbaarheid rekenwerk verrichten, waarvoor anders duizenden "rekenslaven" nodig zouden zijn. Rekenslaven noch computersystemen doen hun werk echter autonoom; beiden moet verteld worden wat van ze verwacht wordt. Verschillen zijn dat werkopdrachten voor computers (rekenautomaten) in detail geformuleerd moeten worden, het aantal verschillende basisopdrachten voor computers vrij klein is, het effect van deze opdrachten beperkt is, computers niet zelfstandig corrigerend kunnen optreden wanneer hun opdrachten onvolledig of dubbelzinnig zijn geweest. Allerlei klachten en grapjes over computersystemen die de pers halen, zijn dan ook vrijwel altijd toe te schrijven aan onvoldoende en/of onjuiste opdrachten aan een computer door mensen en niet aan de rekenautomaten zelf!

Een twintig jaar geleden was het opstellen van opdrachten maatwerk in die zin dat wanneer er ook maar iets veranderde aan hetzij de rekenautomaat, hetzij de betreffende toepassing, vrijwel alle opdrachten opnieuw opgesteld moesten worden. Langzamerhand heeft men echter geleerd allerlei zaken te generaliseren, met het gevolg dat wat onderzoek betreft het hoofdaccent is verschoven van de apparatuur naar problemen samenhangende met de verwerking van gegevens (meestal tot andere gegevens) en in het bijzonder naar de kunst of kunde van het opstellen van opdrachten (meestal "programmeren" genoemd) voor een gewenste gegevensverwerking. Omdat gegevens in een bepaalde context voor de mens informatie bevatten, wordt dit onderzoeksterrein tegenwoordig informatica genoemd. Eigenschappen van rekenautomaten en hun physieke structuur behoren nu tot de randgebieden van de informatica, evenals specifieke toepassingen van informatica methoden in andere vakgebieden. Het eerste is een randgebied met de informatietechniek; bedrijfsinformatica is bijvoorbeeld een toepassing in het vakgebied van de bedrijfskunde en bedrijfsvoering. (Wat hoofd- en randgebieden zijn is natuurlijk betrekkelijk willekeurig; voor iemand die zich op de informatietechniek concentreert, kan de informatica een randgebied zijn.)

Volledigheidshalve zij hier opgemerkt dat we verder slechts over het programmeren van digitale computers zullen spreken. Naast digitale kennen we ook analoge computers (en hybride als combinatie van die twee). De eerste kunnen we opvatten als geautomatiseerde telramen (getalvoorstelling met discrete eenheden), de tweede als veredelde rekenlinealen (getalvoorstelling met een continue physische grootheid).

1. Processen, processoren, programma's en algoritmen

1.1. Inleiding

Opgemerkt kan worden dat de informatica zich vooral bezig houdt met de verwerking van gegevens tot meestal andere gegevens (verwerking gebruiken we als verzamelnaam voor vergaren, vastleggen, verschaffen en de eigenlijke verwerking of transformatie). Al op het eerste gezicht is er zodoende een grote overeenkomst met technische processen, waarbij materiële grondstoffen verwerkt worden tot materiële tussen- of eindproducten (of soms tot energie in al zijn verschijningsvormen). Deze overeenkomst gaat inderdaad zeer ver zodat we gedachten uit de proceskunde kunnen overnemen in de informatica.

Voorbeelden van informaticaprocessen zijn o.a.:

- de uitvoering van (complexe) wiskundige berekeningen, waarbij zowel de grondstoffen als eindproducten getallen zijn (meestal vrij klein in aantal);
- de uitvoering van massale administratieve bewerkingen als bij een girodienst of een magazijnadministratie, waarbij eveneens zowel grondstoffen als eindproducten (grote aantallen) getallen en alfabetische teksten zijn, doch het rekenwerk vrij eenvoudig van aard is;
- de uitvoering van technische processen, waarbij een rekenautomaat in de besturing van het proces ingeschakeld is. De grondstoffen zijn hier overwegend uit het proces komende electrische signalen (en enkele door de toezichthoudende mens ingevoerde getalwaarden). De eindproducten zijn afgezien van enkele getalwaarden voor rapportdoeleinden overwegend weer electrische signalen, waarmee het te besturen proces automatisch geregeld wordt. Voorbeelden zijn o.a. de besturing van chemische of physische fabrieksprocessen, van cyclotrons, van maanlanders, van fotosatellieten (eindproduct nu foto's van planeten, enz.), patiëntenbewakingssystemen, vliegtuigreserveringssystemen;
- de uitvoering van bibliografische processen, waarbij een rekenautomaat als grondstof trefwoorden accepteert en als eindproduct een lijst van relevante publicaties produceert.

Op de details van al deze processen zullen we uiteraard niet ingaan; wanneer we bij latere beschouwingen een voorbeeld nodig hebben zullen we dat meestal ontlenen aan rekenprocessen omdat iedereen daarmee min of meer vertrouwd is.

1.2. Toestandsbeschrijving en toestandsverandering

Karakteristiek voor technische en informatica-processen is hun dynamische karakter, d.w.z. hun afhankelijkheid van de tijd. In de proceskunde is de tijd meestal een continue veranderlijke. In de informatica hanteren we daarentegen als regel de tijd als discrete veranderlijke; we zijn namelijk meestal alleen geïnteresseerd in de toestand van een informatica-proces op bepaalde discrete tijdstippen en in de toestandsveranderingen die achtereenvolgens optreden. De wijze waarop die veranderingen tot stand komen wordt hierbij meestal buiten beschouwing gelaten.

Voor de beschrijving van een toestand in proceskunde en informatica maken we gebruik van benoemde toestandsgrootheden (toestands-variable of kortweg variable). Deze variabelen vormen met elkaar een verzameling, V, van toestandsgrootheden. De heersende (momentane) waarden van alle toestandsgrootheden definiëren een toestand van V. De verzameling van alle mogelijke toestanden wordt de toestandsruimte van V genoemd.

Welke toestandsgrootheden voor een bepaald proces relevant zijn, wordt bepaald door onze belangstelling. Voor de chef van een fabrieksproces is misschien alleen de opbrengst van belang, voor de man die het proces moet regelen zijn druk en temperatuur van het proces de relevante grootheden, terwijl de man die zich wil verdiepen in het moleculaire procesgebeuren geïnteresseerd zal zijn in de plaats- en snelheidscoördinaten van alle bij het proces betrokken moleculen.

Een actie in een toestandsruimte is de toekenning van waarden aan een of meer toestandsgrootheden. Een actie verandert de toestand van V omdat sommige toestandsgrootheden van waarde veranderen. Deze verandering vindt plaats in een eindig tijdsbestek. Uitvoering van een aantal acties achter elkaar bewerkstelligt een reeks opeenvolgende toestanden in de toestandsruimte. Het eerste element van deze reeks is de begintoestand, het laatste element de eindtoestand.

Een toestand is dus bepaald door een rij waarden van variabelen; een toestandsverandering (-transformatie) door de verandering van waarde van één of meer variabelen.

Vaak kan men een actie opvatten als een rij deelacties; uitgaande van een begintoestand leiden de achtereenvolgende toestandstransformaties door de deelacties dan tot de eindtoestand van de totale actie. Elke actie kan op zijn beurt worden beschouwd als deelactie van een omvattende actie. Of we een actie nu als ondeelbaar beschouwen of als een rij deelacties hangt weer af van onze belangstelling. In het eerste geval zijn we alleen geïnteresseerd in begintoestand en eindtoestand; we beschouwen de actie dan alleen "uitwendig". In het tweede geval zijn we tevens geïnteresseerd in de tussentoestanden; we beschouwen de actie dan "inwendig", m.a.w. we zijn dan ook geïnteresseerd in de wijze waarop de transformatie in stappen van begin- naar eindtoestand plaatsvindt. Wordt een actie inwendig bezien, opgedeeld in opeenvolgende deelacties, dan spreken we van een sequentiëel proces.

Sommige acties zullen we niet meer opsplitsen in deelacties; deze acties zijn voor ons elementaire acties (die met elkaar het basisrepertoire vormen).

Acties (en processen) spelen zich af in de tijd, zijn dynamisch. De beschrijving van een actie is echter tijdloos, statisch. De beschrijving van een ondeelbare actie noemen we een opdracht (statement). De beschrijving van een proces bestaat dan uit een rij opdrachten.

Wil de beschrijving eenduidig interpreteerbaar zijn, dan moet er overeenstemming zijn tussen opsteller en uitvoerder t.a.v. de taalelementen waarin de beschrijving wordt vastgelegd.
Is deze taal de wiskundige taal, dan spreekt men i.p.v. procesbeschrijving ook wel van algorithme (zie voor een uitvoeriger beschouwing appendix B § 1). Is deze wiskundige taal wat aangepast omdat een rekenautomaat de uitvoerder is, dan spreekt men i.p.v. een procesbeschrijving van een programma in een of andere programmeeertaal. Een rekenautomaat is hierbij een mechanisme dat een programma uitvoert (zie voor een uitvoeriger beschouwing appendix B § 2).

1.3. Representatie van waarden van toestandsgrootheden

Tot dusver hebben we nog niet gesproken over de wijze waarop de waarden van onze toestandsgrootheden concreet vastgelegd (gerepresenteerd) worden. Dit kan gebeuren door ze te representeren door een of andere physische grootheid, zoals o.a. een lengte (rekenlineaal), een hoekverdraaiing (snelheidsmeter), een electrische spanning. Men spreekt dan van een analoge representatie. Er zijn analoge rekenautomaten geconstrueerd, die (arithmetische) bewerkingen uitvoeren op grootheden die op analoge wijze vastgelegd zijn. Het aantal elementen van zo'n analoge automaat is ruwweg afhankelijk van het product van het aantal voor een probleem relevante toestandsgrootheden en het aantal uit te voeren bewerkingen op iedere grootheid. Voor een complex probleem is zodoende het benodigde aantal elementen zeer groot. Daar het voorts technisch/economisch vrijwel onmogelijk is om analoge grootheden met meer dan vier à zes cijfers nauwkeurigheid vast te leggen, heeft het gebruik van analoge rekenautomaten ondanks hun enorme rekensnelheid niet zo'n grote vlucht genomen.

Het alternatief, waartoe wij ons verder zullen beperken, is de digitale representatie waarbij waarden van toestandsgrootheden in "cijfervorm" worden vastgelegd. Ter verduidelijking van het onderscheid tussen digitaal en analoog: naast de uurwerken met wijzerplaat en wijzers (analoge apparaten) zijn tegenwoordig ook digitale uurwerken verkrijgbaar, die de tijd in cijfers weergeven. Digitale automaten hebben door hun grote flexibiliteit de analoge automaten en de zg. hybride (een combinatie van analoge en digitale) automaten verdrongen.
Ten aanzien van de digitale representatie van toestandsgrootheden in het geheugen van een rekenautomaat bestaat nog een grote mate van vrijheid. Dit is eigenlijk niets bijzonders, want ook op papier zijn verschillende getalrepresentaties mogelijk. Zo kan b.v. het getal dertien worden voorgesteld met 13 (decimaal), 1101 (binair), 15 (octaal), D (hexadecimaal), XIII (Romeins). Wij zullen daarom in hetgeen volgt in het midden laten hoe cijfers, letters, enz. precies op papier, op ponsband, op ponskaart of in een processorgeheugen worden gerepresenteerd. Er bestaan gelukkig wel standaardisaties voor.

1.4. Voorbeelden

Opmerking

Voor procesbeschrijvingen in deze voorbeelden zullen we vooruitlopen op nog te behandelen opdrachten, die echter voor het intuïtieve begrip geen moeilijkheden zullen opleveren.

1.4.1. Voorbeeld 1

Het eerste probleem luidt:
Bepaal van twee natuurlijke getallen het (gehele) quotiënt en de rest bij deling.
De begin- en eindtoestand worden bepaald door vier variabelen: de twee gegeven getallen, die we a en b zullen noemen, en het quotiënt en de rest, die we q en r zullen noemen. De begintoestand kunnen we karakteriseren door:

a en b hebben de gegeven waarden, q en r zijn onbepaald.

De eindtoestand wordt beschreven door:

a en b hebben de gegeven waarden, q bevat het quotiënt en r de rest bij deling van a door b.

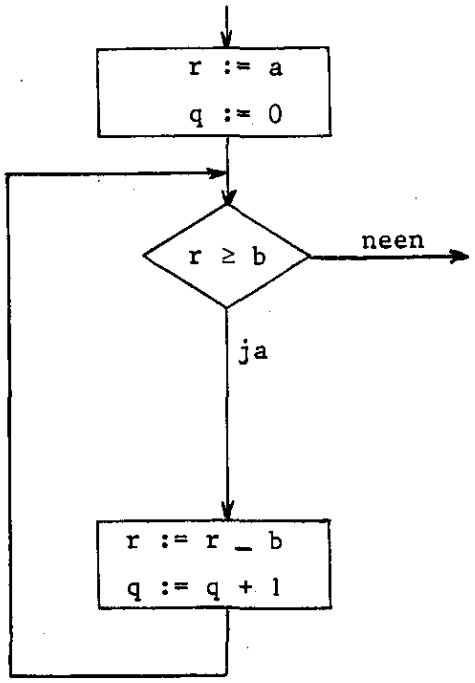
Vaak zullen in de processor de operaties "geheel delen" (met als operator div) en "rest bepalen" (met als operator mod) aanwezig zijn; met deze operaties kan het gewenste resultaat dan als volgt worden bereikt:

```
q := a div b; r := a mod b
```

We zullen er echter vanuit gaan dat deze operaties niet in onze processor aanwezig zijn, maar dat in onze processor slechts de mogelijkheid bestaat tot de operaties optellen en aftrekken. We zullen dan het bepalen van het quotiënt en de rest moeten uitdrukken in sommen en verschillen. Het quotiënt q van a en b kunnen we dan vinden als het aantal malen dat b van a afgetrokken kan worden, zodanig dat er een rest r overblijft, die groter is dan of gelijk is aan 0 maar kleiner is dan b.
We kunnen dit algoritme als volgt vastleggen:

```
r := a; q := 0;
while r ≥ b do begin r := r - b;
                     q := q + 1
               end
```

Een andere representatie is



Opmerking

Voor de interpretatie van de toestanden in een proces is de betekenis van de variabelen belangrijk. Een variabele speelt een bepaalde rol in het proces. Deze betekenis is gedurende het hele proces hetzelfde. De betekenis van de variabele kunnen we tot uitdrukking laten komen in de naam, maar we zullen deze betekenis in ieder geval voor ons zelf moeten vastleggen. Uit de betekenis van de variabelen volgen relaties die tussen de variabelen gelden. In bovenstaand voorbeeld geldt steeds (als de test $r \geq b$ wordt uitgevoerd) de relatie $a = q * b + r$; deze relatie wordt een invariante relatie genoemd. We komen daar later op terug.

Het vasthouden aan de betekenis van de variabelen en aan de invariante betrekkingen, is zeer nuttig zo niet noodzakelijk om tot een correct programma te komen.

1.6.2. Voorbeeld 2

Bepaal de oplossing van $x = \cos(x)$ met $0 \leq x \leq \frac{1}{2}\pi$ (x uitgedrukt in radialen).

Het probleem speelt zich af in de omgeving der reële getallen; voor deze

getallen is de functie cos(x) gedefinieerd. We weten verder dat er gesproken kan worden van dè oplossing, omdat in het gestelde interval er precies één oplossing bestaat van de vergelijking x = cos(x). De oplossing van deze vergelijking is echter een reëel getal, waarvan de waarde niet precies te berekenen is. We volstaan daarom met een benadering van de oplossing, dat wil zeggen dat we tevreden zijn als we een voldoend klein interval [og, bg] (og = ondergrens en bg = bovengrens) gevonden hebben waarin de oplossing ligt.

(bg - og < epsilon) and (oplossing ligt in [og, bg])

Dit is dan ook de te realiseren eindrelatie; hierin is epsilon een maat voor de nauwkeurigheid die we wensen.
We willen verder een punt uit het oplossingsinterval (met name (bg + og)/2) als oplossing opleveren in de variabele x. De gewenste eindrelatie wordt dan:

(bg - og < epsilon) and (oplossing in [og, bg]) and (x = (bg + og)/2)

Dit kunnen we realiseren door

```
while (bg - og) ≥ epsilon
   do "verklein (bg - og) onder de voorwaarde dat
       (oplossing in [og, bg]) blijft gelden"
```

als we er maar voor zorgen dat vooraf al de uitspraak oplossing in [og, bg] geldt. Dit laatste kunnen we bijvoorbeeld realiseren door "og := 0; bg := 3.14/2".
Het verkleinen van (bg - og) kunnen we als volgt realiseren.

(1) m := (bg + og)/2;

(2) als m ≤ cos(m) dan ligt de oplossing van x = cos(x) in het interval [m, bg]; als we dus og gelijk maken aan m houden we "oplossing in [og, bg]" geldig;
als m ≥ cos(m) dan ligt de oplossing van x = cos(x) in het interval [og, m]; als we dus bg gelijk maken aan m houden we "oplossing in [og, bg]" geldig.

We krijgen als totale procesbeschrijving:

```
og := 0; bg := 3.14/2;
while bg - og ≥ epsilon do
      begin m := (bg + og)/2;
            if m ≤ cos(m) then og := m
                         else bg := m
      end;
x := (bg + og)/2
```

2. Variabelen, typen en expressies

2.0. Inleiding

De beschrijving van toestanden vindt plaats door middel van variabelen (toestandsgrootheden). Om aan een variabele te kunnen refereren, zal deze van een naam voorzien moeten zijn, tenzij er slechts sprake is van één variabele. Dit laatste is b.v. het geval bij de roltrap, waar het geheugenelement dat het aantal nog af te werken treden bijhoudt, onbenoemd kan blijven. Zo kun je er in een monogame maatschappij mee volstaan om je echtgenote als "vrouw" toe te spreken. Heb je een harem, dan is dat wat onduidelijk en moet je kunnen aangeven welke van je vele vrouwen je wilt toespreken en daartoe moeten de vrouwen benoemd zijn.

Een variabele wordt dus gekenmerkt door een naam om hem te kunnen benoemen en door een waarde, die mede bepaalt in welke toestand een proces op een bepaald moment verkeert.

In een programma hebben namen van variabelen - de engelse term voor zo'n naam is "identifier" - geen intrinsieke, d.w.z. toestandsbepalende betekenis. Dit is als in het normale leven, waarin meneer De Groot best een klein mannetje zou kunnen zijn. De programmeur is in hoge mate vrij in de keuze van de namen die hij zelf invoert, maar hij doet er goed aan betekenisvolle namen in te voeren en verwarrende namen te vermijden.

In het vervolg geven we de naam van een variabele aan met een rij letters en/of cijfers, altijd beginnende met een letter. Toelaatbaar als naam is dan "seven up", ontoelaatbaar is "7 up". Toelaatbaar is ook "Austin Seven" evenals "Austin 7", maar deze laatste twee namen duiden wel twee verschillende variabelen aan.

De waarde van een variabele is een element van een waardenverzameling (waardebereik). Een waardenverzameling is karakteristiek, voor een bepaalde klasse van variabelen: het type. Bij elk type, hoort een aantal operaties. Wij zullen kennis maken met drie typen, die we aangeven met integer, real en boolean. De eerste twee noemen we arithmetische typen; de derde wordt ook wel logisch type genoemd.

2.1. Typen van variabelen

Zoals gezegd, komen we de volgende typen variabelen tegen:

- integer, d.w.z. variabelen die waarden in Z^* kunnen aannemen, waarbij Z^* een eindige aaneengesloten deelverzameling is van de verzameling Z van de gehele getallen. Onder- en bovengrens van deze deelverzameling liggen voor elke processor vast.
- real, d.w.z. variabelen die waarden in Q^* kunnen aannemen, waarbij Q^* een eindige deelverzameling is der rationele getallen, Q. Deze Q^*-waarden liggen in een interval in Q met voor iedere processor specifieke grenzen.
- boolean, d.w.z. variabelen, die slechts de "boolean waarden" false en true kunnen aannemen; true en false zijn logische waarden, bijv. waarden van "uitspraken".

