STICHTING
MATHEMATISCH CENTRUM
2e BOERHAAVESTRAAT 49
AMSTERDAM
REKENAFDELING

MR 85

Praktijkcursus ALGOL 60

door

L.J.M. Geurts
C.H.A. Koster
L.G.L.Th. Meertens

3e druk

1968

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

PRAKTIJKCURSUS ALGOL

1 Programma, compound statement en block (RR 4.1.1)

Een ALGOL-programma is volgens het "Revised Report on the Algorithmic Language ALGOL 60" (in het vervolg aangeduid met RR), sectie 4.1.1 een compound statement of een block. Een compound statement bestaat uit: "begin", een of meer statements, onderling gescheiden door het symbool ";" en tenslotte "end". Een block heeft dezelfde structuur, maar tussen begin en de eerste statement moeten een of meer declaraties staan, van elkaar en van de statements gescheiden door ";". Zowel compound statement als block zijn statements.
De statements waaruit een compound statement of een block zijn opgebouwd mogen dus op hun beurt een compound statement of een block zijn. Als iedere statement mogen een compound statement en een block gelabeld zijn door een of meer labels. Het programma in zijn geheel mag echter bij de meeste implementaties (d.w.z. realiseringen van ALGOL op een bepaalde computer) niet gelabeld zijn.

2 Declaraties (RR 5)

Een declaratie definieert de betekenis van een of meer identifiers. Deze definitie geldt slechts binnen het block waarin deze declaratie staat.
We kunnen de volgende soorten declaraties en bijbehorende declaratoren onderscheiden:

— Declaratie van simple variables	integer , real , Boolean
— Declaratie van arrays	integer array , real array of array , Boolean array
— Declaratie van switches	switch
— Declaratie van een procedure	procedure , integer procedure , real procedure , Boolean procedure

De declaratie van simple variables en arrays mag worden voorafgegaan door de declarator own (maar niet own array i.p.v. own real array).

3 Procedure declaration (RR 5.4)

De procedure declaration bestaat uit: het symbool procedure, eventueel voorafgegaan door integer , real of Boolean , vervolgens de procedure heading en tenslotte de procedure body. De procedure body bestaat uit een enkele statement (vaak een compound statement of een block). De procedure heading: In de formal parameter part worden, indien aanwezig, de formal parameters opgesomd. Deze zijn altijd identifiers. In de specification part wordt informatie verstrekt over nul of meer der formal parameters. Tussen formal parameter part en specification part kan een value part worden opgenomen, waarin een of meer van de formal parameters uit de specification part voorkomen achter het symbool value.

4 Scope (RR 2.4.3, 2.7, 4.1,3, 4.3.4, 4.7.3.2, 4.7.3.3, 5, 5.3.5, 5.4.3)

Een identifier stelt steeds een entiteit voor: een simple variable, een array, een switch, een procedure, een label of een formal parameter (die behalve op een der voorgaande entiteiten ook betrekking kan hebben op een string of op een expression of op een subscripted variable). Aangezien in principe gelijkluidende identifiers verschillende entiteiten kunnen voorstellen, en de betekenis van een uitdrukking of statement waarin een identifier voorkomt ervan afhangt welke entiteit door die identifier wordt voorgesteld, is het noodzakelijk dat we ondubbelzinnig kunnen vaststellen om welke entiteit het gaat.
Dit kan als volgt:
Volgens de ALGOL-syntaxis zijn een programma of een procedure body alleen een block als aan het begin een of meer declaraties staan, maar we zullen in het vervolg ieder programma en iedere procedure body ook als een block beschouwen. Nu behoort iedere entiteit die door een identifier wordt voorgesteld bij een bepaald block. Het verband tussen entiteit en identifier kan in de tekst van een ALGOL-programma op drie manieren zijn vastgelegd:

— Voor simple variables, arrays, switches en procedures door een declaratie. De entiteit behoort bij het kleinste block waarin die declaratie staat.
— Voor een label door het geplaatst zijn voor een statement, daarvan gescheiden door een ":". De entiteit behoort bij het kleinste block waarbinnen de gelabelde statement staat. (N.B. Is de gelabelde statement een procedure body dan behoort de label bij deze procedure body. In het MC-I ALGOL-systeem is echter voor het geval de procedure body met een declaratie begint, een sprong naar een dergelijke label niet toegestaan)

— Voor een formal parameter door het voorkomen in de formal parameter list.
De entiteit behoort bij de procedure body.

In deze drie gevallen is het zonder meer duidelijk welke entiteit de identifier voorstelt. Verder duidt een identifier in de value part of in de specification part de gelijknamige formal parameter aan. Komt nu een identifier in de programmatekst op een andere wijze voor, dan gaan we na of bij het kleinste block waarin deze identifier voorkwam een entiteit behoort die door een gelijkluidende identifier wordt voorgesteld. Zo ja, dan hebben we de gezochte entiteit, de identifier wordt locaal genoemd; zo nee, dan is de identifier niet-locaal. We gaan dan na of een gelijknamige entiteit behoort bij het kleinste block dat dit block omvat, en vervolgens voor het block dat dat block weer omvat, en we zetten dit proces voort totdat we de gezochte entiteit gevonden hebben. We zoeken dus van binnen uit naar buiten toe. Levert dit proces niet een entiteit op, dan kan de identifier nog een standaardfunctie of een standaardprocedure voorstellen. Is ook dit niet het geval dan is het programma fout. Het is eveneens fout als bij een block twee gelijknamige entiteiten zouden behoren.

Met dit proces is het altijd mogelijk statisch (d.w.z. zonder het programma uit te voeren) vast te stellen welke entiteit een identifier voorstelt. Voor een unsigned integer die een label voorstelt (RR 3.5.1, 3.5.5) kunnen we op analoge wijze vaststellen welke label bedoeld wordt. Het is helaas niet altijd mogelijk statisch te bepalen of een unsigned integer een label dan wel een arithmetic expression voorstelt. In het X1-ALGOL-systeem (d.w.z. MC-II systeem voor teksttesten en MC-I systeem voor uitvoeren) is de unsigned integer als label niet toegestaan.

Opmerking 1. In het officiele ALGOL is deze hele kwestie nog iets ingewikkelder (vgl. het "additional block" uit RR 4.7.3.1). In de praktijk is dit van geen belang.

Opmerking 2. In het X1-ALGOL-systeem wordt in een for statement de statement achter do altijd als een block beschouwd.

Opmerking 3. In het X8-ALGOL-systeem wordt het onderzoek naar de entiteit, voorgesteld door een identifier die voorkomt in de bound pairs van een array declaration, aangevangen bij het block dat het block omvat waarin die declaratie staat (RR 5.2.4).

5 Het procedure-mechanisme (RR 4.7, 5.4)

Een procedure statement wordt uitgevoerd, en de waarde van een function designator wordt bepaald, door een aanroep van de bijbehorende procedure.
Iedere actual parameter bij de aanroep van de procedure komt overeen met een formal parameter bij de procedure declaration. Aan iedere formal parameter die in de value part voorkomt, wordt bij aanroep de waarde toegekend die de overeenkomstige actual parameter op dat moment heeft of blijkt te hebben. Alle overige formal parameters worden overal waar ze (d.w.z. hun identifiers) in de procedure body voorkomen, vervangen door de tekst van de overeenkomstige actual parameter. Is deze geen variable (RR 3.1) maar wel een expression (RR 3) dan wordt hij eerst tussen ronde haakjes "(" en ")" geplaatst.
De aldus verkregen procedure body moet een correcte statement zijn; deze wordt uitgevoerd en vervolgens wordt het programma voortgezet. In geval we met een functie-procedure te doen hebben, moet tijdens de uitvoering een of meer malen expliciet een waarde worden toegekend aan de procedure identifier. De laatste aldus toegekende waarde wordt gebruikt om de berekening van de expression waarin de function designator (de "aanroep" van de functie-procedure) voorkwam, voort te zetten. (N.B. Het is niet juist, binnen de procedure body de procedure identifier in een expression te gebruiken wanneer men daarmee de laatst toegekende waarde wil aanduiden. Een dergelijke vermelding duidt altijd op een expliciete recursieve aanroep van de procedure).
Het is ook toegestaan om een (functie-)procedure door een go to statement te verlaten, en eveneens om een functie-procedure bij wijze van procedure statement aan te roepen. In deze twee gevallen is de aan de procedure identifier toegekende waarde van geen belang; in het eerste geval hoeft een dergelijke toekenning zelfs in het geheel nog niet geschied te zijn.
Het gevolg van bovengenoemd vervangingsproces kan zijn dat een identifier wordt binnengevoerd in een stuk programma waar een gelijkluidende identifier reeds een andere entiteit voorstelt.
Dit is volkomen correct; de door het vervangingsproces binnengevoerde identifier blijft dezelfde entiteit voorstellen als hij blijkens het in paragraaf 3 beschreven zoekproces reeds voorstelde in de actual parameter. Men kan dit ook als volgt inzien: Wanneer in een correct ALGOL-programma systematisch iedere keer dat een identifier voorkomt die een bepaalde entiteit voorstelt, deze identifier wordt vervangen door een passend gekozen andere identifier (die dus niet reeds in deze context een andere entiteit voorstelt) dan verandert de betekenis van het ALGOL-programma niet.

Hieronder volgt een lijst mogelijke actual parameters, en de juiste specificatie (indien aanwezig) van de formal parameter. (N.B. In ieder van de genoemde gevallen mag de actual parameter zelf een formal parameter zijn (het zgn. doorgeven van de formal parameter). Dit is uiteraard alleen mogelijk voor procedure-aanroepen binnen een procedure body).

actual parameter:	formal parameter: specifier	Vermelding in value part toegestaan?
arithmetic expression	integer of real	ja
Boolean expression	Boolean	ja
string	string	nee
designational expression	label	ja, (In het X1-ALGOL-systeem niet)
switch identifier	switch	nee
array identifier	array	nee
voor integer of real array ook	integer array of (real) array	ja
voor Boolean array ook	Boolean array	ja
procedure identifier	procedure	nee
voor integer of real procedure ook	integer procedure of real procedure	nee
voor Boolean procedure ook	Boolean procedure	nee

Opmerking. Uit RR 4.7.5.4 zou men kunnen opmaken dat indien in een der laatste twee gevallen de actual parameter de procedure identifier van een functie-procedure zonder parameters is, vermelding in de value part van de formal parameter wel is toegestaan. In dit geval is de specifier integer procedure, real procedure of Boolean procedure echter zeer onlogisch: de actual parameter is een expression en kan het beste integer , real resp. Boolean gespecificeerd worden; voor het X1-ALGOL-systeem is dit de enige mogelijkheid. N.B. Met een formal parameter die als switch , (type) array of (type) procedure is gespecificeerd, kan alleen een actual parameter corresponderen die uit een enkele identifier bestaat.
Bij alle formal parameters die in de value part worden opgesomd is volledige specificatie vereist. Voor het X1-ALGOL-systeem is voor iedere parameter enige specificatie verplicht.

6 Wanneer "call by name", wanneer "call by value"?

Uit de opsomming in paragraaf 5 blijkt dat bij enkele specificaties de "call by value", d.w.z. de vermelding in de value part, is toegestaan.
In deze paragraaf gaan we ervan uit dat we met een parameter te maken hebben waarbij inderdaad call by value is toegestaan. In dat geval is de specificatie

zo volledig mogelijk: integer , real , Boolean , integer array , (real) array , Boolean array of label. Voor sommige toepassingen is volledige specificatie niet wenselijk; call by value is dan niet mogelijk.
Wat betekent nu call by value in termen van uitvoering van het programma?
In dit geval worden bij de aanroep van de procedure de formal parameters als het ware gedeclareerd vooraan in de procedure body (arrays met dezelfde bound pair waarden als de overeenkomstige actuele array had bij zijn declaratie). Hierin wordt de waarde van de actual parameter gecopieerd. Wanneer we nu de waarde van de formal parameter veranderen, wordt alleen deze "copie" veranderd; de actual parameter wordt niet beinvloed. Evenzo, wanneer de waarde van de actual parameter tijdens uitvoering van de procedure verandert, blijft de waarde van de formal parameter hetzelfde.
We kunnen nu de volgende gevallen onderscheiden:

— De formal parameter is integer , real of Boolean gespecificeerd en bij uitvoering van de procedure wordt aan de formal parameter een waarde toegekend, of de parameter is integer array , real array of Boolean array gespecificeerd en nu wordt aan een of meer van de elementen een waarde toegekend.
Is het nu de bedoeling dat de actual parameter hierdoor van waarde verandert (dat mag in het geval integer , real of Boolean dus alleen als de actual parameter een variable is), dan gebruiken we call by name (d.w.z. we gebruiken call by value niet). Een dergelijke formal parameter kunnen we "output-parameter" noemen.
Is een dergelijke waardeverandering daarentegen niet gewenst, of zelfs ongeoorloofd (nl. als de actual parameter wel een expression maar geen variable is) dan moeten we call by value gebruiken.

— Tengevolge van side effects oftewel neveneffecten van de procedure kan de waarde van de actual parameter veranderen. Dit neveneffect is dan een verandering van de waarde van een niet-locale variable tijdens uitvoering van de procedure body. Is het de bedoeling dat dan ook de formal parameter van waarde verandert, dan gebruiken we call by name, anders call by value.
Een zeer belangrijk geval is Jensen's device geheten. Hierbij vindt het neveneffect plaats door middel van een output-parameter. Daarmee correspondeert dan een actuele variable die in de uitdrukking voor een andere actual parameter voorkomt. Deze laatste is niet value gespecificeerd; door in de procedure een waarde toe te kennen aan de outputparameter verandert de waarde van deze parameter mee. Is de actual parameter een subscripted variable dan kan zelfs zijn "identiteit" veranderen. (zie verder paragraaf 8)

— De bepaling van de waarde van de actual parameter kan neveneffecten met zich meebrengen (de actual parameter is dan de function designator of bevat de function designator van een functie-procedure met neveneffecten).
Zijn deze effecten gewenst (iedere keer dat de betreffende parameter geevalueerd wordt), dan call by name, anders call by value.
— De bepaling van de waarde van de actual parameter vergt een niet verwaarloosbare rekentijd, en veroorzaakt geen gewenste neveneffecten. Uit overwegingen van efficiency is dan call by value op zijn plaats.
— De formal parameter is integer of integer array gespecificeerd, en het is voor de juiste werking van de procedure noodzakelijk dat de parameter van type integer is. Wanneer nu niet met redelijke zekerheid gesteld kan worden dat de corresponderende actual parameter inderdaad het type integer zal hebben dan zal call by value er voor zorgen dat bij de aanroep tijdens het copieren van de waarde van de actual parameter zonodig een automatische overgang van real naar integer plaats vindt. Ook in het omgekeerde geval (type real vereist voor correcte werking, actual parameter mogelijk integer) kan met vrucht call by value gebruikt worden.

