TECHNISCHE HOGESCHOOL EINDHOVEN
Afdeling Algemene Wetenschappen
Onderafdeling der Wiskunde

PROGRAMMEREN met PASCAL

1983

WF

Technische Hogeschool
Eindhoven

Dictaatnummer 2.320
Prijs f. 9,00

Onderafdeling der Wiskunde en Informatica

Programmeren met Pascal

TECHNISCHE HOGESCHOOL EINDHOVEN

Onderafdeling der Wiskunde en Informatica

Programmeren met Pascal

1983

INHOUD

		Pagina
0.	Inleiding	1
1.	Verband tussen procesbeschrijving en toestandsbeschrijving	10
2.	De constructie van repetities	29
3.	Iets over efficiëntie	42
4.	Datatypen en declaraties; expressies en de assignment statement	47
5.	Invoer en uitvoer: communicatie met de "buitenwereld"	63
6.	Verwerking van een invoerrij	69
7.	Samengestelde variabelen: records en arrays	78
8.	Procedures	101
9.	Functies	114
10.	Procedures en functies als parameters	119
11.	Recursie	123
12.	Stepwise refinement	131

0. Inleiding

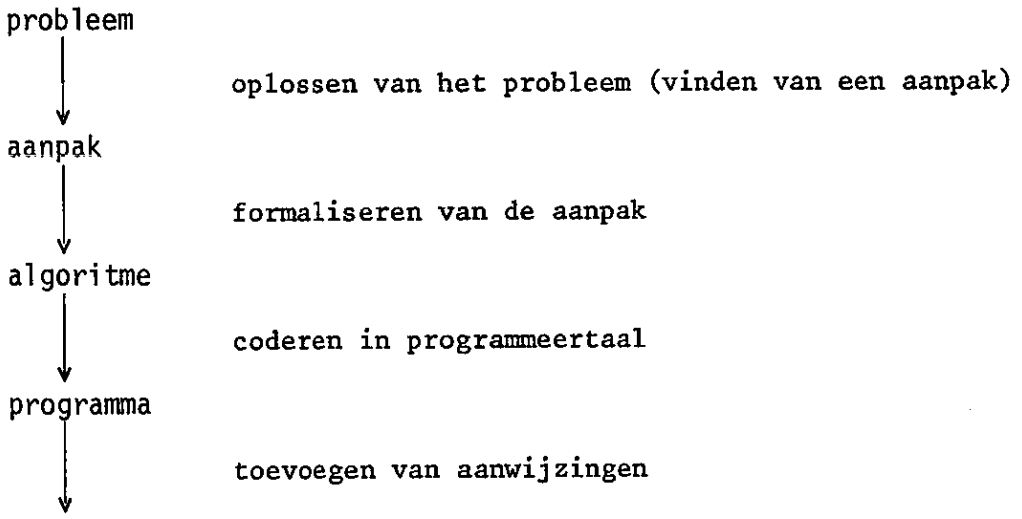
In de *informatica* houdt men zich bezig met het verwerken van gegevens, in het bijzonder met behulp van een computer. Het centrale thema van de informatica is het opstellen van *algoritmen* die manipuleren met *gegevens*. Deze algoritmen moeten worden vastgelegd in een *programmeertaal* om door een *rekenmachine* verwerkt te kunnen worden. De verwerking van een algoritme moet leiden tot de oplossing van een gesteld probleem. *Programmeren* is het geheel van activiteiten dat nodig is om vanuit een gegeven probleem te komen tot een verwerkbaar algoritme.

Een *algoritme* is een rij instructies die bij verwerking leidt tot een gewenst effect. Dit soort rijen instructies is ons uit het dagelijks leven overbekend. Denk bijvoorbeeld maar aan breipatronen, recepten en montagevoorschriften. Het woord algoritme zullen we reserveren voor een rij instructies, die in principe uitgevoerd kan worden door een rekenmachine. Bij een algoritme spelen twee partijen een rol: de opsteller van het het algoritme, de programmeur, en de uitvoerder van het algoritme, de rekenmachine. Tussen opsteller en uitvoerder moet overeenstemming bestaan ten aanzien van de notatie van de algoritmen en ten aanzien van het effect van de uitvoering van de instructies. Deze overeenstemming wordt bereikt door het gebruik van een programmeertaal. Zoals bij iedere taal kunnen we ook bij een programmeertaal de syntaxis (de notatie van een constructie) en de semantiek (de betekenis van een constructie) onderscheiden. De bovenbedoelde overeenstemming geldt ten aanzien van de syntaxis en de semantiek van de gebruikte programmeertaal. Er is ook een groot verschil tussen de opsteller en de uitvoerder van een algoritme. De opsteller moet

(omdat hij een probleem moet oplossen) het effect kennen van zijn algoritme, zonder dat hij het uitvoert. De uitvoerder moet het algoritme alleen kunnen uitvoeren, zonder te "weten" wat er gebeurt.

Als een algoritme geschreven is in een programmeertaal, wordt het een *programma* genoemd. Bij de verwerking van een programma door een rekenmachine moet het programma veelal vergezeld gaan van aanwijzingen voor de rekenmachine over de wijze van verwerking. Zo moet aangegeven worden in welke programmeertaal het programma geschreven is omdat de rekenmachine verschillende programmeertalen "kent".

Recapitulerend kunnen we de werkzaamheden van de programmeur als volgt in beeld brengen:



De nadruk in deze tekst zal liggen op het ontwerpen van algoritmen, het vinden van algoritmen bij gegeven problemen. Soms zal het toepassingsgebied (het vakgebied waaruit het probleem afkomstig is) ons een eerste aanzet tot de oplossing geven, vaak ook is dit niet het geval. Kennis van het toepassingsgebied kan nuttig zijn; in onze beschouwingen zullen toepassingsgebieden een ondergeschikte rol spelen. Het algoritme moet

gecodeerd worden in een programmeertaal. De hier te gebruiken programmeertaal is Pascal. Naast het ontwerpen van algoritmen komt dan ook een aantal facetten van Pascal aan de orde. Er zij echter op gewezen dat de eigenschappen van de te gebruiken programmeertaal bij het ontwerp van het algoritme een ondergeschikte rol moeten spelen.

> "The primary aim of programming courses should not be to teach perfection in the knowledge and use of all features and idiosyncrasies of a specific language."
>
> N. Wirth

> "Thus there is a most important distinction to be drawn between an algorithmic language intended to assist in the definition, design, development and documentation of a program, and the programming language in which the program is eventually conveyed to a computer."
>
> C.A.R.Hoare

Als we iemand een object, een mechanisme of iets dergelijks willen laten realiseren, dan kunnen we het object op twee manieren beschrijven. De eerste manier geeft aan *wat* de eigenschappen van het te realiseren object moeten zijn. De tweede manier geeft aan *hoe* het object geconstrueerd moet worden. De eerste beschrijvingswijze noemen we *toestandsbeschrijving* de tweede noemen we *procesbeschrijving*. (De reden voor het gebruik van de woorden toestand en proces zal later aangegeven worden.) In veel beschrijvingen worden procesbeschrijvingen en toestandbeschrijvingen door elkaar gebruikt. Zo beschrijft een architect in een bestek zowel wat er moet gebeuren als hoe bepaalde constructies gerealiseerd moeten worden.

Vaak ook zien we dat bij een procesbeschrijving toestandsbeschrijvingen als een soort commentaar ter verduidelijking worden opgenomen. Bij een montagevoorschrift bijvoorbeeld, staan vaak tekeningen die aangeven wat de status van het te realiseren voorwerp moet zijn, nadat bepaalde delen van de procesbeschrijving (het montagevoorschrift) zijn uitgevoerd.

Een algoritme heeft de vorm van een procesbeschrijving. Bij het ontwerpen van een algoritme zullen we echter veelvuldig gebruik maken van toestandsbeschrijvingen en deze als commentaar aan het algoritme toevoegen. In het onderstaande voorbeeld staat dit commentaar tussen accolades. Ook het stellen van een programmeerprobleem kan (moet) gebeuren met behulp van toestandsbeschrijvingen.

Voorbeeld

Er moeten twee getallen, i1 en i2 "ingelezen" worden waarvoor geldt $i1 \geq 0$ en geheel en $i2 > 0$ en geheel. Bij deze twee getallen moeten twee andere getallen O1 en O2 bepaald en "geschreven" worden, waarvoor geldt: O1 en O2 geheel en $i1 = O1.i2 + O2$ en $0 \leq O2 < i2$ (O1 is het gehele quotiënt van i1 en i2 en O2 is de rest bij deling van i1 door i2).
Na iedere opdracht staat in de tekst een beschrijving van de toestand zoals die heerst na uitvoering van die opdracht. Later zullen we zien hoe we deze toestandsbeschrijvingen kunnen gebruiken om de procesbeschrijving te vinden.

```
program voorbeeld 1(input,output);
{input = <i1,i2>,i1 ≥ 0 en geheel, i2 > 0 en geheel,
 output = <>}
var x,y,q,r : integer;
```

```
{de waarden van x,y,q en r, die nu nog ongedefinieerd zijn, zullen gehele
 getallen zijn}
begin read(input,x);
      {x = i1, x ≥ 0 en geheel; input = <i2>}
      read(input,y);
      {x = i1,x ≥ 0 en geheel; y = i2, y > 0 en geheel; input = <>}
      q:= 0; r:= x;
      {x = q.y+r en r ≥ 0}
      while r >= y do
        begin {x = q.y+r en r ≥ y > 0}
              r:= r-y; q:= q+1
              {x = q.y+r en r ≥ 0}
        end
      {x = q.y+r en 0 ≤ r < y}
      write(output,x); write(output,y);
      write(output,q); write(output,r)
      {output = <i1,i2, gehele quotiënt van i1 en i2, de rest bij deling
       van i1 door i2>}
end
```

De mate van gedetailleerdheid die in een procesbeschrijving moet worden gebruikt, hangt af van de uitvoerder van de procesbeschrijving. Sommige mensen hebben aan een half woord voldoende, voor anderen moet je tot in alle details uitleggen wat er moet gebeuren. Het niveau van gedetailleerdheid van een programma wordt bepaald door de te gebruiken programmeertaal; deze heeft een bepaald repertoire van instructies en kent bepaalde

soorten gegevens waarmee gemanipuleerd kan worden. Bij het ontwerpen van een algoritme heeft het echter zin om eerst het op te lossen probleem op te splitsen in (min of meer onafhankelijke) deelproblemen, de relaties tussen deze deelproblemen vast te leggen en daarna pas deze deelproblemen tot in detail op te lossen. Deze aanpak bij het ontwerpen van algoritmen wordt *gestructureerd* programmeren, *top-down* programmeren, of programmeren door middel van *stepwise refinement* genoemd. Er wordt dus vanuit een abstract (weinig gedetailleerd) algoritme naar een concreet programma (met alle benodigde details) toegewerkt.

"There once were two watchmakers, named Hora and Tempus, who manufactured very fine watches. Both of them were highly regarded, and the phones in their workshops rang frequently - new customers were constantly calling them. However, Hora prospered, while Tempus became poorer and poorer and finally lost his shop. What was the reason? The watches the men made consisted of about 1.000 parts each. Tempus had so constructed his that if he had one partly assembled and had to put it down - to answer the phone, say - it immediately fell to pieces and had to be reassembled from the elements. The better the customers liked his watches, the more they phoned him and the more difficult it became for him to find enough uninterrupted time to finish a watch.

The watches that Hora made were no less complex than those of Tempus. But he had designed them so that he could put together subassemblies of about ten elements each. Ten of these subassemblies, again, could be put together into a larger subassembly; and a system of ten of the latter subassemblies constituted the whole watch. Hence, when Hora had to put down a partley assembled watch in order to answer **the** phone, he lost only a small part of his work, and he assembled his watches in only a fraction of the man-hours it took Tempus.

It is rather easy to make a quantitative analysis of the relative difficulty of the tasks of Tempus and Hora: Suppose the probability that an interruption will occur while a part is being added to an incomplete assembly is p. Then the probability that Tempus can complete a watch he has started without interruption is $(1 - p)^{1000}$ - a very small number unless p is 0.001 or less. Each interruption

will cost, on the average, the time to assempble 1/p parts (the expected number assembled before interruption). On the other hand, Hora has to complete 111 subassemblies of ten parts each. The probability that he will not be interrupted while completing any one of these is $(1-p)^{10}$, and each interruption will cost only about the time required to assemble five parts.

Now if p is about 0.01 - that is, there is one chance in a hundred that either watchmaker will be interrupted while adding any one part to an assembly - then a straightforward calculation shows that it will take Tempus, on the average, about four thousand times as long to assemble a watch as Hora.

We arrive at the estimate as follows:

1. Hora must make 111 times as many complete assemblies per watch as Tempus; but
2. Tempus will lose on the average 20 times as much work for each interrupted assembly as Hora (100 parts, on the average, as against 5); and
3. Tempus will complete an assembly only 44 times per million attempts ($0.99^{1000} = 44 \times 10^{-6}$), while Hora will complete nine out of ten ($0.99^{10} = 9 \times 10^{-1}$). Hence, Tempus will have to make 20,000 as many attempts per completed assembly as Hora. $(9 \times 10^{-1})/(44 \times 10^{-6}) = 2 \times 10^{4}$. Multiplying these three ratios, we get $1/111 \times 100/5 \times 0.99^{10}/0.99^{1000} = 1/111 \times 20 \times 20{,}000 \sim 4{,}000$."

H.A. Simon
The Architecture of Complexity.

Bij procesbeschrijvingen als recepten en breipatronen laat de exactheid vaak veel te wensen over. De ervaring en welwillendheid van de uitvoerder van de procesbeschrijving zorgen ervoor dat het betreffende werkstuk toch gerealiseerd wordt. Een programma moet echter zeer precies zijn, zowel wat betreft de keuze van instructies om een bepaald effect te bereiken (ontwerp van het algoritme), als wat betreft de notatie (codering in een programmeertaal). Om precies aan te kunnen geven wat de notatie van een constructie in Pascal moet zijn, gebruiken we de volgende definitievorm (*BNF-notatie* genoemd):

<woonplaats met postcode> ::= <postcode> <spatie> <plaatsnaam>

Deze definitie legt vast dat de constructie "woonplaats met postcode" bestaat uit de constructie "postcode", gevolgd door de constructie "spatie", gevolgd door de constructie "plaatsnaam". Om nu te weten hoe een "woonplaats met postcode" wordt geschreven, moeten we van de drie deelconstructies de notatie kennen (we weten alleen nog maar de onderlinge volgorde van de drie constructies). De definitie van "postcode" zou kunnen luiden:

<postcode> ::= <vier cijfers> <spatie> <twee hoofdletters>

met

<twee hoofdletters> ::= <hoofdletter> <hoofdletter>

waarin bijvoorbeeld

<hoofdletter> ::= A|B|C|D|E

Deze definitie legt vast dat de constructie "hoofdletter" óf een A is, óf een B, óf een C, óf een D, óf een E. De verticale streep geeft een keuze aan. Voor een deel van de constructie "woonplaats met postcode" zijn we

nu aangeland op het uiteindelijke notatieniveau; de A, B, C, D en E zijn zogenaamde eindtermen.
De definitie voor "plaatsnaam" zou kunnen zijn:

<plaatsnaam> ::= <hoofdletter> <rij kleine letters>

met

<rij kleine letters> ::= <kleine letter>|
<kleine letter> <rij kleine letters>

Dit is een voorbeeld van een zogenaamde recursieve definitie: rechts van "::=" staat weer het te definiëren begrip. Een "rij kleine letters" is ôf een "kleine letter" (bijvoorbeeld a, als "kleine letter" gedefinieerd is door <kleine letter> ::= a|b|c|d|e) ôf een "kleine letter", bijvoorbeeld b, gevolgd door een "rij kleine letters", bijvoorbeeld a; ba is dus een voorbeeld van een "rij kleine letters". In feite bestaat een "rij kleine letters" uit een of meer kleine letters achter elkaar. Om de totale notatie van "woonplaats met postcode" te kennen moeten nog gedefinieerd worden de constructies "spatie" en "vier cijfers". De laatste is duidelijk, de eerste is:

<spatie> ::= ␣

Omdat in de definities spaties geen betekenis hebben en dus vrij gebruikt mogen worden, moeten we een apart teken afspreken voor de spatie.
In plaats van een definitie als

<rij kleine letters> ::= <kleine letter>|
<kleine letter> <rij kleine letters>

schrijft men ook wel:

<rij kleine letters> ::= <kleine letter>{<kleine letter>}

met als betekenis van de accolades: herhaling van 0 of meer keren van alles wat er tussen staat.