2.2. Toegestane bewerkingen op de verschillende typen

2.2.1. Integers en reals

Hierop zijn de volgende arithmetische bewerkingen mogelijk (met tussen haakjes het operatiesymbool):

- operatie met één operand: inversie (-)
- operaties met twee operanden: optelling (+), aftrekking (-), vermenigvuldiging (*), deling (/), machtsverheffing (↑). Bovenstaande operaties hebben de gebruikelijke betekenis.

Op de natuurlijke getallen zijn verder de volgende operaties mogelijk:

- gehelendeling (div): levert het (gehele) quotiënt bij deling
- restbepaling (mod): levert de rest bij deling.

Als c gelijk is aan div b en d gelijk is aan a mod b dan is a dus gelijk aan b * c + d.

Opmerking. Men zij er op bedacht dat ook andere operatiesymbolen dan bovenstaande worden gebruikt (** i.p.v ↑; // of ÷ i.p.v. div) en dat reële getallen door reals slechts benaderd worden, zodat bij het gebruik van reals afrondingsfouten mogelijk zijn.

2.2.2. Booleans

Hierop zijn de volgende logische bewerkingen mogelijk:
operatie met één operand: ontkenning (not of ¬); bijv. heeft de boolean variabele a de waarde true, dan heeft not a de waarde false;
operaties met twee operanden: het resultaat van enkele logische operaties is uit de volgende tabel af te lezen.

operatie	notaties		a	true	true	false	false
			b	true	false	true	false
conjunctie	a and b	a ∧ b		true	false	false	false
disjunctie	a or b	a ∨ b		true	true	true	false

2.3. Expressies

2.3.1. Arithmetische expressies

Met behulp van operanden (getallen, integer of real variabelen, later te behandelen functies), arithmetische operatoren en ronde haakjes kunnen arithmetische expressies (uitdrukkingen) worden opgebouwd volgens de bekende regels: tussen twee operanden precies één operator (dus geen twee operatoren vlak achter elkaar). Bij de uitwerking (d.i. de berekening van de waarde, evaluatie) van zo'n expressie moeten we rekening houden met de prioriteitsregel dat haakjes de hoogste prioriteit hebben.
Verder (zie nevenstaand operatorenschema) hebben operaties met operatoren op een hogere regel voorrang; operaties met operatoren op eenzelfde regel (b.v. * en /) hebben gelijke prioriteit en moeten van links naar rechts worden afgewerkt.

↑			
*	/	div	mod
+	-		

Zo is nu het volgende te verifiëren

normale schrijfwijze	correcte weergave	incorrecte weergave
(a + b)(c + d)	(a + b)*(c + d)	(a + b)(c + d)
$2n - 1$	2*n - 1	2n - 1
$\frac{1}{1 - a}$	1/(1 - a)	1/1 - a
$(2^3)^4$	(2 ↑ 3) ↑ 4	2 ↑ (3 ↑ 4)
2^{3^4}	2 ↑ (3 ↑ 4)	2 ↑ 3 ↑ 4
$p.\frac{-1}{qr}$	p * (-1)/(q*r)	p * -1/q * r
$a^{(2p)}$	a ↑ (2 * p)	a ↑ 2 * p

2.3.2. Boolean expressies

Op gelijksoortige wijze als bij arithmetische expressies kan men met behulp van boolean operanden (boolean waarden true of false, variabelen of later te behandelen functies), boolean operatoren en ronde haakjes boolean expressies opbouwen. Zo zijn a; a and b; a and (b or c) boolean expressies als a, b en c boolean variabelen zijn.
Een boolean operand kan bovendien worden gevormd door een zogeheten relatie. Een relatie bestaat uit twee arithmetische expressies, verbonden door een "relational operator". Deze operatoren zijn de bekende gelijkheids- en ongelijkheidsoperatoren: $=$, $\neq$, $<$, $\leq$, $\geq$, $>$.

Voorbeelden van relaties

Relatie	Waarde
x = x	true
x = x + 1	false
x ≤ x + 1	true
(x + 3)*(x - 2) ≥ 0	hangt af van de momentane waarden van de variabelen
a + b > c - d	

Voorbeelden van boolean expressies

Expressie	Waarde
x = x and y = y	true
(x + 3)*(x - 2) ≥ 0 and x > - 3 and x < 2	false
(x + 3)*(x - 2) < 0 or x ≤ - 3 or x ≥ 2	true
(x + 3)*(x - 2) ≥ 0 and x = 0.	false

Op gelijksoortige wijze als bij arithmetische bewerkingen is er voor de evaluatie van boolean expressies een prioriteitsregel die zegt dat haakjes de hoogste prioriteit hebben, gevolgd door relaties en vervolgens not, gevolgd door and en tenslotte or.

Bij gelijke prioriteit moet een expressie weer van links naar rechts worden uitgewerkt.

Voorbeelden.

boolean expressie	interpretatie
a > b or c < d	(a > b) or (c < d)
a + b > -5 and c - d < e ↑ 2	((a + b) > -5) and ((c - d) < (e ↑ 2))
a and b or c ≠ d	(a and b) or (c ≠ d)

2.4. Opmerking over functies

Zowel bij arithmetische als boolean expressies kunnen we als operand ook sommige "functies" gebruiken, die nu niet verder behandeld zullen worden. Het is voldoende zo'n operand te omschrijven als de waarde van een functie voor de argumentwaarde, die tussen (ronde) haakjes achter het functiesymbool geplaatst wordt. Voorbeelden zijn de volgende "standaardfuncties" (ae = arithmetische expressie; ae' = waarde van ae)

sign(ae) : +1 als ae' > 0, 0 als ae' = 0 en -1 als ae' < 0 } type integer
entier(ae): de grootste integer niet groter dan ae' }

abs(ae) : de absolute waarde van ae' }
sqrt(ae) : de vierkantswortel uit ae' }
exp(ae) : de exponentiële functie van ae' }
ln(ae) : de natuurlijke logarithme van ae' } type real
sin(ae) : de sinus van ae' (in radialen) }
cos(ae) : de cosinus van ae' (in radialen) }
arctan(ae): de arctan van ae' (hoofdwaarde) }

2.5. Invoer- en uitvoerfuncties

Voor de invoer en uitvoer van gegevens (zie 1.4) worden invoer/uitvoerfuncties gebruikt. Voorlopig maken we gebruik van de volgende (sterk vereenvoudigde) functies

read(a) : deze opdracht bewerkstelligt dat vanuit de invoerstroom de eerstvolgende waarde wordt gelezen, en vervolgens aan de variabele a wordt toegekend.

write(ae): deze opdracht bewerkstelligt dat de waarde van de arithmetische expressie ae in de uitvoerstroom verschijnt.

Opmerking. In werkelijkheid zijn de invoer- en uitvoeropdrachten ingewikkelder dan hier beschreven, hetgeen samenhangt met de vele mogelijkheden van invoer- en uitvoerapparatuur.

2.6. Enkelvoudige en samengestelde variabelen (arrays)

Kenmerkend voor de tot dusver beschouwde variabelen is dat iedere variabele in een bepaalde toestand slechts één waarde bezit. We noemen deze dan ook enkelvoudige of scalaire variabelen. Daarnaast bestaan ook samengestelde variabelen, gekenmerkt door het feit dan hun waarden bestaan uit een samenstel van "componentwaarden". In principe kan iedere componentwaarde weer een samenstel van waarden zijn. De manier waarop de componenten van een samengestelde variabele met elkaar samenhangen noemen we de structuur van de variabele. Zo kan bijvoorbeeld een punt in de driedimensionale ruimte gekarakteriseerd worden door een variabele met drie reële componenten (de coördinaten van dat punt). Een andere toepassing is de volgende. We beschouwen de samengestelde variabele, die we aangeven met de naam "persoonsgegevens". De componenten zijn: naam, geboorteplaats, geboortedatum. Deze laatste component is zelf weer samengesteld uit de componenten: jaar, maand, dag.

We zullen ons hier beperken tot de samengestelde variabele, genaamd array, waarvan de componenten gekarakteriseerd worden door:

- iedere component is enkelvoudig
- alle componenten zijn van hetzelfde type
- iedere component kan individueel genoemd worden in een programma.

2.7. Declaratie

Om de toestandsruimte, waarbinnen een proces zich voltrekt, vast te leggen moeten de nodige toestandsgrootheden, variabelen worden vastgelegd door middel van een zogeheten declaratie. De declaratie van een variabele houdt in feite in dat naam en type worden opgegeven (dit bewerkstelligt tevens dat in het geheugen van de automaat per enkelvoudige component een geheugen plaats wordt gereserveerd). Aan de hand van een aantal voorbeelden zullen we deze declaraties uiteenzetten.

Voorbeeld 1.

```
integer index, jan, n;
boolean afgedaan, vol, c;
real r, som, klaas
```

Met ieder van de drie declaraties declareren we telkens drie enkelvoudige variabelen en hun type.

Voorbeeld 2.

```
integer array a[11:20], b[1:10]
```

Met deze declaratie worden twee array's met ieder 10 componenten gedeclareerd. Deze componenten kunnen geselecteerd worden met zogeheten subscripted variables. Bij gebruik van een bepaalde component zal de naam van het array plus het bij die component horende subscript (ook wel: index) moeten worden vermeld. Bijv. a[13], b[7]. Echter niet a[5], a[21], b[0].

Voorbeeld 3.

```
integer array index, jan[0:10], n[-1:0];
boolean array klaar[-10:2,5:9];
real array piet[0:1,1:2,5:8];
```

Met de eerste declaratie worden drie 1-dimensionale arrays met resp. 11, 11, 2 integer componenten gedeclareerd. De eerste twee zullen geen indexwaarden anders dan 0 t/m 10 mogen aannemen, de derde slechts -1 en 0. Met de tweede declaratie wordt een 2-dimensionale boolean array gedeclareerd, waarbij de eerste index kan lopen van -10 t/m 2 en de tweede index van 5 t/m 9 (naar analogie met matrices zou men van rij en kolomindex kunnen spreken). De array klaar bevat dus 13 * 5 = 65 componenten. Met de derde declaratie wordt een 3-dimensionaal array gedeclareerd met real componenten en de opgegeven onder- en bovengrens voor ieder van de indices. Voor elke dimensieaanduiding geldt dat de bovengrens nóóit kleiner mag zijn dan de ondergrens.

Deze declaratie van een enkelvoudige of een samengestelde variabele betekent niet een waarde toekenning aan de variabele of de componenten (ook niet de waarde nul)! De waarde van de variabele en van elke component blijft ongedefinieerd totdat een waarde is toegekend. De eerste keer dat dit gebeurt noemen we een initialisatie; deze mòet plaats hebben gevonden voordat zo'n va-

riabele of component in een expressie mag voorkomen (anders zou ook de expressie ongedefinieerd zijn).

Wanneer we een bepaalde array-component als operand nodig hebben, dan geven we dit aan met de arraynaam gevolgd door, tussen rechte haken, een aantal indexwaarden, gelijk aan het aantal dimensies van het array. Uiteraard moeten de indexwaarden binnen de gedeclareerde grenzen liggen (anders refereren we aan een nietbestaande component). In plaats van getallen mogen tussen de rechte haken bij het aanroepen van een bepaalde component zelfs arithmetische expressies staan (bij evaluatie mogen zij weer niet buiten de declaratie grenzen liggen).

Voorbeelden van correct en incorrect gebruik van array-componenten op basis van de declaraties van voorbeeld 3.

correct	incorrect
jan[7]	jan[-1]
n[0]	n[1]
klaar[0,6]	klaar[0]
piet[0,2,7]	piet[0,3,8]
	piet[0,2]
--------	---------
--------	---------
a := 1;	a := 2;
q := piet[a,2*a,6*a]	q := piet[a-1,a,a↑2]

3. Taalelementen voor procesbeschrijving

3.1. Inleiding

We hebben enkele primitieve bouwstenen van een programma, zoals variabelen en expressies, al leren kennen. Evenzo zijn ook de declaraties, vastleggingen van een toestandsruimte, al geïntroduceerd. We zullen nu bekijken hoe we acties (processen) kunnen beschrijven. Hierbij onderscheiden we:

- de (statische) beschrijving van acties met behulp van statements. Voorbeelden hiervan zijn de assignment statement en de procedure statement; de laatste komt pas later aan de orde.
  Statements worden in de tekst van een programma onderling gescheiden door een puntkomma; de bijbehorende acties worden als regel uitgevoerd in de volgorde waarin de statements staan.
- elementen voor enerzijds het herhaald en anderzijds het selectief uitvoeren van statements. Men noemt deze elementen wel "sequentiëringsstatements" of "control structures". Wij zullen deze elementen eenvoudigheidshalve ook kortweg statements noemen.

3.2. Assignment statement

De meest elementaire actie die we kennen is "de toekenning van de waarde van een expressie aan een variabele". De statement voor deze actie wordt assignment statement genoemd. Als we aan de variabele a de waarde 5 willen toekennen, wordt dit genoteerd als a := 5. (Als twee (of meer) variabelen van hetzelfde type eenzelfde waarde krijgen, mogen we schrijven a := b := 5).
Met behulp van assignment statements wordt de toestand van een proces vastgelegd (of veranderd).
Bij uitvoering van een assignment statement moet de waarde van de expressie bepaald kunnen worden; dit betekent dat de variabelen, die in de expressie voorkomen, een waarde moeten bezitten.
Zo moeten de arithmetische variabelen x en y een waarde hebben voor de uitvoering van

```
a := x + y   (a arithmetische variabele)
b := x > y   (b boolean variabele)
```

3.3. Voorbeelden van enkele eenvoudige programma's

a) Gegeven is een rij van vijf gehele getallen. Deze getallen, hun som en hun gemiddelde, moeten afgedrukt worden. We hebben vijf getallen en we kunnen zeven variabelen invoeren:

a : krijgt (heeft) de waarde van het eerste getal
b : krijgt (heeft) de waarde van het tweede getal
c : krijgt (heeft) de waarde van het derde getal
d : krijgt (heeft) de waarde van het vierde getal
e : krijgt (heeft) de waarde van het vijfde getal
som: krijgt (heeft) de waarde van de som van a, b, c, d en e.
gem: krijgt (heeft) de waarde van het gemiddelde van a, b, c, d en e.

```
begin integer a,b,c,d,e,som; real gem;
      read(a); write(a); read(b); write(b); read(c); write(c);
      read(d); write(d); read(e); write(e);
      som := a + b + c + d + e;
      write(som);
      gem := som/5;
      write(gem)
end
```

Opmerkingen. In plaats van de hier gebruikte namen van de variabelen kunnen we natuurlijk ook andere namen kiezen. Het afdrukken van de waarde van iedere variabele gebeurt direct nadat iedere variabele een waarde heeft gekregen. We kunnen ook met één variabele volstaan, die steeds de waarde van een getal uit de rij heeft totdat deze bij de som is opgeteld. We krijgen dan:

```
begin integer a,som; real gem;
      read(a); write(a); som := a;
      read(a); write(a); som := som + a;
      read(a); write(a); som := som + a;
      read(a); write(a); som := som + a;
      read(a); write(a); som := som + a;
      write(som);
      gem := som/5; write(gem)
end
```

b) Bereken de hoek tussen de twee vectoren $\underline{a}$ en $\underline{b}$ uit $\mathbb{R}^3$. De componentwaarden zijn in een invoerrij gegeven. We maken gebruik van $(\underline{a}.\underline{b}) = |\underline{a}||\underline{b}|\cos\alpha$ en veronderstellen voorts dat een arccos-functie gebruikt mag worden.
We introduceren als variabelen:

a,b : arrays met grenzen [1:3]; a[i] bevat a_i, b[i] bevat b_i
ab : voor het inproduct
aabs : voor de lengte van de vector $\underline{a}$
babs : voor de lengte van de vector $\underline{b}$
alpha: voor de hoek α.

```
begin real array a,b[1:3]; real ab, aabs, babs, alpha;
      read(a[1]); read(a[2]); read(a[3]);
      read(b[1]); read(b[2]); read(b[3]);
      ab := a[1] * b[1] + a[2] * b[2] + a[3] * b[3];
      aabs := sqrt(a[1] * a[1] + a[2] * a[2] + a[3] * a[3]);
      babs := sqrt(b[1] * b[1] + b[2] * b[2] + b[3] * b[3]);
      alpha := arccos(ab/(aabs * babs));
      write(alpha)
end
```

Opmerkingen. We huiveren natuurlijk al bij het idee dat we op de zojuist geschetste wijze met vectoren $\underline{a}$ en $\underline{b}$ uit $\mathbb{R}^{100}$ zouden moeten werken. Er is behoefte aan de mogelijkheid om een actie te herhalen, zonder dat de (tekst van de) statement herhaald opgeschreven moet worden. Nodig is dus een taalelement om een herhaling aan te kunnen geven.
Het bovenstaande programma zou ontsporen als $\underline{a}$ of $\underline{b}$ de nulvector is; er moet dan iets anders gebeuren. Er is dus ook behoefte om uit alternatieven te kunnen kiezen en dit in de programmatekst te kunnen aangeven (zie voorbeeld a) in 3.7).

3.4. Samenvoeging van meerdere statements tot één statement

Het zal vaak voorkomen dat een groep statements als een eenheid opgevat moet worden. We kunnen dit bereiken door de groep statements te omsluiten door het hakenpaar begin en end; bijvoorbeeld:

```
begin read(g[i]); i := i + 1 end
```

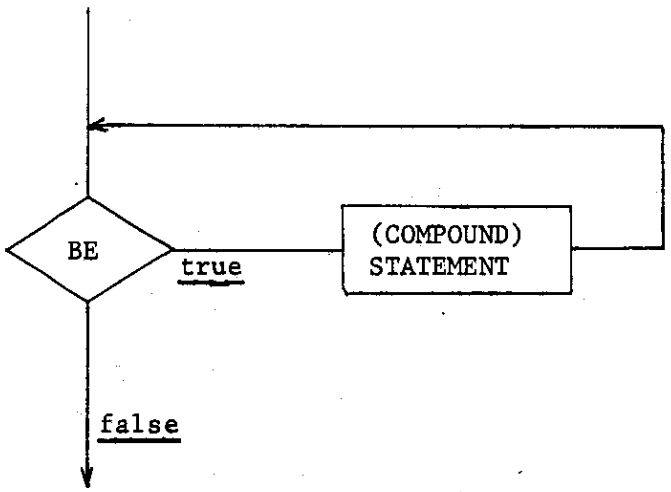
Zo ontstaat een nieuw statement, een compound statement. Het mag op elke plaats staan in een programma waar een statement kan staan.

3.5. Herhaalde uitvoering van statements

Aan de hand van een eenvoudig voorbeeld zullen we aangeven hoe een herhaling in de programmatekst kan worden weergegeven. Daarna zullen we enkele aspecten van deze faciliteit aan de hand van een tweede voorbeeld toelichten.
Stel dat we bij gegeven waarde van n ($n \geq 1$) de kleinste macht van 2 willen bepalen die groter is dan n. Dit kan met behulp van

```
tweemacht := 2;
while tweemacht ≤ n do tweemacht := tweemacht * 2
```

De statement tweemacht := tweemacht * 2 zal dan uitgevoerd worden zolang de waarde van tweemacht niet groter is dan n, dus zolang deze boolean expressie de waarde true heeft.
Algemeen geldt dat de statement na do wordt uitgevoerd, wanneer na berekening van de waarde van de boolean expressie, die tussen while en do staat, blijkt dat deze de waarde true heeft. Dit wordt herhaald totdat deze boolean expressie de waarde false krijgt.