Verder zij nog vermeld dat we bij arrays call by value slechts gebruiken als daar een goede reden voor is, aangezien dan tegelijkertijd zowel de waarde van de actuele array als de waarde van de (mogelijk gewijzigde) copie in de formele array onthouden moeten worden. Dit zou (zonder goede reden toegepast) een overbodige belasting van de geheugenruimte van de uitvoerende computer betekenen.
Daarentegen geldt voor de andere parameters (voor label in mindere mate) waarvoor call by value is toegestaan, dat we call by name slechts gebruiken als er een goede reden voor bestaat. In de gevallen waar call by value en call by name verschillende resultaten zouden opleveren, is natuurlijk slechts een alternatief in overeenstemming met onze bedoeling; maar ook in gevallen waar het resultaat hetzelfde blijft kan call by value een aanzienlijke besparing in de rekentijd geven.

7 Voorbeelden

Ga in de volgende voorbeelden na om welke reden(en) call by value of call by name is gekozen.

```
procedure POLtoCART (x, y, r, fi); value r, fi; real x, y, r, fi;
comment POLtoCART transformeert de poolcoordinaten (r, fi) tot cartesiaanse
                coordinaten (x, y);
begin x:= r × cos (fi); y:= r × sin (fi) end
```

Aanroep bijv. POL to CART (X[k], Y[k], mod ↑ (1 / k), arg / k)

```
procedure TRANSFORM (i, n, Vi); value n; integer i, n; real Vi;
for i:= 1 step 1 until n do Vi:= 1 - 1 / Vi
```

Aanroep bijv. TRANSFORM (j, m, C[j, j])

```
procedure VUL (A, m, n, input); value m, n; integer m, n; real input; array A;
begin    integer i, j;
         for i:= 1 step 1 until m do for j:= 1 step 1 until n do A[i, j]:= input
end
```

Aanroep bijv. VUL (A, 1, read, read) of VUL (A, p, q, 1 / (p + q))

Het laatste geval kan voor grote p en q efficienter:

```
begin    real som1;
         som1:= 1 / (p + q); VUL (A, p, q, som1)
end
```

Dit zou overbodig zijn als "input" value gespecificeerd was. Ga na hoe dan echter de betekenis van VUL (A, 1, read, read) volkomen verandert (aangenomen dat read de function designator is van een functie-procedure die de waarde aanneemt van "het volgende getal op de getallenband" en die als formidabel neveneffect het opschuiven van de band heeft).

```
real procedure DETSOL2 (A, b, n); value A; integer n; real array A, b;
comment DETSOL2:= determinant van array A[1 : n, 1 : n], en array b[1 : n]
                wordt vervangen door de oplossing x van het lineaire stelsel A × x = b;
DETSOL2:= DETSOL (A, b, n)
```

Wanneer men over een procedure DETSOL beschikt die hetzelfde presteert als in het comment van DETSOL2 beschreven staat, maar die zijn tussenresultaten in de array A bewaart, dan kan men beter DETSOL2 gebruiken als de actual parameter voor A ongerept moet blijven, of als deze integer array gedeclareerd is.

8 Jensen's device

Een voorbeeld van het gebruik van Jensen's device is:

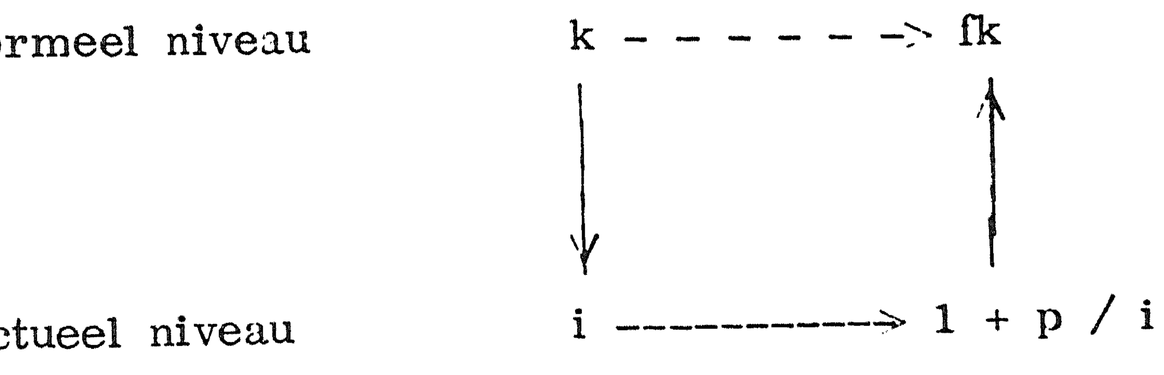
```
real procedure PI (k, a, b, fk); value a, b; integer k, a, b; real fk;
begin    real pi;
         pi:= 1;
         for k:= a step 1 until b do pi:= pi × fk;
         PI:= pi
end
```

Mathematisch uitgedrukt: $PI = \prod_{k=a}^{b} fk$

Bij de aanroep: PI (i, 1, q, 1 + p / i) luidt de procedure body na het invullen van de waarde van de actual parameters "1" en "q", en de tekst van de actual parameters "i" en "1 + p / i":

```
begin    integer a, b; real pi;
         a:= 1; b:= q;
         pi:= 1;
         for i:= a step 1 until b do pi:= pi × (1 + p / i);
         PI:= pi
end
```

Als de formele "k" verandert, verandert de actuele "i", daardoor de actuele "1 + p / i" en daarmee de formele "fk". Schematisch voorgesteld:



Ga na wat in de volgende gevallen wordt berekend:

1. PI (i, 1, q, 1 + p / i)
2. PI (i, 1, ordeA, A[i, i])
3. PI (n, 1, n, n)

9 Recursieve procedures (RR 5.4.4)

Het is uiteraard toegestaan in een procedure body een andere procedure aan te roepen. Het is in ALGOL eveneens toegestaan binnen de body van een procedure deze procedure zelf aan te roepen. Een procedure die zichzelf aanroept heet een recursieve procedure. Ook een niet recursief geschreven procedure kan recursief gebruikt worden, nl. wanneer bij een procedure de bepaling van de waarde van een actual parameter die correspondeert met een call-by-name formal parameter, een aanroep van de procedure ten gevolge heeft, zoals in PI (i, 1, 3, PI (j, 1, 3, M[i, j])). In vrij veel implementaties is het recursief gebruik van procedures niet mogelijk. Zowel in het X1- als in het X8-ALGOL-systeem is het gebruik van recursieve procedures aan geen restrictie gebonden.

Een bekend voorbeeld van een recursieve procedure is dat van de functie-procedure voor het berekenen van de faculteit van een getal:

```
integer procedure fac (n); value n; integer n;
fac:= if n = 0 then 1 else fac (n - 1) × n
```

De waarde van fac (2) wordt ongeveer als volgt berekend:

```
   fac (2)
=  | n:= 2
   | fac:= if n = 0 then 1 else fac (n - 1) × n
   |     = if 2 = 0 then 1 else fac (n - 1) × n
   |     = if false then 1 else fac (n - 1) × n
   |     = fac (n - 1) × n
   |     = | n':= n - 1 = 2 - 1 = 1
   |       | fac:= if n' = 0 then 1 else fac (n' - 1) × n'
   |       |     = if 1 = 0 then 1 else fac (n' - 1) × n'
   |       |     = if false then 1 else fac (n' - 1) × n'
   |       |     = fac (n' - 1) × n'
   |       |     = | n":= n' - 1 = 1 - 1 = 0
   |       |       | fac:= if n" = 0 then 1 else fac (n" - 1) × n"
   |       |       |     = if 0 = 0 then 1 else fac (n" - 1) × n"
   |       |       |     = if true then 1 else fac (n" - 1) × n"
   |       |       |     = 1
   |       |     = 1 × n' = 1 × 1
   |       |     = 1
   |     = 1 × n = 1 × 2
   |     = 2
=  2
```

Dit is geen gelukkig voorbeeld, in zoverre dat het geen navolging verdient:

1. omdat hetzelfde werk sneller en met minder geheugengebruik kan worden gedaan door een niet-recursieve procedure.
2. omdat voor n bijv. groter dan 15 het resultaat van fac (n) wel groter zal zijn dan de grootste representeerbare integer, zodat de procedure beter als een real procedure gedeclareerd kan worden. (Voor het X1-ALGOL-systeem geldt dit bezwaar niet omdat de uitkomst van een integer procedure, als deze te groot is, daar automatisch als een real wordt voorgesteld; in het X8-ALGOL-systeem echter is fac (12) = 479001600 groter dan de grootste waarde die aan een integer variable of procedure identifier kan worden toegekend, nl. 67108863, waardoor de uitvoering van het programma voortijdig wordt beeindigd)
3. omdat de uitvoering van de procedure nooit zal eindigen indien het argument door onvoorziene omstandigheden negatief is. Beter is bijv. te schrijven:
   if n < 0 then go to ALARM else fac:= .....

Voorbeelden van het verantwoord gebruik van recursieve procedures zijn meestal ingewikkeld en liggen doorgaans niet op numeriek terrein. (Een goed voorbeeld is echter de locale procedure I in de procedure QAD, zie AP 251)

Een niet-numerieke toepassing:

Gevraagd wordt welke velden van een schaakbord in minder dan bijv. 4 zetten bezet kunnen worden door een paard dat op het veld a1 staat.

We denken ons een integer array VELD[1 : 8, 1 : 8] gedeclareerd, en vervolgens de recursieve procedure:

```
procedure PAARD (x, y, n); value x, y, n; integer x, y, n;
if x < 1 then else if x > 8 then else
if y < 1 then else if y > 8 then else
if VELD[x, y] < n then
begin    integer dx, dy;
         VELD[x, y]:= n;
         for dx:= -2, -1, 1, 2 do
         for dy:= if abs (dx) = 1 then 2 else 1, - dy do
             PAARD (x + dx, y + dy, n - 1)
end
```

Alle velden van VELD moeten worden gevuld met de waarde 0.

Door de aanroep: PAARD (1, 1, 4) wordt dan een positieve integer ingevuld op die velden, die binnen 4 zetten vanuit VELD[1, 1], d.w.z. a1, door een paard bereikbaar zijn.

Opgave. Voer de statement PAARD (1, 8, 2) uit met een schema als boven is opgesteld voor de berekening van fac (2).
Opmerking. Voor het X1-ALGOL-systeem duurt de uitvoering van de statement PAARD (1, 1, 4) 20 seconden. Wanneer x, y en n niet in de value part vermeld worden vergt de aanroep 69 seconden, dat is dus ruim 3 × zoveel rekentijd. We zien dus hoezeer efficiency-overwegingen hier call by value gebieden.

Een recursieve procedure gebruiken we alleen wanneer het formuleren van een niet-recursieve oplossing zeer ingewikkeld wordt. Bij het schrijven van een recursieve procedure is het gewenst dat men goed nagaat of er aan de recursie een eind komt.

10 Switches en designational expressions (RR 3.5, 4.3.5, 5.3)

Door middel van een switch designator kan men, aan de hand van een subscript expression, uit een lijst designational expressions er een kiezen.
Als gedeclareerd is:
switch S:= L, if p then L else A, S[a]
dan heeft de statement "go to S[n]" voor waarden van de integer variable n binnen het bereik $1 \leq n \leq 3$ dezelfde betekenis als:
go to if n = 1 then L else if n = 2 then (if p then L else A) else S[a]
Voor waarden van n buiten dat bereik is de waarde van de switch designator S[n] ongedefinieerd. Volgens RR 4.3.5 moet dan de go to statement equivalent zijn met een dummy statement, d.w.z. het programma moet achter de go to statement worden voortgezet, alsof er nooit geprobeerd was de waarde van een switch designator te bepalen. Wanneer we bedenken dat de n in S[n] i.p.v. een variable ook best de function designator zou kunnen zijn van een functie-procedure met declaratie:
integer procedure n;
begin PRINTTEXT (‹Dit mag niet op papier verschijnen›); n:= 0 end
waarin PRINTTEXT een code-procedure is die de als argument gegeven string uitprint (en "‹" en "›" de hardware representation voor string quotes zijn),
dan zien we dat het volledig implementeren van deze regel tot grote complicaties kan leiden. Vermoedelijk is dit dan ook in geen enkele implementatie van ALGOL volbracht. Voor het X1- en het X8-ALGOL-systeem is vereist dat de waarde van een switch designator, wanneer deze bepaald wordt, ook gedefinieerd is.

Een eenvoudige maar nuttige toepassing van het switch-mechanisme is:

```
procedure Letter (k); value k; integer k;
begin switch ALFABET:= A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z;
      procedure t (letter); string letter;
      begin PRINTTEXT (letter); go to EXIT end;
      go to if k < 1 ∨ k > 26 then EXIT else ALFABET[k];
      A: t (‹a›); B: t (‹b›); C: t (‹c›); D: t (‹d›); E: t (‹e›); F: t (‹f›);
      G: t (‹g›); H: t (‹h›); I:  t (‹i›); J: t (‹j›); K: t (‹k›); L: t (‹l›);
      M: t(‹m›); N: t (‹n›); O: t (‹o›); P: t (‹p›); Q: t (‹q›); R: t (‹r›);
      S:  t (‹s›); T: t (‹t›); U: t (‹u›); V: t (‹v›); W: t (‹w›); X: t (‹x›);
      Y: t (‹y›); Z: t (‹z›);
EXIT:
end
```

De aanroep "Letter (7)" wordt als volgt uitgevoerd:

```
   go to if 7 < 1 ∨ 7 > 26 then EXIT else ALFABET[7]
= go to ALFABET[7]
= go to G
= G: t (‹g›)
= begin PRINTTEXT (‹g›); go to EXIT end
```

Op papier verschijnt dus de letter "g".