Een andere afkorting is om een constructie al dan niet op te nemen:

<getal> ::= [<teken>] <cijfer>{<cijfer>}

Een "getal" kan met een "teken" beginnen.

1. Verband tussen procesbeschrijving en toestandsbeschrijving

Een algoritme (programma) is een procesbeschrijving, die bij uitvoering leidt tot operaties op gegevens. Een basisbegrip bij de uitvoering van operaties op gegevens is het begrip *actie* of handeling. *Een actie is een gebeurtenis (met een eindige tijdsduur), die een wel gedefinieerd effect realiseert.* Teneinde het effect van een actie te kunnen aangeven, gaan we er van uit dat een actie zich afspeelt in een bepaalde omgeving. Deze omgeving verkeert in een zekere *toestand* (state of the computation). Het effect van een actie kan dan worden vastgesteld aan de hand van het verschil tussen de toestand vóór en de toestand ná de actie. De actie is de oorzaak van deze toestandstransformatie.

De omgeving waarin een actie plaatsvindt, kan worden gekarakteriseerd door een aantal voor de actie relevante grootheden, *variabelen* genoemd. De functie van een variabele kan omschreven worden als het onthouden van (een deel van) de toestand; een variabele is te beschouwen als een geheugenelement waarin een waarde geplaatst kan worden die dan later kan worden gebruikt. We kunnen nu het begrip toestand nader omschrijven: *Een toestand is een aantal variabelen, elk met zijn waarde.*

Met een variabele is verbonden een voor deze variabele karakteristieke *waardenverzameling* met daarop gedefinieerde *operaties*; zo'n waardenverzameling met zijn operaties wordt een *type* genoemd. We beperken ons hier voorlopig tot de volgende Pascal-typen:

- integer: de waardenverzameling is (een deelverzameling van) de verzavan de gehele getallen;
- real : de waardenverzameling is (een deelverzameling van) de verzameling van de reële getallen;

- boolean : de waardenverzameling is de verzameling van de logische waarden: {true, false};
- char : de waardenverzameling is een verzameling van karakters.

Een variabele heeft in een algoritme een *naam*; in Pascal wordt deze naam geschreven als een letter gevolgd door nul of meer letters of cijfers:

<naam variabele> ::= <letter>|
<naam variabele> <letter>|
<naam variabele> <cijfer>

We kunnen deze definitie ook schrijven als

<naam variabele> ::= <letter>{<letter>|<cijfer>}

Iedere variabele in een algoritme wordt geïntroduceerd met een bepaalde bedoeling, iedere variabele heeft een *betekenis*. De betekenis van de geïntroduceerde variabelen tezamen met hun actuele waarde bepalen de interpretatie van een heersende toestand.

Een toestand wordt gekarakteriseerd door een rij variabelen. Een actie veroorzaakt een toestandstransformatie, d.w.z. de verandering van de waarde van één of meer variabelen. Een algoritme bepaalt de waarden van variabelen.

Stel dat in de begintoestand geldt:

- de variabelen x en y hebben een waarde:
  $x \geq 0$ en geheel, $y > 0$ en geheel
- de variabelen q en r (die gehele waarden kunnen aannemen) hebben geen waarde;

en dat de gevraagde eindtoestand is:

- x en y hebben hun oorspronkelijke waarde;
- q en r hebben zodanige waarden dat geldt:
  $x = q \cdot y + r$ en $0 \leq r < y$

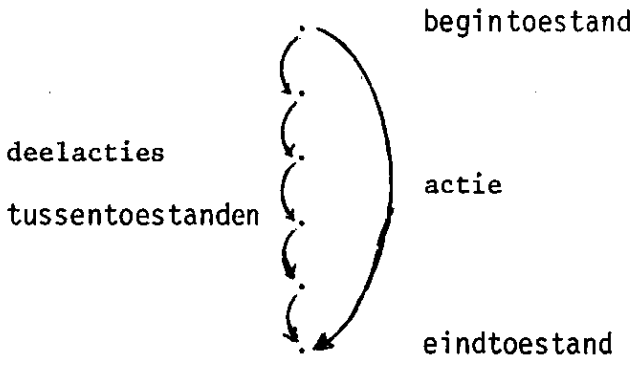
Er wordt nu gevraagd een zodanig algoritme te bedenken (misschien één instructie?), dat de uitvoering hiervan leidt tot een actie die de gegeven begintoestand laat overgaan in de gevraagde eindtoestand:

{x ≥ 0 en geheel, y > 0 en geheel, q en r onbepaald}

bepaal de waarden van q en r

{x = q.y + r, 0 ≤ r < y, x en y onveranderd}

Vaak kan een actie opgevat worden als een rij *deelacties*; uitgaande van een begintoestand leiden achtereenvolgende toestandstransformaties van de deelacties dan - via tussentoestanden - tot een eindtoestand, welke gelijk is aan de toestand die bereikt wordt als de actie als één geheel wordt beschouwd.



Als de actie wordt gezien als een rij deelacties spreekt men van een *proces*. We zijn hier alleen geïnteresseerd in processen waarvan de deelacties strikt na elkaar plaatsvinden; men noemt deze processen wel sequentiële processen.

Iedere actie is op te vatten als een deelactie van een omvattend proces. Omgekeerd is een actie op te splitsen in deelacties. Sommige acties kunnen niet meer opgesplitst worden in deelacties, we spreken dan van *elementaire acties*.

Welke acties elementair zijn wordt bepaald door de gebruikte programmeertaal.

De overgang van proces naar actie geeft ons de mogelijkheid tot abstractie. Bij een actie zijn we alleen geïnteresseerd in het uiteindelijke effect, in wat er gebeurt. Beschouwen we dezelfde gebeurtenis als proces, dan zijn we ook geïnteresseerd hoe de gebeurtenis zich afspeelt, via welke tussentoestanden - vanuit de begintoestand - de eindtoestand bereikt wordt. Het verloop van het proces is geheel vastgelegd door de opeenvolging van toestanden.

Als we nu een algoritme moeten maken voor de oplossing van een probleem, gaan we er in eerste instantie van uit dat de uitvoerder bepaalde machtige acties kan laten plaatsvinden. De instructies voor deze machtige acties splitsen we daarna herhaald op in instructies voor deelacties todat we tenslotte uitkomen bij de instructies (statements) van de programmeertaal. Zo zou "bepaal de waarden van q en r" (zie terug) als volgt opgesplitst kunnen worden:

```
{x ≥ 0 en geheel, y > 0 en geheel}
maak q gelijk aan 0
maak r gelijk aan x
{x = q·y + r, r ≥ 0}
zolang r ≥ y
        doe {x = q·y + r, r ≥ 0, r ≥ y}
            verklein r met behoud van
            x = q·y + r en r ≥ 0
            {x = q·y + r, r ≥ 0}
{x = q·y + r, 0 ≤ r < y}
```

Of, voor zover mogelijk, direct in Pascal:

```
{x ≥ 0 en geheel, y > 0 en geheel}
q := 0; r := x;
{x = q·y + r, r ≥ 0}
while r ≥ y
      do {x = q·y + r, r ≥ 0, r ≥ y}
         "verklein r met behoud van
          x = q·y + r en r ≥ 0"
         {x = q·y + r, r ≥ 0}
{x = q y + r, 0 ≤ r < y}
```

Omdat we zo goed als altijd de gegevens (uit de begintoestand, hier de waarden van x en y) niet veranderen, hebben we dit niet meer in de toestandsbeschrijvingen opgenomen.

De acties, behorende bij q := 0 en r := x zouden gelijktijdig mogen plaatsvinden. Omdat we ons beperken tot sequentiële processen, moeten we een keuze doen voor de volgorde.

Als r met y wordt verminderd in de herhaling (er geldt aan het begin daarvan: r ≥ y), dan blijft gelden r ≥ 0. Wil ook x = q.y + r blijven gelden, dan is de enige mogelijkheid daartoe, omdat x en y onveranderd zouden blijven, dat q met 1 verhoogd wordt.

Eindigt de herhaling? Er geldt y > 0 en r wordt iedere keer met y verminderd. Er moet dus op een gegeven moment wel gaan gelden r < y.

We hebben nu als algoritme gevonden:

```
{x ≥ 0 en geheel, y > 0 en geheel}
q := 0; r := x;
{x = q.y + r, r ≥ 0}
while r >= y do
      begin {x = q.y + r, 0 ≤ r, r ≥ y}
            r := r - y; q := q + 1
            {x = q.y + r, 0 ≤ r}
      end
{x = q.y + r, 0 ≤ r < y}
```

Om tot een programma (in Pascal) te komen moeten nu nog toegevoegd worden

- het lezen van de twee getallen
- het wegschrijven van de resultaten,
- het introduceren (declareren) van de gebruikte variabelen,
- en enige notationele zaken die voor de taal Pascal vereist worden.

Zo krijgen we het eerder gegeven programma.

Opmerking

In Pascal bestaan voor het type integer (en x en y zijn integer) de operaties div en mod. Met behulp van deze operaties zouden we als "equivalent" algoritme kunnen schrijven:

```
{x ≥ 0 en geheel, y > 0 en geheel}
begin q := x div y; r := x mod y end
{x = q.y + r, 0 ≤ r < y}
```

(einde opmerking)

De acties en processen die wij beschouwen, spelen zich af binnen de rekenmachine. Het zijn gebeurtenissen die zich in de tijd afspelen, ze zijn dynamisch. De statische, tijdloze beschrijving van een actie, de instructie, wordt een *statement* genoemd.

Een actie heeft tot gevolg dat één of meer variabelen een (andere) waarde krijgen. Het toekennen van een waarde aan een variabele moet tot het actierepertoire behoren. De statement voor deze actie wordt de *assignment statement* genoemd.

<assignment statement> ::= <variabele> := <expressie>

Het resultaat van de assignment statement is dat de waarde van de expressie is toegekend aan de variabele.

Voorbeelden: q := 0

r := r - y

De waarde van een variabele in een expressie is de waarde van die variabele in de heersende toestand. Door de assignment actie krijgt de variabele in het linkerlid een (nieuwe) waarde, waardoor dus de toestand verandert.

Naast statements kennen programmeertalen *compositiemethoden (besturingsstructuren)* om de volgorde van de acties, behorend bij statements, vast te leggen. We onderscheiden drie methoden van compositie. Hieronder volgt de notatie:

1. *sequentiële compositie*: aaneenrijging van een aantal acties tot een geordende rij van acties.

   <rij statements> ::= <statement>|<statement>;<rij statements>

   vb. q := 0; r := x

   Als een aantal deelacties samen één actie vormen, die ook als eenheid in de tekst moet worden aangegeven, wordt de *compound statement* gebruikt:

   <compound statement> ::= begin <rij statements> end

   vb. begin r := r - y; q := q + 1 end

2. *conditionele compositie*: het uitvoeren van een statement onder een bepaalde conditie. De conditie zal de vorm hebben van een "logische expressie", dit is een expressie die een test op de heersende toestand voorstelt (vb. x + y > 0, a ≠ b), en die de waarde true of false oplevert. We onderscheiden drie vormen van conditionele compositie:

2a. *conditional statement*

<conditional statement> ::=

if <logische expressie> then <statement>

Het effect kan als volgt beschreven worden:

- bereken de waarde van de logische expressie
- als deze waarde true is voer dan de statement uit

vb. if x < 0 then x := -x

2b. *selection statement*

<selection statement> ::=

if <logische expressie> then <statement> else <statement>

Het effect kan als volgt beschreven worden:

- bereken de waarde van de logische expressie
- als deze waarde true is voer dan de statement na then uit, is de waarde false voer dan de statement na else uit

vb. if a > b then max := a else max := b

2c. *case statement*

<case statement> ::=

case <expressie> of <rij case elementen> end

met

<rij case elementen> ::= <case element> |

<case element>; <rij case elementen>

<case element> ::= <case label lijst>:<statement>

<case label lijst> ::= <label>|<label>,<case label lijst>

Het effect kan als volgt beschreven worden:

- de waarde van de expressie wordt berekend
- de statement, die volgt op het label dat gelijk is aan de waarde van de expressie, wordt uitgevoerd.

vb.: case i of

```
    0,1 : x := 0;
    2   : x := y;
    4,5 : x := x+y;
    3   : x := x-y
  end
```

3. *herhalingscompositie*: het herhaald uitvoeren van een statement onder een bepaalde voorwaarde.

Ook hier onderscheiden we drie vormen:

3a. *while statement*

<while statement> ::=

while <logische expressie> do <statement>

Effect:

(1) de waarde van de logische expressie wordt berekend

(2) als deze waarde true is wordt de statement uitgevoerd en er wordt verder gegaan met (1)

vb.: while k < n do k := 2 * k

3b. *repeat statement*

<repeat statement> ::=

repeat <rij statements> until <logische expressie>

Effect:

(1) de rij statements wordt uitgevoerd

(2) de waarde van de logische expressie wordt berekend

(3) als deze waarde false is wordt er verder gegaan met (1)

vb. repeat z:= z+y; x:= x-1 until x = 0

3c. *for statement*

<for statement> ::=

for <variabele> := <expressie> to <expressie> do <statement>

Effect:

(1) de waarden van de twee expressies worden bepaald.

(2) als de waarde van de eerste expressie (vóór to) niet groter is dan de waarde van de tweede expressie wordt er met (3) verder gegaan en anders gebeurt er niets

(3) de variabele wordt gelijk gemaakt aan de waarde van de eerste expressie

(3.1) als de waarde van de variabele niet groter is dan de waarde van de tweede expressie, dan wordt (3.2) uitgevoerd (anders niets)

(3.2) de statement wordt uitgevoerd, de variabele krijgt de waarde van zijn opvolger en er wordt verder gegaan met (3.1)

vb.: for i := 1 to 10 do s := s + i

Opmerkingen.

- Overal waar in de bovenstaande definities <statement> staat, mag elk van de gedefinieerde constructies staan. Dit zijn ook statements in die zin dat ze de toestand transformeren. Deze transformaties zijn echter

steeds het gevolg van de in de constructies voorkomende assignment statements.

- Door de zojuist genoemde combinatiemogelijkheid kan de niet eenduidig interpreteerbare constructie

```
if a ≠ b then if a ≠ c then x := 1 else x := 2
```

ontstaan. In Pascal is hiervan de interpretatie

```
if a ≠ b
   then begin if a ≠ c then x := 1 else x := 2 end
```

Het effect van de andere interpretatie kan bereikt worden door te schrijven:

```
if a ≠ b then begin if a ≠ c then x := 1 end
         else x := 2
```

In al dit soort gevallen waarbij de taal een willekeurige maar vaste keuze doet is het beter om de constructie zodanig te schrijven, dat men niet afhankelijk is van de keuze.

- Om ingewikkelde constructies te voorkomen is het vaak beter om bij het ontwerp van een algoritme kleine stappen te maken (die in de tussentoestanden worden vastgelegd), dan te proberen in één keer de overgang van de begintoestand naar de gewenste eindtoestand te bewerkstelligen.

Stel dat van de variabelen a, b, c en d (die een waarde hebben) de grootste waarde bepaald moet worden en aan de variabele max moet worden toegekend. Dit kan met een constructie als

```
if a > b
   then begin if a > c
                then begin if a > d then max := a
                                    else max := d
                     end
                else begin if c > d then max := c
                                    else max := d
                     end
        end
   else begin if b > c
                then begin if b > d .....
```

Als steeds twee waarden worden vergeleken en het resultaat m.b.v. max wordt vastgelegd, wordt het geheel veel eenvoudiger:

```
max := a; {max ≥ a}
if b > max then max := b; {max ≥ a, max ≥ b}
if c > max then max := c; {max ≥ a, max ≥ b, max ≥ c}
if d > max then max := d; {max ≥ a, max ≥ b, max ≥ c, max ≥ d}
```

- "repeat S until L" is equivalent met "S; while not L do S" waarin not L de ontkenning is van L, d.w.z. als L de waarde true heeft, heeft not L de waarde false , en omgekeerd. Bij de herhaling met de "repeat" wordt de statement S dus altijd ten minste eenmaal uitgevoerd, bij de "while" is dat niet het geval.
- Een punt van zorg bij de repetities is de eindigheid. Als de begintoestand zodanig is dat L in "while L do S" de waarde false heeft, wordt S geen enkele keer uitgevoerd. Is de begintoestand zodanig dat L de waarde true heeft in het begin, dan zal de statement S zodanig moeten

zijn dat L door het herhaald uitvoeren van S eens false wordt. De uitvoering van "while x > 0 do y := y + 1" heeft geen enkele actie tot gevolg of eindigt niet.