Stel nu dat we de kleinste g moeten uitrekenen, waarvoor geldt dat

$$\sum_{j=1}^{g} j^2 \geq 10000 .$$

In paragraaf 1.6 werd reeds gewezen op relaties tussen variabelen, die voortvloeien uit de betekenis van deze variabelen. Met name werd daar gewezen op de invariante relatie als nuttig en zelfs noodzakelijk hulpmiddel om tot een correct programma te komen. We komen daar bij dit voorbeeld op terug.
We kunnen het probleem oplossen door achtereenvolgende deelsommen

$$(\sum_{j=1}^{1} j^2, \sum_{j=1}^{2} j^2, \sum_{j=1}^{3} j^2, \ldots)$$

te bepalen en zodra een deelsom groter of gelijk 1000 is te stoppen. Als we een deelsom hebben (zeg $\Sigma_{j=1}^{n} j^2$), kunnen we gemakkelijk de volgende deelsom bepalen, immers

$$\sum_{j=1}^{n+1} j^2 = \sum_{j=1}^{n} j^2 + (n+1)^2 .$$

Dit suggereert dat, als we een goed startpunt weten te vinden, we het probleem kunnen oplossen met behulp van een herhaling van de volgende twee "statements

"maak volgende term"
"tel deze term op de huidige deelsom en vind zo de nieuwe deelsom"

We voeren twee variabelen in, g en som, waarmee we de toestand van het proces op elk moment vastleggen. De invariante relatie tussen g en som luidt

$$\text{som} = \sum_{j=1}^{g} j^2 .$$

Deze relatie willen we steeds laten gelden voordat en nadat bovenstaande "statements" uitgevoerd worden. We kunnen deze "statements" nu preciseren (en iets anders opschrijven):

```
g := g + 1;
som := som + g ↑ 2 .
```

Als we de statements in deze volgorde laten uitvoeren, zal na uitvoering van g := g + 1 de relatie tussen g en som zijn verstoord; de relatie wordt weer hersteld door uitvoering van de tweede statement.
De herhaling van de uitvoering van deze statements moet gebeuren zolang de som kleiner dan 10000 is. We krijgen dan:

```
while som < 10000 do
      begin g := g + 1;
            som := som + g ↑ 2
      end
```

We moeten er nog voor zorgen dat de relatie tussen g en som ook geldt aan het begin. Er is dan nog geen enkel getal gekwadrateerd en bij de som opgeteld; we kunnen g en som beide de waarde 0 geven. We krijgen dan als programma:

```
begin integer g, som;
      g := 0; som := 0;
      while som < 10000 do
            begin g := g + 1;
                  som := som + g ↑ 2
            end;
      write(som)
end
```

Nog enkele voorbeelden.

a) Gegeven een rij van honderd getallen $g_1, g_2, g_3, \ldots, g_{100}$. Druk deze getallen af met hun som, hun gemiddelde en het verschil van ieder getal met het gemiddelde.
   We introduceren de volgende variabelen:

   g: een array met grenzen [1:100] voor het opbergen van de rij; g[k] krijgt als waarde het k-de getal uit de rij
   i: de index van het laatste array-element, dat een waarde heeft gekregen
   j: de index van het volgende array-element, dat bij het afdrukken een rol speelt

som : de som van de eerste i getallen
gem : het gemiddelde van de honderd getallen.

```
begin integer array a[1:100]; integer i,j,som; real gem;
      som := 0; i := 0;
      while i ≠ 100 do
            begin i := i + 1;
                  read(a[i]); write( a[i]);
                  som := som + a[i]
            end;
      gem := som/100;
      write(gem); write(som);
      j := 1;
      while j ≤ 100 do
            begin write(a[j] - gem);
                  j := j + 1;
            end
end
```

We hebben in dit programma een i en een j geintroduceerd en de twee herhalingen verschillend (i begint met 0, j met 1) geprogrammeerd. Over het algemeen zal men in soortgelijke situaties met één variabele volstaan en de herhalingen op dezelfde wijze noteren.

b) Gegeven een rij gehele getallen, ieder groter dan 10 en kleiner dan 21, afgesloten door een 0. Gevraagd wordt te bepalen hoe vaak iedere waarde voorkomt in de rij.
We introduceren als variabelen:

getal: getal uit de rij dat als volgende verwerkt wordt
tel : een array met grenzen [11:20] dat de tellers voorstelt, die aangeven hoe vaak een waarde voorkomt; tel[i] (10 < i < 21) geeft aan hoe vaak de waarde i geconstateerd is
i : geeft aan
- in de eerste herhaling: de index van het volgende array-element dat op 0 wordt gesteld;
- in de tweede herhaling: de index van het volgende array-element dat afgedrukt wordt.

```
begin integer array tel[11:20]; integer getal,i;
      i := 11;
      while i ≤ 20 do begin tel[i] := 0; i := i + 1 end;
      read(getal);
      while getal ≠ 0 do
            begin tel[getal] := tel[getal] + 1;
                  read(getal)
            end;
      i := 11;
      while i ≤ 20 do begin write(tel[i]); i := i + 1 end
end
```

c) Gegeven een rij van 101 positieve gehele getallen: $g_1, g_2, \ldots, g_{101}$. Gegeven is bovendien dat g_{101} gelijk is aan tenminste één van de getallen $g_1, g_2, \ldots, g_{100}$. Bepaal de kleinste index waarvoor deze gelijkheid optreedt.
Als variabelen introduceren we:

g : een array met grenzen[1:100]; g[i] heeft (krijgt) als waarde g_i ($1 \leq i \leq 100$)

getal: g_{101}

i : - in de eerste behandeling: index van het volgende array-element dat een waarde krijgt;
- in de tweede herhaling: index van volgend element van g dat geprobeerd wordt in de gelijkheidstest.

```
begin integer array g[1:100]; integer getal,i;
      i := 1;
      while i ≤ 100 do
            begin read(g[i]);
                  i := i + 1
            end;
      read(getal);
      i := 1;
      while getal ≠ g[i] do i := i + 1;
      write(i)
end
```

3.6. Het voorwaardelijk of het selectief uitvoeren van statements

Zoals al eerder opgemerkt, is er vaak behoefte aan de faciliteit om, afhankelijk van een bepaalde toestand, een statement al dan niet te laten uitvoeren of een keuze te maken tussen de uitvoering van de ene of de andere statement.

Als we bijvoorbeeld aan de variabele x de absolute waarde van x willen toekennen, dan kan dit door uitvoering van de statement

```
if x < 0 then x := -x
```

Dit is een voorbeeld van een zogeheten conditional statement. In het algemeen zal tussen if en then een boolean expression staan en na then een statement (zie ook het einde van deze paragraaf voor beperkingen). Als de evaluatie van de boolean expression de waarde true oplevert zal de statement na then worden uitgevoerd; is de waarde false dan zal verder worden gegaan bij de statement volgend op de conditional statement. Als meerdere statements onder dezelfde voorwaarde uitgevoerd moeten worden, kunnen we weer gebruik maken van de compound statement.

Een voorbeeld van het kiezen tussen twee statements, is het toekennen van de absolute waarde van de variabele x aan de variabele y:

```
if x < 0 then y := -x else y := x
```

Dit is een zogeheten alternatieve statement. In het algemeen geldt ook hier dat tussen if en then een boolean expression staat, tussen then en else een statement (zie beperkingen verderop) en na else een statement. Als de evaluatie van de boolean expression de waarde true oplevert, wordt de statement tussen then en else uitgevoerd; is deze waarde false dan wordt de statement na else uitgevoerd.

De mogelijkheid van zowel conditional als alternative statements heeft wel consequenties voor het soort constructies dat na then is toegestaan. Schrijven we bijvoorbeeld de volgende constructie op

```
if C1 then if C2 then A else B
```

waarin C1 en C2 boolean expressions zijn en A en B bijvoorbeeld assignment statements, dan zou deze op twee manieren geïnterpreteerd kunnen worden. Het zou geïnterpreteerd kunnen worden als een conditional statement waarbij na

then (na C1) een alternative statement staat (zie fig. 1), of als een alternative statement waarbij na then (na C1) een conditional statement staat (zie fig. 2).

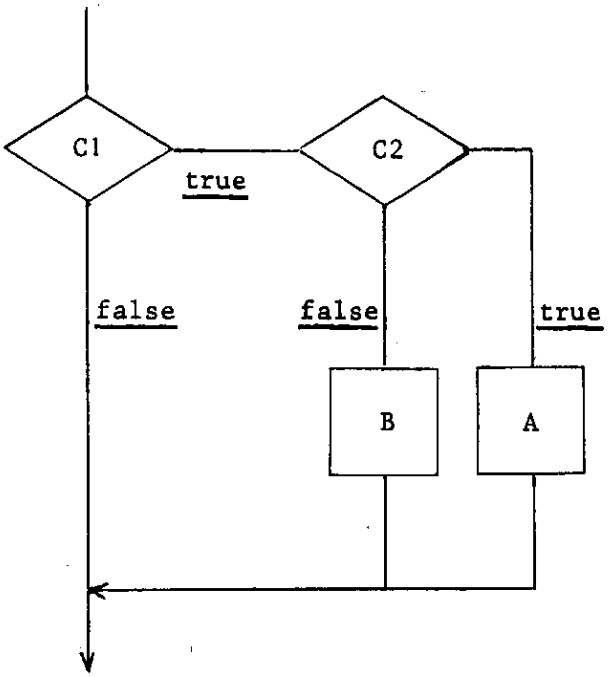


fig. 1

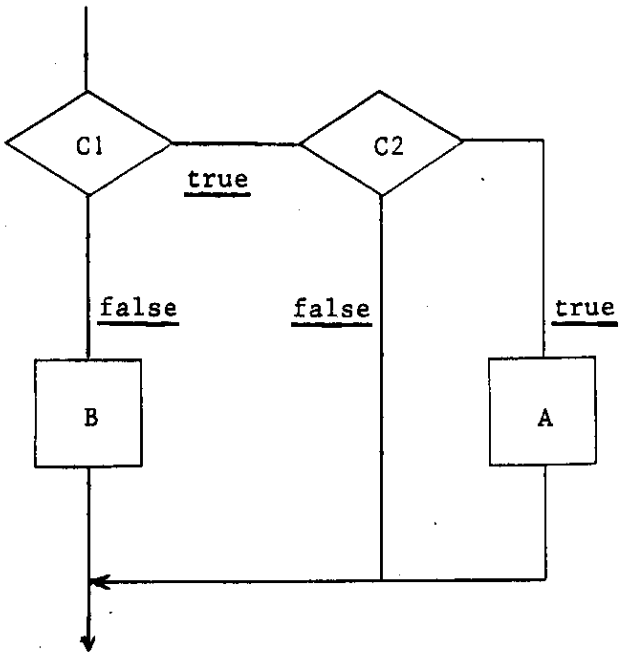


fig. 2

Deze dubbelzinnigheid moet met behulp van afspraken worden opgelost.
We zullen ons houden aan de volgende afspraak: then mag niet gevolgd worden door een alternative of conditional statement. Als men toch zo'n statement wil laten uitvoeren, kan men de statement omsluiten door het hakenpaar begin end, waardoor we een compound statement krijgen. Zo kunnen we de twee hierboven genoemde interpretaties in correct Algol 60 weergeven met:

```
if C1 then begin if C2 then A else B end
resp.  if C1 then begin if C2 then A end else B
```

Na then mag evenmin een herhalingsopdracht staan; een constructie als

```
if C1 then while C2 do if C2 then A else B
```

is namelijk weer op twee manieren te interpreteren. Ook hier kan gebruik gemaakt worden van het hakenpaar begin - end voor het maken van een eenduidig interpreteerbare constructie.

3.7. Voorbeelden

a) Bereken de hoek tussen de twee vectoren $\underline{a}$ en $\underline{b}$ uit $\mathbb{R}^{100}$. De componenten zijn in een invoerrij gegeven. We maken gebruik van $(\underline{a},\underline{b}) = |\underline{a}|.|\underline{b}|\cos\alpha$ en veronderstellen voorts dat een arccos-functie gebruikt mag worden. Als $\underline{a}$ of $\underline{b}$ de nulvector is, drukken we een foutboodschap af. We introduceren als variabelen:

a,b : arrays met grenzen [1:100]; a[i] bevat a_i, b[i] bevat b_i
ab : voor het inproduct $(\underline{a}.\underline{b})$
aabs : voor de lengte van $\underline{a}$
babs : voor de lengte van $\underline{b}$
alpha: voor de hoek α
i : - index van het volgende array-element dat een waarde krijgt
- index van de volgende componenten van $\underline{a}$ en $\underline{b}$ die met elkaar vermenigvuldigd worden (bij het inproduct)
som : bij het inlezen van de componenten bepalen we direct de som van de kwadraten.

```
begin integer i; real som, ab, aabs, babs, alpha;
      real array a,b [1:100];
      som := 0; i := 1;
      while i ≤ 100 do
            begin read(a[i]); som := som + a[i] * a[i]; i := i + 1 end;
      aabs := sqrt(som);
      som := 0; i := 1;
      while i ≤ 100 do
            begin read(b[i]); som := som + b[i] * b[i]; i := i + 1 end;
      babs := sqrt(som);
      ab := 0; i := 1;
      while i ≤ 100 do
            begin ab := ab + a[i] * b[i]; i := i + 1 end;
      if aabs ≠ 0 and babs ≠ 0
         then begin alpha := arccos(ab/(aabs * babs));
                    write(alpha)
              end
         else write("a of b is de nulvector")
end
```

b) Gevraagd wordt een tabel af te drukken voor x = 0(0.1)1 van de sinusfunctie (deze functie mag als operand gebruikt worden, zie par. 2.4). We introduceren als variabele:

i: deze variabele loopt van 0 tot en met 10 met stappen van 1; steeds geeft i de waarde van 10 * x als we voor deze x de waarde van de functie willen afdrukken.

```
begin integer i; real x;
      i := 0;
      while i ≤ 10 do
            begin x := 0.1 * i; write(sin(x)); i := i + 1 end
end
```

We gebruiken in dit soort situaties in de boolean expressie integers in plaats van reals om geen moeilijkheden te krijgen met mogelijke afrondingsproblemen.

c) Gegeven een rij van honderd gehele getallen $a_1, a_2, a_3, \ldots, a_{100}$. Deze rij getallen dient in omgekeerde volgorde afgedrukt te worden. Een beperking bij dit afdrukken is dat er slechts zeven getallen op een regel kunnen. Na het afdrukken van de honderd getallen moeten op een nieuwe regel het maximum en het minimum van de gelezen getallen afgedrukt worden. Voor de overgang op het begin van een nieuwe regel zullen we nlcr (new-line-carriage-return) schrijven.
We voeren de volgende variabelen in:

i : index van het laatste element dat in het array gezet is
k : index van het volgende element dat afgedrukt wordt
j : het aantal getallen dat op een regel geplaatst is
a : array met grenzen [1:100]; hierin worden de honderd getallen geplaatst; a[i] bevat a_i
max: de maximale waarde van in het array geplaatste getallen
min: de minimale waarde van in het array geplaatste getallen.

```
begin integer i,j,k,max,min; integer array a[1:100];
      read(a[1]); max := a[1]; min := a[1];
      i := 1;
      while i ≠ 100 do
            begin i := i + 1;
                  read(a[i]);
                  if a[i] > max
                     then max := a[i]
                     else if a[i] < min then min := a[i]
            end;
      k := 100; nlcr; j := 0;
      while k ≥ 1 do
            begin if j = 7 then begin nlcr; j := 0 end;
                  write(a[k]); j := j + 1;
                  k := k - 1
            end;
      nlcr; write(max); write(min)
end
```

Als we de getallen niet in omgekeerde volgorde moeten afdrukken, dan hebben we het array a niet nodig. We kunnen dan volstaan met de enkelvoudige variabele getal die steeds het getal uit de rij aangeeft dat vergeleken wordt met het heersende maximum en het heersende minimum en afgedrukt wordt. De variabele i geeft dan het aantal verwerkte getallen aan.

```
begin integer i,j,max,min,getal;
      read(getal); max := getal; min := getal;
      i := 1; nlcr; j := 0;
      while i ≠ 100 do
            begin read(getal);
                  if getal > max
                     then max := getal
                     else if getal < min then min := getal;
                  if j = 7 then begin nlcr; j := 0 end;
                  write(getal); j := j + 1;
                  i := i + 1
            end;
      nlcr; write(max); write(min)
end
```

d) Gegeven is een invoerrij van tweehonderd gehele getallen:

$$x_1, x_2, \ldots, x_{100}, y_1, y_2, \ldots, y_{100} .$$

Gevraagd wordt vast te stellen of het rijtje x-en elementsgewijs gelijk is aan het rijtje y-en. Zo ja, dan moet "gelijk" worden afgedrukt; zo nee, dan moet "ongelijk" worden afgedrukt.

Als we de x-en in een array plaatsen, kunnen we daarna de y-en stuk voor stuk verwerken, d.w.z. dat we iedere y vergelijken met de overeenkomende x. We voeren de volgende variabelen in:

x : array met grenzen [1:100]; hierin worden de eerste honderd getallen uit de rij geplaatst: x[i] bevat x_i,

y : de uit de rij geselecteerde waarde die met de overeenkomstige x vergeleken wordt,

i : - index van het volgende element van x dat een waarde krijgt
- rangnummer van het getal uit het tweede deel van de rij dat vergeleken wordt met de x met ditzelfde getal als index.

equal: boolean variabele; deze heeft bij een bepaalde waarde van i ($1 < i \leq 100$), de betekenis dat $x_1 = y_1, x_2 = y_2, \ldots, x_{i-1} = y_{i-1}$ (de waarde van equal is dan true) of dat $x_1 = y_1, x_2 = y_2, \ldots, x_{i-2} = y_{i-2}$ en $x_{i-1} \neq y_{i-1}$ (de waarde van equal is dan false); voor i = 1 geven we equal de waarde true.

Het vergelijken van de getallen kan gestopt worden als alle honderd y-en verwerkt zijn ($i > 100$), of als er een verschil is geconstateerd (equal heeft de waarde false). Er moet dus nog een volgende y met een x vergeleken worden als geldt:

$i \leq 100$ and equal

Als er gestopt is en equal heeft de waarde true (dan zijn we gestopt omdat i = 101) dan geldt dus dat de rijen overeenkomen; zijn we gestopt met voor equal de waarde false dan is er in de (deel)rij verschil geconstateerd.

Aan equal kennen we aanvankelijk toe de waarde true omdat er dan nog geen verschil is geconstateerd.

```
begin integer i,y; boolean equal; integer array x[1:100];
      i := 1;
      while i ≤ 100 do begin read(x[i]); i := i + 1 end;
      i := 1; equal := true;
      while i ≤ 100 and equal do
            begin read(y);
                  equal := y = x[i];
                  i := i + 1
            end;
      if equal then write("gelijk") else write("ongelijk")
end
```

Als er verschil wordt geconstateerd, wordt er geen y meer gelezen. Als het probleem was geweest dat we drie rijen x- en en y-en zouden moeten vergelijken, zouden we in moeilijkheden kunnen komen als we niet steeds ook de hele rij y-en zouden lezen.

e) Gegeven is een rij positieve gehele getallen, afgesloten door een 0. De som moet bepaald worden van het 1^e,3^e,6^e,10^e,... getal uit de rij.

Wat het inlezen betreft, worden alle getallen op dezelfde wijze behandeld. Wat de verwerking betreft, is er verschil in behandeling; er zijn "te sommeren" getallen (het 1^e,3^e,6^e,10^e,... getal), die bij elkaar opgeteld moeten worden (in bijvoorbeeld een variabele som) en "niet te sommeren" getallen, die na de selectie verder genegeerd kunnen worden.
We introduceren een variabele som met als betekenis: som is gelijk aan de som van de te sommeren getallen tot en met het laatst verwerkte te sommeren getal.
Een te sommeren getal is dus een getal waarvan het rangnummer in de rij tot de kandidaten (1,3,6,10,...) behoort. De rij van kandidaten wordt als volgt gevonden: De eerste kandidaat is 1, het verschil tussen de eerste en de tweede kandidaat is 2 en het verschil tussen ieder volgend tweetal opeenvolgende kandidaten is 1 groter dan het verschil tussen het direct daaraan voorafgaand tweetal opeenvolgende kandidaten.