De evaluatie van een switch designator, zelf een designational expression, levert een element van een switch list, dus weer een designational expression, die behalve een label ook een conditionele designational expression of weer een switch designator kan zijn. In de laatste twee gevallen moet de evaluatie worden voortgezet totdat een label wordt gevonden, die dan de waarde van de oorspronkelijke switch designator is. Het volgende voorbeeld toont aan hoe deze voortgezette evaluatie het essentiele deel van een programma kan uitmaken.
Gegeven is een reeks van ALGOL-symbolen, waaraan een code is toegekend volgens de volgende tabel:

code:	symbool:
1	,
2	+ −
3	. .
4	10
5	0 1 2 3 4 5 6 7 8 9

We willen nu een procedure schrijven die controleert of de reeks ALGOL-symbolen een rij correcte ALGOL-getallen is, waarbij de komma als scheider tussen twee getallen kan dienen.

Correct is bijv.: $+2-3,+_{10}-0,,007,5.3+0$

Incorrect daarentegen: $+2_{10}-.0,+317.,++3$

We veronderstellen dat in een integer array CODEvanSYMBOOL[1 : n] de codes van de achtereenvolgende ALGOL-symbolen staan.

De procedure declaratie luidt:

```
Boolean procedure Getalreeks (k, lengte, SYMB); value lengte; integer k, lengte, SYMB;
begin   comment symbool =     ,       + -       .     10      0 t/m 9;
        switch Teken    := fout      , fout       , Punt[s], Tien[s], Integer[s], fout  ;
        switch Integer  := Integer[s], Teken[s]   , Punt[s], Tien[s], Integer[s], einde;
        switch Punt     := fout      , fout       , fout   , fout   , Fractie[s], fout  ;
        switch Fractie  := Integer[s], Teken[s]   , fout   , Tien[s], Fractie[s], einde;
        switch Tien     := fout      , Tekenexp[s], fout   , fout   , Exp[s]    , fout  ;
        switch Tekenexp:= fout       , fout       , fout   , fout   , Exp[s]    , fout  ;
        switch Exp      := Integer[s], Teken[s]   , fout   , fout   , Exp[s]    , einde;
        integer procedure s;
        begin k:= k + 1; s:= if k > lengte then 6 else SYMB end;
              Getalreeks:= true; k:= 0; go to Integer[1];
fout:   Getalreeks:= false;
einde:
end
```

De aanroep is nu: Getalreeks (i, n, CODEvanSYMBOOL[i]).

Opgave: Voer de procedure uit voor de bovenstaande symboolreeksen.
Welke betekenis kan aan de waarde van de output-parameter i na afloop worden toegekend, indien de waarde van de function designator false is?

1.1. Boolean expressions en logische variabelen (RR 3.4)

Een Boolean expression is alles wat in een if clause tussen if en then mag staan. Juist als van een arithmetic expression de evaluatie wordt voortgezet totdat een numerieke waarde is gevonden, en de evaluatie van een designational expression uiteindelijk een label moet leveren, zo wordt ook een Boolean expression geevalueerd totdat een der logische waarden true of false is verkregen. De waarde bijv. van de Boolean expression "$2 \leq 0$" is false, die van "$3 \times 3 + 4 \times 4 = 5 \times 5$" is true.

Met behulp van de arithmetic operators $+$, $-$, $\times$, $/$, $\underline{:}$ en $\uparrow$ kan worden aangegeven hoe uit een of twee getallen een derde getal moet worden berekend, en de relational operators $<$, $\leq$, $=$, $\geq$, $>$ en $\neq$ geven aan hoe bij twee getallen een logische waarde moet worden berekend. Op soortgelijke wijze is het mogelijk bij een of twee logische waarden een derde logische waarde te laten berekenen, nl. met behulp van een der logical operators $\neg$, $\wedge$, $\vee$, $\underline{\neg}$ en $\underline{=}$, en wel volgens de regels die in RR 3.4.5 gegeven worden. Evenals voor de arithmetical operators gelden voor de logical operators prioriteitsregels. De volgorde is: $\neg$, $\wedge$, $\vee$, $\underline{\neg}$, $\underline{=}$.

We geven als voorbeeld de evaluatie van "true $\underline{\neg}$ $\neg$ $3 \times 3 = 9 \vee 1 \geq 1$" :

true $\underline{\neg}$ $\neg$ $9 = 9 \vee 1 \geq 1$
true $\underline{\neg}$ $\neg$ true $\vee$ $1 \geq 1$
true $\underline{\neg}$ false $\vee$ $1 \geq 1$
true $\underline{\neg}$ false $\vee$ true
true $\underline{\neg}$ true
true

Opgave. Ga met behulp van RR 3.3.1 en 3.4.7 na, dat nooit twee arithmetical operators naast elkaar mogen staan, maar dat wel de logical operator $\neg$ door iedere andere logical operator mag worden voorafgegaan.

Een berekende logische waarde kan onthouden worden door toekenning aan een logische variabele, die dan Boolean gedeclareerd moet zijn. Evenzo zijn er Boolean array en Boolean procedure.

Het volgende is een voorbeeld van het gebruik van een Boolean array. In de array PRIEM wordt met de "zeef van Eratosthenes" op de k-de plaats de waarde ingevuld van de uitspraak "k is een priemgetal".

```
begin    integer k, p;
         Boolean array PRIEM[1 : 1600];
         PRIEM[1]:= false;
         for k:= 2 step 1 until 1600 do PRIEM[k]:= true;
         for k:= 2 step 1 until 40 do
         if PRIEM[k] then
         for p:= k × k step k until 1600 do PRIEM[p]:= false;
              Rest van het programma
end
```

Als A en B Boolean expressions voorstellen, dan kunnen we i.p.v. ¬A evengoed schrijven: if A then false else true, en verder (mits de evaluatie van B geen neveneffecten veroorzaakt):

if A then B else false i.p.v. A ∧ B
if A then true else B i.p.v. A ∨ B
if A then B else true i.p.v. A ⊐ B

Indien de berekening van B veel tijd vereist, kan door een dergelijke schrijfwijze soms een besparing in de rekentijd verkregen worden. Belangrijker is, dat het voor een bepaalde waarde van A ongewenst kan zijn dat B geevalueerd wordt. Werken we bijvoorbeeld met een implementatie waarvan de aritmetiek bij deling door 0 een niet gedefinieerd resultaat levert, dan zetten we i.p.v. "a / b < eps1" wellicht liever iets als "b ≠ 0 ∧ a / b < eps1". Wanneer deling door 0 zelfs als een fout beschouwd wordt, dan moeten we hiervoor wel schrijven:
"if b = 0 then false else a / b < eps1".

12 own (RR 5, 5.2.5)

Als tijdens de uitvoering van een programma een block wordt binnengegaan, zijn de waarden van de daar gedeclareerde simple variables en arrays nog ongedefinieerd. Dit geldt ook indien het block reeds eerder is doorlopen en deze grootheden toen wel een waarde hebben gekregen. Beginnen hun declaraties echter met het symbool own, dan blijft hun waarde behouden. Voor simple variables en arrays waarvan de bound pair expressies constanten zijn, kan men dit opvatten alsof zij in het buitenste block gedeclareerd zijn (waarbij de identifier echter alleen betekenis heeft in het block waarin hij own gedeclareerd is). Indien het block waarbij de own gedeclareerde simple variable of array behoort in een procedure body staat, is het mogelijk nog andere interpretaties aan own te geven, met name als het een procedure betreft die recursief gebruikt wordt.

In de zogenaamde dynamische opvatting, waarmee in de MC-II-ALGOL-vertaler geexperimenteerd is, bestaat voor ieder recursie-niveau een eigen set own simple variables en arrays. Volgens een andere dynamische interpretatie bestaat een dergelijke set voor elke plaats in het programma waar de procedure wordt aangeroepen, alsof de procedure body steeds voor zo'n procedure aanroep gesubstitueerd wordt (body replacement). Wij beschouwen hier echter slechts de statische interpretatie, waarbij van elke own grootheid ten hoogste 1 waarde bekend is.
Men kan zich afvragen in welke omstandigheden het zinvol is een grootheid own te declareren i.p.v. deze gewoon in het buitenste of althans een omvattend block te declareren.
Voor simple variables en arrays met vaste bound pair waarden heeft dit alleen zin in een procedure, nl. voor die grootheden waarvan alleen binnen de procedure de waarde bepaald of gevraagd wordt, waarbij toch deze waarde van de ene aanroep tot de andere behouden moet blijven: in dit geval zou declaratie in een buitenblock tot gevolg hebben dat de procedure niet als een zelfstandige eenheid beschouwd kan worden. Een grootheid als wagenstand (hoeveel symbolen staan er al op de regel) en case-definitie (hoofdletters of kleine letters) in een procedure die aangeboden symbolen omzet in ponsingen voor een Flexowriterband, kunnen daar het beste own gedeclareerd worden. We moeten er wel op een of andere wijze voor zorgen dat die grootheden eerder een waarde krijgen dan er naar hun waarde gevraagd wordt.
Dynamische own arrays, met variabele bound pair waarden dus, kunnen gebruikt worden om geheugenruimte te besparen in het geval dat de subscript expressions in principe over een groot bereik kunnen varieren, terwijl toch steeds over een beperkt bereik tegelijk de waarden onthouden behoeven te worden. (Bijv. een cyclisch magazijn of een zogenaamde stapel) In het X8-ALGOL-systeem zijn op het ogenblik own arrays nog niet geimplementeerd. Bij de meeste implementaties, waaronder het X1-ALGOL-systeem, zijn dynamische own arrays niet toegestaan. Dit is wel het geval bij de MC-II-vertaler.

13 for statements (RR 4.6)

In een for statement moet de statement achter do worden uitgevoerd met die waarden van de lopende variabele, die door de for list elements worden bepaald.
Er zijn drie soorten for list elements:

1. Arithmetic expression.
2. Step-until-element. In veel implementaties wijkt de werking op onderdelen af van de werking beschreven in RR 4.6.4.2. Om die reden vermijden we gevallen waarin bijv. de evaluatie van de stapwaarde neveneffecten geeft.
3. While-element.

Deze drie mogen door elkaar in een for list gebruikt worden.

Enige voorbeelden:

1. for z:= - arctan (b / a), sin (z), 1 - z ↑ 2 do print (z)

2. Invullen van een codetabel:

```
wijzer:= -1;
for code:= 32, 1, 2, 19, 4, 21, 22, 7, 8, 25 do
begin    VERTAAL[code]:= wijzer:= wijzer + 1;
         CODE[wijzer]:= code
end
```

3. j moet de getallen 1 t/m n doorlopen, maar het getal i (waarbij $1 \leq i \leq n$) overslaan.

   for j:= 1 step 1 until i - 1, i + 1 step 1 until n do ...

4. c moet de cijfers 0 t/m 9 cyclisch doorlopen, te beginnen bij cijfer c0.

   for c:= c0 step 1 until 9, 0 step 1 until c0 - 1 do ...

5. Oplossen van x = f (x) met x0 als beginschatting, waarbij $|f'(x)| \ll 1$.

   for z:= x0, f (x) while abs (z - x) > $_{10}$-5 × abs (z), z do x:= z

14 Hoe wordt een ALGOL-programma uitgevoerd?

Men zou kunnen denken dat een rekenmachine bij de uitvoering van een ALGOL-programma de in zijn geheugen opgeslagen ALGOL-tekst doorloopt, en bij het vinden van een identifier steeds weer bepaalt welke entiteit bedoeld wordt, etc. Dit is niet erg efficient: veel dat eens en voor al gedaan kan worden, gebeurt bij deze aanpak steeds opnieuw, hetgeen onnodig veel tijd kost.
In de praktijk gaat men dan ook anders te werk: men gebruikt een compiler of vertaler, dat wil zeggen een programma dat als input een ALGOL-programma heeft en als output een programma in de code van de machine, het zgn. object-programma. Dit objectprogramma wordt dan uitgevoerd.
Men kan voor een machine verschillende vertalers construeren: de ene zal sneller werken en minder geheugenruimte beslaan, de andere zal een efficienter objectprogramma leveren.
Door verschillende vertalers zal bijvoorbeeld de statement "x:= a × b + c / d" verschillend vertaald kunnen worden. In ALGOL beschreven zijn dan twee mogelijke vertalingen ervan (waarbij iedere statement een machinecode-opdracht is):

1.
```
begin real F, S1, S2, S3;
        F:= a; S1:= F;
        F:= b; S2:= F;
        F:= S1; F:= F × S2; S1:= F;
        F:= c; S2:= F;
        F:= d; S3:= F;
        F:= S2; F:= F / S3; S2:= F;
        F:= S1; F:= F + S2; S1:= F;
        F:= S1; x:= F
end
```

2.
```
begin real F, S1;
        F:= a;
        F:= F × b;
        S1:= F;
        F:= c;
        F:= F / d;
        F:= F + S1;
        x:= F
end
```

In het objectprogramma zullen steeds stereotiepe opeenvolgingen van machine-code-opdrachten voorkomen, corresponderend met een bepaalde "actie-eenheid", bijvoorbeeld het ALGOL-symbool "↑". Om nu het objectprogramma kort te houden, is het nuttig dat de vertaler in plaats van zo'n opeenvolging een opdracht schrijft, die een (parameterloze) machinecodeprocedure aanroept, die dan bestaat

uit de stereotiepe opeenvolging van opdrachten. Met "abs" in de ALGOL-tekst zou bijvoorbeeld een aanroep kunnen corresponderen van de (hier in ALGOL beschreven) machinecodeprocedure:

```
procedure ABS;
begin     F:= Si;
          if F ≥ 0 then go to L;
          F:= -F;
  L:      Si:= F
end
```