- Voor het contact met de buitenwereld staan ons de statements "read (<variabele>)" en "write (<expressie>)" ter beschikking (ook wel geschreven als read (input,<variable>) en write (output,<expressie>). Uitvoering van de read heeft tot gevolg dat de eerstvolgende waarde uit een zogenaamde invoerrij (met de naam input) wordt toegekend aan de variabele (de toestand verandert). Uitvoering van de "write" plaatst een kopie van de waarde van de expressie in een z.g. uitvoerrij (met de naam output).

  Stel dat de invoerrij bestaat uit een rij positieve getallen afgesloten door een 0. Schrijf een algoritme dat het maximum van de positieve getallen in de uitvoerrij plaatst:

```
m := 0; read(g);
{m is het maximum van alle getallen uit de invoerrij, voorafgaand
 aan g}
while g <>0 do
   begin if g > m then m := g;
         read(g)
         {m is het maximum .....  aan g}
   end;
{m  is het maximum ..... aan g en g = 0}
write(m)
```

(einde opmerkingen).

De mens moet een programma kunnen begrijpen zonder het uit te voeren, de rekenmachine moet een programma kunnen uitvoeren zonder het te begrijpen. Hierboven is het effect van de statements uitgelegd in termen van machine-

reacties. Om het effect van een programma te kennen, zou je het als een machine moeten uitvoeren. We willen echter het effect van statements begrijpen (en vastleggen) in termen van toestandstransformaties.

We beginnen met een voorbeeldje. Gevraagd wordt een algoritme dat bij een gegeven waarde van n (geheel en ≥ 0) de faculteit van n bepaalt:

$\{n \geq 0,$ geheel$\}$ algoritme $\{f = n!\}$

Voor de faculteit geldt:

$0! = 1$ en $n! = n * (n-1)!$ voor $n > 0$.

Uit $0!$ kan $1!$ berekend worden, hieruit $2!$ enzovoorts.

Stel dat de variabele i aangeeft van welke waarde als laatste de faculteit berekend is en dat f deze faculteitswaarde is. Het algoritme wordt dan:

```
{n ≥ 0 en geheel}
i := 0; f := 1; {f = i!}
while i<> n
      do {f = i!, i ≠ n}
         "verhoog i, maar zorg dat f = i! blijft gelden"
         {f = i!}
{f = i! en i = n}
```

(In Pascal is <> het teken voor de ongelijkheid.)

Om de repetitie te beëindigen moet de statement na do de toestand veranderen (i ophogen). Maar belangrijk bij de repetitie is ook dat een deel van de (beschrijving van de) toestand, in ons geval $f = i!$, blijft gelden. Deze beschrijving van het niet wijzigende deel van de toestand wordt de *invariante relatie* genoemd. De uitwerking van "verhoog .... gelden" moet dus voor twee zaken zorgen: 1. de repetitie eindigt; 2. de invariant blijft gelden. Als we trachten de repetitie eindig te maken door

i met 1 op te hogen, dan geldt:

```
{f = i!, i ≠ n}
i := i + 1
{f = (i - 1)!}
```

De invariant kan hersteld worden door f met i te vermenigvuldigen:

```
{f = (i - 1)!}
f := f * i
{f = i!}
```

Zo hebben we als algoritme gevonden:

```
{n ≥ 0 en geheel}
i := 0; f := 1; {f = i!}
while i <>n do
   begin {f = i!, i ≠ n}
         i := i + 1;
         f := f * i
         {f = i!}
   end
{f = i! en i = n}
```

Laten we eens kijken naar een aantal constructies en de transformatie van de toestand die zij bewerkstelligen. Om het effect van een statement (of heel algoritme) te beschrijven, doen we een uitspraak P over de begintoestand en een uitspraak Q over de eindtoestand. Met

$$\{P\}S\{Q\}$$

geven we aan dat als in de begintoestand P (de *preconditie*) geldt, dan geldt na afloop van de actie S een toestand waarvoor Q (de *postconditie*)

geldt. P en Q leggen het effect van S vast.

Omgekeerd moeten we bij een programmeerprobleem bij gegeven P en Q de S vinden; P en Q vormen dan de *specificatie* van S.

We zullen nu eerst de pre- en postconditie gebruiken om het effect (de *semantiek*) van enkele constructies vast te leggen. In het volgende hoofdstuk komt het gebruik van de pre- en postconditie voor het vinden van een algoritme aan de orde.

We zullen P en Q ook vaak de begin- respectievelijk de eindrelatie noemen.

Intermezzo

Voor we de semantiekbeschrijvingen bekijken, eerst iets over uitspraken. Uitspraken zijn logische expressies. In logische expressies kunnen constanten, variabelen en relaties ($x < y$, $a \neq 0$, e.d.) als operand optreden. De operatoren die hier gebruikt zullen worden zijn:

and (ook wel genoteerd als $\wedge$), or (ook wel genoteerd als $\vee$) en

not (ook wel genoteerd als $\neg$) . Omdat de operanden slechts twee waarden kunnen hebben, true en false, kan het resultaat van de operatoren eenvoudig in een tabel weergegeven worden:

én

a	b	a and b
true	true	true
true	false	false
false	true	false
false	false	false

a and b heeft de waarde true als zowel a als b de waarde true hebben, anders heeft a and b de waarde false.

óf

a	b	a or b
true	true	true
true	false	true
false	true	true
false	false	false

a or b heeft de waarde false als zowel a als b de waarde false hebben, anders heeft a or b de waarde true.

niet	a	not a
	true	false
	false	true

Deze operaties worden gebruikt voor uitspraken, zoals de geïntroduceerde P en Q, en ook voor de logische expressies die in de statements voorkomen. Einde Intermezzo.

Semantiekregels.

De onderstaande regels geven de pre- en postcondities van de constructies. Deze condities behoeven de constructies niet volledig vast te leggen, maar de regels zijn voor het gebruik dat wij ervan willen maken voldoende.

- *assignment statement*

  {Q(E(x))} x := E(x) {Q(x)}

  Als voor de toestand na uitvoering van x := E(x) Q(x) moet gelden, dan moet voor de begintoestand de uitspraak Q gelden met daarin voor x gesubstitueerd E(x).

  Voorbeelden:

  {x = 3} x := x + 2 {x = 5}

  {x = q.y + r} r := r - y {x = (q + 1).y + r}

  {f = (i - 1)!} f := f * i {f = i!}

- *sequentiële compositie*

  Als S1 en S2 beschreven worden door {P}S1{Q} en {Q}S2{R}, dan geldt

  {P}S1;S2{R}

  Voorbeelden:

  {x = q.y + r} r := r - y; q := q + 1 {x = q.y + r}

  {n = 7} m := 4; m := m + n {m = 11 ∧ n = 7}

- *selection statement*

  Als voor S1 en S2 gelden $\{P \wedge B\}S1\{Q\}$ en $\{P \wedge \neg B\}S2\{Q\}$ dan geldt

  $\{P\}$ if B then S1 else S2 $\{Q\}$

- *while statement*

  Als S beschreven wordt door $\{P \wedge B\}S\{P\}$, dan geldt:

  $\{P\}$ while B do S $\{P \wedge \neg B\}$

P wordt de *invariante relatie* genoemd. Daarnaast speelt de uitspraak B (wel eens de *variante relatie* genoemd) een rol. Uitvoering van S moet ervoor zorgen dat het false worden van B " dichterbij" komt.

Voorbeelden:

```
omdat: {x = q.y + r ∧ r ≥ y}
       r := r - y; q := q + 1
       {x = q.y + r}
geldt: {x = q.y + r}
       while r >=y do
          begin r := r - y; q := q + 1 end
       {x = q.y + r ∧ r < y}
```

(in dit voorbeeld is P : x = q.y + r, en B : r ≥ y).

```
omdat:{f = i! ∧ i ≠ n}
      i := i + 1; f := f * i
      {f = i!}
geldt:{f = i!}
       while i <> n do
          begin i := i + 1; f := f * i end
       {f = i! ∧ i = n}
```

(in dit voorbeeld is P: f = i!, en B : i ≠ n).

Opmerking.

De uitspraken over de toestanden zijn in de voorbeelden "nette" wiskundige formuleringen. In het vervolg zullen we ook wel eens gebruik maken van gewone Nederlandse zinnen of van plaatjes om deze uitspraken vast te leggen.

2. De constructie van repetities

In het vorige hoofdstuk zijn regels gegeven die het effect vastleggen van eerder geïntroduceerde constructies. Met deze regels kan ook worden nagegaan of een gegeven algoritme correct is. De regels zullen hier gebruikt worden als hulpmiddelen bij de constructie van algoritmen (in dit hoofdstuk beperkt tot repetities), waarvan dan ook direct de correctheid is verzekerd.

De aanpak hiervan is als volgt. Van het te construeren algoritme A zijn de begin- en eindtoestand gegeven., bijvoorbeeld door S respectievelijk F. Voor A moet gelden: $\{S\}A\{F\}$. Probeer F te schrijven als $P \wedge \neg B$, waarvan P als invariant gaat optreden en B als logische expressie van de repetitie. (Anders gezegd: P is een beschrijving van de tussentoestanden en in de eindtoestand wordt iets extra geëist, nl. $\neg B$). P moet ook gelden vóór de repetitie. Dit zal, vanuit S door een of meer *statements* gerealiseerd moeten worden (de zogenaamde initialisatie van de repetitie). A wordt dus gesplitst in een A1 en een A2 zodanig dat

$$\{S\}A1;\{P\}\ \underline{\text{while}}\ B\ \underline{\text{do}}\ A2\ \{P \wedge \neg B\}$$

A2 moet zodanig zijn dat de repetitie eindigt en moet er bovendien voor zorgen dat de invariant blijft gelden.

Stel dat gevraagd wordt een algoritme dat de waarde berekent van

$$1 + 2^2 + 3^2 + \ldots + n^2 = \sum_{k=1}^{n} k^2$$

bij gegeven waarde van n (geheel en ≥ 1). De berekende waarde zal aan een variabele toegekend moeten worden, noem die s. Voor het gevraagde algoritme moet dus gelden

$$\{n \geq 1 \text{ en geheel}\}\ A\ \{s = \sum_{k=1}^{n} k^2\}$$

De eindrelatie, $s = \sum_{k=1}^{n} k^2$, is te schrijven als $s = \sum_{k=1}^{p} k^2 \wedge p = n$. Als invariant proberen we $s = \sum_{k=1}^{p} k^2$ en als stopcriterium (de ontkenning van de logische expressie: B): $p = n$.

Laten we de invariant aangeven met P(p), dus P(p): $s = \sum_{k=1}^{p} k^2$. De invariant is, vanuit de begintoestand, eenvoudig waar te maken:

$$\{n \geq 1 \text{ en geheel}\}\ p := 1;\ s := 1\ \{P(p)\}$$

De repetitie is van de vorm:

$$\underline{\text{while}}\ p <> n\ \underline{\text{do}}\ \ldots$$

De eindigheid van de repetitie kan gegarandeerd worden door in de repetitie p met 1 op te hogen. Hierdoor wordt echter de invariant verstoord:

$$\{P(p)\}\ p := p + 1\ \{P(p - 1)\}$$

Deze kan hersteld worden door s te verhogen met p^2. Zo hebben we gevonden:

$\{n \geq 1$ en geheel$\}$

$p := 1;\ s := 1;\ \{P(p)\}$

<u>while</u> $p <> n$ <u>do</u>

  <u>begin</u> $\{p \neq n \wedge P(p)\}$

    $p := p + 1;\ \{P(p - 1)\}$

    $s := s + p * p\ \{P(p)\}$

  <u>end</u>

$\{P(p) \wedge p = n\}$

De hier gevolgde werkwijze zullen we steeds hanteren:

- vind uit de eindrelatie de invariant en het stopcriterium; de invariant is een afzwakking van de eindrelatie (in het voorbeeld bereikt door de

"constante" n te vervangen door de variabele p, $1 \leq p \leq n$)

- maak de invariant in het begin waar
- zorg voor de eindigheid van de repetitie
- herstel, zonodig, de invariant.

De invariant hoeft niet de vorm te hebben van een wiskundige formule; vaak zal de invariant gegeven worden door een beschrijving - in het Nederlands - van de betekenis van de variabelen, zoals in het onderstaande voorbeeld.

Stel dat de variabelen x en y een waarde hebben (geheel en > 0). Gevraagd wordt een algoritme dat aan de variabele g een waarde toekent: de grootste gemene deler van x en y. (a is een deler van b als geldt b mod a = 0; mod bepaalt de rest bij deling, zie terug).

{x > 0, y > 0, beide geheel}A{g = GGD(x,y)}

De eindrelatie kunnen we ook formuleren als:

(x mod g = 0) and (y mod g = 0) and geen enkel getal groter dan g is deelbaar op x en deelbaar op y.

Als invariant P nemen we de betekenis van g: geen enkel getal groter dan g is deelbaar op x en deelbaar op y.

Het stopcriterium is dan (x mod g = 0) and (y mod g = 0); de logische expressie in de while-statement is dan not ((x mod g = 0) and (y mod g = 0)) en dit is equivalent met (x mod g <> 0) or (y mod g <> 0).

Om de invariant in het begin waar te maken, kunnen we g gelijk maken aan het minimum van x en y.

```
{x > 0, y > 0, beide geheel}
if x < y then g := x else g := y
{P}
```

De eindigheid kunnen we garanderen door g in de repetitie met 1 af te lagen. De invariant blijft gelden want

$$\{((x \underline{\text{mod}}\ g <> 0)\ \underline{\text{or}}\ (y \underline{\text{mod}}\ g <> 0)) \wedge P\}\ g := g-1\ \{P\}\ .$$

De repetitie eindigt omdat er een ondergrens is voor g, namelijk 1 (en voor de repetitie geldt $g \geq 1$). Zo hebben we gevonden:

$\{x > 0,\ y > 0,$ beide geheel$\}$

$\underline{\text{if}}\ x < y\ \underline{\text{then}}\ g := x\ \underline{\text{else}}\ g := y;\ \{P\}$

$\underline{\text{while}}\ (x \underline{\text{mod}}\ g <> 0)\ \underline{\text{or}}\ (y \underline{\text{mod}}\ g <> 0)\ \underline{\text{do}}\ g := g-1\{P\}$

$\{(x \underline{\text{mod}}\ g = 0)\ \underline{\text{and}}\ (y \underline{\text{mod}}\ g = 0) \wedge P\}$

"After having devoted a considerable number of years of my scientific life to clarifying the programmer's task, with the aim of making it intellectually better manageable, I found this effort at clarification to my amazement (and annoynace) repeatedly rewarded by the accusation that "I had made programming difficult". But the difficulty has always been there, and only by making it visible can we hope to become able to design programs with a high confidence level, rather than "smearing code", i.e. producing texts with the status of hardly supported conjectures that wait to be killed by the first counterexample."

E.W. Dijkstra

- Doen we inderdaad niet overdreven moeilijk? Moet zelfs voor een eenvoudige opgave, zoals de eerste uit dit hoofdstuk, de methodische aanpak gevolgd worden? Kun je niet gewoon beginnen:

```
k := ...; s := ...;
while k...n do
   begin k := k + 1;
         s := s +...
   end
```

of:

```
s :=...; k := ...;
while k...n do
   begin s := s + ...;
         k := k + 1
   end
```

en al experimenterend vaststellen wat er ingevuld moet worden? Natuurlijk kan het, maar zelfs bij kleine problemen blijken al vaak fouten gemaakt te worden, laat staan bij grote.

We bekijken nog enkele voorbeelden.

Gevraagd wordt een oplossing voor

$$x = \cos(x) \text{ met } 0 \leq x \leq \frac{\pi}{2} .$$

De oplossing is een reële waarde die benaderd moet worden. Daarom zoeken we een interval [d,b] waarin de oplossing ligt. De grootte van het interval moet kleiner zijn dan de waarde van de variabele e(> 0).