De toestand is nu op elk moment volledig bepaald door de waarden van de volgende variabelen:

getal: het laatst geselecteerde getal uit de rij
i : het rangnummer van dit laatst gelezen getal
k : het rangnummer van het eerst volgende te sommeren getal
v : het verschil tussen k en het rangnummer van het voorafgaand te sommeren getal
som : zie boven.

Uit deze beschrijving volgt nu direct het programma:

```
begin integer getal,i,k,v,som;
      k := 1; v := 1; som := 0;
      read(getal); i := 1;
      while getal ≠ 0 do
            begin if i = k then begin som := som + getal;
                                      v := v + 1;
                                      k := k + v
                                end;
                  read(getal); i := i + 1
            end;
            write(som)
end
```

3.8. Structuur van een programma

In de voorbeelden hebben we al gezien dat een programma begint met begin, dan volgen de declaraties (onderling gescheiden door een puntkomma), dan volgen de statements (waarbij de eerste statement van de declaraties wordt gescheiden door een puntkomma en ook de statements onderling worden gescheiden door een puntkomma) en tenslotte volgt end.

3.9. Processen en deelprocessen

3.9.1. Inleiding

Een actie hebben we in 1.2 leren kennen als een transformatie van een toestand die in een eindige tijd plaatsvindt en wel gedefinieerd is. In deze omschrijving van een actie wordt alleen gesproken over wàt er plaatsvindt, te weten de transformatie van begin- naar eindtoestand. Als we er ook in geïnteresseerd zijn hòe deze transformatie in detail tot stand komt, spreken we van een deelproces, dat optreedt als onderdeel van het totale proces waarvan de actie deel uitmaakt. De beschrijving van een deelproces kan op twee manieren in de beschrijving van het totale proces worden opgenomen.

De eerste manier is door op de plaats waar de actiebeschrijving moet staan, het hele deelproces uit te schrijven. Meestal spreken we dan van een blok. Het blok kent voor de procesbeschrijving binnen dat blok een eigen toestandsruimte, die aan het begin van het blok vastgelegd wordt: de declaratie van de locale variabelen. Daarnaast zullen in het blok ook variabelen uit de omgeving (globale variabelen) een rol spelen; dit zijn variabelen waarvan de waarden in het blok gebruikt worden en variabelen waaraan in het blok waarden kunnen worden toegekend die later in de omgeving worden gebruikt. Nadat het deelproces, zoals beschreven in het blok, is uitgevoerd, bestaat de locale toestandsruimte niét meer en het enige dat overblijft zijn de resultaten die in globale variabelen zijn achtergelaten.

De tweede manier waarop de deelprocesbeschrijving in de totale procesbeschrijving kan worden opgenomen, laat beter dan bij het blok het verschil zien tussen wàt en hòe. In de procesbeschrijving wordt wàt er moet gebeuren, beschreven door middel van een procedure statement. De uitvoering hiervan heeft een activering tot gevolg van een (in een zogeheten procedure declaratie beschreven) mechanisme, dat vertelt hòe het moet gebeuren. Mechanismen die we kunnen activeren (zonder dat we ze zelf behoeven te declareren) hebben we al gezien: het read-mechanisme en het write-mechanisme.

3.9.2. Het blok

Aangezien een blok een procesbeschrijving bevat met een eigen (locale) toestandsruimte, ligt het voor de hand dat een blok als volgt opgebouwd is:

begin declaraties; statements end

(dus op dezelfde wijze als een programma). We zullen, mede met behulp van enkele voorbeelden, verschillende facetten van locale en globale variabelen in een blok nader toelichten.
Een variabele wordt gekenmerkt door een naam. Het is toegestaan om dezelfde naam zowel voor een globale als voor een locale variabele te gebruiken. Deze variabelen kunnen dan uiteraard niet naast elkaar in het blok een rol vervullen. Dit houdt in dat deze gemeenschappelijke naam

- buiten het blok uitsluitend betrekking zal hebben op de globale variabele
- binnen het blok uitsluitend betrekking zal hebben op de locale variabele.

De locale variabele heeft slechts betekenis in het blok ("de scope van een locale variabele is tot het betreffende blok beperkt"). De globale variabele met dezelfde naam als de locale variabele is binnen het blok onbekend en onbereikbaar. Bij het verlaten van een blok verliezen de locale variabelen hun betekenis en ook hun waarde; ze bestaan niet meer. Dit impliceert dat op het moment van het opnieuw binnenkomen van hetzelfde blok (hetgeen mogelijk is daar een blok overal mag staan waar een statement mag staan) de waarden van de lokale variabelen ongedefinieerd zijn.
In het voorbeeld

```
begin integer a,b,c;
      ----------
      begin integer c,d,e;
            ----------
      end;
      ----------
end
```

bestaan de variabelen a, b en c van het buitenblok (programma) in het gehele programma, terwijl de variabelen c, d en e (in het binnenblok gedeclareerd) alleen bestaan in dit binnenblok; de variabele c van het buitenblok is in het binnenblok niet bereikbaar; als in het binnenblok een c voorkomt dan is dit de c van het binnenblok; na het verlaten van het binnenblok is de c van het buitenblok weer beschikbaar.

De grenzen in een array-declaratie worden bij activering van een blok (opnieuw) berekend. Dit geeft ons de mogelijkheid om dynamisch (tijdens de uitvoering van het programma) de grenzen van een array te laten bepalen:

```
begin integer n;
      read(n);
      begin real array A[1:n];
            ----------
      end
end
```

We kunnen dit niet bereiken met bijvoorbeeld:

```
begin integer n; read(n); real array A[1:n] ..... end
```

omdat na een statement geen declaraties kunnen volgen in hetzelfde blok. Het grote voordeel van de blokstructuur is dat we bij de programma-opbouw kunnen denken in acties die we later als blokken uitschrijven.

3.9.3. De procedure

Bij het begrip procedure spelen drie aspecten een rol:

- Het effect van de procedure, wàt bewerkstelligt de procedure. Diegene die een al gedeclareerde procedure wil gebruiken zal een eenduidige definitie wensen van het effect van de procedure. De exacte beschrijving van het effect van de procedure is echter ook van belang voor de "constructeur", want aan de hand hiervan moet hij de procedure "construeren".
  Men kan om het lezen van een programma te vereenvoudigen, in de naam van de procedure proberen uit te drukken wat het effect is van de procedure.
- De relatie van de procedure met de omgeving. Deze relatie wordt gerealiseerd via parameters. In de declaratie van de procedure worden "formele" parameters (namen) vastgelegd. Bij de activering van de procedure worden "actuele" parameters opgegeven. Via deze actuele parameters krijgt de procedure invoerwaarden uit de omgeving (invoerparameters) en kan de procedure de berekende waarden afleveren aan de omgeving (uitvoerparameters). Juist in de parameters kan het effect van de procedure uitgedrukt worden.

- De beschrijving van de wijze waarop de procedure de voor het gewenste effect geëiste relatie tussen invoer- en uitvoerparameters tot stand brengt (hòe werkt de procedure). Dit komt neer op een beschrijving van het proces, een schema van acties dat, uitgaande van een bepaalde begintoestand (bepaald door de invoerwaarden) een bepaald eindtoestand (vastgelegd in de uitvoervariabelen) oplevert. Deze beschrijving van het effect wordt de body van de procedure genoemd. De body is een blok waarin naast de parameters ook locale variabelen een rol kunnen spelen.

In het voorafgaande zijn we uitgegaan van een algemeen schema om, door invulling van de actuele parameters te komen tot een bepaalde actie. We kunnen ook in de omgekeerde richting de zaak bekijken. Zo kan voorbeeld 1.6.1 worden opgevat als de beschrijving van de actie: "Bepaal quotiënt en rest van a en b (positief geheel) en leg de resultaten vast in q en r ($a = q * b + r$ and $0 \leq r < b$)". Deze actie is een bijzonder geval van de algemenere actie (het algemene schema). Bepaal quotiënt en rest van twee positieve gehele getallen (de invoerparameters) en ken de resultaten toe aan twee variabelen (de uitvoerparameters)". Deze algemene actie kunnen we opvatten als de activering van een procedure, die we de naam divmod geven. De specifieke actie op a, b, q en r noteren we dan als divmod (q,r,a,b).
We gaan nu de procedure divmod construeren (declareren). In de procesbeschrijving van de procedure noemen we de invoerparameters deeltal en deler, de uitvoerparameters noemen we quot en rest. Van de parameters moeten we aangeven van welk type ze zijn. Van deeltal en deler geven we bovendien (d.m.v. value) aan dat het gaat om de waarden van deze parameters (want het zijn invoerparameters). De declaratie ziet er als volgt uit:

```
procedure divmod(quot,rest,deeltal,deler);
        value deeltal,deler; integer quot,rest,deeltal,deler;
        begin quot := 0; rest := deeltal;
              while rest ≥ deler do
                    begin rest := rest - deler; quot := quot + 1 end
        end
```

Uitvoering van de procedure statement divmod(q,r,a,b) houdt de activering in van het in de proceduredeclaratie beschreven proces. Deze activering bestaat uit twee stappen:

(1) parameterinitialisering
Dit houdt in dat de namen deeltal en deler staan voor de waarden van a, resp. b, en dat de namen quot en rest staan voor de variabelen q resp. r.

(2) uitvoering van de body
Bij deze uitvoering zullen voor deeltal en deler steeds de waarden van a en b genomen worden en zal bij toekenning aan quot en rest in feite een toekenning aan q en r plaatsvinden.

Opmerking 1. Een programma kan niet alleen als blok opgevat worden, maar ook als een procedure (zonder parameters). Het verschil is, dat een procedure geactiveerd wordt door een aanroep, terwijl een programma geactiveerd wordt door het aanbieden aan een computersysteem. Een procedure is ingebed in een programma, een programma in een "omgeving", die voor de uitvoering van het programma zorgt. In deze omgeving zijn bepaalde (standaard) procedures gedeclareerd, die we dan ook in onze programma's mogen gebruiken zonder ze daarin te declareren.

Opmerking 2. De aanroep van een procedure heeft tot gevolg, dat zoals bij een blok een deelproces wordt uitgevoerd, dat een eigen toestandsruimte heeft. Na uitvoering van het deelproces bestaat deze toestandsruimte niet meer.

3.9.4. Routine en functieprocedures

In het voorgaande is een beeld geschetst van wat we verder "routineprocedures" zullen noemen. Uitvoering van een dergelijke procedure heeft tot gevolg dat uit één of meer invoerparameters de waarden voor één of meer uitvoervariabelen worden berekend volgens het in de body van de procedure vastgelegde patroon. De functie van één uitvoerparameter kunnen we laten overnemen door de procedurenaam. In dat geval spreken we van een functieprocedure die we, zoals we reeds in 2.4 opgemerkt hebben, overal mogen aanroepen op de plaatsen, waar een variabele in een expressie mag staan. De functienaam in zo'n aanroep noemen we een function designator. Een aantal veel gebruikte functieprocedures met hun standaardnamen is in de programmeertaal beschikbaar.

Daar in een functieprocedure een berekende waarde toegekend wordt aan de procedurenaam, moet in de declaratie aangegeven worden van welk type deze waarde zal zijn. (Vandaar dat functieprocedures ook wel typed procedures genoemd worden en routineprocedures non-typed) Voorbeelden:

a) een procedure om de grootste van 2 getallen te bepalen:

```
real procedure max (x,y); value x,y; real x,y;
max := if x > y then x else y;
```

Opmerking. In de body mag begin - end weggelaten worden omdat er maar 1 statement is.

b) een functieprocedure om de som van de elementen van een vector a te bepalen

$$\sum_{i=m}^{n} a_i$$

```
integer procedure som (a,m,n); value m,n; integer m,n; integer array a;
        begin integer s,i;
              s := 0; i := m;
              while i ≤ n do begin s := s + a[i]; i := i + 1 end;
              som := s
        end
```

c) een functieprocedure om na te gaan of in het array a[h,k] een element voorkomt met de waarde w:

```
boolean procedure found (a,h,k,w); value h,k,w; integer h,k,w; integer array a;
        begin integer i; boolean seen; seen := false; i := h;
              while not seen and i ≤ k do begin seen := a[i] = w;
                                                i := i + 1
                                          end;
              found := seen
        end
```

Enkele details van deze procedures, zoals het gebruik van value en locale variabelen, zoals s en seen, moeten nader toegelicht worden.

5. ALGOL 60

5.1. Inleiding

In de voorgaande hoofdstukken lag het accent op het programmeren, daarbij gebruik makend van slechts enkele programmeringsprimitiva die geïnspireerd zijn door Algol. In de praktijk maakt men gebruik van uitgebreider programmeringsgereedschap. Dit opent de mogelijkheid om vaak voorkomende programmaconstructies snel op te schrijven. In dit hoofdstuk zullen daarom nog enkele algemeen aanvaarde hulpmiddelen besproken worden, evenals enige programmeringsprimitiva die ook wel eens van pas komen.

Voor een behandeling van de exacte definitie van de toegelaten schrijfwijze van Algol-teksten verwijzen wij naar de appendix over de syntaxis van Algol.

5.2. Variabelen, expressies

5.2.1. Typen, variabelen, getalrepresentatie

Reeds genoemd zijn de variabelen van het type:

- boolean, die slechts de waarden true en false kunnen aannemen;
- integer, die waarden uit Z^* kunnen aannemen, waarbij Z^* een aaneengesloten eindige deelverzameling van de verzameling Z der gehele getallen is;
- real, die waarden uit Q^* kunnen aannemen, waarbij Q^* een eindige deelverzameling is van de verzameling Q der rationale getallen en dus ook van de verzameling R der reële getallen.

Hierbij is nog het volgende op te merken:

- dat Z^* een eindige deelverzameling is van Z en Q^* van Q en dat reële getallen benaderd moeten worden met rationele getallen, hangt samen met de slechts discrete representatiemogelijkheid in rekenautomaten. Op deze representatie van integers en reals en de onder- en bovengrens van Z^* en Q^* gaan we hier niet in. In de meeste grote rekenautomaten zijn getallen wel met een nauwkeurigheid van ongeveer 8 decimalen te representeren, terwijl de automaten een of andere waarschuwing geven wanneer men buiten de grenzen van Z^* en Q^* komt. De numeriek wiskundige die wil garanderen dat zijn programma's feilloos werken, zal wèl precies op de hoogte moeten zijn

van de representatie en de arithmetiek van integers en reals in de betreffende automaat.
- soms komt men "double length integers" en "double length reals" tegen. Dit betekent dan dat zowel de representatienauwkeurigheid van reals ("het aantal decimalen") veel groter is, alsook de grenzen van Z^* en Q^* veel ruimer zijn dan bij de corresponderende "enkele lengte" grootheden.
- om te vermijden dat bewerkingen met complexe getallen uitgeschreven moeten worden voor de reële en imaginaire componenten afzonderlijk, komt men soms variabelen van het type "complex" tegen.

Voor veel programmeringsopgaven (bij niet-numerieke toepassingen) is behoefte aan variabelen van het type string, die als waarde een "tekst" kunnen aannemen. Een tekst is daarbij een rij van karakters uit een gegeven "alfabet" van bijvoorbeeld letters, cijfers en leestekens. Bewerkingen op deze variabelen zijn dan o.a. de volgende:
- concatenatie, d.w.z. aaneenkoppeling van twee strings,
- splitsing van een string in deelstrings,
- het opzoeken van een bepaalde deelstring in een string.

In Algol 60 is het begrip string wel ingevoerd, maar zijn geen bewerkingen op strings gedefiniëerd. Strings kunnen zodoende alleen maar als actuele parameters van procedures (bijvoorbeeld bij read en write) gebruikt worden. Gezien dit uiterst beperkte nut van strings gaan we niet verder op de in Algol 60 voorgeschreven representatie ("schrijfwijze") van strings in.

De in Algol 60 toegelaten schrijfwijze voor getallen kunnen we als volgt kort samenvatten.

$$\pm\ [\text{natgetal}]\ [\ .\ \text{natgetal}\ ]\ [_{10}\ \pm\ \text{natgetal}]$$

met de volgende interpretetie:

$\pm$	een van de twee tekens wordt desgewenst gekozen;
natgetal	natuurlijk getal (inclusief 0);

[...] constructie tussen deze haken mag in zijn geheel ontbreken, doch tenminste één van de drie [··] constructies moet gekozen worden;

$_{10}$ symbool voor grondtal 10.

Toegelaten zijn dus getallen als

. 123 - 1.23 + 0123 $+ _{10} - 5$ $- 12.3 _{10} + 9$

maar niet "getallen" als

. 12.3 ++ 123 - 123. $+ . _{10} - 5$ $- 12.3 _{10} + 9.8$

hetgeen vrijwel geheel aansluit bij onze normale schrijfwijze.

Opmerking. De ponscode voor $_{10}$ kan plaatselijk verschillen.

5.2.2. Arithmetische expressies

Hoe arithmetische expressies (AE) worden gevormd, is reeds besproken, echter niet wat het type van een AE is. Het type van een AE is afhankelijk van de operanden en de operatie (zie onderstaande tabel, waarin i een integer operand, r een real operand en a een integer of real operand voorstelt). Verder moet nog het volgende worden opgemerkt:

- Bij uitvoering kan de waarde van een expressie buiten de grenzen van het betreffende type (dus buiten de grenzen van Z^* of Q^*) komen te liggen. Als regel zal dan door de rekenautomaat een "overflow"-boodschap worden geproduceerd.
- Ook kan de situatie "ongedefiniëerd" optreden, als n.l. een operatie niet gedefiniëerd is voor bepaalde operand-waarden. Zo is de deling ongedefiniëerd als de deler de waarde 0 heeft.

Opmerking. Voor integers m en n geldt

$m \underline{\text{div}} n = \text{sign}(m * n) * q$

$m \underline{\text{mod}} n = \text{sign}(m * n) * r$

als $|m| = q * |n| + r$ met $0 \leq r < |n|$ en $n \neq 0$

bewerking	operandpaar	resultaat	
		type	ongedefinieerd als
optelling aftrekking vermenigvuldiging	i i	i	
	i r r r	r	
deling	a a	r	deler = 0
gehelendeling restbepaling	i i	i	deler = 0
machtsverheffing	a^i	a als i > 0 a als i = 0 r als i < 0	a = 0
	a^r	r als a > 0 r als a = 0 -	r ≤ 0 a < 0

5.2.3. Boolean expressies

Genoemd zijn al de variabelen van het type
- boolean, die slechts de 2 waarden true en false kunnen aannemen en waarvoor de operatoren not, and en or genoemd zijn.

Een volledige tabel van Algol 60 binaire boolean operatoren in prioriteitsvolgorde is:

operatie	notatie			a: f f t t b: f t f t
	progr. taal	wiskundig	"omgangstaal"	
conjunctie	a and b	a ∧ b	a èn b	f f f t
disjunctie	a or b	a ∨ b	a of b	f t t t
implicatie	a imp b	a → b	als a dan b	t t f t
equivalentie	a eqv b	a ≡ b	b alleen dan als a	t f f t

Zoals door uitschrijven is aan te tonen, kunnen we met de not operator en (o.a. de eerste) twee operatoren de werking van de andere operatoren representeren. Bijvoorbeeld:

a eqv b door (a and b) or (not a and not b)
a imp b door not a or b

Opmerking. In de wiskundige logica komt men nog de in de volgende tabel genoemde operatoren tegen. De laatste 2 operatoren hebben de bijzondere eigenschap dat, met ieder van hen alleen, de andere operatoren te representeren zijn. Bijvoorbeeld:

not a door a nor a
a and b door (a nor a) nor (b nor b)
a or b door (a nor b) nor (a nor b)

De nor and nand operatoren worden dan ook gebruikt in elektronische schakelingen (o.a. in een rekenautomaat).