Het arsenaal van machinecodeprocedures dat het objectprogramma completeert heet het complex. Het objectprogramma uit het voorbeeld kan met behulp van een complex als volgt gerealiseerd worden:

```
begin     integer i; real F; real array S[1 : 100];
          comment complex;
          procedure STAPEL; begin S[i]:= F; i:= i + 1 end;
          procedure ONTSTAPEL; begin i:= i - 1; F:= S[i] end;
          procedure MUL; begin ONTSTAPEL; F:= F × S[i - 1]; S[i - 1]:= F end;
          procedure DIV; begin i:= i - 1; ONTSTAPEL; F:= F / S[i + 1]; STAPEL end;
          procedure ADD; begin ONTSTAPEL; F:= F + S[i - 1]; S[i - 1]:= F end;
          comment objectprogramma; i:= 1;
          begin     real x, a, b, c, d;
                    . . . . . .
                    F:= a; STAPEL;
                    F:= b; STAPEL;
                    MUL;
                    F:= c; STAPEL;
                    F:= d; STAPEL;
                    DIV;
                    ADD;
                    ONTSTAPEL; x:= F
          end
end
```

De complexprocedures zullen in het algemeen corresponderen met eenvoudige actie-eenheden. Sommige machines hebben zodanig aan ALGOL aangepaste opdrachten, dat eenheden als "MUL", "DIV" en "ADD" uit 1 opdracht bestaan, zodat ze niet meer in het complex behoeven te worden opgenomen: in plaats van de aanroep van de procedure zet men dan liever de opdracht zelf. Deze machines kunnen een ALGOL-programma sneller en met minder geheugengebruik uitvoeren dan andere machines. Zo duurt op de X1 een ALGOL-programma 10 tot 20 maal zo lang als een zorgvuldig met de hand geschreven equivalent machinecodeprogramma. Bij de X8, met zijn meer op ALGOL toegespitste opdrachten, bedraagt deze factor 2 tot 3.
Waarom werkt nu bij een vertaler het objectprogramma langzamer dan noodzakelijk? In het algemeen omdat ook in de eenvoudigste gevallen rekening gehouden wordt met de grootste complexiteit die kan optreden:

-- Komt in een arithmetic expression een formal parameter voor, dan kan evaluatie daarvan een recursieve aanroep van de procedure met zich brengen, maar in verreweg de meeste gevallen zal dat niet zo zijn.
-- Bij een for statement kan tijdens uitvoering de lopende variabele van identiteit veranderen, maar doet dat meestal niet.
-- Een go to statement kan buiten het block voeren, maar doet dat vaak niet.

De compiler geeft dus steeds een algemene vertaling, en het complex bevat de meest algemene routines.

15 Tijdsduur en plaatsruimte van het programma

Hoelang uitvoering van een ALGOL-programma duurt, en hoeveel ruimte het programma in zal nemen, hangt sterk af van het gebruikte vertalersysteem. We zullen hier dus niet een volledige handleiding geven voor het berekenen van de benodigde plaats en tijd voor het gehele programma.
Wel is het nuttig, te kunnen bepalen voor korte stukken ALGOL-programma, wat uit een oogpunt van efficiency de beste formulering is.
In ruwe benadering kan men de tijdsduur resp. lengte van een statement schatten door de tijdsduur resp. ruimte van delimiters en identifiers op te tellen. Voor de X1 en de X8 bedragen plaats en ruimte:

		X1 systeem			X8 systeem		
integers	0, i	2 plaatsen	.5	msec	1 plaats	5	mmsec
	:=	1 plaats	1	msec	2 plaatsen	6	mmsec
	+, −	1 plaats	.5	msec	1 plaats	7	mmsec
	×	1 plaats	1	msec	1 plaats	40	mmsec
	:	1 plaats	1	msec	1 plaats	190	mmsec
reals	3.14, r	2 plaatsen	1	msec	1 plaats	7.5	mmsec
	:=	1 plaats	1.5	msec	0 plaatsen	0	mmsec
	+, −	1 plaats	1	msec	1 plaats	10	mmsec
	×	1 plaats	3	msec	1 plaats	40	mmsec
	/	1 plaats	3	msec	1 plaats	65	mmsec
	↑ 2	3 plaatsen	10	msec	3 plaatsen	285	mmsec
Booleans	true, b	2 plaatsen	.5	msec	1 plaats	5	mmsec
	:=	1 plaats	1	msec	1 plaats	4	mmsec
	¬	1 plaats	.5	msec	1 plaats	5	mmsec
∧, ∨, ≡, ¬		1 plaats	1	msec	4 plaatsen	15	mmsec
subscripts	[i1]	3 plaatsen	3	msec	3 plaatsen	50	mmsec
	[i1, i2]	5 plaatsen	5	msec	5 plaatsen	135	mmsec
	[i1, i2, i3]	7 plaatsen	7	msec	7 plaatsen	220	mmsec
					(Boolean arrays	100	mmsec extra)
if...then...else		3 plaatsen	1	msec	2 plaatsen	6	mmsec
for i:= 1 step 1 until n do							
(tijd per slag)		22 plaatsen	7.5	msec	21 plaatsen	80	mmsec
go to L		1 plaats	0	msec	1 plaats	5	mmsec

blockingang + blockverlating (zonder arraydeclaratie)

4 msec 45 mmsec

voorbeeld Loont het de moeite voor het X8 systeem de for statement uit te schrijven?

"for i:= 1 step 1 until 1000 do" vergt 80 msec.

"i:= 1; BB: if i = 1000 then else begin i:= i + 1; go to BB end" vergt 67 msec.

In het algemeen loont het niet de moeite, voor zo'n kleine tijdwinst de overzichtelijkheid van het programma op te offeren.

<u>voorbeeld</u> Op een bepaald punt in het programma moet vaak worden uitgerekend, voor verschillende waarden van i, j en k:

```
s1:= s1 + A[i, j, k];                    1+1.5+1+2.5+1+7= 14 msec
s2:= s2 + A[i, j, k] × A[i, j, k]        14+3+1+7= 25 msec
```

Op de X1 duurt dit per keer ongeveer 39 msec. Schrijven we echter:

```
a:= A[i, j, k];                          1+1.5+1+7= 10.5 msec
s1:= s1 + a;                             1+1.5+1+2.5+1= 7 msec
s2:= s2 + a × a                          7+3+1= 11 msec
```

dan duurt de berekening 28.5 msec.
Voor de X8 zijn deze tijden .77 en .35 msec. Blijkbaar loont het bij sommige machines in sommige gevallen de moeite om de waarden van een array-element, waarnaar vele malen gevraagd zal worden, aan een hulpvariabele toe te kennen.

Opvallend is, dat de X1 niet alleen veel langzamer is dan de X8, maar dat de tijden voor de verschillende X1 operaties niet veel van elkaar verschillen, terwijl bij de X8 wel degelijk aanmerkelijk verschil in tijd is tussen de operatoren onderling.
Bij de X1 neemt het eigenlijke rekenwerk minder tijd in beslag dan de daarvoor vereiste administratie. Bij de X8 is de administratie meestal veel geringer (Bij subscripted variables, waar de administratie vrij uitgebreid is, is de snelheidsverhouding van de twee machines dan ook duidelijk minder groot). Het X1 en het X8 systeem vormen hierin twee extremen, andere systemen bevinden zich daartussen, of niet ver daarvandaan.

Uit bovenstaande cijfers blijkt dat overigens equivalente schrijfwijzen vaak een verschillende duur van uitvoering hebben, bijvoorbeeld "a × a" kan equivalent zijn met "a ↑ 2" maar de tijden hoeven niet gelijk te zijn (X8: 55 mmsec resp 292.5 mmsec).
Blijkbaar is het mogelijk, door het kiezen van een geschikte schrijfwijze een programma enigzins te versnellen. Hoewel verschillende systemen duidelijk verschillende eigenschappen hebben, is het toch mogelijk hiervoor enkele suggesties te geven, die een grotendeels algemene geldigheid zullen hebben. Een gevolg van dit versnellen kan zijn dat de constructie van het programma minder overzichtelijk wordt. Men moet zich er steeds rekenschap van geven of dit verlies aan elegantie en het extra benodigde denkwerk opwegen tegen de te behalen tijdwinst. Besteed dus de meeste energie aan het optimaliseren van die stukken programma die het meest worden doorlopen, bijvoorbeeld binnen (geneste) for statements.

suggestie Laat geen werk dubbel doen.

Wordt naar de waarde van een bepaald array element enkele malen gevraagd, ken hem dan toe aan een hulpvariabele. Hebben enige expressies subexpressies gemeen, ken die dan toe aan een hulpvariabele. Maak bij procedure declaration, daarvoor in aanmerking komende parameters value.

suggestie Binnen geneste for statements: reken een van een of meer lopende variabelen afhangend tussenresultaat uit zo gauw dat kan. Stel de berekening niet uit tot in de "kern" van de for statements.

suggestie Bij de meeste ALGOL systemen staan codeprocedures ter beschikking, die aanmerkelijk sneller werken dan de corresponderende ALGOL procedures. Gebruik die procedures dan ook.

suggestie Vaak geeft een eenvoudige mathematische herschrijving een efficienter proces.

suggestie Maak niet onnodig een vaak doorlopen compound statement tot block, door er een declaratie te laten plaatsvinden. Een compound statement binnen een for statement is niet noodzakelijk een block. Wel is de body van een procedure in de meeste systemen een block. Schrijf desnoods een procedure aanroep in een veeldoorlopen traject uit.

voorbeeld Berekend moet worden "(exp (x) - exp (-x)) / (exp (x) + exp (-x))"
Ingewikkelde standaardroutines als "sin", "arctan" en "exp" duren erg lang. We kunnen de uitdrukking herschrijven tot
"1 - 2 / (exp (2 × x) + 1)", welke uitdrukking driemaal zo snel berekend wordt.

voorbeeld Stel, in het gebruikte systeem hebben we de beschikking over de standaardprocedures SOM en INTEGRAAL, met voor de hand liggende betekenis.
"SOM (i, 1, n, SOM (j, 1, m, INTEGRAAL (x, x0, x1,
S(i) × P (j) × Q (i, j) × R (i, j, x), tolerantie)))"
is beter te berekenen als
"SOM (i, 1, n, S (i) × SOM (j, 1, m, P (j) × Q (i, j) ×
INTEGRAAL (x, x0, x1, R (i, j, x), tolerantie)))".
Bovendien loont het bij grote n waarschijnlijk de moeite de waarden van P in een hulparray op te slaan, omdat ze anders voor iedere waarde van i opnieuw berekend moeten worden.

Over geheugengebruik valt algemeen het volgende te zeggen:
Tijdens uitvoering van het programma moet het geheugen bevatten

-- het complex
-- het objectprogramma
-- de ruimte voor de variables
-- de werkruimte voor de administratie.

De lengte van het complex heeft de programmeur niet in de hand. De benodigde werkruimte speelt alleen een rol bij een programma dat uitgebreid gebruik maakt van recursiviteit. Om het objectprogramma noemenswaardig korter te maken zou de programmeur al vrij grote wijzigingen in zijn programma aan moeten brengen. Blijft over de ruimte voor de variables, zowel simple als subscripted variables. Het aantal simple variables zal nooit groot zijn, al was het maar omdat de programmeur ze stuk voor stuk moet declareren. Een array daarentegen van 10 000 elementen kan de programmeur met niet meer moeite declareren dan een van 10 elementen.

Komt een programma in ruimtenood, dan moet de programmeur in eerste instantie proberen ruimte te winnen door de ruimte ingenomen door arrays te verminderen. Hij kan overbodig geworden arrays verwijderen door blokverlating, voordat hij nieuwe declareert, of een array op verschillende plaatsen verschillende functies laten vervullen. Integer arrays van kleine integers kan hij "pakken", dat wil zeggen meer dan een integer in een arrayplaats opbergen. In het algemeen moet hij een equivalente schrijfwijze van het programma vinden, die minder arrayruimte vereist.
In het ergste geval kan het programma in autonome stukken gesplitst worden, die na elkaar gedraaid worden, waarbij ieder deel een korter objectprogramma en misschien minder geheugenbehoefte heeft.

16 Nog iets over tijden en efficiency

De tijdsduur van de standaardfuncties voor het X1- en het X8-ALGOL-systeem:

	X1	X8
sqrt	4.4 msec	340 mmsec
ln	26.7 msec	580 mmsec
exp	24.7 msec	735 mmsec
cos	28.2 msec	450 mmsec
sin	24.6 msec	470 mmsec
arctan	109.2 msec	725 mmsec
Ter vergelijking:		
r:= 3.14	2.9 msec	15 mmsec

Als bijzonderheid van het X8-systeem kan vermeld worden dat een schrijfwijze van expressies waarbij zoveel mogelijk de eenvoudige subexpressies achteraan staan, zowel in tijd als in geheugengebruik aanzienlijk efficienter kan zijn.
Het uitrekenen bijv. van $m + n \times (n - 1)$ vergt op de X8 ongeveer 120 mmsec en 8 plaatsen. Voor $(n - 1) \times n + m$ is dit ongeveer 90 mmsec en 4 plaatsen. We kunnen dit begrijpen als we weten hoe beide expressies vertaald worden:

m + n × (n − 1):	(n − 1) × n + m:
F:= m;	F:= n;
S1:= F;	F:= F − 1;
F:= n;	F:= F × n;
S2:= F;	F:= F + m
F:= n;	
F:= F − 1;	
F:= F × S2;	
F:= F + S1	

Dit kan bijvoorbeeld bij de berekening van een polynoom van groot belang zijn:
a0 + x × (a1 + x × (a2 + x × (a3 + x × (a4 + x × a5)))): ± 495 mmsec; 29 plaatsen
((((a5 × x + a4) × x + a3) × x + a2) × x + a1) × x + a0: ± 360 mmsec; 11 plaatsen
Uiteraard schrijven we niet:
a0 + a1 × x + a2 × x ↑ 2 + a3 × x ↑ 3 + a4 × x ↑ 4 + a5 × x ↑ 5: ±2000 mmsec; 41 plaatsen

Nog een voorbeeld:
We willen de restterm berekenen die ontstaat wanneer we de Taylorreeks voor sin (x) na de term met x ↑ 9 afbreken:

```
sin (x) = x - x ↑ 3 / fac (3) + x ↑ 5 / fac (5) - x ↑ 7 / fac (7) + x ↑ 9 / fac (9) + R (x)

        real procedure R (x); value x; real x;
        begin  real x2;
               x2:= x × x;
               R:= sin (x) - ((((x2 / 72 - 1) × x2 / 42 + 1) × x2 / 20 - 1) × x2 / 6 + 1) × x
        end
```

Opmerking. De tijden in het voorafgaande zijn vermeld om een indruk te geven van de tijdsverhoudingen tussen verschillende operaties e.d. We gebruiken ze daarom niet om de efficientste te bepalen van twee versies, indien dat niet op het oog kan. Indien het verschil in tijdsduur niet onmiddellijk te zien valt zal dit waarschijnlijk te klein zijn om de moeite van het narekenen te belonen. Het gemak van ALGOL, nl. dat de programmeur zich niet om administratieve bijkomstigheden hoeft te bekommeren, zou hierbij verloren gaan.
Men kan het zich ook, ten koste van de efficiency, te moeilijk maken, door iets in ALGOL te omschrijven wat daarin ook rechtstreeks kan worden uitgedrukt.