We kunnen de gewenste eindtoestand

$$\text{oplossing in } [d,b] \wedge b - d < e$$

splitsen in een invariant: oplossing in [d,b] (beschrijving van alle (tussen)toestanden), en een stopcriterium b - d < e (extra eis voor de eindtoestand. De invariant is te realiseren door

```
d := 0; b := 3.14/2;
```

De eindigheid van de repetitie kunnen we garanderen door het interval

steeds kleiner te maken:

```
d := 0; b := 3.14/2;
while b - d >=e
   do "verklein [d,b], maar zorg dat de oplossing in het inter-
       val blijft"
```

Het verkleinen van het interval en het behouden van de invariant kan als volgt:

- verdeel het interval in twee gelijke delen [d,m] en [m,b] met m = (d+b)/2;
- als m ≤ cos(m) dan ligt de oplossing in [m,b]; door d gelijk te maken aan m blijft de invariant geldig; als m ≥ cos(m) dan ligt de oplossing in [d,m], door b gelijk te maken aan m blijft de invariant geldig.

```
d := 0; b := 3.14/2;
while b - d >=e do
   begin m := (d+b)/2;
         if m <=cos(m) then d := m;
         if m >=cos(m) then b := m
   end;
x := (d+b)/2
```

Als de repetitie eindigt levert het algoritme de oplossing. Of de repetitie eindigt hangt af van de wijze waarop met getallen van het type real wordt gerekend en van de waarde van e.

In het bovenstaande algoritme zijn we er van uit gegaan dat in Pascal de functie cos(x) zonder meer gebruikt kan worden. Dit is ook het geval en dat geldt eveneens voor de functies

abs(x)	$\|x\|$
sin(x)	
arctan(x)	
exp(x)	e^x
ln(x)	de natuurlijke logaritme
sqrt(x)	$\sqrt{x}$
sqr(x)	x * x

Stel dat cos(x) berekend zou moeten worden m.b.v. de reeksontwikkeling:

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots . \qquad (0 \leq x \leq \frac{\pi}{2})$$

Schrijf nu een algoritme dat bij een gegeven x en eps de waarde van cos(x) benadert met een afbreekfout die ten hoogste eps is. We weten dat als de reeks na een eindig aantal termen wordt afgebroken, de afbreekfout ten hoogste gelijk is aan de (absolute waarde van de) eerstvolgende term.

De eindrelatie is

$$c = \sum_{k=0}^{p-1} (-1)^k \frac{x^{2k}}{(2k!)} \wedge \frac{x^{2p}}{(2p)!} \leq \text{eps} \ .$$

Als invariant kiezen we

$$c = \sum_{k=0}^{n-1} (-1)^k \frac{x^{2k}}{(2k)!} \wedge t = \frac{x^{2n}}{(2n)!}$$

waarvan we het eerste deel aangeven met P(n) en het tweede deel met Q(n).

Het stopcriterium is dan $t \leq$ eps.

De eerste opzet van het algoritme wordt dan:

```
n := 1; c := 1; t := x * x/2; {P(n) ∧ Q(n)}
while t > eps do
   begin n := n + 1; {P(n - 1) ∧ Q(n - 1)}
         "verwerk t in c"; {P(n) ∧ Q(n - 1)}
         "bepaal nieuwe t" {P(n) ∧ Q(n)}
   end
{P(n) ∧ Q(n) ∧ t ≤ eps}
```

De repetitie eindigt omdat elke nieuwe t kleiner is dan zijn voorganger. Het verwerken van t in c komt neer op c := c + t als n even is en op c := c - t als n oneven is. We kunnen ook zeggen dat t afwisselend opgeteld en afgetrokken moet worden. Om te registreren welke operatie uitgevoerd moet worden, voeren we een variabele s in (die de waarden 1 en -1 kan aannemen) met als betekenis: de uit te voeren assignment op c is c := c + s * t. De betekenis van s voegen we toe aan de invariant, met als aanduiding S.

```
n := 1; c := 1; t := x * x/2; s := -1; {P(n) ∧ Q(n) ∧ S}
while t > eps do
   begin n := n + 1; {P(n - 1) ∧ Q(n - 1) ∧ S}
         c := c + s * t; {P(n) ∧ Q(n - 1) ∧ ¬S}
         "bepaal nieuwe t" {P(n) ∧ Q(n) ∧ ¬S}
         s := -s {P(n) ∧ Q(n) ∧ S}
   end
{P(n) ∧ Q(n) ∧ t ≤ eps}
```

De nieuwe waarde van t kan verkregen worden door de oude waarde van t te vermenigvuldigen met x^2 en het resultaat te delen door 2n * (2n - 1). Nu

blijkt dat in het algoritme n niet nodig is, maar dat met vrucht gebruik kan worden gemaakt van een variabele r, die steeds gelijk is aan 2n. De n verdwijnt uit het algoritme, maar wordt nog wel gebruikt om daarin de invariant uit te drukken; zo'n variabele wordt ook wel een "*ghost variable*" genoemd.

```
r := 2; c := 1; t := x * x/2; s := -1; {P(n) ∧ Q(n) ∧ S, r = 2n}
while t > eps do
   begin r := r+2; {P(n-1) ∧ Q(n-1) ∧ S, r = 2n}
         c := c+s * t; {P(n) ∧ Q(n-1) ∧ ¬S}
         t := t * x * x/(r * (r - 1)); {P(n) ∧ Q(n) ∧ ¬S}
         s := -s {P(n) ∧ Q(n) ∧ S}
   end
{P(n) ∧ Q(n) ∧ t ≤ eps}
```

In het volgende voorbeeld zal het overgaan op nieuwe variabelen nogmaals toegepast worden. Het gaat hierbij om een algoritme dat <u>in</u> de rekenmachine toegepast zou kunnen worden. Daar de rekenmachine binair is, moeten in zo'n algoritme de rekenkundige bewerkingen zijn afgestemd op binair rekenen. Dat betekent o.a. dat vermenigvuldigen en delen beperkt blijven tot operaties waarvan de tweede operand 2 is.

Gegeven is dat x en y een waarde hebben, waarvoor geldt: $0 \leq x \leq y \leq 1$. Gegeven is bovendien de variabele tol met een waarde (> 0).
Gevraagd wordt een algoritme dat $z = x/y$ bepaalt binnen de opgegeven tolerantie tol:

$$\{0 \leq x < y \leq 1 \wedge tol > 0\}\ A\ \{z \leq x/y < z + tol\}\ .$$

Een invariant vinden we door de "constante" tol uit de eindrelatie te vervangen door de variabele d: $z \leq x/y < z + d$. De eindrelatie geldt als naast de invariant ook nog geldt: $d \leq tol$.

```
z := 0; d := 1; {z ≤ x/y < z + d}
while d > tol do "verbeter z en d met behoud van invariant"
{z ≤ x/y < z + d ∧ d ≤ tol}
```

Het verbeteren van z en d komt neer op het verkleinen van het interval waarin de oplossing ligt. Dit verkleinen kan door het interval te halveren (zie terug bij opgave x = cos(x)). Het midden van het interval is $z + \frac{1}{2}d$.

als $z + \frac{1}{2}d > x/y$ dan $z \leq x/y < z + \frac{1}{2}d$

als $z + \frac{1}{2}d \leq x/y$ dan $z + \frac{1}{2}d \leq x/y < z + d$

```
z := 0; d := 1; {z ≤ x/y < z + d}
while d > tol do
   begin d := d/2; {z ≤ x/y < z + d ∨ z + d ≤ x/y < z + 2d}
         if z + d <= x/y then z := z + d
         {z ≤ x/y < z + d}
   end
```

In "$z + d \leq x/y$" moet de deling door y voorkomen worden (alleen deling door 2 is geoorloofd): $z * y + d * y \leq x$. We voeren nu de extra variabelen u en v in: $u = z * y$ en $v = d * y$. Als d nu verandert, moet ook v veranderen, zo ook voor z en u:

```
z := 0; d := 1; u := 0; v := y;
{z ≤ x/y < z + d ∧ u = z * y ∧ v = d * y}
while d > tol do
   begin d := d/2; {v = 2d * y} v := v/2; {v = d * y}
         if u + v <=x then
            begin z := z + d; {u = (z - d) * y}
                  u := u + v {u = z * y}
            end
         {z ≤ x/y < z + d ∧ u = z * y ∧ v = d * y}
   end
{z ≤ x/y < z + d ∧ d ≤ tol}
```

In dit voorbeeld hebben we de semantiekregels niet alleen gebruikt voor de constructie van de repetitie, maar ook om in de deelconstructies de juiste assignments te vinden. Daarvan geven we tenslotte nog een voorbeeld.

Een (wiskundige) rij getallen, die veel toepassingen kent is de rij van Fibonacci:

0,1,1,2,3,5,8,13,21,...

Als we de getallen aangeven met f_i (i = 0,1,2,...), dan geldt: $f_0 = 0$, $f_1 = 1$, $f_i = f_{i-1} + f_{i-2}$ voor $i \geq 2$.

Gevraagd wordt een algoritme dat bij gegeven waarde van n (≥ 0 en geheel) f_n bepaalt.

$\{n \geq 0 \text{ en geheel}\}\ A\ \{t = f_n\}$

Als invariant kiezen we $t = f_k$; het stopcriterium is dan k = n. Dit stopcriterium speelt alleen een rol als $n \geq 2$, want voor $n < 2$ is geen repetitie nodig.

{$n \geq 0$ en geheel}

if $n \leq 1$

then $t := n$ {$(n = 0 \vee n = 1) \wedge t = n$, dus $t = f_n$}

else {$n \geq 2$}

"bepaal f_n m.b.v. een repetitie" {$t = f_n$}

{$t = f_n$}

We beperken ons verder tot de repetitie.

Daar een nieuwe term te vinden is als som van twee vorige, introduceren we ook de variabele s: $s = f_{k-1}$.

$s := 0$; $t := 1$; $k := 1$; {$s = f_{k-1} \wedge t = f_k$}

while $k <> n$ do

begin $k := k + 1$; {$s = f_{k-2} \wedge t = f_{k-1}$}

$t := t + s$; {$s = f_{k-2} \wedge t = f_k$}

$s := t - s$ {$s = f_{k-1} \wedge t = f_k$}

end

{$t = f_k \wedge k = n$}

De repetitie eindigt door de ophoging van k met 1.

Vaak ziet men voor dit probleem de volgende oplossing

$s := 0$; $t := 1$; $k := 1$;

while $k <> n$ do

begin $k := k + 1$;

$v := s$;

$s := t$;

$t := v + s$

end

Men introduceert dan de hulpvariabele v omdat men in de repetitie op de moeilijkheid stuit dat noch "t := t + s; s := t" noch "s := t; t := t + s" het gevraagde realiseert. Uit onze afleiding blijkt duidelijk wat er moet en kan gebeuren. Gebruikt men geen toestandsbeschrijvingen dan is dit moeilijker in te zien.

In dit hoofdstuk hebben we gezien hoe we regels voor de beschrijvingen van de effecten van constructies kunnen gebruiken voor het vinden van algoritmen. Vooral bij de repetitie blijkt dit van nut te zijn. De stappen die worden gemaakt zijn:

- *Bepaal de gewenste eindrelatie R*
- *Vind door afzwakking van R een invariante relatie P met daarnaast een stopcriterium S zodanig dat P ∧ S de eindrelatie R oplevert. Bovendien moet P "zo sterk" mogelijk gekozen worden en zodanig dat P eenvoudig initieel is waar te maken.*
- *Zorg dat de repetitie eindigt onder behoud van de invariant.*

Daarmee krijgen we als repetitie:

"maak P waar";

<u>while</u> ¬S *<u>do</u> "zet stap in de richting van het stoppen (S) onder behoud van de invariante relatie".*

3. Iets over efficiëntie

In een voorbeeld van het vorige hoofdstuk werd een nieuwe waarde van een variabele steeds berekend uit de vorige waarde: t := t * x * x/(p * (p - 1)). Men kan de gewenste waarde $x^p/p!$ natuurlijk ook bij iedere slag van de repetitie helemaal van de grond af aan opbouwen. Dat zou echter niet efficiënt zijn. Van de programmeur wordt verwacht dat hij dit soort inefficiënties vermijdt. Maar het vinden van een efficiënt algoritme moet niet overdreven worden en zeker niet ten koste gaan van de correctheid.

"Program correctness and cost of execution are two so important concerns that the programmer who has to give full attention to both of them, should be given the mental tools to separate these two concerns."

E.W. Dijkstra

"Whereas a teacher should not and must not pay attention to "percent issues" as to efficiency while explaining and exemplifying methods of composing well-structured programs, a professional programmer may well be forced to do so."

N. Wirth

Vaak kan kennis van het vakgebied, waaruit het probleem afkomstig is, of kennis van het probleem zelf gebruikt worden om tot efficiëntere algoritmen te komen.
In het eerste voorbeeld van het vorige hoofdstuk werd gevraagd een algoritme voor de bepaling van s: $s = \sum_{k=1}^{n} k^2$. Uit de wiskunde zouden we kunnen weten dat $1^2 + 2^2 + 3^2 + \ldots + n^2 = n(n+1)(2n+1)/6$ (ga na met volledige inductie), waardoor het algoritme is terug te brengen tot de assignment statement:

s := n * (n + 1) * (2 * n + 1)/6 .

In het tweede voorbeeld van het vorige hoofdstuk hebben we een recht-toe-recht-aan algoritme voor de grootste gemene deler afgeleid. Als de gevraagde g = GGD(x,y) klein is ten opzichte van min(x,y) zal de while statement een groot aantal herhalingen tot gevolg hebben. Misschien zijn er minder stappen nodig als gebruik wordt gemaakt van eigenschappen van de GGD:

1. als a > b dan GGD(a,b) = GGD(a - b,b)
2. GGD(a,b) = GGD(b,a)
3. GGD(a,a) = a

Om te voorkomen dat de gegeven waarden van x en y veranderen, worden deze waarden eerst toegekend aan a respectievelijk b.

{x > 0 ∧ y > 0}

a := x; b := y

{a > 0 ∧ b > 0 ∧ GGD(a,b) = GGD(x,y)}

De eindrelatie voor de repetitie is

a > 0 ∧ b > 0 ∧ GGD(a,b) = GGD(x,y) ∧ a = b

Het eerste deel, dat al gerealiseerd was door de assignement statements gebruiken we als invariant en a = b als stopcriterium.
In de repetitie moet voor de eindigheid gezorgd worden. Eigenschap 1 suggereert hoe dit kan zonder de invariant te verstoren.

```
while a <> b
    do if a > b then a := a - b
                else b := b - a
```

Hierbij is ook gebruik gemaakt van eigenschap 2. Dat de repetitie echt

eindigt kunnen we aantonen aan de hand van de waarde van a + b. Steeds geldt a + b > 0. Dit is immers vóór de repetitie waar (want $a > 0 \wedge b > 0$) en de assignments in de repetitie laten het onveranderd waar ($a > 0 \wedge b > 0$ is immers een deel van de invariant). In iedere "slag" van de repetitie wordt a + b echter kleiner. Wil a + b > 0, bij gehele a en b, blijven gelden, dan moet het algoritme wel eindigen.

Het aantal herhalingen van dit algoritme zal vaak kleiner zijn dan het aantal herhalingen van het vorige algoritme, maar niet voor elke combinatie van x en y is dit het geval, neem maar (x,y) = (6,81).

Kan het bovenstaande algoritme versneld worden?

Neem bijvoorbeeld:

```
a := x; b := y;
while a<> b do
   begin while a > b do a := a - b;
         while a < b do b := b - a
   end;
g := a
```

Het aantal malen dat een van de twee assignment statements wordt uitgevoerd is hetzelfde als bij het vorige algoritme. Het aantal keren dat a en b vergeleken wordt zal nu echter vaak kleiner zijn; al met al geen indrukwekkende winst. Maar

```
while a ≥ b do a := a - b
```

komt overeen met

```
a := a mod b
```

en als deze laatste assignment statement nu efficiënter is dan de repetitie dan krijgen we wellicht een efficiënter algoritme:

```
a := x; b := y;
while a <> b do
   begin if a > b then a := a mod b;
         if a < b then b := b mod a
   end;
g := a
```

Dit algoritme is echter fout! Waardoor?
De les is dat we op moeten passen met het aanpassen van algoritmen, hier met het oog op efficiëntie, zelfs als we uitgaan van een correct algoritme.