Let goed op het verschil tussen de ("niet - uitsluitende") or en de ("wel - uitsluitende ") eor. Vergelijk hiermee het gebruik van het woord "of" in de volgende zinnen:
"Een paspoort of een rijbewijs is nodig om zich te legitimeren".
"De eerste kerstdag valt op dinsdag of woensdag".

operatie	notatie		a: f f t t b: f t f t
	wiskundig	"omgangstaal"	
differentie	a \ b	a wel, b niet	f f t f
antivalentie	a eor b	òf a, òf b	f t t f
Peircefunctie	a nor b	noch a, noch b	t f f f
Schefferfunctie	a nand b	a niet of b niet	t t t f

5.2.4. Eenvoudige en conditionele expressies

Hetgeen wij hier een (arithmetische of boolean) expressie genoemd hebben, heet in de officiële Algol 60 beschrijving een "eenvoudige" ("simple") expressie. Dit komt omdat in Algol 60 het begrip expressie, in afwijking van het normale wiskundige gebruik, ook wordt gebruikt voor hetgeen wij een "conditionele" expressie zullen noemen. Deze heeft de gedaante:

if BE then E1 else E2

waarin BE een (eventueel weer conditionele) boolean expressie voorstelt, E1 een (eenvoudige!) boolean of arithmetische expressie en E2 een (eenvoudige of conditionele) boolean of arithmetische expressie. E1 en E2 moeten bovendien beide òf arithmetisch òf boolean zijn, terwijl vanzelfsprekend het else gedeelte nooit mag ontbreken omdat de expressie dan ongedefinieerd zou zijn. De waarde van een conditionele expressie hangt af van een voorwaarde (BE).

Aangeraden wordt om de conditionele expressie alleen te benutten voor eenvoudige gevallen van het type:

if SBE then SE1 else SE2

waarin de letter S een afkorting is voor "simple" en waarin de SE's betrekkelijk korte (arithmetische of boolean) expressies zijn. (Van een conditionele expressie kunnen we echter door deze tussen ronde haakjes te plaatsen een eenvoudige expressie maken!)

Gebruik in een assignment als

```
x := if x > 0 then y else z
```

is dan direct te doorgronden. Hetzelfde wordt echter bereikt met

```
if x > 0 then x := y else x := z
```

De conditionele expressie lijkt overbodig, doch wordt toch nog al eens gebruikt als bijvoorbeeld actuele parameter in een procedure aanroep (omdat (conditionele) statements niet als actuele parameters toegestaan zijn) of als subscript van een array variabele.

5.2.5. Programma commmentaar en opmaak

Ter verhoging van de leesbaarheid van een programma wil men graag in de programmatekst commentaar schrijven. Dit kan in Algol als volgt:

```
; comment "toelichtende tekst";is equivalent met  ;
begin comment "toelichtende tekst"; is equivalent met begin
end "toelichtende tekst";       is equivalent met end;
end "toelichtende tekst" end    is equivalent met end end
end "toelichtende tekst" else   is equivalent met end else
```

Het toegevoegde commentaar heeft geen effect bij de uitvoering van een programma.

Minstens zo belangrijk als het toevoegen van commentaar is het duidelijk op papier neerschrijven van een programma. Bij korte programmateksten op de plaats van de puntjes zal men bijvoorbeeld schrijven

```
if ..... then begin ..... end else begin ..... end;
if ..... then begin ..... end;
while .... do   begin ......... end;
```

doch bij lange teksten zal men om niet te snel aan de rechter marge van het papier te komen, in deze gevallen schrijven

```
if .....................
      then begin .....
                 .....
           end
      else begin .....
                 .....
           end;
if ..................... then
      begin ..........
            ..........
      end;
while ..................... do
      begin ..........
            ..........
      end;
```

Bij elkaar horende begin-end en then-else symbolen schrijft men dan zoveel mogelijk onder elkaar.

5.3. Assignment en sequentiërings statements

5.3.1. Enkelvoudige en meervoudige asssignments; dummy statements

Ten aanzien van de ("enkelvoudige") assignment statement

variabele: = expressie (of korter V: = EX)

is direct op te merken dat linker- en rechterlid beide óf van het type boolean óf van het type arithmetisch moeten zijn. Bij arithmetische typen is het toegestaan dat V en EX niet beiden real of integer zijn. Als EX van het type integer is en V van het type real, dan wordt de waarde van EX als getal van het type real aan V toegekend. Is daarentegen V van het type integer en EX van het type real, dan wordt de waarde van EX afgerond op het dichtstbijzijnde gehele getal (uit Z^*), dat dan als integer aan V wordt toegekend. Is de waarde van EX echter binnen de precisie van de arithmetiek gelijk aan n + ½, dan kan de waarde van V ongedefinieerd zijn (is die nl. n of n + 1?) Deze situatie moet dan ook bij het programmeren vermeden worden!

Wanneer aan een aantal variabelen van hetzelfde type(!) eenzelfde waarde moet worden toegekend, dan kan dit met behulp van de zogenoemde multiple assignments. Voorbeeld:

```
x : = y : = z : = expressie
```

Aangezien onder de variabelen ook subscripted variabelen kunnen voorkomen, geldt de volgende afspraak t.a.v. de betekenis van deze constructie

- bepaal van links naar rechts gaande de identiteit van de variabelen,
- bepaal de waarde van de expressie,
- ken deze waarde toe aan de gevonden variabelen.

Ga aan de hand hiervan zelf na dat

```
i : = 3; i := a[i]=:i + 1;
```

ten gevolge heeft dat a[3] = i = 4 wordt

De dummy statement is een lege rij Algol-symbolen. Zoals bekend is de punt-komma een scheidingsteken, o.a. tussen statements. Hieruit volgt dat tussen twee opeenvolgende punt-komma's of tussen een punt-komma en een daarop volgend end-symbool een dummy statement staat. Dummy statements hebben geen effect bij de verwerking van een programma en zijn dus vrij zinloos.

5.3.2. Herhaalde uitvoering van statements

De while-do statement en de hierna te behandelen do-until statement behoren officieel niet tot Algol 60. Zij vormen echter een dusdanig essentieel gereedschap voor de programmeur dat zij een modificatie op Algol 60 rechtvaardigen. Zij worden daarom eerst behandeld en zijn in veel Algol-systemen gerealiseerd. Omdat deze twee statements niet officieel zijn, treft men in de literatuur ook wel alternatieve schrijfwijzen voor deze constructies aan, nl. do-while, respectievelijk repeat-until. Voor het begrip van deze constructies is dit natuurlijk niet essentieel.
Aan het eind van deze paragraaf worden tenslotte slechts die aspecten van de wel officiële Algol 60 for statement behandeld die nu nog zinvol lijken.

Zoals we gezien hebben is de while statement uitermate geschikt voor cycli waarbij tijdens het schrijven niet vaststaat of en hoeveel herhalingen zullen moeten plaatsvinden. In die gevallen waarbij door de aard van het probleem het in ieder geval zeker is dat een herhaling tenminste 1 keer moet worden uitgevoerd, is een do-until statement van het type

```
do S until B
```

vaak eenvoudig in het gebruik. De interpretatie ("semantiek") hiervan is dat de (als regel compound) statement S moet worden uitgevoerd zolang de boolean expressie B false is (dit betekent natuurlijk dat S iets doet dat B een keer true maakt). Een voorbeeld voor het gebruik van dit statement is de nulpuntsbenadering van een functie volgens Newton:

```
begin real x0, x1; x1 := start;
      do begin x0 := x1; x1 := x0 - f(x0)/afgeleide f(x0) end
              until abs (x1 - x0) < eps;
      write (x1)
end
```

Bij de voorbeelden bij de while statement zal opgevallen zijn dat deze vaak het volgende karakter hebben

```
i := 1; while i ≤ n do begin S; i := i + 2 end
```

We kunnen dit vervangen door een enkel statement

```
for i := 1 step 2 until n do begin S end
```

In het algemeen geldt dat de for statement

```
for i := p step q until r do S
```

waarin i een (integer) variabele is en p,q en r arithmetische expressies (van het type integer) zijn, hetzelfde effect heeft als

```
i := p; while (r - i) * q ≥ 0 do begin S; i := i + q end
```

Na het verlaten van de lus is volgens het officiële Algol 60 de waarde van de variabele i ongedefinieerd!!

Een andere officiële vorm van een for statement is die waarbij voor een aantal niet equidistante waarden een berekening moet worden uitgevoerd; bijv.

```
for i : = ex1, ex2, ...., exn do S
```

waarin i, ex1, ... exn variabelen, resp. expressies van het type integer zijn.

Tot slot als groter voorbeeld voor het gebruik van een for statement de berekening van het product C van een m × n matrix A en een n × p matrix B met

$$c_{ij} = \sum_{k=1}^{n} a_{ik}\, b_{kj}$$

```
for i : = 1 step 1 until m do
    for j : = 1 step 1 until p do
        begin c[i,j] : = 0;
              for k : = 1 step 1 until n do
                  c[i,j] : = c[i,j] + a[i,k] * b[k,j]
        end
```

5.3.3. Selectieve uitvoering van meer statements

Met de alternatieve statement

```
if B then S1 else S2
```

hebben we een middel in handen gekregen om afhankelijk van de waarde van de boolean expressie B hetzij (de als regel compound statement) S1 of S2 uit te voeren.
In programmeringsproblemen komen we echter ook de situatie tegen van het type

"als de waarde van een zekere integer variabele (of expressie) A
1 is, voer dan statement S1 uit
2 is, voer dan statement S2 uit
3 is, voer dan statement S3 uit
een andere dan de genoemde is, voer dan statement S0 uit".

Met een keten van alternatieve statements is dit natuurlijk uit te schrijven

```
if A = 1 then S1
         else if A = 2 then S2
                       else if A = 3 then S3
                                     else S0;
```

Vooral als het aantal alternatieven groot is, krijgen we problemen met het duidelijk opschrijven van een programma. Om die reden is naast de alternatieve statement (ook wel de " logische if" genoemd) de zgn. "arithmetische if" voorgesteld, die uitgeschreven voor het bovenstaande voorbeeld de volgende gedaante heeft:

```
if A then S1; S2; S3 else S0;
```

waarin A dus i.h.a. een (integer) arithmetische expressie is, S0 t/m S3 willekeurige (maar niet-conditionele) enkelvoudige of compound statements zijn en waarvan de interpretatie hierboven met de logical if statements is weergegeven. In overeenstemming daarmee kan dus slechts één van de statements S0 t/m S3 worden uitgevoerd. Het spreekt vanzelf dat andere representaties van de arithmetische if ook mogelijk zijn; zo komt men in plaats van if - then ook wel case - of tegen, waarbij else dan wordt weggelaten omdat S0 altijd een dummy statement is.

5.3.4. Andere sequentiërings-statements

Zonder hier verder in detail op in te gaan, vermelden we dat in Algol 60 ook een goto-statement is opgenomen, die ontleend is aan machinetalen (en die ook in de meeste andere programmeertalen voorkomt). Omdat het gebruik van dit statement aanleiding geeft tot zgn."spaghetti" programma's, waarvan de correctheid vrijwel niet is aan te tonen, raakt dit type statement snel in de vergetelheid.

5.4. Procedures

5.4.1. Procedure declaratie

In de declaratie van een procedure worden achtereenvolgens geschreven:

(1) procedure

bij routineprocedures of

integer procedure, real procedure of boolean procedure

bij functieprocedures.

(2) De door de programmeur zelf te kiezen naam voor de procedure.

(3) Tussen haakjes en onderling gescheiden door komma's de parameters van de procedure.

(4) Het zogenaamde value-part; hiermede wordt opgegeven welke formele parameters als invoerwaarden gebruikt worden. Uitvoerparameters mogen nóóit in het valuepart staan!

(5) De specificatie van de parameters; hiermede wordt de soort van de formele parameters opgegeven en wel met behulp van zogenaamde specifiers. Als zodanig kunnen optreden: real, integer, boolean, real array, integer array, boolean array, procedure, real procedure, integer procedure en boolean procedure.
Alle formele parameters dienen gespecificeerd te worden. Op te merken is ook nog, dat in de specificatie alleen de naam van de formele parameter wordt genoemd en niet bijkomende informatie (als arraygrenzen bij een array, of parameter en body bij een procedure).

(6) De body van de procedure. De body is een statement; deze statement zal vaak, zoals in twee van de drie voorbeelden uit de vorige paragraaf, de vorm van een blok hebben.

Na de parameterlijst (3), na de valuelijst (4) en na de specificaties wordt een puntkomma geschreven; bovendien worden de specificaties onderling gescheiden door een puntkomma (real a; integer m).

De variabelen die in de body gedeclareerd zijn, noemen we locale variabelen. In de body mogen naast formele parameters en locale variabelen ook variabelen uit de omgeving, waarin de procedure is gedeclareerd, gebruikt worden. Dit worden globale variabelen genoemd. Het gebruik van globale variabelen moet als regel afgeraden worden, omdat door de globale variabelen de procedure niet los is te maken van zijn omgeving (wat wel gaat via de parameters).

De waarde berekend aan een function designator, is de waarde die (dynamisch gezien) als laatste wordt toegekend aan de procedurenaam. De procedurenaam mag dus meermalen in de body voorkomen in het linkerlid van een assignment statement. We hadden de procedure max uit de vorige paragraaf dan ook kunnen declareren als:

```
real procedure max2(x,y); value x,y; real x,y;
if x > y then max2 := x else max2 := y
```

In de procedure som mogen we echter niet schrijven som := som + a[i], omdat som dan in een expressie zou voorkomen, hetgeen een nieuwe aanroep van de procedure som zou betekenen.

We hebben in vorige hoofdstukken een strikte scheiding aangehouden tussen invoer- en uitvoerparameters. Het komt echter ook voor dat parameters zowel een invoer- als een uitvoerfunctie vervullen:

```
procedure wissel(a,b); real a,b;
begin real hulp;
      hulp := a; a := b; b := hulp
end
```

In de valuelijst worden alle enkelvoudige variabelen opgenomen, die louter als invoerparameters optreden. Bij de aanroep van de procedure (zie 5.4.2) moet de overeenkomende actuele parameter een waarde hebben. Deze waarde wordt bij de aanroep bepaald en aan de formele parameter toegekend (zie 5.4.4). Vandaar ook dat array parameters niet in de valuelijst worden opgenomen, omdat volgens de regels het actuele array dan in het formele array gecopieerd zou moeten worden, hetgeen veel geheugenruimte kost.

5.4.2. Procedure-aanroep

Een procedure kan alleen aangeroepen worden in het blok (of een binnenblok hiervan), waarin de procedure gedeclareerd is.
Een routineprocedure wordt aangeroepen door middel van een procedurestatement. Deze bestaat uit de naam van de procedure met daarachter de actuele parameters (tussen haakjes en onderling gescheiden door komma's).
Een functieprocedure wordt aangeroepen door middel van een function designator, die in een expressie voorkomt. De function designator bestaat uit een procedurenaam gevolgd door de tussen haakjes staande lijst van actuele parameters.

Bij aanroep moeten alle actuele parameters gedeclareerd zijn en moeten alle invoerparameters (in de valuelijst) een waarde bezitten. Aantal, type en volgorde van de actuele parameters moeten in overeenstemming zijn met de formele parameters en hun specificaties.
Bij de verwerking van een routine-aanroep door middel van een procedurestatement (bij routineprocedures) en een function designator (bij functieprocedures) wordt als het ware op de plaats van de aanroep in het programma de body van de procedure uitgevoerd. Daarbij moet een correspondentie gelegd worden tussen formele en actuele parameters. Hoe dit gebeurt, bekijken we in de volgende paragraaf.

5.4.3. Het parametermechanisme

Voordat de body van de procedure als het ware op de plaats van aanroep wordt uitgevoerd, worden achtereenvolgens de volgende handelingen uitgevoerd:

- de body van de procedure wordt ingebed in een (fictief) blok, waarin de formele parameters uit de valuelijst worden gedeclareerd (volgens de overeenkomstige specificatie) en waarin aan deze parameters de waarden van de overeenkomstige actuele parameters wordt toegekend;
- de formele parameters, die niet in de valuelijst voorkomen, worden in de body vervangen door de overeenkomstige actuele parameters.

Hierna wordt het fictieve blok uitgevoerd in de omgeving (in het blok) van de aanroep. Dit laatste is van belang als er globale variabelen

in de body voorkomen.
De parametervervanging legt aan de actuele parameter de eis op, dat de door bovenstaande handelingen ontstane Algol-tekst correct is (hetgeen bijvoorbeeld niet het geval zou zijn als een formele parameter die niet in de valuelijst voorkomt en in de body in het linkerlid van een assignment statement staat, actueel overeen zou komen met een getal).

Als voorbeeld zullen we voor de procedure som uit 4.3.1 het fictieve blok geven dat ontstaat bij de aanroep som (b,p,q), waarbij we veronderstellen dat b, p en q gedeclareerd zijn. Dit fictieve blok luidt:

```
begin integer m,n;
      m := p; n := q;
      begin integer s,i;
            s := 0;
            for i := m step 1 until n do s := s + b[i];
            som := s
      end
end
```

Om het verschil tussen parameters in de valuelijst en parameters die niet in de valuelijst voorkomen nog eens te illustreren bekijken we de procedures:

```
procedure P(n); integer n ;
begin n := n * n; n := n + n end
```

en

```
procedure Q(n); value n; integer n;
begin n := n * n; n := n + n end
```

en hun aanroepen

```
i := 5; P(i);
j := 5; Q(j);
```

Na afloop van dit stukje programma heeft i de waarde 50 en j de waarde 5. We spreken bij de twee manieren van parameteroverdracht van respectievelijk call-by-name en call-by-value.

5.4.4. Jensen's device

Stel dat de volgende procedure gedeclareerd is:

```
real procedure sigma(var,first,last,term);
                value first, last; real term; integer var, first, last;
begin real h;
      h := 0;
      for var := first step 1 until last do h := h + term;
      sigma := h
end
```

Op het eerste gezicht lijkt het alsof deze procedure (last - first + 1) * term berekent (tenminste het actuele equivalent hiervan). Als we echter precies de regels van de vorige paragraaf toepassen op de aanroep

```
sigma(n,1,5n ↑ 2)
```

dan zien we dat door deze aanroep wordt berekend $\sum_{n=1}^{5} n^2$ en door de aanroep

```
sigma(k,10,30,a[k])
```

de waarde wordt berekend van $\sum_{k=10}^{30} a[k]$.

We bereiken deze resultaten doordat twee actuele parameters (waarvan de overeenkomstige formele parameters niet in de valuelijst voorkomen) van elkaar afhankelijk zijn.