1. We schrijven

niet	if x < 0 then -x else x	maar	abs (x)
niet	exp (ln (a) × b)	maar	a ↑ b
niet	e:= 2.718; z:= e ↑ q	maar	z:= exp (q)
niet	a ↑ (1 / 2)	maar	sqrt (a)
niet	1.5 × 10 ↑ (-3)	maar	$1.5_{10}-3$
niet	(-1) × r	maar	-r
niet	if s > 0 then true else false	maar	s > 0

2. Verder bedenken we dat het geen zin heeft de waarde van een expressie aan een variabele toe te kennen, indien de waarde van deze variabele vervolgens slechts 1 keer gebruikt wordt. Men schrijve dus niet:

   p:= ZERO (n, b, e, fn, pr)

   om vervolgens p alleen te gebruiken voor: print (p).
   We schrijven dan: print (ZERO (n, b, e, fn, pr)).
   Ook niet: t:= entier (k × f); n:= k × k - 1
   als vervolgens t en n alleen gebruikt worden in: q:= t : n.
   Men schrijve dan: q:= entier (k × f) : (k × k - 1).
   Nog een voorbeeld: q:= f / p; w:= 1 / d; q:= abs (q); m:= n × q ↑ w.
   Beter is: q:= abs (f / p); m:= n × q ↑ (1 / d)
   en nog beter nu: m:= n × abs (f / p) ↑ (1 / d).

3. For statements die dezelfde for list hebben en waarbij de ene niet op het resultaat van de gehele andere for statement behoeft te wachten, kunnen samengetrokken worden. In plaats van:

```
for j:= 1 step 1 until m do C[j, j]:= 1 / C[j, j];
for j:= 1 step 1 until m do C[j, j]:= 1 - C[j, j]
```

schrijven we beter:

```
for j:= 1 step 1 until m do begin C[j, j]:= 1 / C[j, j]; C[j, j]:= 1 - C[j, j] end
```

en nu zien we dat we nog beter kunnen schrijven:

```
for j:= 1 step 1 until m do C[j, j]:= 1 - 1 / C[j, j]
```

4. Er zijn twee vaak voorkomende gevallen aan te geven waarbij het niet zinvol is een array te gebruiken of waarbij althans een der indices overbodig is.

— De eerste overbodige array-soort is die, waarbij na toekenning van een waarde aan een element de waarden van de eerder berekende elementen niet meer gebruikt worden.

Als we bijvoorbeeld 1/1 + 1/2 + 1/3 + ... + 1/n willen berekenen als volgt:

```
A[0]:= 0;
for j:= 1 step 1 until n do A[j]:= A[j - 1] + 1 / j
```

dan kunnen we de array A door een simple variable vervangen:

```
A:= 0;
for j:= 1 step 1 until n do A:= 1 / j + A
```

— Van de tweede overbodige array-soort komen de elementen steeds voor met als index-expressie een getal. Hierbij is de array een verzameling op zichzelf staande variabelen, geen gestructureerde verzameling.

Voorbeeld:

```
A[1]:= read; A[2]:= A[1] + 1; A[3]:= A[1] / A[2];
for k:= 4, 5 do A[k]:= ln (A[k - 2]);
z:= b × A[4] / A[2] + c × A[5] / A[3];
w:= a × A[1] + b × A[4] ↑ 2 + c × A[5] ↑ 2
```

We gebruiken hier liever 5 simple variables:

```
A:= read; B:= A + 1; C:= A / B;
D:= ln (B); E:= ln (C);
z:= b × D / B + c × E / C;
w:= a × A + b × D × D + c × E × E
```

Wanneer in dit tweede geval naast de overbodige index ook zinvolle indices voorkomen, bijv. `real array` PARAM[1 : 5, 1 : k, 1 : n], dan declareren we natuurlijk: `real array` PA, PB, PC, PD, PE[1 : k, 1 : n].

Als voorbeeld van het overbodig gebruik van arrays en van nog enkele andere genoemde fouten zullen we een programma geven voor het berekenen van e en 1/e.

Zoals bekend: $e = 1 + 1 / fac(1) + 1 / fac(2) + 1 / fac(3) + \ldots$

$1/e = 1 - 1 / fac(1) + 1 / fac(2) - 1 / fac(3) + \ldots$

We zullen eerst overbodige arrays gebruiken en het programma vervolgens stap voor stap verbeteren. Ga na hoe meestal een verbetering pas dank zij de vorige stap mogelijk wordt.

```
begin   integer n; n:= read;
        begin   integer k; real p; real array A, B[1 : 2, 0 : n], FAC[0 : n];
        FAC[0]:= 1;
        for k:= 1 step 1 until n do FAC[k]:= FAC[k - 1] × k;
        A[1, 0]:= A[2, 0]:= 1;
        for k:= 1 step 1 until n do
        begin   B[1, k]:= 1 / FAC[k];
                B[2, k]:= (-1) ↑ k / FAC[k];
                A[1, k]:= A[1, k - 1] + B[1, k];
                A[2, k]:= A[2, k - 1] + B[2, k]
        end;
        for k:= 1 step 1 until n do
        begin NLCR; print (A[1, k]); print (A[2, k]) end;
                NLCR; p:= A[1, n] × A[2, n]; print (p)
        end
end
```

We zien in het programma drie parallelle for statements: for k:= 1 step 1 until n do
We trekken deze samen en schrappen tevens de variabele p, waarvan de waarde immers slechts 1 keer wordt gebruikt.

```
begin   integer n; n:= read;
        begin   integer k; real array A, B[1 : 2, 0 : n], FAC[0 : n];
                FAC[0]:= 1; A[1]:= A[2]:= 1;
                for k:= 1 step 1 until n do
                begin   FAC[k]:= FAC[k - 1] × k;
                        B[1, k]:= 1 / FAC[k];
                        B[2, k]:= (-1) ↑ k / FAC[k];
                        A[1, k]:= A[1, k - 1] + B[1, k];
                        A[2, k]:= A[2, k - 1] + B[2, k];
                        NLCR; print (A[1, k]); print (A[2, k])
                end;
                NLCR; print (A[1, n] × A[2, n])
        end
end
```

In bovenstaande tweede versie zien we dat de array FAC tot de eerste soort der overbodige arrays behoort: Heeft FAC[k] eenmaal een waarde, dan worden FAC[k - 1] en de daaraan voorafgaande elementen van FAC niet meer gebruikt. De index van FAC kan dus achterwege blijven. Hetzelfde geldt voor de tweede index van A en B.

```
begin   integer n; n:= read;
        begin   integer k; real FAC; real array A, B[1 : 2];
                FAC:= A[1]:= A[2]:= 1;
                for k:= 1 step 1 until n do
                begin   FAC:= FAC × k;
                        B[1]:= 1 / FAC;
                        B[2]:= (–1) ↑ k / FAC;
                        A[1]:= A[1] + B[1];
                        A[2]:= A[2] + B[2];
                        NLCR; print (A[1]); print (A[2])
                end;
                NLCR; print (A[1] × A[2])
        end
```

We zien nu dat de arrays A en B tot overbodige arrays van de tweede soort zijn geworden: alleen A[1], A[2], B[1] en B[2] verschijnen in de tekst. (In feite was de eerste index van A en B al vanaf het begin overbodig). We nemen dus liever vier variabelen: e1, e2, term1 en term2. Verder is het nu onnodig een apart binnenblock te introduceren.

```
begin   integer n, k; real FAC, e1, e2, term1, term2;
        n:= read; FAC:= e1:= e2:= 1;
        for k:= 1 step 1 until n do
        begin   FAC:= FAC × k;
                term1:= 1 / FAC; term2:= (–1) ↑ k / FAC;
                e1:= e1 + term1; e2:= e2 + term2;
                NLCR; print (e1); print (e2)
        end;
        NLCR; print (e1 × e2)
end
```

De uitdrukking (-1) ↑ k zorgt voor het alternerend teken van term2. Dit kan veel efficienter door term2 steeds te vervangen door -term2 / k, waardoor FAC overbodig wordt.

```
begin   integer n, k; real e1, e2, term;
        n:= read; e1:= e2:= term:= 1;
        for k:= 1 step 1 until n do
        begin   term:= -term / k;
                e1:= e1 + abs (term);
                e2:= e2 + term;
                NLCR; print (e1); print (e2)
        end;
        NLCR; print (e1 × e2)
end
```

17 Efficiency van het proces

Tot nu toe is steeds gesproken over efficiency in detailpunten: steeds werd getoond hoe een bepaald proces op efficientere wijze kon worden geschreven. Van hoger belang is, dat het proces zelf efficient is, en in het volgende zullen we aanwijzingen en voorbeelden in die richting geven.

Voorbeeld. Laten we het probleem aanpakken van het naar grootte rangschikken van de elementen van een array. Eerst zetten we het grootste element van de array op de laatste plaats, vervolgens het op 1 na grootste op de op 1 na laatste plaats, enz. We krijgen het grootste element achteraan door van voren af elk tweetal opeenvolgende elementen te verwisselen als het niet in de juiste volgorde staat.

```
procedure SORT1 (A, n); value n; integer n; array A;
begin   integer k, j; real z;
        for k:= n step -1 until 2 do
        for j:= 2 step 1 until k do
        begin   if A[j] < A[j - 1] then
                begin z:= A[j]; A[j]:= A[j - 1]; A[j - 1]:= z end
        end
    end
```

We kunnen dit proces efficienter maken door minder verwisselingen te doen: i.p.v. elk verkeerd geordend tweetal opeenvolgende elementen te verwisselen, kunnen we ineens het maximum van de reeks zoeken, en dit grootste element met het laatste verwisselen. Zo krijgen we:

```
procedure SORT2 (A, n); value n; integer n; array A;
begin   integer k, j, jmax; real Amax;
        for k:= n step -1 until 2 do
        begin   jmax:= 1; Amax:= A[jmax];
                for j:= 2 step 1 until k do
                begin   if A[j] > Amax then
                        begin jmax:= j; Amax:= A[jmax] end
                end;
                A[jmax]:= A[k]; A[k]:= Amax
        end
    end
```

Dat het proces nog veel efficienter kan worden ingericht blijkt uit de procedure Quickersort (Comm. ACM 8 (1965) 11, Algorithm 271).

Ter vergelijking de tijden voor het X1-ALGOL-systeem voor een array van 50 elementen:

SORT1 (A, 50)	45 sec
SORT2 (A, 50)	26 sec
Quickersort (A, 50)	10 sec

Hier blijkt dat het goed is voor dit standaardprobleem een oplossing uit de literatuur te nemen. Dit geldt ook voor veel numerieke problemen, zoals het vinden van een nulpunt, het benaderen van een functie door een polynoom enz.

Voorbeeld. We beschouwen de integratie van een reele functie f(x) over het interval $a \leq x \leq b$. We verdelen het interval in stukjes van lengte h, bepalen de integraal over ieder stukje met een benaderingsformule, en vinden een benadering van de totale integraal door de som van deze waarden te nemen. Als benaderingsformule voor de integraal van x0 tot x0+h kunnen we nemen:

1. blokjesmethode	$h \times f(x0)$
2. trapeziumregel	$h/2 \times (f(x0) + f(x0 + h))$
3. regel van Simpson	$h/6 \times (f(x0) + 4 \times f(x0 + h/2) + f(x0 + h))$

We passen deze formules toe om 1/x te integreren over $1 \leq x \leq 2$, hetgeen als resultaat ln(2) = .693147180560 moet opleveren. We bepalen het antwoord in een bepaalde precisie met de verschillende methoden, en kijken dan hoe efficient ze zijn. Een goede maatstaf voor de efficiency geeft het aantal malen dat de integrandfunctie berekend moet worden.

We krijgen dan de volgende tabel:

	aantal evaluaties benodigd bij		
aantal decimalen relatieve precisie	blokjes-methode	trapezium-regel	regel van Simpson
1	4	2	3
2	37	4	3
3	361	11	5
4	3607	32	7
5	$\pm$36000	96	11
6	302	17	
7	951	27	
8	$\pm$3000	49	
9	$\pm$9500	83	
10		147	
11		247	

Uit deze tabel blijkt wel duidelijk dat de regel van Simpson verreweg het efficientste proces geeft, en dat de blokjesmethode niet praktisch bruikbaar is.