" Why should we spend a lot of time trying to program optimally, only to discover that we do not have a working program, rather than getting it working correctly first and worry about any improvements later?"

P. Landin

Er bestaat wel een algortime dat gebruik maakt van de restbepaling. Dit is een reeds lang bekend algoritme, het algoritme van Euclides. De invariant P van dit algoritme luidt:

$$a \geq b > r \geq 0 \wedge GGD(a,b) = GGD(x,y) \wedge r = a \bmod b$$

Het bijbehorende stopcriterium is $r = 0$.

```
{x > 0 ∧ y > 0}
if x >= y then begin a := x; b := y end
          else begin a := y; b := x end;
{a ≥ b > 0 ∧ GGD(a,b) = GGD(x,y)}
r := a mod b;
{a ≥ b > r ≥ 0 ∧ GGD(a,b) = GGD(x,y) ∧ r = a mod b}
while r <> 0 do
```

```
   begin {P ∧ r ≠ 0 d.w.z P ∧ GGD(a,b) = GGD(b,r) = GGD(x,y)}
         a := b; b := r; {a ≥ b > 0 ∧ GGD(a,b) = GGD(x,y)}
         r := a mod b
         {P}
   end;
{P ∧ r = 0, d.w.z. GGD(a,b) = b}
g := b
```

4. Datatypen en declaraties; expressies en de assignment statement

Bij de verwerking van een programma wordt door de machine gemanipuleerd met waarden, waarden van constanten, variabelen en expressies. Iedere waarde in een programma is van een bepaald *type*. Een type is een waardenverzameling waarop in het algemeen een aantal operaties is gedefinieerd. Het typebegrip komt niet alleen bij het programmeren voor. Ook in de wiskunde kent men een soort typebegrip. Zo zal een wiskundige voor iedere "variabele" die hij introduceert direct het type vastleggen, bijvoorbeeld: "laat n een natuurlijk getal zijn."

Van het typeconcept in de informatica kunnen we de volgende karakteristieken noemen:

- het type bepaalt de verzameling van waarden;
- het type bepaalt de uit te voeren operaties;
- iedere waarde in een algoritme is van precies één type;
- het type van een constante, variabele of expressie kan bepaald worden aan de hand van de tekst van het algoritme;
- iedere operator verwacht operanden van een bepaald type en levert ook een resultaat van een bepaald type.

Een type is *geordend* als er een volgorde gedefinieerd is op de waardenverzameling. Het type van een variabele wordt vastgelegd door middel van een *declaratie*. Naast het type wordt in de declaratie ook de naam van de variabele vastgelegd. De declaratie van een variabele moet plaatsvinden voordat de variabele gebruikt wordt in een statement. Voordat een variabele gedeclareerd kan worden moet het betreffende type gedefinieerd zijn, of het moet in de declaratie zelf worden vastgelegd. Er zijn vier stan-

daardtypen die niet gedefinieerd behoeven te worden: integer, real, boolean en char. Alle andere typen moeten wel gedefinieerd worden. We komen later terug op het definiëren van typen.
De declaraties van de variabelen worden gegeven in het "variable declaration part":

<variable declaration part> ::=
var <variable declaration>; {<variable declaration>}
<variable declaration> ::= <identifier list> : <type>

Voorbeeld:

```
var k,m,n: integer;
    x,y: real;
    c: char;
    p,q: boolean;
```

Na de declaratie bestaat de variabele, maar heeft nog geen waarde. De variabele krijgt zijn waarde door een assignment statement of een read-opdracht.
Voorbeeld van een programma:

```
program division (input, output);
   var a,b,q,r: integer;
 begin read(a,b);
       r := a; q := 0;
       while r >= b do
          begin r := r - b; q := q + 1 end;
       write(a,b,q,r)
 end.
```

We komen later terug op de vorm van een programma (en op de read- en write-opdracht). De declaraties van een programma bepalen de *toestandsruimte* van het proces, dit is de verzameling van toestanden die tijdens de uitvoering van het programma kunnen optreden. In het voorbeeld bestaat iedere toestand uit vier waarden van het type integer; deze waarden zijn de waarden van a,b,q en r. Na afloop van het proces bestaat de toestandsruimte (bestaan de variabelen) niet meer.
Voor de interpretatie van een toestand is de betekenis van de variabelen van belang. Deze betekenis kunnen we tot uitdrukking laten komen in de naam van de variabele, door deze zinvol te kiezen en niet altijd x of y te noemen en de variabele niet geel te noemen als hij een lengte voorstelt. Het is vaak echter niet mogelijk de betekenis volledig in de naam tot uitdrukking te laten komen, een nadere omschrijving (buiten het programma) is dan nodig.
Een constante in een programma kan gegeven worden door steeds zijn waarde uit te schrijven op de plaatsen waar hij nodig is. We kunnen de constante ook een naam geven en eenmaal aangeven voor welke waarde deze naam staat. Daarna is iedere verschijning van de naam in het programma equivalent aan de waarde. Ook de definities van de constanten worden samengebracht, vooraan in het programma.

<constant definition part> ::=
const <constant definition>; {<constant definition>}

<constant definition> ::= <identifier> = <constant>

Voorbeeld:

```
const pi = 3.14159;
      e = 2.7182;
```

De standaardtypen integer, real, boolean en char

Het type integer

De waardenverzameling is een deelverzameling van de verzameling der gehele getallen. In feite is het een gesloten interval waarvan de ondergrens en de bovengrens door de rekenmachine bepaald worden. Wel bestaat de standaard constante "maxint" die als waarde heeft de grootste waarde van het type integer op de gebruike machine (in de gebruikte implementatie).

Operaties (voor de beschrijving geldt dat i en j van het type integer zijn):

	notatie	voorbeeld	waarde
unair:	+i	+3	3
	-i	-3	-3
binair:	i+j	7+3	10
	i-j	7-3	4
	i*j	7*3	21
	i div j	7 div 3	2
	i mod j	7 mod 3	1

Voor de definities van div (geheel delen) en mod (rest bepalen) verwijzen we naar hoofdstuk 1. Daar wordt alleen het resultaat voor positieve operanden gegeven; voor negatieve operanden gebruiken we de gebruikelijke tekenregels zodat (-7) mod 3 = -. (Pascal is hierover onduidelijk).
Deling door 0 is ongedefinieerd.
De (resultaten van de) operaties zijn alleen gedefinieerd als de waarde ligt binnen het interval dat door de implementatie wordt bepaald. Als dit niet het geval is, spreken we van overflow (resp. underflow).

De ordening welke bestaat in de verzameling der gehele getallen, bestaat ook in het type integer. Deze ordening geeft de mogelijkheid om twee waarden te vergelijken: groter, ten minste gelijk, ongelijk, ten hoogste en kleiner. Zo'n vergelijking levert een waarde op van het type boolean (en we zullen daar de vergelijkingsoperatoren behandelen). Bovendien bestaan in Pascal voor geordende typen (uitgezonderd voor het type real) de functies succ en pred:

succ(x) levert de directe opvolger van x, als deze opvolger bestaat (voor integer: succ(x) = x + 1)

pred(x) levert de directe voorganger van x, als deze voorganger bestaat (voor integer: pred(x) = x - 1).

Naast deze functies bestaan in Pascal nog twee standaardfuncties voor het type integer:

notatie	resultaat
abs(x)	|x|
sqr(x)	x * x

Voor de schrijfwijze van constanten van het type integer geldt de volgende syntaxis:

<integer number> ::= [<sign>]<unsigned integer>

<sign> ::= + | -

<unsigned integer> ::= <digit>{<digit>}

(In feite kent Pascal alleen niet-negatieve constanten van het type integer; -5 bijvoorbeeld is een expressie.)

Het type real

De waardeverzameling van het type real is een deelverzameling van de verzameling der reële getallen, bepaald door twee restricties:

- Iedere waarde van het type real ligt in het gesloten interval [min,max],
- waarbij de waarden van min en max (geen standaard constanten) worden bepaald door de implementatie.
- Het interval [min,max] bevat slechts een eindig aantal waarden van het type real. Welke dat zijn, wordt bepaald door de implementatie.

Een gevolg van de tweede restrictie is dat slechts een beperkt aantal reële getallen (met een waarde waarvoor geldt min ≤ waarde ≤ max) exact overeenkomt met een waarde van het type real.

Operaties (voor de beschrijving geldt dat r en s van het type real zijn):

	notatie	voorbeeld	waarde
unair:	+r	+2.7	2.7
	-r	-2.7	-2.7
binair:	r + s	2.7 + 2.4	5.1
	r - s	2.7 - 2.4	0.3
	r * s	2.7 * 2.4	6.48
	r/s	2.7/2.4	1.125

De genoemde operaties zijn de bekende operaties op reële getallen, met dien verstande dat ze als resultaat een benadering van het exacte resultaat geven (waardoor bijvoorbeeld a - b + b niet altijd exact a oplevert). De operaties zijn alleen gedefinieerd als het resultaat in het interval [min,max] ligt; is dit niet het geval, dan spreken we weer van overflow. De waarden van het type real zijn geordend, zodat de vergelijkingsoperatoren ook voor het type real gedefinieerd zijn; dit geldt echter niet voor de functies succ en pred.

Wel zijn een aantal andere standaardfuncties gedefinieerd voor het type real:

notatie	resultaat
sin(x)	"spreekt voor zich" (x in radialen)
cos(x)	"spreekt voor zich" (x in radialen)
arctan(x)	hoek y, $-\pi/2 \leq y \leq \pi/2$, waarvoor tan(y)=x
ln(x)	$^{e}\log x$ voor x > 0
exp(x)	e^{x}
sqrt(x)	$\sqrt{x}$ voor $x \geq 0$
abs(x)	$\|x\|$
sqr(x)	x * x

In Pascal mogen de operanden bij een "real operator" ook van het type integer zijn, zo zijn 3.7 - 2, 8.5 * 2 en ook 18/5 toegestaan. Het resultaat is van het type real. Er vindt als het ware een automatische type-overgang plaats van het type integer naar het type real. Andersom, van real naar integer, is niet anders mogelijk dan via de functies (type transfer functies genoemd):

notatie	resultaat
trunc(x)	afkapping van x; 7.6 wordt 7 -7.6 wordt -7
round(x)	afronding van x; 7.6 wordt 8, 7.3 wordt 7, -7.6 wordt -8 en -7.3 wordt 7

Voor de schrijfwijze van constanten van het type real geldt de volgende syntaxis:

<real number> ::= [<sign>]<unsigned real>

<unsigned real> ::= <unsigned integer>.<digit sequence>|
    <unsigned integer>.<digit sequence>E<scale factor>|
    <unsigned integer>E<scale factor>

<digit sequence> ::= <digit>{<digit>}

<scale factor> ::= <integer number>

Zo zijn 0.456, 45.6E -2 en 0.0456 E1 correcte getallen van het type real, alle met dezelfde waarde. (Ook hier geldt dat bijvoorbeeld -0.5 een expressie is.)

Het type boolean

De waardenverzameling bestaat uit twee waarden true en false (standaard constanten).

Operaties (voor beschrijving geldt dat a en b van het type boolean zijn):

	notatie	naam
unair:	not a	ontkenning ("niet")
binair:	a and b	conjunctie ("en")
	a or b	disjunctie ("of")

De resultaten van deze operaties kunnen we vastleggen in tabelvorm (zie ook hoofdstuk 1):

a	not a
true	false
false	true

a	b	a and b	a or b
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

Als operanden kunnen optreden: de constanten true en false, variabelen van het type boolean (tot zover is er geen verschil met de operaties van het type integer en van het type real), maar ook relaties van de vorm: x < y (kleiner), x <= y (ten hoogste), x = y (gelijk), x <> y (ongelijk), x >= y (ten minste) en x > y (groter). Hierin kunnen x en y van de typen

integer en real zijn, maar - in Pascal - ook van het type boolean, omdat in Pascal het type boolean geordend is, waarbij false "kleiner" is dan true. Zo zijn bijvoorbeeld a = b, a <> b en a< = b, waarin a en b van het type boolean zijn, toegestaan. De waardentabel voor deze drie operaties is:

a	b	a = b	a <> b	a< = b
true	true	true	false	true
true	false	false	true	false
false	true	false	true	true
false	false	true	false	true

en ze worden (achtereenvolgens) genoemd: equivalentie, exclusieve disjunctie en implicatie.

Naast de relatie-operatoren bestaat in Pascal nog een functie die van een integer argument een boolean waarde maakt: odd(x) (met het vanzelfsprekende resultaat).

Het type char

Tot de waardenverzameling van het type char behoren alle karakters die in de machine, waarop het programma verwerkt wordt, beschikbaar zijn. De notatie van een constante van het type char geschiedt door het betreffende karakter tussen aanhalingstekens te plaatsen: 'x', 'p', '2', ' ' (spatie), ',', '?'.

Er zijn twee type- transferfuncties "tussen" de typen char en integer, ord(c) en chr(i):

ord(c) het resultaat is het rangnummer van karakter c in de geordende verzameling der karakters (collating sequence genoemd)

chr(i) het resultaat is het karakter dat in de collating sequence rangnummer i heeft ($1 \leq i \leq$ aantal karakters van de collating sequence)

Er geldt: ord(chr(i)) = i (als chr(i) bestaat) en chr(ord(c)) = c.

Met behulp van de functie ord kunnen we de ordening van het type char definiëren: $c_1 R c_2$ (waarin R staat voor <, <=, =, <>, >= of > en c_1 en c_2 constanten van het type char zijn) is equivalent met $ord(c_1)\ R\ ord(c_2)$.

Voor het type char zijn ook de functies succ en pred gedefinieerd, waarvoor geldt:

pred(c) = chr(ord(c)-1)

succ(c) = chr(ord(c)+1) .

Expressies

Een expressie is een notatie voor een samengestelde operatie: een rij operanden onderling gescheiden door operatoren waarbij eventueel ook (ronde) haakjes gebruikt kunnen worden. Als operanden kunnen optreden: constanten, variabelen en functies. Een expressie is van een bepaald type; dit type kan bepaald worden aan de hand van de regels die gegeven zijn bij de bespreking van de operatoren van de verschillende typen.

Voor de berekening van de waarde van een expressie wordt voor iedere variabele die er in voorkomt de waarde genomen uit de heersende toestand. (Iedere variabele moet dus een waarde hebben.)

De waarde van een expressie is eenduidig bepaald doordat er een volgorde is afgesproken voor de operaties (doordat de operatoren bepaalde prioriteiten hebben). De prioriteiten zijn:

prioriteiten	operatoren
4	not
3	*, /, div, mod, and
2	+, -, or
1	>,>=, =,<>,<=, <

De volgorde van uitwerking van een expressie is nu als volgt:

- bij gelijke prioriteit wordt de expressie van links naar rechts geevalueerd;
- operaties met een hogere prioriteit worden eerst uitgevoerd (onderling ook weer van links naar rechts);
- met haakjes kunnen de bovenstaande regels doorbroken worden, doordat hetgeen tussen haakjes staat eerst wordt uitgerekend (volgens de gegeven regels).

De laatste regel suggereert al dat we steeds haakjes moeten gebruiken als we voor onszelf niet zeker zijn van de volgorde (of de volgorderegels niet willen onthouden).

Afsluitend nog enkele opmerkingen.

Twee arithmetische operatoren (operatoren van het type real en van het type integer) mogen nooit naast elkaar voorkomen: 2 * -3 moet dus geschreven worden als 2 * (-3).

De boolean expressie not (a and b) is equivalent met not a or not b. Zo ook not (a or b) en not a and not b.

De boolean operatoren and en or zijn in de tabel commutatief gedefinieerd, d.w.z. dat a and b hetzelfde is als b and a, zo ook a or b en b or a.

De waarde van a and b is false als de eerste operand false is, ongeacht de waarde van de tweede operand. Zo is a or b true als de eerste operand true

is, ongeacht de waarde van de tweede operand. Pascal legt niet vast of in zulke gevallen de tweede operand nog wel bepaald moet worden (en sommige implementaties zullen het niet doen en andere wel). Als het niet gebeurt spreekt men wel van de "conditionele and" en de "conditionele or" .
De twee berekeningswijzen kunnen verschillen opleveren. Zo zal bij de "gewone and" de expressie false and y een fout opleveren als y geen waarde heeft, terwijl dat bij de "conditonele and" niet het geval zou behoeven te zijn. Het beste is om de expressies zo te noteren dat er geen verschillen optreden voor de twee berekeningswijzen.