5.4.5. Recursiviteit

In 5.4.1 is voor de procedure som gesteld dat de procedurenaam van een functieprocedure niet in het rechterlid van een assignment statement mag staan, omdat dit een nieuwe aanroep van de procedure inhoudt. Dit laatste was in die procedure niet de bedoeling, maar in het algemeen is het wel toegestaan dat in een procedure een aanroep van de procedure zelf voorkomt. Men spreekt in zo'n geval van een recursieve procedure.
Wanneer is de recursiviteit zinvol te gebruiken? Men kan natuurlijk elk probleem dat recursief gedefinieerd is met behulp van een recursieve procedure oplossen.
Als we als definitie van n! nemen dat n! gelijk is aan n * (n - 1)! voor $n \geq 1$ en gelijk is aan 1 voor $n \leq 0$ dan kunnen we voor n! als procedure schrijven:

```
integer procedure nfac(n); value n; integer n;
        if n ≤ 0 then nfac := 1 else nfac := n * nfac(n - 1);
```

We kunnen de probleemstelling echter gemakkelijk niet-recursief geven door te definiëren dat n! gelijk is aan 1 * 2 * 3 * ..... * (n - 1) * n voor n ≥ 1 en gelijk is aan 1 voor n ≤ 0. Zoals de probleemstelling nu is, kunnen we n! veel gemakkelijker berekenen met behulp van:

```
nfac := 1;
for k := 1 step 1 until n do nfac := nfac * k
```

Het volgende voorbeeld laat echter zien dat er problemen zijn waarbij de omzetting van een repetitieve oplossingsmethode naar een reptitieve oplossingsmethode niet (zo eenvoudig) gaat.
Stel dat we een rij positieve getallen, afgesloten door -1, moeten inlezen en in omgekeerde volgorde moeten afdrukken zonder van een array gebruik te maken. Dit probleem kunnen we oplossen met behulp van het volgende programma:

```
begin procedure printrij;
      begin integer a; read(a);
            if a ≠ - 1 then begin printrij; write(a) end
      end;
      printrij;
end
```

6. Programmeren

6.1. Inleiding

In het voorgaande zijn bouwstenen voor het programmeren besproken. Aan de hand van een aantal voorbeelden zullen we nu zien hoe deze bouwstenen gebruikt kunnen worden bij het ontwerpen van grotere programma's voor verschillende problemen. Numeriek wiskundige problemen laten we daarbij buiten beschouwing omdat die in de betreffende colleges aan de orde worden gesteld.
Zoals altijd bij het ontwerpen kunnen we niet een gedetailleerd voorschrift geven voor het ontwerpproces. Wel zijn enkele vuistregels te geven, zoals onder andere:

- zorg ervoor dat het probleem precies gespecificeerd en geformuleerd is (dit is niet eens zo triviaal).
- probeer een idee te krijgen van de moeilijkheden die zich bij het oplossen van het probleem kunnen voordoen en tracht dan een logische splitsing in (kleinere) deelproblemen te definiëren. Voor de keuze van de deelproblemen kunnen we als criterium nemen dat de interactie tussen de deelproblemen zo "gering mogelijk" moet zijn (een zekere interactie blijft natuurlijk altijd bestaan).

6.2. Worteltrekken

Gevraagd wordt een programma te maken dat voor een gegeven niet-negatief geheel getal a de naar beneden afgeronde vierkantswortel van a bepaalt, i.e. entier $(\sqrt{a})$. Daarbij is de voorwaarde dat geen beroep wordt gedaan op real-arithmetiek.

Uit de vraagstelling blijkt dat wij een programma moeten maken dat een (integer-)variabele b een zodanige waarde geeft, dat de relatie

$$R: \quad b^2 \leq a < (b+1)^2 \quad \wedge \quad b \geq 0$$

geldt na afloop van de uitvoering van het programma.
Aangezien a willekeurig is, en wij geen in Algol formuleerbare expressie kennen die de waarde van de gevraagde b heeft, mogen we vermoeden dat b gestadig berekend zal moeten worden d.m.v. een repetitief proces. De

vraag is dan welke invariante relatie (cf. pag. 29, e.a.) dit rekenproces zou kunnen beheersen. De invariant moet in ieder geval een generalisatie van de eindtoestand R zijn.
Een makkelijke manier waarop een relatie kan worden gegeneraliseerd (afgezwakt) is door het achterwege laten van een restrictie. Laten we eens onderzoeken wat we hier kunnen bereiken bij het beschouwen van de invariant

$$P: \quad b^2 \leq a \quad \wedge \quad b \geq 0 \ .$$

In elk geval is deze relatie initieel eenvoudig waar te maken, nl. door b:= 0, want dan staat er: $a \geq 0$, en dat is waar op grond van het gegeven dat $a \geq 0$ is. Het eindresultaat R volgt uit P indien bovendien nog zou gelden $a < (b + 1)^2$.
Op grond van deze observeringen lijdt het dan geen twijfel dat een programmaatje van de vorm:

```
b:= 0 ;
while (b + 1)^2 ≤ a do "verhoog b onder invariantie van P"
```

de gewenste waarde b zal bepalen.
Omdat na afloop van "verhoog b onder invariantie van P" nog steeds moet gelden dat $b^2 \leq a$ waar is, mag de verhoging van b niet te groot zijn. Omdat de verhoging van b tenminste 1 moet bedragen, moet vlak voor "verhoog b onder invariantie van P" zeker voldaan zijn aan $(b + 1)^2 \leq a$, anders wordt P geschonden. De conditie waaronder in het programmaatje de uitvoering van "verhoog b onder invariantie van P" ter hand wordt genomen is echter precies $(b + 1)^2 \leq a$, en dat betekent dat de verhoging van b maximaal 1 mag bedragen, willen we zeker zijn dat na afloop P nog geldt.

Hiermee komen we tot het programma:

```
b:= 0 ;
while (b + 1)*(b + 1) ≤ a  do   b:= b + 1
```

Opmerking:

Als we de vermenigvuldiging er nog uit zouden willen werken, dan kunnen we een additionele variabele p introduceren waarvoor steeds zal gelden:

$p = (b + 1)^2$. Daarmee transformeert het programma in

```
b:= 0 ; p:= 1 ;
while p ≤ a do begin b:= b + 1 ; p:= p + b + b + 1 end .
```

(Einde opmerking).

Het ontworpen programma is hopeloos inefficiënt: voor bv. $a = 10^6$ vergt het liefst 10^3 repetities. Deze inefficiëntie is het gevolg van de keuze van de invariant P, die zoals we gezien hebben geen drastischer ophogingen van b toeliet dan een ophoging met 1. Wij gaan daarom nu omzien naar een andere invariant.

Een tweede manier waarop een relatie kan worden gegeneraliseerd is door vervanging van een constante door een variabele. Laten we daarom de consequenties onderzoeken indien we voor de invariant kiezen

$$P: \quad b^2 \leq a \quad \wedge \quad a < c^2 \quad \wedge \quad c > b \geq 0$$

Deze is initieel makkelijk waar te maken, nl. door b:= 0 ; c:= a + 1 , aangezien voor elke $a \geq 0$ geldt dat $(a + 1)^2 > a$. Verder volgt R op triviale wijze uit P als bovendien geldt: $b + 1 = c$.
Hiermee komen we dan tot een programmaatje van de vorm

```
b:= 0 ; c:= a + 1 ;
while b + 1 ≠ c do "verklein (c - b) onder invariantie van P" .
```

dat gewisselijk aan b de gevraagde waarde zal toekennen. De verkleining van c - b kan geschieden door een verlaging van c en/of een verhoging van b. De analyse van hoe groot de vermindering van c - b mag zijn is iets gecompliceerder, dan de analyse hoe groot de verhoging van b mocht zijn in het eerste programma van deze paragraaf. Beschouw eens een verhoging van b met d. Wil dit een effectieve verhoging zijn dan moet in elk geval $d > 0$ zijn. Wil de statement b:= b + d de invariant niet verstoren dan moet eveneens voldaan zijn aan

$$(b + d)^2 \leq a \quad \wedge \quad d < c - b \ .$$

Op analoge wijze kunnen we inzien dat een verlaging van c met d alleen is toegestaan als

$$a < (c - d)^2 \quad \wedge \quad d < c - b \ .$$

Indien nu voor d een zodanige keuze wordt gemaakt dat $c - d \geq b + d$, ofwel $2d \leq c - b$, dan is steeds aan tenminste één van de condities $(b + d)^2 \leq a$ of $a < (c - d)^2$ voldaan. (en uiteraard aan $d < c - b$) .
Nu kunnen we "verklein (c - b) onder invariantie van P" zonder aarzelen uiteenrafelen in:

```
"kies d zodanig dat 0 < d en 2d ≤ c - b" ;
if (b + d)^2 ≤ 0 then b:= b + d
                 else c:= c - d
```

Het is nu evident dat hoe groter de keuze van d, hoe sneller het programma. Uiteraard is het toegestaan om d = 1 te kiezen, want uit het feit dat $b + 1 \neq c$ is volgt samen met de invariant dat $c - b \geq 2$. Maar bij de keuze d = 1 zouden we een programma krijgen, dat net zo langzaam is als het allereerste uit deze paragraaf.

De maximale d die is toegestaan, rekening houdend met de beperking,wordt hier verkregen door d:= (c - b) div 2. Daarmee wordt het totale programma:

```
b:= 0 ; c:= a + 1 ;
while b + 1 ≠ c do
begin d:= (c - b) div 2 ;
      if (b + d) * (b + d) ≤ a then b:= b + d ;
                               else c:= c - d
end .
```

Oefening 1: Tracht ook uit dit programma de vermenigvuldiging te elimineren.
(Einde oefening 1.)

Oefening 2: Het is niet uitgesloten dat de condities $(b + d)^2 \leq a$ en $a < (c - d)^2$ elkaar niet uitsluiten, en in dat geval zou zowel een ophoging van b als een aflaging van c te overwegen zijn. Ga na onder welke omstandigheden dat het geval is.
(Einde oefening 2.)

Opmerking: Vorenstaand programma vergt voor bv. a = 10^6 circa 20 repetities, en dat is een aanzienlijke winst vergeleken met de 10^3 repetities uit het eerste programma. Het maken van programma's is uiteraard een activiteit van correct ontwerpen, maar niet alleen dat :oog voor redelijk efficiënt gedrag is ook gewenst, en dat maakt de activiteit vaak erg gecompliceerd. (Einde opmerking.)

Tenslotte zij nog vermeld, dat de redenering die aanleiding geeft tot het voorgaande programma afkomstig is van E.W. Dijkstra.

6.3. De eerste duizend W-getallen

(Voorbeeld ontleend aan: N. Wirth, Systematic Programming, pag. 154, opgave 15.12)

Laat de verzameling der W-getallen gedefinieerd zijn door de volgende drie axioma's:

a) 1 behoort tot de verzameling
b) als x tot de verzameling behoort dan ook 2x + 1 en 3x + 1
c) alleen getallen die op grond van de axioma's a) en b) tot de verzameling behoren, behoren tot de verzameling.

Gevraagd wordt een programma te maken dat de duizend kleinste W-getallen print. (Deze vraagstelling is zinvol aangezien de verzameling W-getallen niet eindig is.)

Aangezien wij een programma moeten maken voor een sequentiële machine, die slechts op scalaire grootheden kan opereren, is het ondenkbaar uit de voeten te komen met een niet-repetitief rekenproces. En dat betekent dat wij met een cykeltje te maken krijgen en wij ons derhalve de vraag dienen te stellen hoe wij de tussentijdse toestanden zullen karakteriseren, i.e. hoe de invariante relatie bij deze repetitie er uitziet. Omdat er na afloop van het programma duizend getallen gegenereerd moeten zijn, en omdat er om mee te beginnen nog (bijna) geen enkel gevraagd getal bekend is, gokken wij dat typerend voor de tussentijdse toestand zal zijn dat een aantal - zeg k - der gevraagde getallen reeds berekend is. Het blijkt om

administratieve redenen prettig te zijn dat we ons nog een extra beperking opleggen, en die is dat de k berekende getallen niet zomaar k van de gevraagde W-getallen zijn, maar de k kleinsten: $w_0, w_1, \ldots, w_{k-1}$. $(w_i < w_{i+1})$

Het ligt nu op onze weg om te analyseren hoe het $k+1^{ste}$ W-getal w_k gekarakteriseerd is. Voor $k = 0$ is het eenvoudig: $w_0 = 1$ (axioma a)).
Voor $k > 0$ is $w_k = 2w_i + 1$ of $w_k = 3w_j + 1$ (axioma b)).
Aangezien zowel de functie $2x + 1$ als de functie $3x + 1$ monotoon stijgend in x is, wordt w_k voortgebracht door een w-getal dat kleiner dan w_k is, en --- gezien de veronderstelling dat we de k kleinste W-getallen berekend hebben --- dus bekend is.
Het is nu niet meer moeilijk om in te zien dat, wanneer we i definiëren als de kleinste index waarvoor geldt: $0 \leq i < k \wedge 2w_i + 1 > w_{k-1}$ en wanneer we j definiëren als de kleinste index waarvoor geldt:
$0 \leq j < k \wedge 3w_j + 1 > w_{k-1}$, dat dan

$$w_k = \min(2w_i + 1,\ 3w_j + 1) .$$

Na dit voorbereidend werk is het al mogelijk summier een programmastructuur te gaan geven. Laten we de volgende afspraken maken:
Er is een integer array w[0:999], en er zijn drie integers i, j en k zodanig dat:

P: $0 \leq k \leq 1000$,
w[0], ..., w [k - 1] zijn in monotoon stijgende volgorde de k kleinste W-getallen,
i is de kleinste index waarvoor geldt:
$0 \leq i < k \wedge 2w[i] + 1 > w[k - 1]$,
j is de kleinste index waarvoor geldt:
$0 \leq j < k \wedge 3w[j] + 1 > w[k - 1]$.

Dan kan het programma luiden

```
"kies makkelijke waarden van w, i, j en k zodat P geldt" ;
while k ≠ 1000 do
begin "bereken in bv. m het k + 1ste W-getal";
      "breid w uit met m en verhoog k met 1";
      "pas i en j aan zodat P weer geldt"
end;
"print w" .
```

De makkelijkste beginwaarden voor w, i, j en k worden verkregen voor k = 1. (Ga na waarom k = 0 niet mogelijk is.) We krijgen dan:

```
k:= 1 ; w[0]:= 1 ; i:= 1 ; j:= 1.
```

Het berekenen van m wordt op grond van de karakterisering van het k + 1ste W-getal:

```
m:= 2 * w[i] + 1 ; if m > 3*w[j] + 1 then m:= 3*w[j] + 1 .
```

De uitbreiding van w met m is bijna triviaal:

```
w[k] := m ; k:= k + 1 .
```

De aanpassing van i en j geschiedt door gebruik te maken van de definities van i en j. Daaruit volgt namelijk dat, gezien de monotonie van de rij w[0], ..., w[k - 1] , de waarden van i en j door uitbreiding van w nooit te groot kunnen zijn geworden. En dat betekent dat we alleen maar ophogingen van i resp. j hoeven te beschouwen. De aanpassing van i en j wordt daarmee:

```
while 2*w[i] + 1 ≤ m do i:= i + 1 ;
while 3*w[j] + 1 ≤ m do j:= j + 1
```

Als we nu nog twee integers x en y invoeren waarvoor invariant zal gelden: $x = 2w[1] + 1 \wedge y = 3w[j] + 1$, dan ziet het totale programma er als volgt uit:

```
begin integer k,i,x,j,y; integer array w[0:999];
      k := 1; w[0]:= 1; i := 1; j := 1;  x := 3; y := 4;
      while k ≠ 1000 do
            begin integer m;
                  m := x; if m > y then m := y;
                  w[k] := m; k := k + 1;
                  while x ≤ m do begin i := i + 1; x := 2*w[i]  + 1 end;
                  while y ≤ m do begin j := j + 1; y := 3*w[j]  + 1 end
            end;
      k := 0;
      while k ≠ 1000 do begin print (w[k]); k := k + 1 end
end
```

Oefening 1: Ga na waar we gebruik gemaakt hebben van het feit dat de rij w[0], ..., w[k - 1] monotoon stijgend is.
(Einde oefening 1.)

Oefening 2: De aanpassing van i (en die van j) hebben we gerealiseerd met een cykeltje. Dit brengt heel duidelijk tot uiting dat na afloop voldaan is aan x > m, ofwel 2w[i] + 1 > w[k - 1]. Het is hier echter zo dat het cykeltje maximaal een keer doorlopen hoeft te worden om de juiste waarde van i te verkrijgen. Toon dit aan.
(Einde oefening 2.)

Oefening 3: Had de uitbreiding van w en de aanpassing van i en j ook in omgekeerde volgorde gekund?
(Einde oefening 3.)

Oefening 4: Tracht een programma te maken dat voor een gegeven natuurlijk getal n nagaat of het tot de collectie der W-getallen behoort.
(Einde oefening 4.)

6.4. Mediaan bepalen

Gevraagd wordt van een rij gehele getallen a_0, ..., a_{n-1} de mediaan te bepalen. De mediaan is ruwweg dat getal x uit de rij waarvoor het aantal getallen uit de rij die niet groter dan x zijn ongeveer gelijk is aan het aantal getallen uit de rij die niet kleiner dan x zijn. Deze definitie is niet erg precies en daarom onhandzaam. Beter is:

zij m = n div 2 , dan is de mediaan x het getal a'_m waarbij $a'_0, \ldots, a'_{n-1}$ een monotoon niet afnemende opsomming van de getallen $a_0, \ldots, a_{n-1}$ voorstelt.

Deze definitie van de mediaan x suggereert al onmiddellijk een oplossing, namelijk: sorteer de rij $a_0, \ldots, a_{n-1}$ en kies dan voor x het m^{de} element. Logisch gezien is er geen enkel bezwaar tegen dit voorstel, maar economisch gezien wel. Immers als wij de rij zouden sorteren zouden wij niet alleen het m^{de} element bepaald hebben voor m = n div 2 maar voor alle waarden $0 \leq m < n$, en dat is duidelijk veel meer dan de bedoeling is.

Een andere karakterisering van de mediaan zou uitkomst kunnen bieden:

zij m = n div 2 , en zij $a'_0, \ldots, a'_{n-1}$ een zodanige rangschikking van de getallen $a_0, \ldots, a_{n-1}$ dat: $a'_i \leq a'_m$ voor $0 \leq i < m$ en $a'_i \geq a'_m$ voor $m < i \leq n - 1$, dan is a'_m de mediaan x .

Uit deze formulering blijkt dat het niet nodig is dat de getallen aan weerszijden van a'_m gesorteerd zijn. Het getal $x = a'_m$ fungeert in de rij $a'_0, \ldots, a'_{n-1}$ als een soort spil: links ervan is geen enkel getal groter, rechts ervan geen enkel getal kleiner.
Het is teveel om te hopen dat wij in een klap de gevraagde waarde van x kunnen berekenen, en wij zullen onze toevlucht dan ook moeten zoeken in een repetitief proces. De vraag is dan wederom welke invariante betrekking we zullen kiezen. Het onderhavige probleem is niet zo eenvoudig dat we meteen een goede gok kunnen wagen, en daarom zullen we eerst iets meer inzicht trachten te verwerven.
Een effectieve manier om een probleem klein te krijgen is door het in omvang te verkleinen. Als we de klasse van getallen waaruit x moet komen stapsgewijs weten te verkleinen, dan komt er een ogenblik waarop de klasse zó klein is, bv. omvang 1 heeft, dat het antwoord triviaal is.
Laat s eens een willekeurig getal uit de rij $a_0, \ldots, a_{n-1}$ zijn. Beschouw vervolgens een rangschikking $a'_0, \ldots, a'_{n-1}$ van de oorspronkelijke getallen die zodanig is dat:

de getallen $a'_0, \ldots, a'_{i-1}$ alle kleiner dan s,
de getallen $a'_i, \ldots, a'_{j-1}$ alle gelijk aan s, en
de getallen $a'_j, \ldots, a'_{n-1}$ alle groter dan s zijn,
dan kunnen zich drie gevallen voordoen:

- $m < i$, d.w.z. de spil s is te groot en x is het m^{de} getal uit de rij $a'_0, \ldots, a'_{i-1}$ wanneer die gesorteerd zou zijn
- $i \leq m < j$, d.w.z. de spil s is precies de gevraagde x
- $j \leq m$, d.w.z. de spil s is te klein en x is het $(m-j)^{de}$ getal uit de rij $a'_j, \ldots, a'_{n-1}$ wanneer die gesorteerd zou zijn.

Hieruit zien we dat we in de algemene situatie x kunnen typeren als het k^{de} element uit een rij $a_p, \ldots, a_{q-1}$ (en hiermee bedoelen in de rest van deze paragraaf steeds het element a'_{p+k} uit de monotoon niet afnemende opsomming $a'_p, \ldots, a'_{q-1}$ van de rij $a_p, \ldots, a_{q-1}$) .