In het volgende voorbeeld wordt een veel eenvoudiger en efficienter proces verkregen door een formule iets anders te schrijven. Gevraagd wordt gemiddelde en spreiding te berekenen van getallen x[1] t/m x[n], die op een getalband staan, voorafgegaan door de waarde van n. Dit kan als volgt:

```
begin   integer n; real gemiddelde, spreiding;
        n:= read;
        begin   integer i; real s, sx; real array x[1 : n];
                sx:= 0;
                for i:= 1 step 1 until n do
                begin x[i]:= read; sx:= sx + x[i] end;
                gemiddelde:= sx / n; s:= 0;
                for i:= 1 step 1 until n do s:= s + (x[i] - gemiddelde) ↑ 2;
                spreiding:= sqrt (s / (n - 1))
        end;
        ..........
end
```

Nu is (x[i] - gemiddelde) ↑ 2 = x[i] ↑ 2 - 2 × x[i] × gemiddelde + gemiddelde ↑ 2. Wanneer we dus reeds tijdens het lezen ook nog de som van de x[i] ↑ 2 opbouwen, kunnen we de spreiding achteraf zonder sommatie berekenen en is het niet nodig de (mogelijk zeer vele) afzonderlijke x[i] tegelijk te onthouden:

```
begin   integer i, n; real x, sx, sx2, gemiddelde, spreiding;
        n:= read; sx:= sx2:= 0;
        for i:= 1 step 1 until n do
        begin x:= read; sx:= sx + x; sx2:= sx2 + x × x end;
        gemiddelde:= sx/n; spreiding:= sqrt ((sx2 - gemiddelde × sx)/(n - 1));
        ..........
end
```

18 Efficiency door betrouwbaarheid

Doordat de programmeertaal ALGOL in vergelijking met eerdere programmeertalen voor de programmeur een veel gemakkelijker hanteerbaar instrument is, is het lonend geworden programma's te schrijven die slechts enkele malen of zelfs eenmaal behoeven te worden uitgevoerd voor problemen die vroeger met een tafelrekenmachine zouden zijn opgelost. Voor zulke programma's is het al weinig zinvol veel moeite te besteden aan het verkorten van de tijd, benodigd voor uitvoering van het programma; hier is vooral van belang hoeveel tijd verstrijkt tussen de oorspronkelijke formulering van het probleem en het gereed komen van de resultaten.
In de praktijk blijkt dat deze tijd bijna altijd veel groter is dan nodig zou zijn, doordat er zoveel fouten gemaakt worden. Wanneer we door een bepaalde manier van programmeren minder fouten zouden maken, of minder tijd nodig zouden hebben om een gemaakte fout op te sporen, kan daarvoor gerust iets worden prijsgegeven van de snelheid van het programma. In feite geldt ditzelfde ook voor produktieprogramma's, die vaak uitgevoerd moeten worden: de resultaten zijn niet meer waard dan de mate waarin wij op hun juistheid mogen vertrouwen, en helaas is het nooit mogelijk voor 100 procent de correctheid van een programma te garanderen (wat al blijkt uit het feit dat ook na lange tijd nog fouten worden ontdekt in goed geteste en vaak gebruikte programma's).
De belangrijkste drie soorten fouten zijn wel:

— fouten tegen ALGOL
— vergissingen en verschrijvingen
— onvoldoende geldigheid van het proces

Op de eerste soort fouten zal hier niet verder worden ingegaan.
De beste remedie tegen vergissingen en verschrijvingen is vermoedelijk het verhogen van de leesbaarheid van de programmatekst, zodat de fouten eerder opvallen.
Wat de laatste soort fouten betreft: Allereerst moet getracht worden steeds een proces te kiezen en te programmeren dat zo algemeen mogelijke geldigheid heeft, waarbij dus voor de toepasbaarheid van het proces geen restricties gelden; als dat niet lukt moet duidelijk omschreven worden welke de restricties zijn en moet het programma nagaan of aan de voorwaarden voor de toepasbaarheid voldaan is.

19 Leesbaarheid

Een goede lay-out kan veel bijdragen tot de overzichtelijkheid en leesbaarheid van het programma. Sterk aanbevolen wordt de volgende regel, die de structuur van het programma in de lay-out tot uitdrukking brengt:
Wanneer een symbool begin en het bijbehorende symbool end niet op dezelfde regel staan, zorg dan dat begin recht boven end komt te staan, en dat het stuk programma daartussen rechts van de kolom tussen begin en end komt (alsof na het symbool begin de kantlijn ongeveer 8 plaatsen opschuift, en voor het symbool end weer 8 plaatsen teruggaat). Schrijf labels (gevolgd door ":") in de kolom tussen begin en end van het block waarbij ze behoren (zie paragraaf 4).
Bijvoorbeeld:

```
begin    real a;
START: ..........
         begin ..........  end;
         begin    integer b;
                  ..........
                  begin    ..........
                           ..........
                  end;
                  ..........
                  begin    ..........
                           begin ..........  end;
      ZOEK:               ..........
                  end
      end;
      ..........
EINDE:
end
```

Hierbij staan begin of end dus aan het begin van een regel. Geschikte punten om een regel af te breken zijn dan ook: voor begin, end of een label, en verder na een ";", do, else of then. (Probeer dus niet else recht onder het bijbehorende then te plaatsen).

Een redelijke spatiering tussen de symbolen is ook van groot belang voor de lay-out. Hiervoor kunnen de volgende regels gebruikt worden, die weliswaar enigzins willekeurig zijn, maar waarvan het konsekwent gebruik een rustige tekst garandeert:
Geef tussen ieder tweetal symbolen een spatie, behalve
— tussen de afzonderlijke symbolen van een identifier of een getal.
— tussen een array of switch identifier en het symbool "[".
— tussen een label identifier en ":".
— tussen de unaire operatoren "+" en "-" en het operand.
— links van ; , :=)]
— rechts van (en [
In het volgende voorbeeld komen alle uitzonderingen voor:
VOORBEELD: z:= P (-A[0], 3.14_{10}-2);

In ALGOL bestaat een grote vrijheid voor de keuze van de identifiers. Door hier verstandig gebruik van te maken kunnen we de kans op verschrijvingen reduceren. In het volgende stukje programma:

```
begin   integer array F[1 : m, 1 : n]; real array Fg[1 : m];
        ..........
        for k:= 1 step 1 until m do Fg[k]:= SUM (j, 1, n, F[k, j]) / n;
        ..........
end
```

zal een verwisseling tussen m en n veel minder opvallen dan wanneer we (gebruik makend van onze kennis van wat m en n betekenen) schrijven:

```
begin   integer array Freq[1 : nletters, 1 : nauteurs]; real array Fgem[1 : nletters];
        ..........
        for letter := 1 step 1 until nletters do
        Fgem[letter]:= SUM (auteur, 1, nauteurs, Freq[letter, auteur]) / nauteurs;
        ..........
end
```

Meestal zal het niet moeilijk zijn een geschikte naam te kiezen. Wel moeten we bedenken, dat voor een array vaak beter een naam gekozen kan worden die karakteristiek is voor de afzonderlijke elementen dan voor het geheel, dus
niet PUNT[1 : 3] maar COORDINAAT[1 : 3],
niet BORD[1 : 8, 1 : 8] maar VELD[1 : 8, 1 : 8],
niet WOORD[1 : lengte] maar LETTER[1 : lengte],
aangezien bijv. LETTER[i] een letter, en niet een woord voorstelt.

20 Algemene geldigheid

We moeten proberen voor onze processen een formulering te vinden die zo algemeen mogelijk geldig is, zodat dezelfde formulering in principe ook geschikt is indien sommige details van het probleem veranderen. We moeten dus geen gebruik maken van toevallige eigenschappen van onze gegevens, die niet uit de probleemstelling volgen. Willen we bijvoorbeeld een proces formuleren voor het vinden van de kleinste uit een rij getallen waarvan we verwachten dat er wel een getal kleiner is dan $_{10}6$, dan zouden we kunnen schrijven:

```
min:= 10.6; for k:= 1 step 1 until n do if M[k] < min then min:= M[k]
```

maar als we dit stukje programma een keer gebruiken voor een reeks getallen die alle groter zijn dan $_{10}6$, dan gaat dit proces mis. De volgende formulering is geldig voor iedere verzameling:

```
min:= M[1]; for k:= 2 step 1 until n do if M[k] < min then min:= M[k]
```

Wanneer we toch om de een of andere reden een methode moeten gebruiken die niet in alle denkbare mogelijkheden voorziet dan moeten we:

— de voorwaarden waaronder de methode geldig is in een commentaar vermelden.

— in het programma er op testen of aan de voorwaarden voldaan is.

Voorbeelden.

—
```
real procedure f (z); value z; real z;
comment Het is vereist dat z > 0;
if z > 0 then begin real w; w:= sqrt (z); f:= sin (w) / w end else
begin NLCR; PRINTTEXT (‡argument van w niet positief‡); EXIT end
```
In dit voorbeeld wordt gebruik gemaakt van standaardprocedures van het X8-ALGOL-systeem.

— De rij getallen X[1 : m] moet met een zodanig getal vermenigvuldigd worden, dat X[q] gelijk aan 1 wordt, waarbij q een van de band te lezen getal is.
Eerste, foutieve oplossing:

```
begin   procedure scalmult (A, n, labda);
        value labda; integer n; real labda; real array A;
        begin   integer i;
                for i:= 1 step 1 until n do A[i]:= A[i] × labda
        end;
        scalmult (X, m, 1 / X[read])
end
```

Het bezwaar hiertegen is, dat indien door een leesfout of door een fout op de invoerband het gelezen getal niet in het bereik 1 t/m m ligt, dat dan het resultaat van dit stukje programma ongedefinieerd is, wat onder meer wil zeggen dat elke implementatie van ALGOL hier weer verschillend op kan reageren.
In de volgende oplossing geldt dit bezwaar niet meer:

```
begin   integer q;
        procedure scalmult (A, n, labda);
        value labda; integer n; real labda; real array A;
        begin   integer i;
                for i:= 1 step 1 until n do A[i]:= A[i] × labda
        end;
        q:= read;
                if 1 ≤ q ∧ q ≤ m then scalmult (X, m, 1 / X[q]) else
                begin NLCR; PRINTTEXT (‡q niet juist‡); EXIT end
end
```

Merk op dat voor de parameter labda vermelding in de value list absoluut noodzakelijk is: de waarde van X[q] en dus ook die van 1 / X[q] wordt tijdens de aanroep van scalmult gelijk aan 1.

Er moet niet alleen gestreefd worden naar "algemene geldigheid" van het gehele programma; we moeten er ook voor zorgen dat het onwaarschijnlijk is dat een der deelprocessen niet correct zou blijven functioneren als elders in het programma een wijziging wordt aangebracht: algemene geldigheid dus ook voor de delen van het programma. Indien bijvoorbeeld bovenstaande procedure scalmult geschreven zou zijn voor een geval waarin de actual parameter voor labda een simple variable is, dan zou toch opneming in de value list uit oogpunt van betrouwbaarheid gewenst zijn.

Voorbeeld.

In een programma wordt met een aantal symbolen gewerkt, die als integers zijn voorgesteld met de volgende codetabel:

symbool:	0	1	2	3	4	5	6	7	8	9	?	,	$_{10}$	+	–	.
code:	12	3	9	6	2	13	7	8	4	11	1	14	10	5	15	0

De vraag of nu een symbool een der symbolen $_{10}$, +, – of . is kan met behulp van de toevalligheid dat juist de codes van deze symbolen een vijfvoud zijn, als volgt worden gesteld: code : 5 × 5 = code.

Het is echter juister te vragen: code = 10 ∨ code = 5 ∨ code = 15 ∨ code = 10.

Het beste is echter vooraan in het programma te schrijven:

```
integer nul, een, twee, drie, vier, vijf, zes, zeven, acht, negen, vraagteken, komma,
        tientje, plus, min, punt;
        nul:= 12; een:= 3; twee:= 9; drie:= 6; vier:= 2; vijf:= 13; zes:= 7;
        zeven:= 8; acht:= 4; negen:= 11; vraagteken:= 1; komma:= 14; tientje:= 10;
        plus:= 5; min:= 15; punt:= 0;
```

en dan te vragen: code = tientje ∨ code = plus ∨ code = min ∨ code = punt.

Blijkt nu dat we het programma met een andere codetabel willen gebruiken, dan hoeft maar op een plaats in het programma iets veranderd te worden.

De voor de algemene geldigheid wenselijke relatieve zelfstandigheid van een deel-proces kan men vaak bereiken door het in de vorm van een block of zelfs van een procedure te schrijven. Daardoor kan men ervoor zorgen dat de hulpgrootheden van het deelproces, die voor de rest van het programma niet ter zake doen, gedeclareerd worden op de plaats waar ze nodig zijn. Het is dan gemakkelijker na te gaan of een grootheid wel gedeclareerd is, en de kans dat men ten onrechte een identifier in twee betekenissen gebruikt, is aanzienlijk verminderd.

Er zijn velerlei redenen die het wenselijk kunnen doen zijn een bepaald proces in de vorm van een procedure te schrijven, waaronder het volgende drietal:

1. Het proces komt meermalen voor: we gebruiken de procedure bij wijze van afkorting.
2. Het proces is een conceptuele eenheid, die als bouwsteen zou kunnen dienen voor andere programma's.
   Wanneer we reeds over een acceptabele procedure beschikken die het gevraagde proces verricht, gebruiken we uiteraard die procedure: niet alleen bespaart ons dat de moeite van het programmeren, maar ook het risico dat we daarbij een fout maken. Dit geldt des te meer voor bibliotheek-procedures, waarvan we niet alleen redelijkerwijs mogen veronderstellen dat ze efficient zijn, maar ook dat ze grondig getest zijn.
3. Het proces levert een grootheid af waarmee we verder willen rekenen. In dit geval gebruiken we een function procedure zodat de function designator als primary in een arithmetic expression gebruikt kan worden.