Assignment statement

Uitvoering van de assignement statement houdt in:
Berekening van de waarde van de expressie in de heersende toestand en toekenning van deze waarde aan de variabele in het linker lid, waardoor de toestand verandert. Het type van de waarde van de expressie moet gelijk zijn aan het type van de variabele, met als uitzondering dat een waarde van het type integer mag worden toegekend aan een real variabele (waarbij een impliciete type transfer plaatsvindt van integer naar real).

Definitie van ongestructureerde typen

Met de vier standaardtypen kan elk proces beschreven worden. Het is echter wenselijk de mogelijkheid te hebben eigen, nieuwe typen te definiëren.
Enkele redenen hiervoor zijn:

- door een nieuw type kan de waardenverzameling aangepast worden aan het probleem;
- eventueel is een efficiëntere implementatie van de waarden mogelijk.

We beperken ons hier voorlopig tot enkelvoudige typen. De waarden van een enkelvoudig type worden als ondeelbaar beschouwd, zo is 325 één ondeelbare waarde. Zo'n enkelvoudige waarde wordt ook wel een scalair genoemd. De vier standaardtypen zijn enkelvoudig.

De eerste methode om een nieuw enkelvoudig type te definiëren is door *opsomming* (*enumeratie*) van de waarden, die (in Pascal) namen zijn:

<enumerated type> ::= (<identifier list>)

<identifier list> ::= <identifier>{,<identifier>}

De vorm van de *type-definitie* is vastgelegd in het volgende stukje syntaxis:

<type definition part> ::= type <type definition>;
{<type definition>;}

<type definition> ::= <identifier> = <type>

Een van de vormen van "type" is "enumerated type". Zo kunnen we bijvoorbeeld definiëren:

```
type kleur = (geel, groen, rood, blauw);
     kaartkleur = (klaveren, ruiten, harten, schoppen);
     dag = (maandag,dinsdag,woensdag,donderdag,vrijdag,zaterdag,zondag);
```

De waardenverzameling is geordend in de volgorde van het opnoemen van de waarden. Er geldt dus bijvoorbeeld: geel < groen < rood < blauw. Alle relatie-operatoren zijn van toepassing en ook de functies succ en pred zijn gedefinieerd (uitgezonderd succ voor de laatste waarde en pred voor de eerste waarde uit de opsomming). Daarnaast is ook de functie ord voor enumeratietypen gedefinieerd met als resultaat een waarde tussen 0 en n - 1 als het aantal waarden van het type n is; zo is ord(geel) gelijk aan 0 en ord(harten) is gelijk aan 2.

Na de definitie van een type kan een variabele van dat type gedeclareerd worden:

```
var k: kleur ;
```

De definitie van het type mag ook direct in de declaratie van de variabele opgenomen worden:

```
var d: (maandag, dinsdag, woensdag, donderdag, vrijdag,
        zaterdag, zondag);
```

Zijn bovenstaande variabelen gedeclareerd, dan kunnen via assignment acties waarden aan deze variabelen worden toegekend:

```
k := succ(geel); d := woensdag
```

Een tweede mogelijkheid voor het definiëren van een nieuw type is het opgeven van de waardenverzameling van dit nieuwe type als deelverzameling (*interval*) van de waardenverzameling van een standaardtype (uitgezonderd real) of van een reeds bekend enumeratietype.

```
<subrange type> = <constant> .. <constant>
```

Voorbeelden van typedefinities met subrange types:

```
type cijfer = '0'..'9';
     jaar = 1900..2000;
     werkdag = maandag .. vrijdag;
     nat = 1.. maxint;
```

Op waarden van een subrange type zijn alle operaties van het "omvattende" type gedefinieerd. Als zo'n operatie een waarde buiten het interval oplevert, is dit de waarde van het omvattende type. In een assignment mogen de tussenresultaten van de expressie tot het omvattende type behoren, als de waarde van de expressie maar ligt in het interval. Een waarde van een subrange type mag worden toegekend aan een variabele van het omvattende type.

Definities en declaraties

Een Pascalprogramma heeft de volgende opbouw:

<program> ::= <program heading> <block>.

<program heading> ::= program <identifier>(<program parameters>);

<program parameters> ::= <identifier list>

<block> ::= <declaration part> <statement part>

<declaration part> ::= [<constant definition part>]
[<type definition part>]
[<variable declaration part>]

<statement part> ::= begin <statement>
{;<statement>} end

(Enkele constructies die nog niet behandeld zijn, zijn weggelaten.)
In de definities en declaraties worden betekenissen toegekend aan namen (identifiers). Deze betekenissen moeten uniek zijn voor iedere naam, d.w.z. dat aan geen enkele naam twee verschillende betekenissen mogen worden toegekend in het programma. Alle namen van variabelen, constanten en typen moeten een betekenis krijgen in het definitie- en declaratiegedeelte van het programma (uitgezonderd de standaardtypen en -constanten).
De definities en declaraties moeten, als ze voorkomen, in de aangegeven volgorde voorkomen. Bij het definiëren van typen kan gebruik worden gemaakt van gedefinieerde constanten en bij het declareren van variabelen kan gebruik worden gemaakt van de gedefinieerde typen. Uit het Pascalrapport is niet op te maken of binnen ieder van de definitie- en declaratiedelen ook de strikte volgorde "eerst definiëren, dan gebruiken" geldt. Zou dit het geval zijn dan zijn

```
const  negeen = - poseen;
       poseen = 1;
```

en

```
type een = twee;
     twee = 14 .. 21;
```

niet toegestaan.

In dit hoofdstuk ligt de nadruk op syntactische aspecten. Bij het ontwerpen van een programma zijn deze niet van belang. Wel als het uiteindelijke programma door een rekenmachine verwerkt wordt.

5. Invoer en uitvoer: communicatie met de buitenwereld

In een programma voor de berekening van de "wortels" x_1 en x_2 van de vierkantsvergelijking $ax^2 + bx + c = 0$ dienen de waarden van a, b en c bekend te zijn. Dit zou kunnen door deze waarden in het programma op te nemen als constanten. Daarmee zou het programma alleen geschikt zijn voor de oplossing van die ene vergelijking. Er bestaat echter ook de mogelijkheid om tijdens de verwerking van een programma van buiten af de waarden toe te kennen aan variabelen. Dit gebeurt via de uitvoering van de opdracht:

read(<variable list>)

waarin:

<variable list> ::= <variable>{,<variable>}

De variabelen moeten van het type integer, real of char zijn, of van een type dat een subrange is van de typen integer of char. De benodigde waarden worden achtereenvolgens "gelezen" uit een invoerrij die bij het programma behoort en die buiten de verwerking van het programma om zijn waarden krijgt toegevoerd. Uitvoering van "read(a)" heeft als effect:

{input = $(s_1,s_2,\ldots,s_n)$} read(a) {a = s_1, input = $(s_2,\ldots,s_n)$}

waarin de invoerrij de naam input heeft, die ook gebruikt wordt in de "kop" van het programma (zie het begin van hoofdstuk 4).
Naast een invoerrij behoort bij een programma ook een uitvoerrij met de naam output. Waarden (van constanten, variabelen en expressies) kunnen "geschreven" worden in de uitvoerrij door middel van uitvoering van de opdracht:

write(<output list>)

waarin:

<output list> ::= <output value>{,<output value>}

De waarde moet van het type integer, real, boolean of char zijn (of het is een string; komt later aan de orde). Een waarde van het type boolean wordt in de uitvoerrij geplaatst als TRUE of FALSE.

read(a,b,c) heeft hetzelfde effect als: read(a); read(b); read(c). Zo ook heeft write(w1,w2,w3) hetzelfde effect als write(w1); write(w2); write(w3).
De invoerrij en de uitvoerrij kunnen opgebouwd gedacht worden uit "regels" (ponskaart, of regel op een regeldrukker of regel op een beeldscherm). Met de regelopbouw kan in het programma rekening worden gehouden.

Invoer

De opdracht read(v) is ten aanzien van de variabele v equivalent met de assignment statement "v := ...", alleen krijgt v nu zijn waarde door-dat de volgende waarden uit de invoerrij wordt genomen. Iedere uitvoering van een leesopdracht heeft tot gevolg dat de volgende waarde uit de invoerrij wordt toegekend aan de betreffende variabele. Een waarde kan dus niet tweemaal gelezen worden.
Een waarde van het type integer of van het type real moet voldoen aan de syntaxis van die waarden. Een karakter wordt in de invoerrij niet omsloten door aanhalingstekens. Een getal mag vooraf worden gegaan door spaties, een karakter uiteraard niet omdat bij het lezen dan aan de variabele de spatie als waarde wordt toegekend.
Uitvoering van

read(i1,i2,c1,c2,r1,r2)

met i1 en i2 van het type integer, c1 en c2 van het type char en r1 en r2 van het type real heeft bij invoerrij

27 + 38ab 34.4 - 12.8 E -4

voor de variabelen hetzelfde effect als uitvoering van de assignment statements

i1 := 27; i2 := 38; c1 := 'a'; c2 := 'b'; r1 := 34.4; r2 := -12.8 E -4

De invoerrij kan opgebouwd zijn uit regels. Voor het constateren van het einde van een regel bestaat de boolean functie eoln die de waarde true oplevert als van de huidige regel het laatste karakter gelezen is (en anders false oplevert).

Overgang op een nieuwe regel van de invoerrij wordt automatisch gerealiseerd bij het lezen van een volgende waarde, tenzij men bij het eind van de regel een karakter wil lezen, dan krijgt men eerst een spatie voordat de overgang op de nieuwe regel wordt gerealiseerd,
Met de opdracht "readln" kan men de rest van de huidige regel overslaan. Deze opdracht kan gecombineerd worden met de leesopdracht. Uitvoering van readln(v1,v2) heeft tot gevolg dat v1 en v2 de volgende waarden uit de invoerrij krijgen toegekend en dat de dan heersende regel voor de rest wordt overgeslagen. De opdracht heeft dus hetzelfde effect als:
"read(v1,v2); readln
Zoals er een boolean functie is waarmee het einde van een regel geconstateerd kan worden, zo is er ook een boolean functie waarmee het einde van de totale invoerrij geconstateerd kan worden: eof. Stel dat in de invoerrij een onbekend aantal waarden (zeg van het type integer) staan en dat op al deze waarden de operatie P moet worden uitgevoerd. Dit kan met behulp van het volgende programmafragment:

```
while not eof do begin read(i); P(i) end
```

Uitvoer

Verwerking van de opdracht

write(e)

heeft tot gevolg dat de waarde van e in de uitvoerrij wordt geplaatst; e mag een willekeurige expressie zijn.
Ook de uitvoerrij kan in regels opgedeeld worden. Uitvoering van de opdracht "writeln" heeft tot gevolg dat er overgegaan wordt op een nieuwe regel. Deze opdracht kan weer gecombineerd worden met een schrijfopdracht: writeln(x+y); uitvoering van deze opdracht heeft tot gevolg dat de waarde van x+y wordt geplaatst op de huidige regel van de uitvoerrij en dat er overgegaan wordt op het begin van een nieuwe regel.
Achtereenvolgende regels van de uitvoerrij kunnen samengenomen worden, ze vormen een "bladzijde". (Dit is bijvoorbeeld van belang als de uitvoerrij met behulp van een regeldrukker op papier wordt afgedrukt. We komen daar straks nog even op terug.) De opdracht page heeft tot gevolg dat de eerstvolgende waarde die in de uitvoerrij geplaatst wordt de eerste waarde is van de regel van een nieuwe bladzijde.

Een waarde wordt in de uitvoerrij vastgelegd met behulp van een vast aantal karakters, waarbij dit aantal bepaald wordt door het type van de waarde; zo zal een karakterwaarde altijd één plaats beslaan. Dit aantal plaatsen noemt men de breedte. Vaak is het gewenst om de breedte van een waarde niet aan het systeem over te laten (die dan dus een standaardbreedte neemt), maar om die zelf te kunnen bepalen. Ditzelfde geldt voor het aantal cijfers achter de decimale punt bij een waarde van het type real. De machine zal zo'n waarde altijd als 0.... of -0.... in de uitvoerrij plaatsen,

dus 48.53 wordt in de uitvoerrij geplaasts als 0.485300 E +02, waarbij het aantal nullen na de decimale punt bepaald wordt door de standaard-breedte voor waarden van het type real (in het voorbeeld is die breedte 12). Wellicht zouden we in dit geval prefereren dat er 48.53 in de uit-voerrij geplaatst wordt. Zowel de breedte als het aantal cijfers na de decimale punt kunnen in de schrijfopdracht vastgelegd worden.

```
<output value> ::= <expression>[:<field width>
                                [:<fraction length>]]
<field width> ::= <expression>
<fraction length> ::= <expression>
```

De(eerste)expressie is de waarde die in de uitvoerrij geplaatst wordt. De "field width" is de breedte en de "fraction length" is het aantal cijfers na de decimale punt bij een waarde van het type real. De field width en de fraction length zijn expressies die een positieve gehele waarde moeten opleveren. Als de opgegeven breedte groter is dan het werkelijke aantal benodigde karakters worden voor de waarde zelf eerst zoveel spaties ge-zet dat het totale aantal karakters gelijk is aan de opgegeven breedte. Als de opgegeven breedte kleiner is dan het werkelijke aantal benodigde karakters is het effect ongedefinieerd (de benodigde breedte zou gebruikt kunnen worden of de standaardbreedte).

Voorbeelden:

```
write(123:8)       ␣ ␣ ␣ ␣ ␣123   (␣ = spatie)
write(123:3)        123
write(-48.53)      -0.485300 E +02
write(-48.53:6:2)  -48.53
write(-48.53:11)   -0.4853 E +02
```

Eén vorm van uitvoerwaarde hebben we nog niet bekeken: de string. We kunnen een willekeurige tekst in de uitvoerrij laten plaatsen door de tekst tussen aanhalingstekens in de schrijfopdracht op te nemen:

```
write('dit is een voorbeeld')
write('het kwadraat van', n, 'is',n * n)
```

Afsluitende opmerkingen

De invoerrij krijgt zijn waarde via een invoerapparaat (bijvoorbeeld een kaartlezer). De uitvoerrij beeldt zijn waarde af via een uitvoerapparaat (bijvoorbeeld een regeldrukker of een beeldscherm). Uitvoering van lees- en schrijfopdrachten zorgen voor de communicatie tussen het programma en de invoer- en uitvoerrij. Het systeem zorgt voor de koppeling tussen invoerrij en invoerapparaat en tussen uitvoerrij en uitvoerapparaat. De terminologie van de lees- en schrijfopdrachten is echter afgestemd op de apparaten.

In de beschrijvingen in dit hoofdstuk zijn alleen de standaard invoerrij "input" en de standaard uitvoerrij "output" gebruikt. Ook andere invoer- en uitvoerrijen zijn mogelijk. De namen hiervoor moeten dan in de lees- en schrijfopdrachten opgenomen worden. (Van de standaardrijen mogen de namen opgenomen worden.)

"There does not now, nor will there ever, exist a programming language in which it is the least bit hard to write bad programs."

Lawrence Flon

6. Verwerking van een invoerrij

De verwerking van een rij objecten, die wat deze verwerking betreft gelijksoortig zijn, is een vaak voorkomend (deel)probleem. We zullen dit soort problemen hier bekijken en er "standaard" algoritmen voor geven. De aard van de objecten is daarbij in eerste instantie niet van belang.
Een rij kan voorgesteld worden als $e_1, e_2, \ldots, e_n$, waarbij ieder object (element) een (fictief) rangnummer heeft gekregen.
Voor iedere rij geldt:

- Er is een laatste element: e_n .
- Elk element, behalve het laatste, heeft een opvolger.
- De elementen zijn sequentieel selecteerbaar en daardoor ook sequentieel verwerkbaar.