Neem nu aan dat er een integer **array a[0 : n - 1]** is, en dat er drie integers p, q en k zijn, zodat steeds **zal gelden:**

P: $0 \leq p < q \leq n$, $0 \leq k < q - p$
x = het k^{de} element uit de rij a[p] ,..., a[q - 1]

dan kan bij deze afspraak de volgende programmastructuur horen:

```
"initialiseer a, p, q en k zodanig dat P geldt";
while p ≠ q - 1 do
    begin "kies een willekeurig element s uit de rij
                a[p], ..., a[q - 1]" ;
          "bepaal a, i en j zodanig dat:
                a[ℓ] < s voor alle p ≤ ℓ < i
                a[ℓ] = s voor alle i ≤ ℓ < j
                a[ℓ] > s voor alle j ≤ ℓ < q " ;
          "verklein q - p en verander k zodanig dat P weer geldt"
    end;
print (a[p])
```

Een kortstondige bezinning leert ons dat de driedeling rond de spil s het lastigste deelprobleem is dat we hebben overgehouden. Dit is een goede reden om met dat probleem te beginnen. Maar aangezien schrijver dezes weet dat onze oplossing op dat onderdeel niet meer stuk zal lopen, gaan we eerst naar de eenvoudige deelproblemen kijken.

Wat betreft de initialisatie zullen we aannemen dat het array a al een waarde heeft gekregen in een omvattend blok. De beginwaarden voor p, q en k zijn triviaal:

```
p := 0; q := n; k := n div 2;
```

Een willekeurig element uit de rij a[p], ..., a[q - 1] kiezen kan doorgaans op vele manieren. Wij nemen de eenvoudigste manier

```
s := a[p] .
```

De verkleining van q - p en de aanpassing van k berust op de drie gevallen die we reeds bekeken hebben. Deze drie gevallen sluiten elkaar uit, derhalve:

```
if p + k < i then  q := 1  else
if p + k < j then  begin p := p + k; q := p + 1; k := 0 end
              else  begin k := p + k - j; p := j end
```

(We zien aan bovenstaande programmatekst dat p + k constant blijft, en wanneer we straks het volledige programma zullen opschrijven introduceren we in plaats van k de integer r = p + k. We passen dan als het ware een coördinatentransformatie toe.)

Tenslotte de driedeling, een probleem dat inmiddels bekend is geworden onder de naam van "the Dutch National Flag". Een aparte paragraaf uit dit hoofdstuk gewijd aan de Nederlandse Vlag zou alleszins te rechtvaardigen zijn geweest, echter we zullen ons in het kader van het vigerende vraagstuk een kortere behandeling permitteren. De oplossing die nu volgt is afkomstig van E.W. Dijkstra.
De getallen a[p], ..., a[q - 1] moeten zodanig gerangschikt worden dat linksaangeschoven de getallen kleiner dan s komen te staan, rechtsaangeschoven de getallen groter dan s en ertussen in de getallen gelijk aan s. We zullen

proberen dit karwei te klaren door alle getallen precies één keer te inspecteren (en als dat lukt hebben we de beste algorithme gevonden, aangezien elke getal ook minstens één keer geinspecteerd zal moeten worden). Als gedurende een inspectie een getal kleiner dan s bevonden wordt zullen we het (in gedachte) rood kleuren, indien het groter dan s bevonden wordt blauw, en anders wit. In deze beeldspraak is de vraag dan de getallen a[p], ..., a[q - 1] zo te permuteren dat de rode links staan, de blauwe rechts en de witte in het midden. Aanvankelijk is de situatie zo dat alle getallen kleurloos zijn, en een goede generalisatie van begin- en eindtoestand lijkt dan ook de toestand waarin een aantal getallen nog kleurloos is, terwijl de overige getallen reeds kleur hebben bekend. We zullen de volgende afspraak maken:

P_{dnf}:

a[ℓ] is rood	voor alle ℓ :	$p \le \ell < i$
a[ℓ] is wit	voor alle ℓ :	$i \le \ell < w$
a[ℓ] is blauw	voor alle ℓ :	$j \le \ell < q$
a[ℓ] is kleurloos	voor alle ℓ :	$w \le \ell < j$.

Het aantal kleurlozen bedraagt hiermee j - w. Voor j - w = 0 heeft de gewenste driedeling zich voltrokken. Laten we aannemen dat we voor j - w > 0 het element a[w] kleur laten bekennen. Er doen zich drie gevallen voor:

- a[w] is rood: een verwisseling van a[i] en a[w] , gevolgd door een ophoging van zowel i als w met 1 laat P_{dnf} invariant .

- a[w] is blauw: een verlaging van j met 1, gevolgd door een verwisseling van a[w] en a[j] laat P_{dnf} invariant .

- a[w] is wit: een ophoging van w met 1 (voorafgegaan door een verwisseling van a[w] en a[w]) laat P_{dnf} invariant .

We zien dat in elk der gevallen a[w] met een mogelijk ander element verwisseld dient te worden. Teneinde de code voor de verwisseling slechts één keer in de programmatekst te laten voorkomen zullen we in een (hulp)-variabele v1 de waarde van w administreren, en in een (hulp)variabele v2 de identiteit van het element waarmee a[v1] verwisseld dient te worden.

Hiermee wordt het programma voor de driedeling:

```
i := p; j := q; w := p;
while j - w ≠ 0 do
      begin v1 := w;
            if a[w] < s then begin v2 := i; i := i + 1; w := w + 1 end else
            if a[w] = s then begin v2 := w; w := w + 1 end
                        else begin j := j - 1; v2 := j end;
            z := a[v1]; a[v1]:= a[v2]; a[v2]:= z
      end
```

Opmerking: Het is goed wanneer we ons realiseren dat het stukje programma dat de inspectie van a[w] en de nodige verwisselingen moet doen op een groot aantal manier gecodeerd kan worden. In bovenstaande code wordt begonnen met de test of a[w] < s is. Maar we hadden evengoed kunnen beginnen met de test a[w] = s of a[w] > s. Indien we zouden weten dat het merendeel der getallen uit de rij a[p], ..., a[q - 1] groter dan s is, vergt het gegeven programma meestal twee tests per getal,en waren we liever begonnen met de test a[w] > s. Het is de notatiewijze van Algol 60 (en van heel veel andere programmeertalen) die ons confronteert met in feite onbeslisbare beslissingsproblemen. Aangezien ons uitdrukkingsvermogen op de duur ons denkvermogen beïnvloedt, is dit een angstige gedachte. In verband met het bovenstaande: het is voor een oplettend oog reeds waar te nemen dat menig ervaren programmeur lijdt aan wat genoemd zou mogen worden het if-then-else-syndroom. (Einde opmerking.)

Oefening. De beslissing om onder de kleurlozen het element a[w] kleur te laten bekennen, is niet arbitrair. Een ander candidaat zou a[j - 1] zijn geweest. Analyseer de gevallen die zich dan zouden hebben voorgedaan. (Einde oefening)

Tenslotte volgt nu het totale programma voor de berekening van de mediaan. Opgemerkt zij nog dat deze algorithme uitgevonden is door C.A.R. Hoare, en nu wereldbekend is onder de naam FIND.

```
p := 0; q := n; r := n div 2;
while p ≠ q - 1 do
      begin s := a[p];
            i := p; j := q; w := p;
            while j - w ≠ 0 do
                  begin v1 := w;
                        if a[w] < s then begin v2 := i; i := i+1; w := w + 1 end else
                        if a[w] = s then begin v2 := w; w := w + 1 end
                                    else begin j := j - 1; v2 := j end;
                        z := a[v1]; a[v1] := a[v2]; a[v2] := z
                  end;
            if r < i then q := i else
            if r < j then begin p := r; q := p + 1 end else p := j
      end;
print (a[p])
```

De efficiëntie van deze algorithme hangt sterk af van de keuze van s. Het loopt allemaal erg slecht als a[p] steeds bv. het minimum van a[p],..., a[q - 1] is. Bij de driedeling wordt dan geen enkel element kleiner dan s gevonden. De algorithme loopt echter als een lier als a[p] steeds in de buurt van het gemiddelde van de getallen a[p],..., a[q - 1] ligt.
We zullen nu echter niet meer ingaan op slimmere strategieën ter bepaling van s, dan de hierboven gegeven s := a[p].

6.5. Een eenvoudig loketsysteem

Van een loket wordt gebruik gemaakt door n ≥ 1 klanten, genummerd van 1 t/m n. De klanten worden bediend in volgorde van aankomst. Het komt nooit voor dat er klanten in de rij voor het loket staan, terwijl de lokettiste niemand bedient. Het loket opent op tijdstip 0, en sluit zodra de laatste klant bediend is. Als het aankomstmoment van klant i nu gegeven is door a[i], de bedieningstijd van klant i door b[i] > 0, en als $0 \leq a[1] < a[2] < \ldots < a[n]$, bereken dan het sluitingstijdstip van het loket, en de lengte van de langste rij die tijdens openstelling voor het loket heeft gestaan.

Een moment van bezinning leert ons dat de twee gevraagde antwoorden ons voor een dilemma dreigen te plaatsen. Immers enerzijds suggereert het

feit dat de sluitingstijd, i.e. het vertrekmoment van klant n, gevraagd wordt ons om vooral op de vertrekmomenten te gaan letten, terwijl anderzijds de langste rij ons dwingt tot aandacht voor de aankomstmomenten, aangezien op geen enkel ander tijdstip de lengte van een rij voor het loket kan toenemen. Echter de bepaling van de langste rij vergt bij nader inzien dat we kennis hebben over het totaal aantal personen in het systeem ten tijde van een aankomst, en daartoe zullen we dan tevens op vertrekmomenten moeten letten.
Neem aan dat v_k het vertrekmoment van klant k voorstelt ($k < n$). Ter bepaling van v_{k+1} onderscheiden we dan twee gevallen.
Als klant k + 1 arriveert in een leeg systeem, d.w.z. als $a[k + 1] > v_k$, dan is $v_{k+1} = a[k + 1] + b[k + 1]$.
Als klant k + 1 arriveert in een niet-leeg systeem, d.w.z. als $a[k + 1] \leq v_k$, dan is $v_{k+1} = v_k + b[k + 1]$.
Met behulp hiervan is het niet moeilijk de successieve vertrekmomenten te berekenen.

Wat we nu kunnen proberen te berekenen is het aantal mensen, w_j , in het systeem op het moment van aankomst van klant j.
Het is duidelijk dat

$$w_j = \text{aantal aankomsten t/m tijdstip } a[j] - \text{aantal vertrekken tot tijdstip } a[j] .$$

Als nu k het hoogste klantnummer voorstelt waarvoor geldt dat $v_k \geq a[j]$ (en zo'n klant bestaat, nl. $k = j$) dan is

$$w_j = j - k + 1 .$$

Indien we in ons te maken programma erin slagen de bij j behorende waarde van k te bepalen, dan is w_j slechts een triviaal afgeleide grootheid, en kunnen we de eventueel nieuwe maximum rijlengte op tijdstip a[j] bijwerken onder controle van de expressie j - k + 1.
Zodra het moment a[n] bereikt is kan de maximum lengte niet meer groeien, en rest nog de berekening van het sluitingstijdstip van het loket. Dit is echter heel eenvoudig als geen aankomst meer te verwachten is, want het moment waarop de ene klant vertrekt valt dan steeds samen met het moment waarop de bediening van de volgende uit de rij begint, enz., tot de rij is uitgeput.

Na deze aftastingen komen we tot de volgende afspraken voor de invariant:

P: $1 \leq j \leq n$,
k = minimale waarde van $1 \leq i \leq n$ waarvoor geldt dat
$v_i \geq a[j]$
$v = v_k$
m = maximale rijlengte opgetreden t/m tijdstip $a[j]$

We schrijven nu meteen het programma op:

```
j := 1; k := 1; v := a[1] + b[1]; m := 1;
while j ≠ n do
      begin j := j + 1;
            while v < a[j] do
                  begin k := k + 1;
                        if a[k] > v then v := a[k];
                        v := v + b[k]
                  end;
            if m < j - k + 1 then m := j - k + 1
       end;
while k ≠ n do begin k := k + 1; v := v + b[k] end;
print (v); print (m)
```

De efficiëntie van het programma is zodanig dat de hoeveelheid rekenwerk evenredig is met n. Dit is eenvoudig in te zien door na te gaan hoe vaak de statements in het binnenste cykeltje gedurende het gehele rekenproces kunnen worden uitgevoerd, bijvoorbeeld de statement k := k + 1. Gezien de betekenis van k geldt voor het buitenste cykeltje dat steeds $k \leq j$ zal zijn. Echter omdat k := k + 1 de enige verandering is waaraan k onderworpen wordt, geldt tegen de tijd dat $j = n$ bovendien : $k \leq n$. Omdat k met de waarde 1 begint betekent dit dat de statement k := k + 1 maximaal n - 1 keer kan zijn uitgevoerd.

Oefening: In tegenstelling tot de voorbeelden in de vorige paragrafen hebben we hier terstond na de formulering van P het volledige programma opgeschreven. Ga tot in detail na hoe het bovenstaande programma gerechtvaardigd is op grond van de gegeven invariant.
(Einde oefening.)

6.6. Een "administratief" vraagstuk

Laat twee rijen v en w gegeven zijn door de integer array's v[0 : P] en w[0 :Q] , laat bovendien gelden: $v[0] \leq \ldots \leq v[P-1] < v[P]$ en $w[0] \leq \ldots \leq w[Q-1] < w[Q]$, en tenslotte $v[0] = w[0]$ en $v[P] = w[Q]$. Gevraagd wordt een programma te schrijven dat elk getal v[i] uitprint dat niet in de rij w voorkomt, en ook elk getal w[j] dat niet in de rij v voorkomt.

(Een probleem van bovenstaande soort is in zekere zin typerend voor een bepaalde klasse van zogenaamde administratieve problemen: daarbij is v dan geen rij integers maar bv. een alfabetisch gesorteerde rij namen van werknemers die aan een bepaalde eigenschap voldoen, en w een alfabetisch gesorteerde rij van werknemers die aan een andere eigenschap voldoen, etc. : vul maar in !
Het feit dat een probleem van deze soort wel eens in administratieve werelden voorkomt is nog geen argument om het dan te bestempelen als een probleem van een administratieve signatuur, wat dit laatste dan ook moge betekenen. Klasseindeling van programmeerproblemen naar toepassingen is misleidend en daarom gevaarlijk. Bovendien is het nut ervan niet duidelijk.)

Het meest voor de hand liggende idee ter verkrijging van het gewenste antwoord is om voor elk getalletje v[i] uit de v-rij maar eens na te gaan of het in de w-rij voorkomt, zo nee het dan uit te printen, zo ja het te vergeten. En vervolgens andersom, met de rollen van v en w verwisseld. Het wordt dan al snel duidelijk dat op deze manier een grote hoeveelheid overtollig werk gedaan wordt. Indien immers $v[p] > w[q]$ dan is zeker ook $v[p+1] > w[q]$, en hoeven wij wat betreft het voorkomen van v[p + 1] in de w-rij ons slechts te beperken tot het onderzoeken van de w-rij vanaf w[q + 1].
Aangezien in dit vraagstuk die getallen die zowel in de ene als in de andere rij voorkomen een centrale rol spelen, doen we er voorlopig goed aan deze getallen eens van een naam te voorzien :

$$g_0 < g_1 < \ldots < g_{k-1} < g_k$$

Hierbij is g_0 de kleinste gemeenschappelijke waarde, g_1 de op een na kleinste gemeenschappelijke waarde, etc., en g_k de grootste gemeenschappelijke waarde. We weten dat

$$g_0 = v[0] = w[0] ,$$

en

$$g_k = v[P] = w[Q] .$$

De getallen x uit de v-rij en de w-rij die uitgeprint moeten worden zijn nu precies die getallen die voldoen aan $g_i < x < g_{i+1}$ met $0 \leq i < k$.

Er is nu hoop dat we in het kader van de overgang van g_i naar g_{i+1} de te printen getallen x : $g_i < x < g_{i+1}$ kunnen isoleren. Immers: zowel deze getallen x als het getal g_{i+1} zijn alle van de vorm v[p] met $v[p] > g_i$ of van de vorm w[q] met $w[q] > g_i$.

We zullen uitgaan van de volgende afspraak. Laat er integers c, p en q zijn zodat

Pr:
- er is een $0 \leq i \leq k$ zodat $c = g_i$
- p is de minimale index m in de v-rij waarvoor geldt: v[p] = c
- q is de minimale index m in de w-rij waarvoor geldt: w q = c
- alle elementen v[i] met $0 \leq i < p$ waarvoor geldt dat v[i] niet voorkomt in de w-rij zijn geprint
- alle elementen w[j] met $0 \leq j < q$ waarvoor geldt dat w[j] niet voorkomt in de v-rij zijn geprint.

Dan is een acceptabele programmastructuur.

```
"Initialiseer c, p en q zodanig dat Pr geldt";
while "c ongelijk g_k" do
      begin "Verhoog c tot de eerstvolgende gemeenschappelijke
            waarde, onder invariantie van Pr"
      end.
```

Bovenstaand programma zal zeker eindigen omdat per slag c wordt opgehoogd, en zal ook het gewenste eindresultaat bewerkstelligen omdat voor $c = g_k$ (= v[P] = w[Q]) alle gevraagde getallen geprint zijn.

Voor de verhoging van c tot de eerstvolgende gemeenschappelijke waarde is het in elk geval voldoende ons onderzoek te beperken tot die waarden v[i] waarvoor v[i] > c, en tot die waarden w[j] waarvoor w[j] > c. Omdat de v-rij en de w-rij beide monotoon niet-dalend zijn, en omdat c = v[p] en c = w[q], kan het zijn dat we met uitsluitend ophogingen van p en van q uit de voeten kunnen komen. Alle getallen v[i] en w[j] waarvoor v[i] = w[j] = c kunnen we al zonder meer buiten beschouwing laten omdat die niet uitgeprint mogen worden, en derhalve lijkt een ophoging van p tot de kleinste index waarvoor v[p] > c, en een ophoging van q tot de kleinste index waarvoor w[q] > c, op zijn minst geboden. Deze twee indices zijn gedefinieerd want c is noch in de v-rij noch in de w-rij het maximale element. Rest nog de vraag hoe we uitgaande van deze nieuwe waarden van p en q het kleinste gemeenschappelijke element van de (deel)rijen (v[p], ...,v[P]) en (w[q], ..., w[Q]) kunnen vinden. Welnu: voor v[p] < w[q] is p nog te klein. Het getal v[p] komt in de rij (w[q],...w[Q]), en dus (ga na!) in de hele w-rij niet voor, en mag worden geprint. Het probleem reduceert dan tot het vinden van het kleinste gemeenschappelijke element van de rijen (v[p + 1], ..., v[P]) en (w[q], ... w[Q]). Op het geval w[q] < v[p] is een soortgelijke beschouwing van toepassing.

Nu volgt het volledige programma bijna onmiddellijk:

```
c := v[0]; p := 0; q := 0;
while c ≠ v[P] do
      begin while c ≤ v[p] do p := p + 1;
            while c ≤ w[q] do q := q + 1;
            while v[p] ≠ w[q] do
               if v[p] < w[q] then begin print (v[p]); p := p + 1 end
                              else begin print (w[q]); q := q + 1 end;
            c := v[p]
      end
```

Oefening 1: Controleer nauwgezet de juistheid van bovenstaande programmatekst. Ga na waarom het laatste van de drie cykeltjes binnen de grote cykel eindigt en correct werkt.
(Einde oefening 1.)

Oefening 2: Waarin wijzigt het programma indien gegeven zou zijn dat zowel de v-rij als de w-rij strict stijgend zijn.
(Einde oefening 2.)

Oefening 3: Hoe hangt de benodigde hoeveelheid rekentijd van de constanten P en Q af?
(Einde oefening 3.)

Oefening 4: Ga na waar we gebruik hebben gemaakt van de gegevens v[0] = w[0] en v[P] = w[Q] .
Zonder deze extra gegevens zou het probleem aanzienlijk lastiger zijn geweest, redeneertechnisch en (dus !) ook codeertechnisch.
Probeer dit in te zien, desnoods door het vraagstuk voor dit geval te maken.
(Einde oefening 4.)

Appendix A

Syntaxis van Algol

1. Inleiding

Algol 60 was de eerste programmeertaal waarvoor een bevredigend strenge definitie werd gegeven voor de

- syntaxis; d.i. de beschrijving van de in de taal toegelaten constructies (de grammatica van de taal zou men kunnen zeggen)
- semantiek; d.i. de "betekenis" van die toegelaten constructies in de zin van het resultaat bij uitvoering van het programma.