Natuurlijk kan er meer dan een reden tegelijk gelden: voor de real procedure SUM (i, a, b, fi) bijvoorbeeld alle drie bovenstaande redenen.
In geval we de procedure louter gebruiken als afkorting, is het toch vaak zinvol een algemenere schrijfwijze te bezigen door gebruik te maken van de faciliteiten die een procedure ons nu eenmaal biedt. Met name moeten we zoveel mogelijk het gebruik van niet-locale namen vermijden door hulpgrootheden binnen de procedure te declareren en andere grootheden als parameter mee te geven. (Het gebruiken van niet-locale hulpgrootheden geeft juist bij een procedure, door de impliciëtheid van het gebruik bij de aanroep, gemakkelijk aanleiding tot vergissingen)
Al dit geldt des te sterker voor een bouwsteen-procedure. Alleen als het om een speciale reden gewenst is gebruiken we een niet-locale naam, en daarvan maken we dan melding in een commentaar bij de procedure heading.

Het is goed hier ervoor te waarschuwen dat vele numerieke processen, ook beproefde standaardprocessen, slechts geldig zijn onder bepaalde voorwaarden die vaak moeilijk te formuleren zijn en waaraan doorgaans wel, maar soms onverwachts niet is voldaan.

Zo is het in principe niet mogelijk een waterdichte
real procedure INTEGRAAL (x, a, b, fx, epsilon) te schrijven:
om de integraal te bepalen zal een dergelijke procedure voor een gegeven functie de parameter fx evalueren voor een eindig aantal waarden van x op het interval $a \leq x \leq b$. De dan voor de integraal afgeleverde waarde is uiteraard gelijk aan de waarde die wordt afgeleverd voor iedere functie die in die basispunten x dezelfde functiewaarden fx heeft. Het is echter mogelijk een functie te construeren waarvan de grafiek door al die punten (x, fx) gaat, maar waarvan de integraal over het interval $a \leq x \leq b$ sterk verschilt van die van de oorspronkelijke functie. Aangezien nu de procedure INTEGRAAL voor beide functies dezelfde waarde levert, zal die voor tenminste een van de twee functies onjuist zijn.
Voorbeeld voor de procedure QAD (AP 251):
Bij de aanroep QAD (x, 0, 1, 0, $_{10}$-12) wordt fx geevalueerd voor x = 0, x = .25, x = .50, x = .75 en x = 1; de waarde van de function designator wordt dan 0. Hetzelfde resultaat zal ook worden geleverd voor iedere functie die in die vijf basispunten gelijk aan 0 is, zoals sin (4 × 3.14159265359 × x) ↑ 2.
QAD (x, 0, 1, sin (4 × 3.14159265359 × x) ↑ 2, $_{10}$-12) geeft inderdaad (nagenoeg) de waarde 0 in plaats van de waarde 0.50.

In het begin van deze paragraaf is er al op gewezen dat we in een programma tests horen in te bouwen om zoveel mogelijk na te gaan of aan alle nodige en bekende voorwaarden voldaan is. Om het belang hiervan te onderstrepen volgt een opsomming van voordelen van het grondig zelf-testen van een programma:

— Besparing van machine-tijd doordat de uitvoering van een programma waarin een fout is ontdekt wordt beeindigd.
— Het opsporen van fouten wordt gemakkelijker doordat de fout zo dicht mogelijk bij de bron wordt opgespoord en doordat de fout wordt gespecificeerd.
— De resultaten van een programma met tests zijn betrouwbaarder dan die van hetzelfde programma zonder tests.

Het is gebleken dat onderstaande procedure goed voldoet voor het inlassen van tests:

```
procedure stop (fout, diagnose, output, indicatie);
Boolean fout; string diagnose; procedure output;
if fout then
begin   NLCR; PRINTTEXT (‹fout: ›); PRINTTEXT (diagnose);
        SPACE (1); output (indicatie); EXIT
end
```

Het voorbeeld in het begin van deze paragraaf wordt, bij gebruik van de procedure:

```
real procedure f (z); value z;
begin    real w;
         stop (z ≤ 0, ‡argument van f is niet positief:‡, PRINT, z);
         w:= sqrt (z); f:= sin (w) / w
end
```

21 Real-aritmetiek (RR 3.3.6)

Door het RR worden helaas geen regels gegeven voor het rekenen met real-grootheden, zodat voor een real-grootheid x niet kan worden afgeleid $x + 1 \geq x$.
We zullen in het volgende toch gebruik moeten maken van een aantal regels die voor vele implementaties, waaronder het X8-ALGOL-systeem, wel gelden. (Voor het X1-ALGOL-systeem gelden niet alle gegeven regels)
Als de implementatie waarin onze programma's worden uitgevoerd voor de berekeningen met real-grootheden een relatieve nauwkeurigheid van bijv. 12 decimalen heeft, dan zullen de resultaten doorgaans ook zonder speciale maatregelen wel een nauwkeurigheid van 5 decimalen hebben. We kunnen bijvoorbeeld al honderd biljoen getallen met elkaar vermenigvuldigen (voor de X8 nog altijd een werk van eeuwen) voor we hoeven te verwachten dat het door de gezamenlijke afrondingsfouten zover komt. Toch is een summiere analyse van de afrondingsfouten beslist niet overbodig.
Voorbeeld: We proberen de functie j1(z) = sin(z)/z↑2 - cos(z)/z te berekenen met behulp van de real procedure j1 (z); value z; real z; j1:= (sin (z) / z - cos (z)) / z.
We hoeven dan niet te verwachten dat evaluatie van de function designator "j1 ($_{10}$-6)" een waarde oplevert waarvan zelfs maar de eerste decimaal correct is: tengevolge van afrondingsfouten is de relatieve nauwkeurigheid tot 0 decimalen teruggelopen.
De afrondingsfouten bij het rekenen met real-grootheden ontstaan doordat niet alle reele waarden als real voorstelbaar zijn (d.w.z. door een real-grootheid kunnen worden aangenomen).
Het resultaat van een real-aritmetische bewerking, bijvoorbeeld een real-vermenigvuldiging, kan als volgt worden verkregen:

1. De operanden worden volgens de regels van de real-aritmetiek geevalueerd, zodat hiervoor een reele waarde verkregen is die als real voorstelbaar is.
2. Het exacte mathematische resultaat van de bewerking wordt gevormd.
3. Deze waarde wordt afgerond tot een als real voorstelbare waarde.

Wanneer we het resultaat van de afronding van x voorstellen als R(x), dan kunnen we de volgende regels opstellen waaraan de functie R onder meer moet voldoen:

1. R(R(x)) = R(x), d.w.z. reeds als real voorstelbare waarden worden niet meer afgerond.
2. Als x > y dan R(x) ≥ R(y).
3. Voor integerwaarden i binnen de integercapaciteit (d.w.z. waarden die door een integer-grootheid kunnen worden aangenomen): R(i) = i.

(In vrij veel systemen wordt het real-getal 0 apart behandeld, zodat bijvoorbeeld zou kunnen gelden: (3.14 - 3.14) × 2 ≠ 0. Bij de X8 komt hier wel degelijk 0 uit)

Uit de gegeven relaties is af te leiden: 3.14 - 3.14 = 0 ∧ 3.14 / 3.14 = 1.

Niet af te leiden valt 1 / 3 × 3 = 1; dit zal dan ook doorgaans niet opgaan.

Voor het X8-ALGOL-systeem geldt verder nog:

— Als $_{10}$-604 ≤ |x| ≤ $_{10}$628, dan is |R(x) - x| < $_{10}$-12 × |x|,
en |R(x) - x| ≤ |R(y) - x| (afronding naar dichtstbijzijnde real-waarde).

— R(x) = 0 indien en slechts indien x = 0.

— Voor integerwaarden i waarvoor |i| < 1 099 511 627 776 = 2 ↑ 40, geldt R(i) = i.

— Voor het resultaat van de standaardfuncties kan niet worden gegarandeerd dat ze voor de volle 12 decimalen juist zijn.
De operatie a ↑ b (b real) wordt berekend als exp (ln (a) × b) en zal dus ook niet altijd in 12 decimalen nauwkeurig zijn.
Als a en b echter als real-voorstelbare waarden zijn (d.w.z. R(a) = a en R(b) = b), en in mathematische zin geldt f(a) = b, waarbij f een der standaard-functies is, dan geldt ook voor het X8-ALGOL-systeem f (a) = b.
Hieruit valt af te leiden: exp (0) = 1 ∧ sqrt (4) = 2 ∧ 3.14 ↑ (3.14 - 3.14) = 1.
Niet afleidbaar is: ln (exp (1)) = 1 ∨ sqrt (2) × sqrt (2) = 2.

Een groot verlies in relatieve nauwkeurigheid ontstaat altijd door het van elkaar aftrekken van ongeveer even grote getallen:

	exacte waarde	voor de X8	relatieve fout
sin($_{10}$-6)/$_{10}$-6	.999 999 999 999 833 333 33	1.000 000 000 000 000 000 00	.17$_{10}$-12
cos($_{10}$-6)	.999 999 999 999 500 000 00	.999 999 999 999 090 505 30	.41$_{10}$-12
verschil	.000 000 000 000 333 333 33	.000 000 000 000 909 494 70	1.7

In plaats van j1($_{10}$-6) = .000 000 333 333 33 wordt met de eerdergenoemde procedure j1 door de X8 .000 000 909 494 70 berekend. Voor nog kleinere waarden van het argument z kan niet alleen de relatieve, maar ook de absolute fout zeer groot worden: voor j1 ($_{10}$-20) bijvoorbeeld krijgen we -90 949 470 i.p.v. .333 333$_{10}$-12 !
In dit geval kunnen we voor kleine waarden van z de onnauwkeurigheid opheffen door de functie te benaderen door een polynoom. We zullen onderzoeken in welk gebied we deze benadering nodig hebben. Laten we ervan uitgaan dat voor ons probleem 6 decimalen voldoende nauwkeurig is. We bepalen nu de onnauwkeurigheid van de oorspronkelijke methode als functie van z:

De fout in sin (z) is relatief ongeveer $_{10}$-12. Het mathematische resultaat na deling door z zal -aangenomen dat z exact wordt voorgesteld- dezelfde relatieve fout van $_{10}$-12 hebben. De afronding naar een real getal levert nog eens een fout van $_{10}$-12, tezamen 2_{10}-12. Aangezien sin(z)/z voor kleine z ongeveer 1 is, is de absolute fout 2_{10}-12 × 1 = 2_{10}-12. De fout in cos (z) is relatief ongeveer $_{10}$-12, dus absoluut $_{10}$-12 × 1 = $_{10}$-12. De aftrekking sin (z) / z - cos (z) geeft dan ten hoogste een absolute fout van 3_{10}-12. (De afronding van dit resultaat geeft een verwaarloosbare fout $_{10}$-12 × (sin(z)/z - cos(z)), ongeveer $_{10}$-12 × z↑2/3) Na deling door z is de absolute fout tenslotte 3_{10}-12/z. Omdat j1(z) ongeveer z/3 is, is de relatieve fout 9_{10}-12/z↑2.

Voor welke waarden van z is dit klein genoeg?

9_{10}-12/z↑2 < $_{10}$-6: |z| > 3_{10}-3.

We willen de polynoombenadering dus gebruiken in het gebied -3_{10}-3 ≤ z ≤ 3_{10}-3

De Taylorreeks voor j1(z) is: j1(z) = z/3 - z↑3/30 + z↑5/840 - .....

Breken we na de eerste term af, dan maken we een absolute fout van ongeveer z↑3/30, en dus een relatieve van z↑2/10. Als |z| ≤ 3_{10}-3 is z↑2/10 ≤ .9_{10}-6, dus voldoende nauwkeurig. Het volstaat hier dus de eerste term te nemen.

De beste grenswaarde voor de omschakeling van de ene op de andere formule vinden we door de fouten aan elkaar gelijk te stellen: 9_{10}-12/z↑2 = z↑2/10.

Dit levert ons weer nagenoeg ± 3_{10}-3. De uiteindelijke procedure wordt dan

```
real procedure j1 (z); value z; real z;
j1:= if abs (z) ≤ 3₁₀-3 then z / 3 else (sin (z) / z - cos (z)) / z
```

Nog meer voorbeelden:

— sqrt (a × a + a) - a: een wortel van x↑2 + 2×a×x - a = 0. Voor grote waarden van a is mathematisch sqrt (a × a + a) - a ongeveer 0.50, maar voor a ≥ $1.6_{10}12$ is bij de X8 a × a + a = a × a, zodat we niet hoeven te verwachten dat daar veel van overblijft.
Als we stellen: z = 1/x krijgen we de vergelijking z↑2 - 2×z - 1/a = 0,
met een wortel z = sqrt (1 / a + 1) + 1, dus x = 1 / (sqrt (1 / a + 1) + 1).
Voor grote waarden van a is deze formule numeriek heel wat beter.

— De expressie 2 / sin (2 × x) - 1 / sin (x) is voor waarden van x dicht bij 0 numeriek onbruikbaar. Beter is sin (x) / cos (x) / (1 + cos (x)).

— De real procedure arsinh (x); value x; real x; arsinh:= ln (sqrt (x × x + 1) + x) is voor grote negatieve waarden van x in absolute zin zeer onnauwkeurig, en voor waarden van x dicht bij 0 in relatieve zin.
Het eerste punt lossen we eenvoudig op door gebruik te maken van de relatie arsinh (-x) = -arsinh (x). De tweede moeilijkheid komt doordat bij de berekening van sqrt (x × x + 1) + x de relatieve precisie grotendeels verloren gaat.