Het is duidelijk dat de verwerking van zo'n rij door middel van een repetitie gebeurt waarin voorkomen "selecteer element" en "verwerk element". Voor het vaststellen van de in de repetitie voorkomende stopconditie is de herkenning van e_n, het laatste element, belangrijk. Er doen zich hierbij twee gevallen voor:

1. Van een verwerkt element is vast te stellen of dit het laatste element is. Dit geval treedt bijvoorbeeld op als a priori het aantal elementen van de rij bekend is. Het treedt ook op als de te verwerken rij gegeven is als een invoerrij, want dan zal na de selectie van dit laatste element de boolean functie eof de waarde true opleveren.
2. De rij van verwerkbare elementen wordt afgesloten door een extra element dat niet verwerkt behoeft te worden en dat als zodanig herkenbaar is. Zo'n extra element wordt een afsluiter genoemd.

Deze situatie treedt bijvoorbeeld op als er een als zodanig herkenbare deelrij van een rij verwerkt moet worden.

De verwerking van de elementen geeft een bepaald resultaat, dat wij in de algemene beschouwing die nu volgt "result" zullen noemen. De eindrelatie voor de verwerking van een rij elementen kunnen we nu voor beide hierboven genoemde gevallen aangeven:

result is het resultaat van de verwerking van de elementen e_1 tot en met e_n.

De invariante relaties P en Q voor beide gevallen laten zich eenvoudig afleiden uit de eindrelatie:

1. P(k): result is het resultaat van de verwerking van alle elementen tot en met e_k $(k \leq n)$;
2. Q(k): result is het resultaat van de verwerking van alle elementen tot en met $e_k (k \leq n)$ en de opvolger van e_k is geselecteerd.

Opmerking.

In beide gevallen zou de rij van te verwerken elementen leeg kunnen zijn. In dit geval kan worden voorzien door result te definiëren voor een lege rij. Wat we hiervoor moeten kiezen volgt uit de aard van de "echte" elementen.

De variante relatie is in beide gevallen: "het laatst geselecteerde element <>"laatste" element", waarbij het laatste element in het eerste geval een verwerkbaar element is en in het tweede geval de afsluiter.

We hebben dus gevonden:

1. *laatste element is herkenbaar*

eindrelatie R: $e_1, e_2, \ldots, e_n$ zijn geselecteerd en verwerkt;

invariant P(k): $e_1, e_2, \ldots, e_k$ zijn geselecteerd en verwerkt;

variant B: er is nog een verwerkbaar element dat geselecteerd kan worden.

1a. *aantal elementen is bekend (=n)*

```
k := 0;
"definieer result voor lege rij"; {P(0)}
while k <> n do
   begin  {P(k)}
         k := k + 1; {P(k-1)}
         "selecteer k-de element";
         "verwerk geselecteerde element {P(k)}
   end {P(n)}
```

1b. *invoerrij (aantal niet bekend)*

```
"definieer result voor lege rij"; {P(0)}
while not eof do
   begin  {P(k)}
         "selecteer volgende element";
         "verwerk geselecteerde element {P(k)}
   end  {P(n)}
```

2. *afsluiter die niet verwerkt wordt*

eindrelatie R: $e_1, e_2, \ldots, e_n$ zijn geselecteerd en verwerkt en de afsluiter is geselecteerd;

invariant Q(k): $e_1, e_2, \ldots, e_k$ zijn geselecteerd en verwerkt en de opvolger van e_k is geselecteerd;

variant B: geselecteerde element is verwerkbaar.

```
"definieer result voor lege rij";
"selecteer eerste element"; {Q(0)}
while "geselecteerde element is nog verwerkbaar do
   begin {Q(k)}
         "verwerk geselecteerde element";
         "selecteer volgende element" {Q(k)}
   end {Q(n)}
```

In de gevallen 1b en 2 treedt k in de gegeven beschrijving op als ghost variable.

We gaan nu, aan de hand van de voorgaande algemene beschouwingen, enkele voorbeelden bekijken.

Voorbeeld 1

De invoerrij bestaat uit een rij gehele getallen.

Gevraagd wordt het minimum van deze getallen te bepalen.

De gewenste eindrelatie is: min = "het minimum van de getallen". De invariante relatie P is: min = "het minimum van de getallen tot en met het laatst geselecteerde getal". Hoe houden we bij verwerking van een volgend getal P invariant? Stel $min_{k-1} = min(e_1,e_2,\ldots,e_{k-1})$, dan geldt $min_k = min(min_{k-1},e_k)$. Blijft nog over de situatie beschreven door P(0), met andere woorden wat is het minimum van een lege rij. We kunnen dit minimum definiëren als maxint.

```
min := maxint;
while not eof do
   begin read(getal);
         if getal <min then min := getal
   end
```

Opmerking.

In veel gevallen staat vast dat een rij die verwerkt moet worden niet leeg is. In dat geval kan worden gezorgd dat voor de repetitie P(1) (of Q(1)) geldt. In het voorbeeld:

```
read(getal); min := getal;
while not eof do
   begin read(getal);
         if getal < min then min := getal
   end
```

Voorbeeld 2

De invoerrij bestaat uit een stijgende rij getallen gevolgd door een dalende rij. (Stel dat de invoerrij $e_1,e_2,\ldots,e_n$ is dan is er dus een m zodanig dat $e_1,e_n,\ldots,e_m$ stijgend is en $e_m,e_{m+1},\ldots,e_n$ dalend.)
Beide deelrijen zijn niet leeg ($m > 1, n > m$).
Gevraagd wordt de som te bepalen van de stijgende deelrij.
Eindrelatie R: som = $e_1 + e_2 + \ldots + e_m$ en e_{m+1} is geselecteerd (om te constateren dat de dalende rij begint)
Invariante relatie Q: som = $e_1 + e_2 + \ldots + e_k$ en e_{k+1} is geselecteerd.
Variante relatie B: $e_{k+1} > e_k$
Omdat beide deelrijen niet leeg zijn kunnen we voor de repetitie Q(1) realiseren. Steeds hebben we voor B zowel e_k als e_{k+1} nodig; we zullen deze aangeven met de variabelen g1 respectievelijk g2.

```
read(g1); som := g1; read(g2);
while g2 > g1 do
   begin som := som + g2;
         g1 := g2;
         read(g2)
   end
```

Voorbeeld 3

De invoerrij bestaat uit een rij positieve gehele getallen: $e_1, e_2, \ldots, e_n$.
Gevraagd wordt de som te bepalen van $e_1, e_3, e_6, e_{10}, \ldots$

De gewenste eindrelatie kunnen we formuleren als:

som = "de som van de elementen van de deelrij $e_1, e_3, e_6, e_{10}, \ldots$".

Wat de selectie betreft zijn alle getallen gelijkwaardig.
Wat de verwerking betreft zijn de getallen niet gelijkwaardig, er zijn "sommeerbare" getallen ($e_1, e_3, e_6, e_{10}, \ldots$), die in de variabele som verwerkt moeten worden en "niet-sommeerbare" getallen, die na de selectie genegeerd moeten worden.
We kunnen als invariante relatie nemen:

som = "de som van de sommeerbare elementen tot en met het sommeerbare getal dat een rangnummer heeft dat ten hoogste gelijk is aan het rangnummer i van het laatst geselecteerde getal".

Als algoritme krijgen we:

```
som := 0; i:= 0;
while not eof do
   begin read(getal); i:= i+1;
         if "getal is sommeerbaar " then som := som + getal
   end
```

Een sommeerbaar getal is een getal waarvan het rangnummer tot de kandidaten 1,3,6,10,... behoort. De rij van kandidaten kan als volgt gevonden worden: de eerste kandidaat is 1, het verschil tussen de tweede en de eerste kandidaat is 2 en elk volgend tweetal kandidaten heeft een verschil dat 1 groter is dan het verschil tussen de vorige twee.
Om nu het rangnummer van de geselecteerde getallen te kennen en te weten of dit rangnummer tot de kandidaten behoort, breiden we de invariant uit met:

kan: het rangnummer van het eerstvolgende sommeerbare getal

v: het verschil tussen 'kan' en het rangnummer van het voorafgaande sommeerbare getal.

Voor de lege rij kunnen we kan de waarde 1 geven en v de waarde 1 (op grond van de rest van de rij).

Het algoritme wordt nu:

```
kan := 1; v := 1; i := 0; som := 0;
while not eof do
   begin read(getal);
         i := i + 1;
         if i = kan
            then begin som := som + getal;
                       v := v + 1;
                       kan := kan + v
                 end
   end
```

We kunnen dit probleem ook op een andere manier aanpakken. In de voorgaande oplossing hebben we de invoerrij opgevat als een rij elementen die - wat de verwerking betreft - niet allemaal gelijkwaardig zijn:
er zijn sommeerbare en niet-sommeerbare elementen. We kunnen de invoerrij ook opvatten als een rij deelrijen. Een deelrij begint na een vorige deelrij en eindigt met een (in termen van de vorige oplossing) sommeerbaar element. Dit geldt echter niet persé voor de laatste deelrij. We denken daarom de totale rij uitgebreid met de waarde -1; de laatste deel-

rij eindigt nu met -1. De invoerrij bestaat dan uit een rij verwerkbare deelrijen, afgesloten door een niet verwerkbare deelrij (afsluiter).
Het algoritme wordt:

```
"selecteer deelrij"; som := 0;
while "dit is een verwerkbare deelrij" do
   begin som := som + "laatste element van deelrij";
         "selecteer deelrij"
   end
```

Van een deelrij behoeft alleen de laatste waarde behouden te blijven; van een verwerkbare deelrij om deze op te tellen bij de som en anders om te constateren dat het om de afsluiter gaat (-1). Deze laatste waarde leggen we vast in de variabele w. De variante relatie is dan $w \neq -1$.
Om de volgende deelrij te kunnen selecteren moeten we de lengte van de laatst geselecteerde deelrij kennen, want iedere volgende deelrij is 1 langer dan de vorige deelrij (eventueel afgezien van de laatste). Daarom nemen we in de invariant op

ℓ: lengte van de laatst geselecteerde deelrij.

Het algoritme wordt nu:

```
som := 0;
if eof then w := -1 else read(w);
ℓ := 1;
while w <>-1 do
   begin {verwerk de geselecteerde deelrij:}
         som := som + w;
         {selectie van volgende deelrij:}
         ℓ := ℓ + 1; {lengte van te selecteren rij}
         i := 0; {aantal geselecteerde elementen}
         while i ≠ ℓ ∧ w ≠ -1 do
```

```
            begin if eof then w := -1 else read(w);
                  i := i + 1
            end
end
```

7. Samengestelde variabelen: records en arrays

De tot nu toe besproken variabele kan alleen enkelvoudige waarden aannemen. Dat wil zeggen dat de waarden op het niveau van de operaties van het type niet op te splitsen zijn in delen.

Voor de enkelvoudige variabele kunnen we resumeren:

- een variabele is een toestandsgrootheid;
- een variabele wordt geïdentificeerd door een naam;
- een variabele is van een bepaald type;
- het type en de naam van een variabele worden vastgelegd door middel van een declaratie.

Er zijn grootheden die door meer dan één waarde gekarakteriseerd worden. Naast enkelvoudige variabelen bestaan er ook samengestelde variabelen. Een samengestelde variabele heeft ook de bovenstaande eigenschappen, maar kenmerkend voor een samengestelde variabele is dat zijn waarde bestaat uit een samenstel van componentwaarden. Een samengestelde variabele kunnen we dan ook opvatten als een samenstel van componenten. (In plaats van componenten spreken we ook wel van elementen, velden of items.) Iedere component heeft een waarde, die een enkelvoudige waarde is of weer een samenstel van waarden. De manier waarop de componenten van een samengestelde variabele samenhangen, noemen we de *structuur* van de variabele. De structuur bepaalt de wijze waarop de componenten selecteerbaar zijn. We bekijken hier de record-structuur en de array-structuur.

Het aantal componenten, het type van de componenten en de eventueel andere operaties op de structuur bepalen het *type* van de variabele. Met een zelfde structuur kunnen dus verschillende typen gedefinieerd worden.

Een punt in de driedimensionale ruimte kan in ons programma gerepresenteerd worden door een variabele, die bestaat uit drie componenten, die respectievelijk de x-, de y- en de z-coördinaat van het punt voorstellen. Zo ook zou in een bepaalde toepassing de variabele persoonsgegevens kunnen optreden waarvan de componenten de naam, het adres en de geboortedatum van een persoon aangeven; hierbij zou de laatste component op zich weer kunnen bestaan uit drie componenten: de dag, de maand en het jaar. In een andere toepassing zouden we bijvoorbeeld van een groep mensen willen vastleggen hoe vaak de lengtes 150,151,...,205 (in centimeters) voorkomen en willen we een tabel van lengtes en aantallen (i.p.v. 56 afzonderlijke variabelen).

Bij het spreken over samengestelde waarden kunnen we drie niveaus onderscheiden:

1. Het wiskundig niveau (modelniveau).

   Op dit niveau kijken we naar de wiskundige eigenschappen van een structuur. Op dit niveau wordt alleen gebruik gemaakt van wiskundig gereedschap.

2. Het algoritmisch niveau.

   Op dit niveau gaan we uit van de structuren met hun bijbehorende operaties zoals die in onze programmeertaal (Pascal) aanwezig zijn. Met behulp van deze structuren definiëren we nieuwe typen.

3. Het implementatieniveau.

   Het kan zijn dat voor bepaalde soorten wiskundige verzamelingen geen structuur in de taal aanwezig is. We zouden zo'n structuur dan kunnen simuleren in ons programma.

We beperken ons hier tot het algoritmische niveau. We definiëren een nieuw

type (met een structuur uit de taal) als het programmeren op het algoritmische niveau gemakkelijker wordt door het gebruik van variabelen van het nieuwe type.

De record-structuur

Het komt vaak voor dat een grootheid gekarakteriseerd wordt door een aantal, eventueel samengestelde, waarden die niet noodzakelijk tot het zelfde type behoeven te behoren; deze componenten krijgen een naam om ze van elkaar te onderscheiden. Zo kunnen we een punt in het platte vlak karakteriseren door de x- en de y-coördinaat en personen kunnen we beschrijven door een aantal eigenschappen zoals naam, geboortedatum, geslacht en dergelijke.
We zeggen in zo'n geval dat een variable (een waarde) een record-structuur heeft. We zullen nu eerst deze structuur aan de hand van enkele voorbeelden introduceren en daarna de syntaxis formeel geven.
Voor de punten in het platte vlak kunnen we als type definiëren:

```
type punt = record x,y: real end
```

Daarna kunnen variabelen van dit type gedeclareerd worden:

```
var p1,p2: punt
```

De variabele p1 bestaat uit twee componenten; selectie van de componenten, die beide van het type real zijn, noteren we als p1.x en p1.y (zo ook voor p2).
Stel dat gedefinieerd zijn de typen:

```
type dag = 1..31;
     maand = 1..12;
     jaar = 1900..1999
```

Deze drie enkelvoudige typen gebruiken we voor de definitie van het samengestelde type:

```
type datum = record d: dag;
                    m: maand;
                    j: jaar
             end
```

Een variabele van het type datum wordt gedeclareerd als:

```
var dat: datum.
```

De syntaxis voor een record-type in Pascal luidt:

```
<record type> ::= record <field list> end
<field list> ::= <fixed part>[; <variant part>]|
                 <variant part>
<fixed part> ::= <record section>{; <record section>}
<record section> ::= [<identifier list>: <type>]
<identifier list> ::= <field identifier>{, <field identifier>}
<field identifier> ::= <identifier>
```

In een record-type vormen de symbolen record en end dus een hakenpaar, waartussen de componenten opgesomd worden elk met een eigen naam (field identifier) en met een type. Alle componentnamen van een record-type moeten verschillend zijn (in verschillende record-typen mogen dezelfde componentnamen voorkomen).
Het variante deel komt straks aan de orde.

Als een record-type gedefinieerd is kunnen variabelen van dit type gedeclareerd worden. We kunnen de definitie van het type ook in de declaratie zelf opnemen:

```
var p3,p4: record x,y,z: real end
```

(nu krijgt het type dus geen naam).

Waardetoekenning kan plaatsvinden op het niveau van de variabelen:

```
p3 := p4
```

waarbij p3 en p4 van hetzelfde record-type moeten zijn (wat hier het geval is).