Voor de semantische beschrijving werd nog gebruik gemaakt van onze gewone spreektaal, doch voor de syntactische beschrijving werd een zogenaamde meta-taal ingevoerd. Deze BNF notatie (als afkorting van Backus Naur Form naar de namen van twee bij de ontwikkeling van Algol betrokken onderzoekers) maakt gebruik van slechts enkele taalbeschrijvings- of "meta"-symbolen (die zelf niet tot de te beschrijven taal behoren), te weten

:: =	voor "bestaat uit" of "gedefinieerd als"
|	voor "(niet uitsluitende) of"
<...>	de omsluitende "metahaken", waartussen de te definiëren grootheid staat.

Voorbeeld

<cijfer> :: = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<cijfer duo> :: = (<cijfer>, <cijfer>)

<natuurlijk getal> :: = <cijfer> | <natuurlijk getal> <cijfer>

Met behulp van deze definities is te verifiëren dat (9,8), resp. 981 voorbeelden van cijferduo, resp. natuurlijk getal zijn, doch 9.8 en 98.1 niet. Merk op dat de derde definitie een zogenaamde recursieve definitie is omdat rechts weer de linksstaande te definiëren grootheid staat. Een "cirkel-

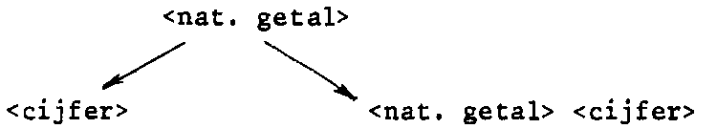
definitie" is het echter niet omdat rechts ook het alternatief <cijfer> genoemd is. We zouden deze recursieve definitie daarom ook kunnen lezen als een verkorte schrijfwijze voor

<cijfer> | <cijfer> <cijfer> | <cijfer> <cijfer> <cijfer>|... ... ...

Volgens de definitie is ook 00981 een natuurlijk getal, hoewel we in het dagelijkse leven gewend zijn om "vooropstaande nullen" (leading zeroes) weg te laten.
In latere jaren zijn nog andere representaties voor syntax definities ontwikkeld, vooral om deze meer aanschouwelijk te maken. We noemen daarvan slechts:

- de syntaxgraaf methode

  Voorbeeld



- de stroomdiagram methode

  Voorbeeld

Opmerking Nooit tegen de pijlrichting in gaan!

In de volgende paragrafen zullen we ons slechts van de BNF notatie en de stroomdiagram methode bedienen. Voor andere talen (waarin met name het aantal herhalingen in een recursieve definitie sterk beperkt is) dan Algol moet men veelal nog andere metasymbolen invoeren (voor Cobol bijvoorbeeld accoladen om grootheden waaruit één keus gemaakt moet worden; rechte haken om grootheden die 0 of 1 keer mogen voorkomen). Een verdienste van Algol is tenslotte nog dat spaties geen betekenis hebben en dat Algol (dank zij zijn onderstrepingssymboliek) geen keywords kent, dat wil zeggen dat bepaalde woorden als begin en end rustig als identifier gebruikt mogen worden (niet dat dit raadzaam is!).

In het volgende zullen we slechts de syntaxis van de in dit college in detail behandelde Algolconstructies weergeven; voor een volledige behandeling zij verwezen naar het zogenaamde Revised Report, gepubliceerd in onder andere Numerische Mathematik 4 (1963) 420-453 en C ACM 6 (1963). Om deze weergave zo duidelijk en compact mogelijk weer te geven maken we daarbij gedeeltelijk gebruik van de BNF, gedeeltelijk van de stroomdiagram notatie. De gebruikte terminologie sluit wel aan bij de syllabus, doch niet in alle details bij het Revised Report. Om ruimte te sparen worden soms afkortingen gebruikt, bijvoorbeeld ident. voor identifier, nat. getal voor natuurlijk getal, enz. In sommige gevallen zullen we geen formele definities geven, maar een informele tussen aanhalingstekens. We doen dit wanneer de formele definitie veel schrijf- en leeswerk zou betekenen en de informele definitie niet tot misverstanden kan leiden.

Appendix B

De volgende paragrafen over algorithmen en rekenautomaten zijn ontleend aan een oude syllabus van prof. d r E.W. Dijkstra over algorithmen.

1. Algorithmen

In het begin van de negende eeuw na Christus werkte in de bibliotheek van het toen pas gestichte Bagdad een zekere Mohammed ben Musa, bijgenaamd al-Khwarizmi, die het eerste Arabische boek over algebra schreef. (Het woord "algebra" is een verbastering van de Arabische titel!). Behalve dat schreef hij een handleiding voor het zogenaamde Indische rekenen, dat wil zeggen het rekenen in het tientallig stelsel. Enige eeuwen heeft dit boek (in Latijnse vertaling) een grote rol gespeeld bij de invoering van het tientallig stelsel in Europa; de bijnaam van de auteur, al-Khwarizmi, leeft tot op de huidige dag voort in het woord "algoritme", dat sedert die tijd in zwang is voor "rekenvoorschrift", in het algemeen voor "handelingsvoorschrift".

Ook al komt het woord algoritme in het dagelijks leven nu niet zoveel voor, met het idee van een algoritme worden we dagelijks geconfronteerd. Sprekende voorbeelden van algoritmen uit het dagelijkse leven zijn breipatronen, recepten en montagevoorschriften. Wie een ander naar de weg vraagt, vraagt in wezen naar een algoritme: hij vraagt naar een handelingsvoorschrift dat hij kan opvolgen om zijn doel te bereiken.

Aan deze alledaagse algoritmen kunnen wij al een aantal typische eigenschappen illustreren. Essentieel is, dat een of andere totale verrichting ontleed wordt, dan wel opgebouwd wordt uit een aantal op zichzelf simpelere handelingen: het breien van een truitje wordt uitgedrukt in termen van het breien van toeren, die op hun beurt beschreven worden door een bepaalde opeenvolging van speciale steken, minderingen en meerderingen. Als onderdeel van een recept van ragout zal men vinden, dat een niet te grote ui gesnipperd en bruin gefruit moet worden, etc.

Verder observeren wij, dat hele klassen algoritmen herleid worden tot handelingen uit het repertoire, dat duidelijk bij die klasse hoort, waarbij als regel elke handeling uit het repertoire in allerlei algoritmen voorkomt. Zo zal het deelvoorschrift om een uitje te fruiten als onderdeel in allerlei recepten voorkomen, zo komen typische breihandelingen (zoveel steken zus, zoveel steken zo, minderen etc.) in allerlei breipatronen voor.

Vervolgens zien wij, dat een algoritme alleen dan als zodanig functioneert, wanneer tussen opsteller en uitvoerder geen misverstand of verschil van mening bestaat over het repertoire handelingen, waaruit de totale verrichting moet worden opgebouwd. Breipatronen en montagevoorschriften zijn in dit opzicht vrij zuivere algoritmen; recepten zijn in dit opzicht al wat troebeler ("zout naar smaak toevoegen" is niet zo duidelijk; het is dan ook geen toeval, dat een gerecht eerder mislukt dan een breiwerk). Bij het vragen naar de weg is de verwarring over wat toegelaten handelingen zijn doorgaans volkomen; ze zijn als algoritmen dan ook zelden zonder dubbelzinnigheid uitvoerbaar.
Tenslotte moet een algoritme nog een zekere innerlijke logica hebben. Als er in een recept, waarin nog niet over bouillon geschreven is, staat "Laat vervolgens de bouillon afkoelen", dan is er iets mis, ook al weet de uitvoerder op zich zelf best, hoe hij bouillon volgens de regels van de kunst moet laten afkoelen! Bij het naar de weg vragen is een antwoord als "Rechtuit en dan bij de laatste stoplichten linksaf " niet te gebruiken.

Hebben wiskundigen zich dus al sinds eeuwen met algoritmen beziggehouden, in de twintigste eeuw zijn algoritmen opnieuw in het middelpunt van de wiskundige belangstelling komen staan.
De eerste aansporing daartoe kwam van het grondslagenonderzoek, waarin men zich de vraag ging stellen aan welke eisen een deugdelijk wiskundig bewijs nu eigenlijk moest voldoen. Dit was een soort "wiskundig gewetensonderzoek", dat nodig was geworden omdat men tegen allerlei paradoxen aanstootte. Dat men hierbij algoritmen tot voorwerp van studie maakte is

niet zo verwonderlijk als we bedenken, dat een bewijs een bepaald soort handelingsvoorschrift is, namelijk een redeneervoorschrift om tot een bepaalde conclusie te komen. De moeilijkheden bleken voort te komen uit bewijsstappen, waarbij niet gezegd werd, hoe je ze moest uitvoeren. Wij zullen deze logische schermutselingen niet verder vervolgen. Ter aanduiding van de aansluiting met wat volgt vermelden we slechts, dat de gedachteconstructie van een van de geniaalste onderzoekers op dit gebied Alan M. Turing, onder de naam "Turing Machine" wereldvermaard is geworden. De tweede aansporing om zich op algoritmen te concentreren kwam door de ontwikkeling van de electronische rekenautomaten (ook wel computers of rekenmachines genaamd). Deze machines zijn namelijk in staat om algoritmen (van een bepaalde klasse) in zich op te nemen en vervolgens getrouwelijk uit te voeren. Rekenautomaten zijn dus machines, die algoritmen kunnen uitvoeren; voor algoritmen, die bestemd zijn om door een rekenautomaat uitgevoerd te worden is de naam "programma" in zwang gekomen; het opstellen van programma's heet "programmeren" en degene, die dit doet, heet "programmeur". Door de komst van de rekenautomaten is de belangstelling voor algoritmen drastisch veranderd: van een uiterst theoretische belangstelling is het, door de grote toepassingsmogelijkheden van rekenautomaten en hun economische importantie geworden tot een buitengewone practische aangelegenheid. Bovendien, was aanvankelijk de belangstelling voor algoritmen in hoofdzaak analytisch, nu is de belangstelling in hoge mate synthetisch: er moeten algoritmen van allerlei soort gemaakt worden, willen wij de mogelijkheden van rekenautomaten kunnen benutten.

2. Rekenautomaten

In deze paragraaf zullen we min of meer een indruk geven van wat een rekenautomaat kan, uit welke functionele componenten zo'n machine is opgebouwd. Uit het dagelijkse leven zijn dergelijke machines nauwelijks bekend, we kunnen wel diverse automatische mechanismen noemen, die ieder een of ander aspect illustreren.

Te noemen zijn de roltrappen van een voetgangerstunnel. 's Nachts staan deze roltrappen stil. Wie bij de ingang een neergaande of bij de uitgang

een opgaande roltrap oploopt onderbreekt daarbij een lichtstraal die op een fotocel pleegt te vallen en ten gevolge van deze onderbreking zet de roltrap zich in beweging en rolt hij (ruim) eenmaal zijn eigen lengte af. Wie zich dan daardoor laat transporteren en aan de andere kant de trap verlaat, merkt dat even later de trap achter hem weer tot stilstand komt, tenzij inmiddels opnieuw iemand de trap is opgegaan en de straal heeft onderbroken: de roltrap rolt zijn eigen lengte af sinds de laatste onderbreking van de straal.
Dit betekent, dat vanaf een bepaald ogenblik (namelijk de onderbreking van de straal) het mechanisme automatisch een vaste handeling verricht, autonoom "een vast programma" afwerkt, namelijk zijn eigen lengte eenmaal afrollen. Er moet een bepaald aantal treden "verrold" worden; als de trap tot ergens halverwege in de afrolling gevorderd is, ergens halverwege in de afwerking van zijn programma, dan zit kennelijk ergens in het mechanisme een geheugenelement, dat bijvoorbeeld bijhoudt over hoeveel treden nog doorgerold moet worden, totdat de trap weer moet stoppen. Wij kunnen ons voorstellen, dat de roltrap is uitgerust met een zogenaamde teller, een geheugenelement, waarin genoteerd staat het aantal treden, waarover de trap nog moet rollen. Wie de lichtstraal onderbreekt, zorgt daardoor dat dit geheugenelement gevuld wordt met het maximale bedrag (= aantal zichtbare treden van de trap + een beetje extra), als de trap loopt wordt de inhoud van dit geheugenelement per treetransport met 1 verminderd en als de inhoud van dit geheugenelement terugkeert tot nul, stopt de trap.

In de praktijk worden dergelijke geheugenelementen gerealiseerd door een *object dat zich in een groot* aantal discrete toestanden kan bevinden, waarbij men met elke toestand van het geheugenelement een waarde associeert. "Het element vullen met een bepaald bedrag" betekent "het element brengen in de met dit bedrag geassocieerde toestand". Typerend voor het hele automatisme is:
1. de *aanwezigheid van een dergelijk* geheugenelement
2. de beïnvloeding van de inhoud van dit geheugenelement door de activiteit

van het mechanisme (elke tredeverschuiving gaat gepaard met een vermindering van de inhoud met 1)

3. de beïnvloeding van het mechanisme door de inhoud van het geheugenelement (trap stoppen als deze inhoud tot nul is teruggekeerd)
4. het startsignaal (hier de onderbreking van de lichtstraal) waardoor het geheugenelement in de aanvangstoestand gebracht wordt.

Vergeleken bij de processen, die van moderne rekenautomaten gevergd worden, is wat van de roltrap verlangd wordt kinderachtig eenvoudig. Waar de roltrap uitkomt met een enkel geheugenelement, kan men in een moderne rekenautomaat duizenden van dergelijke geheugenelementen aanwijzen, die allemaal dienen ter vastlegging, ter onderscheiding van de interne toestanden, waarin het rekenproces zich bevinden kan. Mede door dit enorme aantal geheugenelementen is een rekenmachine een heel andere automaat geworden dan het besturingsmechanisme van een roltrap; de functie van de geheugenelementen is in principe echter dezelfde.

Er is een ander, drastisch verschil tussen de moderne rekenautomaat en de roltrap. Wat van de roltrap gevergd wordt is eigenlijk altijd hetzelfde, namelijk een standaard reactie op een uniek startsignaal. Van de moderne rekenautomaat verlangt men, dat hij allerlei rekenprocessen kan uitvoeren, allerlei algoritmen in zich kan opnemen en gehoorzamen. Men drukt dit wel uit door te zeggen, dat een rekenautomaat een "general purpose" machine is; het wordt iets duidelijker tot uitdrukking gebracht door te zeggen, dat moderne rekenmachines "programmeerbare automaten" zijn.

Er zijn andere apparaten, waarin we dit aspect van programmeerbaarheid al terugvinden. In de technische sfeer vinden we dit bij het weefgetouw van Jacquard, in de amusementssfeer vinden we dit terug bij het beter bekende draaiorgel. Beide worden gestuurd door een "boek", dat bestaat uit een lange reep kaarten, waarin gaatjes de uit te voeren handelingen voorschrijven. Bij het weefgetouw van Jacquard leggen de gaatjes (dat wil zeggen de aan- respectievelijk afwezigheid van gaatjes) vast, welke draden van de schering dit keer omhoog moeten, opdat het uiteindelijke damasten tafelkleed de Franse lelie, dan wel het familiewapen vertoont; bij het draaiorgel leggen de gaatjes vast welke pijpen moeten worden aangeblazen en welke balgen voor trommels en belletjes bediend moeten worden.

Enerzijds zijn deze apparaten specifiek; je kunt het weefgetouw alleen maar voor weven gebruiken en het draaiorgel alleen maar om muziek te maken. Zo is ook de "general purpose" rekenmachine specifiek: je kunt hem rekenprocessen laten uitvoeren.

Anderzijds zijn ze, op hun gebied, tamelijk algemeen! Voor het weefgetouw van Jacquard betekent dit, dat het allerlei patroontjes kan weven, van Franse lelies tot familiewapens toe en juist door die algemeenheid door het boek gestuurd moeten worden om te bepalen, wat er dit keer geweven moet worden. Voor het draaiorgel betekent dit, dat het zoals het uit de fabriek komt, allerlei muziek kan maken, van psalmen en volksliederen tot tophits en juist door die algemeenheid het draaiboek nodig heeft opdat vastligt, wat er dit keer voor muziek gemaakt moet worden. Zo ook voor de moderne programmeerbare rekenautomaat: hij kan allerlei algoritmen uitvoeren en het is juist deze algemeenheid die maakt, dat hij steeds een programma nodig heeft ter beschrijving van de gedragslijn, die dit keer gevolgd moet worden.

De rekenautomaat kan dus programma's uitvoeren, mits zo'n programma aan de machine is toegevoerd. Analoog aan het boek bij het weefgetouw en het draaiorgel wordt het programma als regel geponst in hetzij ponskaarten (bekend van de postgiro) dan wel in ponsband (sinds lang bekend bij het telexverkeer). De rekenmachine is uitgevoerd met een zogenaamde "lezer" (kaartlezer of ponsbandlezer), waardoor de informatie in een pak kaarten dan wel een ponsband kan worden afgetast. Geschiedt dit bij het weefgetouw mechanisch (namelijk door tastpinnetjes, die afhankelijk van de aanwezigheid van een gat al of niet worden tegengehouden) en bij het draaiorgel pneumatisch (waar afhankelijk van de aanwezigheid van een gat een pijp wel of niet wordt aangeblazen), bij moderne rekenautomaten geschiedt het aftasten (om der wille van de snelheid) meestal optisch: wel een gat laat meer licht door dan geen gat.

Hier houdt overigens de analogie tussen weefgetouw en draaiorgel enerzijds en rekenautomaat anderzijds op. Weefgetouw en orgel beschikken niet over

een groot geheugen (dat wil zeggen een groot aantal geheugenelementen) en elke regel van het draaiboek wordt "uitgevoerd" op het moment, dat deze regel onder de aftasters van het leesstation ligt. Bij een draaiorgel heeft dit bijvoorbeeld tot gevolg dat de enige manier om zestien maten muziek te laten herhalen is om deze zestien maten twee keer in het immers doordraaiende boek te laten voorkomen. De rekenautomaat beschikt echter wel over een groot geheugen, dat onder andere gebruikt wordt om eerst de hele algoritme op te nemen, voordat aan de werkelijke uitvoering ervan begonnen wordt. In een programma kunnen we, zoals we later zullen zien, wel aangeven, dat bijvoorbeeld een stukje nog een keer herhaald moet worden (zoals ook in bladmuziek!).

We zijn inmiddels twee componenten van de rekenmachine tegengekomen, het leesstation via hetwelk het geponste programma aan de machine wordt toegevoerd en het geheugen, waarin dit programma wordt opgenomen. Als het programma in zijn geheel is opgenomen, dan begint het eigenlijke rekenwerk: het handelingsvoorschrift wordt opgevolgd, de berekening wordt uitgevoerd. Hierbij speelt een derde onderdeel van de machine een belangrijke rol, het zogenaamde "rekenorgaan". Tijdens het rekenproces speelt het geheugen een dubbele rol: ten eerste wordt het geraadpleegd omdat het uit te voeren programma er in is opgeslagen, ten tweede wordt het gebruikt als "kladpapier" om voor het rekenproces belangrijke tussenresultaten er zolang in op te slaan. (Dit tweede gebruik van het geheugen is analoog aan het geheugenelement van de roltrap.)

Tenslotte bevat de rekenmachine apparatuur voor de vierde functie, namelijk het aan de buitenwereld terugmelden van de gevraagde uitkomsten. Dit kan via bandponsers, kaartponsers, regeldrukkers, tekentafels, kathodestraalbuizen etc. (In de wandeling heten dit "uitvoerorganen" omdat via deze apparatuur de informatie weer de machine wordt uitgevoerd; de lezers heten "invoerorganen". Omdat men in het Nederlands ook spreekt over "het uitvoeren van een programma" zou in plaats van invoer- en uitvoerorganen "import- en exportorganen" een prettige terminologie geweest zijn. In het Engels spreekt men van "Input", "Output" en "Execution".)