Wel kunnen we berekenen h = sqrt (x × x + 1) + x – 1 in de volle relatieve precisie, namelijk met h:= x × x / (sqrt (x × x + 1) +1) + x. Hoe berekenen we nu ln (h + 1) in volle relatieve precisie? Zogauw we immers h + 1 berekenen zijn we weer alles kwijt.
Indien h + 1 dicht bij r ligt, dan is ln (r) een benadering voor ln (h + 1), maar voor h $\leq$ 3.9 is de waarde gevonden door lineaire inter- of extrapolatie in ln (x + 1) tussen x = 0 en x = r – 1, namelijk y = h / (r – 1) × ln (r), een betere benadering. (We nemen aan dat h > 0). Voor r nemen we dan R(h + 1), het real-getal waarop h + 1 wordt afgerond. Voor h > 3.9 wordt de fout bij toepassing van de correctie zo al dan toch nauwelijks groter dan de fout die toch al aanwezig is. We nemen dus voor ln (h + 1) steeds h / (r – 1) × ln (r), behalve natuurlijk als r = 1.

```
real procedure arsinh (x); value x; real x;
begin   real h, r;
        h:= x × x / (sqrt (x × x + 1) + 1) + abs (x); r:= h + 1;
        arsinh:= sign (x) × (if r = 1 then h else h / (r – 1) × ln (r))
end
```

22 Iteratieve processen

Een soort proces dat vaak voorkomt is het oplossen van een vergelijking door middel van iteratie, d.w.z. door opeenvolgende verbeteringen van een schatting van de oplossing. Uitgaande van een beginschatting x[0] berekenen we x[1]:= f(x[0]), dan x[2]:= f(x[1]) enz. Een iteratiestap bestaat dan uit x[i]:= f(x[i–1]).
Wanneer nu de waarden x[i] naar de limiet x convergeren, en f is continu in x, dan is x een wortel van de vergelijking x = f(x).
Doorgaans wordt bij een iteratiestap de schatting van een aantal grootheden verbeterd, zoals in:

```
begin   x[i]:= f (x[i – 1], y[i – 1], z[i – 1]);
        y[i]:= g (x[i – 1], y[i – 1], z[i – 1]);
        z[i]:= h (x[i – 1], y[i – 1], z[i – 1])
end
```

In dit geval zijn de limieten x, y en z de oplossing van de vergelijking:

$$x = f(x, y, z) \wedge y = g(x, y, z) \wedge z = h(x, y, z).$$

Het bekendste iteratieproces is wel de methode van Newton voor het berekenen van sqrt (a), met f(x) = (x + a / x) / 2.

Kiezen we bijvoorbeeld x[0] = 1, a = 4, dan krijgen we achtereenvolgens: x[1] = 2.50000, x[2] = 2.05000, x[3] = 2.00061, x[4] = 2.00000. Het is duidelijk dat dit snel convergeert. Zouden we door een vergissing a = –4 genomen hebben, dan krijgen we: x[1] = –1.50000, x[2] = .58333, x[3] = –3.13690, x[4] = –.93088 enz. In dit geval convergeert het proces niet. Zouden we nu in het programma het proces pas beeindigen, zodra twee opeenvolgende waarden van x minder dan $_{10}$–5 van elkaar verschillen, dan zou door deze vergissing het programma niet beeindigd worden. (Het verschil is telkens minstens 2). Nu is in dit voorbeeld de berekening van de functie f weinig tijdrovend, maar in de praktijk komen hiervoor vaak functies voor waarvan de berekening per keer veel tijd kost. In zo'n geval zou pas na lange tijd bemerkt worden dat het programma niet goed werkt. We moeten er dus voor zorgen dat in het programma de convergentie gecontroleerd wordt. We moeten er ook op letten dat we geen grotere precisie eisen dan gezien de precisie van de gegevens zinvol is.

We kunnen nu de volgende drie tests opstellen die in een iteratieproces moeten worden opgenomen:

1. Beeindiging van het proces zodra het aantal iteratiestappen een zeker maximum, bijv. 100, zou overschrijden. Dit is een soort noodrem die af en toe nuttige diensten kan bewijzen.
2. Test op de convergentie.

   De convergentie van het iteratieproces kunnen we vaak beschouwen aan de hand van de volgende stelling:

   Als op een zeker interval $a \leq x \leq b$ de functie f(x) voldoet aan:

   1. $a \leq f(x) \leq b$
   2. $|f(x1) - f(x2)| < |x1 - x2|$ (x1 en x2 op het interval [a, b], $x1 \neq x2$)

   dan convergeert de reeks x[i] = f(x[i – 1]) (met beginschatting x[0] op het interval) naar een limiet x (op het interval) die voldoet aan: f(x) = x.

   De tweede voorwaarde is essentieel (zoals het niet convergerende geval f(x) = –x aantoont). In het programma testen we daarom of abs (x[i + 1] – x[i]) < abs (x[i] – x[i – 1]), anders beeindigen we het proces. Deze voorwaarde ligt numeriek erg moeilijk; niet eraan voldaan zijn hoeft niet op een fout te wijzen. Neem bijvoorbeeld een aritmetiek waarin die getallen representeerbaar zijn die in decimale voorstelling achter de punt 5 cijfers hebben en verder nullen, en met de afronding naar de dichtstbijzijnde mogelijke garde. Beschouw nu $f(x) = .6 \times (1 - x)$. Het iteratieproces moet dan convergeren naar .375. Mathematisch geldt: $|f(x1) - f(x2)| = .6 \times |x1 - x2| < |x1 - x2|$.
   Neem nu x1 = .37499, x2 = .37501. Evaluatie van f(x1) en f(x2) geeft:

f(x1) = R(.6 × R(1 − .37499)) = R(.6 × R(.62501)) = R(.6 × .62501) =
= R(.375006) = .37501 = x2.
f(x2) = R(.6 × R(1 − .37501)) = R(.6 × R(.62499)) = R(.6 × .62499) =
= R(.374994) = .37499 = x1.

Komen we nu met het iteratieproces in een van de punten x1 of x2 terecht, dan convergeert het proces niet langer. Het is dus niet voldoende langs mathematische weg te bewijzen dat |f(x1) − f(x2)| < |x1 − x2|; het blijft nodig hierop in het programma zelf te testen. Is aan de ongelijkheid niet meer voldaan dan beeindigen we het proces; is het verschil niet al te groot dan hebben we de oplossing, anders is er iets fout.

Voor het geval we drie vergelijkingen tegelijk oplossen moet de uitdrukking |x1 − x2| vervangen worden door een functie d ((x1, y1, z1), (x2, y2, z2)), bijv. sqrt ((x1 − x2) ↑ 2 + (y1 − y2) ↑ 2 + (z1 − z2) ↑ 2), of max (|x1 − x2|, |y1 − y2|, |z1 − z2|), als maar geldt:

1. d ((x1, y1, z1), (x2, y2, z2)) = 0 indien en slechts indien
   x1 = x2 ∧ y1 = y2 ∧ z1 = z2.
2. d ((x1, y1, z1), (x2, y2, y2)) ≤
   d ((x1, y1, z1), (x3, y3, z3)) + d ((x2, y2, z2), (x3, y3, z3)) voor alle mogelijke waarden van x1, y1 enz. (een dergelijke functie d heet metriek)

Vaak kan de convergentie van het proces op een andere wijze bewezen worden, bijvoorbeeld als er een monotone functie F bestaat zodat de waarden F(x[i]) met toenemende i kleiner worden, maar toch nooit negatief kunnen worden, terwijl de x[i] in een zeker gesloten gebied zullen liggen. In dit geval testen we of de F(x[i]) inderdaad dalen en de x[i] inderdaad in dat gebied liggen.
Als de bedoeling van het iteratieproces juist is F(x) zo klein mogelijk te krijgen zijn de voorwaarden van monotonie en dat gesloten gebied niet nodig; we zijn dan tevreden als de waarden F(x[i]) naar een minimum convergeren.

3. Gewenste precisie.

Stel dat het door meetonnauwkeurigheden in gegevens niet mogelijk is de exacte waarde van f(x) te bepalen, maar dat hierbij een zekere fout eps gemaakt wordt. I.p.v. f(x) wordt dus f(x) + eps berekend. Veronderstel nu dat bij het iteratieproces een van de waarden x[i] juist de waarde x is waarvoor x = f(x).
We krijgen dan x[i + 1] = f(x[i]) + eps = f(x) + eps = x + eps; met andere woorden: een fout in x ter grootte van eps wordt tenslotte wel ongeveer gemaakt.
We stoppen dus als de fout |x − x[i]| < eps, waarbij x de limiet voorstelt.
Hoe bepalen we die fout in x[i]? We nemen aan dat de fout bij iedere stap een factor r kleiner wordt. We hebben dan: x[i − 2] = x + d; x[i − 1] =
= x + r × d; x[i] = x + r↑2 × d. Hieruit kunnen we x schatten, en dus ook de

fout |x - x[i]|. Omdat het hier maar om een schatting gaat is het vrij riskant de eerste de beste keer dat |x - x[i]| < eps het proces te beeindigen; liever wachten we nog even om te zien of dat zo blijft. Geldt dit dan bijvoorbeeld drie maal achtereen dan beeindigen we alsnog het proces.

Als voorbeeld wordt hier een procedure voor kleinste-kwadratenaanpassing gegeven. Bij de meetpunten x[1] t/m x[n] zijn de waarden y[1] t/m y[n] gemeten, met vermoedelijke meetfout eps[1] t/m eps[n]. We willen nu de parameters p[1] t/m p[m], die gebruikt worden in de berekening van een theoretische functie f(x, p) met partiele afgeleiden naar p[k]: partafg (x, p, k), zo bepalen dat de uitdrukking
U = sqrt (SUM (i, 1, n, ((f (x[i], p) - y[i]) / eps[i]) ↑ 2) / n) minimaal is.
De aanroep van de procedure zou hiervoor kunnen luiden:
Least Squares (i, n, x[i], y[i], eps[i], k, m, p[k], x, f (x, p), partafg (x, p, k)).
De inputparameters x[i], y[i] en eps[i] hangen via Jensen's device af van i (1 t/m n).
p[k] hangt via Jensen's device af van k (1 t/m m) en heeft de volgende functies:
1. inputparameter die bij de aanroep gevuld moet zijn met een beginschatting.
2. outputparameter die na de aanroep de beste gevonden benadering bevat.
3. parameter waar f(x, p) en partafg (x, p, k) van afhangen.
De parameter f (x, p) hangt via Jensen's device af van x en p[1] t/m p[m].
De parameter partafg (x, p, k) hangt via Jensen's device af van x, k en p[1] t/m p[m].
De parameters overeenkomend met i, k, x en p[k] moeten van de vorm <variable> zijn. Het is dus niet mogelijk i.p.v. p[k] te schrijven: if k = 1 then A else r.

Het iteratieproces wordt hier beeindigd zodra aan een der volgende voorwaarden voldaan is:
1. U wordt bij een iteratiestap niet kleiner.
2. U ≤ 1. (In dat geval zijn de benaderingsfouten zo groot als de meetfouten)
3. Drie maal achtereen is een schatting van de limiet van U hoogstens 10 procent lager dan de reeds behaalde waarde.
4. Er zijn reeds 100 iteratiestappen verricht.

Least Squares is een Boolean procedure. Wordt de procedure beeindigd doordat U in niet onaanzienlijke mate stijgt, dan is het resultaat false, anders true.

Een voorbeeld van een aanroep: aanpassing van A × (exp (r × x) - 1) aan 25 gemeten waarden y.

```
p[1]:= 1; p[2]:= .7; comment schattingen van A en r;
Least Squares (i, 25, x[i], y[i], eps[i], k, 2, p[k], x,
p[1] × (exp (p[2] × x) - 1),
if k = 1 then exp (p[2] × x) - 1 else p[1] × p[2] × exp (p[2] × x))
```

Tot slot de procedure:

```
Boolean procedure Least Squares (i, n, xi, yi, epsi, k, npar, pk, x, fx, dfxdpk);
comment gebruikt de niet-locale procedures SYMDET2 en SYMSOL2 (AP 228/229);
value n, npar; integer i, n, k, npar; real xi, yi, epsi, pk, x, fx, dfxdpk;
begin    integer count, maxcount, c, tim, j;
         real small, minwfac, maxrelerr, ssq, ssq1, d1, d2, desfrac;
         real array X, Y, W[1 : n], F, paro, delta[1 : npar],
                    S[1 : (npar + 1) × npar : 2];

tim:= 3; small:= 10-100; minwfac:= 10-3; maxrelerr:= 10-3; desfrac:= .1;
maxcount:= 100;  comment small en minwfac zijn zo gekozen dat
minwfac ↑ 2 ≫ relatieve precisie en
(small × minwfac) ↑ 4 ≫ kleinste positieve real-getal;

begin    real h, max, min;
         max:= small;
         for i:= 1 step 1 until n do
         begin    X[i]:= xi; Y[i]:= yi;
                  w[i]:= h:= abs (epsi); if h > max then max:= h
         end;
         min:= max × minwfac;
         for i:= 1 step 1 until n do
         begin h:= W[i]; W[i]:= 1 / (if h < min then min else h) end
end;
desfrac:= (2 - desfrac) × desfrac; c:= tim; Least Squares:= true;

for count:= 1 step 1 until maxcount do
begin    for k:= (npar + 1) × npar : 2 step -1 until 1 do S[k]:= 0;
         for k:= 1 step 1 until npar do delta[k]:= 0; ssq:= 0;
         for i:= 1 step 1 until n do
         begin    integer t; real df, dy, w;
                  x:= X[i]; w:= W[i]; dy:= (Y[i] - fx) × w;
                  ssq:= dy × dy + ssq; t:= 0;
                  for k:= 1 step 1 until npar do
                  begin    df:= F[k]:= w × dfxdpk;
                           delta[k]:= delta[k] + df × dy;
                           for j:= 1 step 1 until k do
                           begin t:= t + 1; S[t]:= S[t] + F[j] × df end
                  end
end;
```

```
if ssq ≤ n then go to exit;
if count > 1 then
begin   d1:= ssq1 - ssq;
        if d1 ≤ 0 then
        begin   if d1 < 0 then for k:= 1 step 1 until npar do
                pk:= paro[k];
                Least Squares:= ssq ≤ (1 + maxrelerr) × ssq1;
                go to exit
        end;
        if count > 2 then
        begin   if d1 × d1 > (d2 - d1) × desfrac × ssq then
                c:= tim else
                if c > 1 then c:= c - 1 else go to exit
                end;
                d2:= d1
        end;

        SYMDET2 (S, npar, 1); SYMSOL2 (S, npar, 1, delta);
        for k:= 1 step 1 until npar do
        begin paro[k]:= pk; pk:= paro[k] + delta[k] end;
        ssq1:= ssq
    end;
exit:
end
```