We kunnen met een record-variabele ook manipuleren op het niveau van de componenten:

```
<field designator> ::= <record variabele>.<field identifier>
<record variabele> ::= <variabele>
```

Bij de bovenstaande variabele dat van het type datum, kunnen we drie componenten selecteren:

```
dat.d  van  het type dag
dat.m  van  het type maand
dat.j  van  het type jaar
```

Een component van een bepaald type (van een record variabele) mag overal gebruikt worden waar ook een "gewone" variabele van zo'n type gebruikt mag worden:

```
dat.d  :=  26
if dat.m = 2 then ...
write(dat.d, dat.m, dat.j)
```

We mogen dus individuele componenten van een record-variabele van waarde veranderen. We moeten er daarbij wel aan denken dat daarmee de totale record-waarde verandert. We noemen de verandering van één component de selectieve toekenning.

De with-statement

Stel dat we de variabele dat van het type datum, een waarde willen toekennen. Dit kan met:

```
dat.d := 22; dat.m := 5; dat.j := 1982
```

Het in zo'n geval steeds maar weer moeten opschrijven van de naam van de variabele kan in Pascal voorkomen worden door gebruik te maken van de zogenaamde with statement:

```
<with statement> ::= with <record variable>
                         do <statement>
```

Ons voorbeeld zou hiermee worden:

```
with dat do begin d := 22; m := 5; j := 1982 end
```

Voorbeeld

Het onderstaande programmafragment bepaalt bij een gegeven waarde van het type datum (deze waarde is de waarde van de variabele dat) de waarde die de volgende dag voorstelt (ook weer als waarde van de variabele dat). We gebruiken een variabele aantal van het type 28..31 die het aantal dagen van een maand zal voorstellen. Verder gaan we er van uit dat het type datum gedefinieerd is en dat de variabele dat gedeclareerd is en een waarde heeft.

```
with dat do
  begin case m of
          1,3,5,7,8,10,12: aantal := 31;
          4,6,9,11       : aantal := 30;
          2: if (j mod 4 = 0) and (j <> 1900)
               then aantal := 29
               else aantal := 28
     end;
     if d = aantal
        then begin d := 1;
                   if m = 12
```

```
                    then begin m := 1;
                               j := j + 1
                         end
                    else m := m + 1
               end
          else d := d + 1
     end
```

Het variante record

Stel dat we hebben:

```
type manp = record gebdat : datum;
                   hk     : (blond,zwart,grijs);
                   gewicht: 1..200;
                   baard  : boolean
            end
```

en

```
type vrouwp  = record gebdat: datum;
                      hk    : (blond,zwart,grijs);
                      b,t,h : 1..150
               end
```

Met behulp van het in de syntaxis aangegeven variante deel kunnen we één type persoon definiëren waarbij nog wel onderscheid gemaakt kan worden tussen mannen en vrouwen

```
<variant part> ::= case <tag field><type identifier> of
                       <variant>{; <variant>}
<tag field> ::= [<identifier>:]
<variant> ::= [<case label list> : (<field list>)]
```

Ons ene type zouden we kunnen definiëren door

```
type sexe = (man, vrouw);
type persoon = record gebdat : datum;
                      hk     : (blond,zwart,grijs);
                      case s : sexe of
                        man  : (gewicht: 1..200; baard: boolean);
                        vrouw: (b,t,h: 1..150)
               end
```

Iedere variabele van het type persoon bestaat in elk geval uit de componenten gebdat, hk en s; als s de waarde man heeft, heeft de variabele nog de componenten gewicht en baard en anders (als s de waarde vrouw heeft) heeft de variabele nog de componenten b, t en h. Een selectieve toekenning als p.b := 90 is dus alleen toegestaan als p.s de waarde vrouw heeft.
Uit de syntaxis voor record-typen blijkt dat het vaste deel mag ontbreken. In het variante deel mag het "tag field" ontbreken, dat wil zeggen dat deze niet als component voorkomt in het record.

De array-structuur

Bij de typen met record-structuur spelen de namen van de componenten de rol van selectoren. Een structuur waarin alle componenten van hetzelfde type zijn en waarin de selectorverzameling een type is, is de array-structuur. De waarde van een variabele van een type met de array-structuur kunnen we beschrijven als een afbeelding van het selectortype in het componenttype. Het selectortype wordt ook wel het indextype of domeintype genoemd.

We kunnen dus een type met array-structuur gebruiken als we een verband tussen twee waardenverzamelingen (typen) willen aangeven. Als we in ons programma bijvoorbeeld aantallen willen bijhouden bij bepaalde lengtes, zoals

lengte	aantal
150	3
151	7
152	12
.	.
.	.
.	.
.	.
.	.
204	21
205	6

dan kunnen we deze gegevens (deze tabel) vastleggen in een array. Dit array wordt dan gedeclareerd als

```
var verdeling array [50..205] of integer
```

Algemeen geldt voor een array-type:

```
<array type> ::= array [<index type>{, <index type>}]
                        of <type>
<index type> ::= <type>
```

In deze syntaxis hebben we een vermenging van de te definiëren taal (Pascal) en de definitietaal (metataal). De rechte haken in deze definitie zijn namelijk "Pascal-haken".

Het index type mag ieder enkelvoudig type zijn, uitgezonderd het type real. Een variabele van een array-type heeft zoveel componenten als het index type waarden heeft. Het index type zal vaak een subrange van de integers zijn, maar dit hoeft niet.

```
type loket = (a,b,c);
     situatie = (bezet,vrij);
     loketbezetting = array [loket] of situatie
```

De waardenverzameling van het type loketbezetting bestaat uit 8 waarden. Een component van een array-variabele wordt geselecteerd door de naam van de variabele gevolgd door - tussen rechte haken - een waarde van het indextype; dit kan zijn een constante, een variabele of een expressie, als er maar een waarde wordt opgeleverd van het indextype.

<indexed variable> ::= <array variable>[<expression>
{,<expression>}]

<array variable> ::= <variable>

Zo zijn verdeling[175], verdeling[i] (met 150 ≤ i ≤ 205) en verdeling[p * q + r] (met 150 ≤ p * q + r ≤ 205) componenten van het array verdeling. Een component mag overal in het programma voorkomen waar een variabele die van hetzelfde type is als het componenttype, mag voorkomen.

```
verdeling[175] := 542
verdeling[k] := verdeling[k] + 1
if verdeling[i] > 400 then ....
```

Een waarde van het indextype wordt de *index* genoemd. Met behulp van een index wordt dus een component geselecteerd.

Een array (een variabele met een array-type) kan gebruikt worden om een groot aantal "variabelen" (indexed variabelen) in een keer te declareren. Bovendien kan daar op een gemakkelijke manier mee gemanipuleerd worden doordat de index een expressie mag zijn. Door de declaratie

```
var x: array [1..100] of boolean
```

worden honderd variabelen gedeclareerd (x[1],x[2],...,x[100]), alle van het type boolean. Als al deze variabelen de waarde true moeten krijgen kan dat eenvoudig met de for statement (zie hoofdstuk 1, p.19):

```
for i := 1 to 100 do x[i] := true
```

Als de variabele

```
var a: array [1..100] of integer
```

een waarde heeft (alle honderd componenten hebben een waarde) dan kan het maximum van de componenten bepaald worden door:

```
max := a[1]; i := 1;
while i <> 100 do
   begin i := i + 1;
         if a[i] > max then max := a[i]
   end
```

Met als invariant: max = maximum(a[1],a[2],...,a[i]).

Het kan ook met de for statement

```
max := a[1];
for i := 2 to 100 do if a[i] > max then max := a[i]
```

De for statement is niets anders dan een "afkorting" voor de while statement. Het bovenstaande stukje programmatekst met de for statement komt overeen met

```
max := a[1];
i := 2;
while i <= 100 do
   begin if a[i] > max then max := a[i];
         i := i + 1
   end
```

met als invariant : max = maximum(a[1],a[2],...,a[i-1]).

De twee stukjes programma zijn niet geheel equivalent. Na verwerking van de for statement is de waarde van de variabele, in het voorbeeld i, ongedefinieerd.

In de statement na do in de for statement mag de variabele (in het voorbeeld i) niet gewijzigd worden.

De variabele mag niet van het type real zijn.

Stel dat we hebben

```
type lengte = 150..205;
     lengteverdeling = array [lengte] of 1..maxint;
var lvd: lengteverdeling
```

Dan geeft lvd[170] het aantal behorende bij lengte 170 (als lvd een waarde heeft).

Het is mogelijk dat het component-type op zich weer een array-structuur heeft. Zo kunnen we bijvoorbeeld introduceren

```
type gewicht = 40..200;
     gewichtsverdeling = array [gewicht] of lengteverdeling;
var gvd: gewichtsverdeling
```

Er geldt dat gvd[180] van het type lengteverdeling is en dus zelf weer bestaat uit 56 componenten, die weer geselecteerd kunnen worden. Zo stelt gvd[80][175] het aantal voor bij gewicht 80 en lengte 175.

We zouden ook kunnen definiëren en declareren (zie de syntaxis):

```
type lgverdeling = array [gewicht,lengte] of 1..maxint;
var lgvd: lgverdeling
```

Een component van lgvd wordt geselecteerd met behulp van twee indices: lgvd[80,175]. Let goed op het verschil met de vorige definities!

Het bovenstaande array lgvd wordt een twee-dimensionaal array genoemd. Uit de syntaxis blijkt dat een willekeurig aantal dimensies is toegestaan. Bij een n-dimensionaal array wordt een component geselecteerd met behulp van n indices.

Een twee-dimensionaal array kan gebruikt worden om een matrix te representeren (zoals een één-dimensionaal array gebruikt kan worden voor de representatie van een vector).

Bijvoorbeeld:

```
var a: array [1..10,1..15] of real;
    b: array [1..15,1..20] of real;
    c: array [1..10,1..20] of real;
    i: 1..10; j: 1..15; k: 1..20;
    som: real;
begin {lees 10 * 15 matrix in en zet die in a}
      for i := 1 to 10 do
          for j := 1 to 15 do read(a[i,j]);
      {lees 15 * 20 matrix in en zet die in b}
      for j:= 1 to 15 do
          for k:= 1 to 20 do read(b[j,k]);
      {vorm productmatrix c: c[i,k] = sum_{j=1}^{15} a[i,j] * b[j,k] voor
       i = 1,2,...,10 en k = 1,2,...,20}
      for i:= 1 to 10 do {rijen 1,2,...,i-1 van c zijn gevormd}
          for k:= 1 to 20 do {van rij i van c zijn kolommen
                                        1,2,...,j-1 gevormd}
          begin som := 0;
                for j := 1 to 15 do
                    som:= som + a[i,j] * b[j,k];
                c[i,k]:= som
          end
end
```

In het bovenstaande voorbeeld wordt met de arrays gemanipuleerd op componentniveau. Dit kan ook niet anders want de enige operatie die is toegestaan op arrayniveau is de assignment

```
a1 := a2
```

Hierin moeten a1 en a2 van hetzelfde type zijn.

Opmerking

Over het algemeen zal iedere component van een array (als het een enkelvoudig type is) een machinewoord vragen voor de representatie van de waarde, zelfs als - gezien de waardenverzameling (bv. {true,false}) - die niet noodzakelijk is. Het is in Pascal mogelijk zuiniger te zijn door:

var a := packed array[1..n] of type

Het systeem zal dan proberen zuiniger te zijn. We gaan er niet verder op in.

(einde opmerking)

Voorbeelden met arrays

Voorbeeld 1

Het array

var a: array [1..n] of integer

waarin n een constante is, heeft een zodanige waarde dat geldt: $a[i] \leq a[i+1]$ voor $i = 1,2,\ldots n-1$. (Het array is niet-dalend gesoteerd.)

Gevraagd wordt om door herrangschikking van de waarden van de componenten te bereiken dat vanaf index 1 eerst (in stijgende volgorde) alle onderling verschillende waarden komen en daarna de waarden die in het oorspronkelijke array meer dan eens voorkomen.

(Geen hulparray gebruiken en de "enige" operatie is dat twee componenten hun waarden omwisselen.)

We introduceren de variabelen p en q en nemen als invariant:

a[1..p] : verschillende (a[i] < a[i+1])

a[p+1..q] :"dubbele"(komen ook voor in a[1..p])

a[q+1..n] : nog te bekijken"

Deze invariant is een afzwakking van de eindrelatie, omdat wat in de eindrelatie moet gelden voor a[1..n] in de invariant geldt voor a[1..q].

(De array-elementen a[q+1.. n] hebben nog hun oorspronkelijke waarde en er geldt dus met name a[q+1] ≥ a[j] voor j = 1,2,...,q.)

Initialisatie : p := 1; q := 1 {invariant geldt}

Variant: q <> n

Eindigheid: q met 1 ophogen

Invariant herstellen:

a[q] ≥ a[j] voor j = 1,2,...,q-1

Twee mogelijkheden:

a[q]= a[p]; dan geldt invariant, of

a[q] > a[p]; invariant te herstellen door

-a[q] met a[p+1] te verwisselen

-p met 1 op te hogen.

Algoritme:

```
p := 1; q := 1;
while q <> n do
   begin q := q+1;
         if a[q] > a[p]
            then begin h := a[p+1];
                       a[p+1] := a[q];
                       a[q] := h;
                       p := p+1
                 end
   end
```

Opmerkingen

In de repetitie wordt driemaal p+1 uitgerekend; dit kan vermeden worden door met de assignement p := p+1 te beginnen.

Uit de eindrelatie zijn door afzwakking ook andere invariante relaties af te leiden. Probeer dat eens en kijk wat voor verschillen dat oplevert in het algoritme.

Voorbeeld 2

Gegeven is dat de variabelen

```
var a: array [1..n] of integer;
    x: integer
```

waarin n een constante is, een waarde hebben en dat er geldt: $a[1] \leq a[2] \leq \ldots \leq a[n]$.

Gevraagd wordt om een programma(fragment) te schrijven dat aan de variabele k de waarde i toekent als er een i is zodanig dat $a[i] = x$ en dat aan k de waarde 0 toekent als er geen enkele i is met $a[i] = x$ $(1 \leq i \leq n)$.

Als invariant kiezen we de volgende afzwakking van de eindrelatie:

$(k = 0 \wedge$ er is geen i, $1 \leq i \leq p$, met $a[i] = x) \vee$

$(k = i \wedge$ er is een i, $1 \leq i \leq p$, met $a[i] = x)$

Initialisatie : k := 0; p := 0 {invariant geldt}

Variant: $p <> n$

Eindigheid: p met 1 ophogen

Algoritme:

```
k := 0; p := 0;
while p <> n
   begin p := p+1;
         if a[p] = x then k := p
   end
```

Uit de invariant zien we dat $p = n$ inderdaad nodig is voor het eerste deel van de disjunctie. De invariant levert echter ook de eindrelatie op als voor een $p < n$ geldt dat er i is, $1 \leq i \leq p$, met $a[i] = x$. Deze andere

variant leidt tot de oplossing:

```
k := 0; p := 0;
while p <> n  and k = 0 do
   begin p := p + 1;
         if a[p] = x then k := p
   end
```

In de bovenstaande oplossingen is totaal geen rekening gehouden met het feit dat het array niet-dalend gesorteerd is. We zouden hier bij het formuleren van de invariant en/of de variant al rekening mee kunnen houden.

Invariant: voor alle i, $1 \leq i < p$, geldt: $a[i] \neq x$

We kunnen bij deze invariant de eindrelatie realiseren als $a[p] \geq x$ of als $p = n+1$. Als $p = n+1$ zouden we in het vergelijken van a[p] met x refereren aan een niet-bestaand array-element. We nemen daarom als variant: $(a[p] < x)$ and $(p<>n)$. Waarna, als de repetitie eindigt met $p = n$ nog apart gekeken moet worden naar a[n]. Maar omdat na de repetitie k nog zijn waarde moet krijgen, moest dit toch al gebeuren (ook als $p \neq n$).

Algoritme:

```
p := 1;
while (a[p] < x) and (p<> n ) do p := p + 1;
if a[p] = x then k := p else k := 0
```

In dit algoritme wordt het "gebied", waarin x moet liggen àls x voorkomt, steeds 1 kleiner (het gebied is a[p..n]). Van de sortering is gebruik gemaakt om te kunnen stoppen zodra de gewenste informatie (x komt voor of niet) bekend is. We zouden kunnen proberen van de sortering gebruik te maken om het gebied steeds met meer dan 1 te verkleinen. (Denk bijvoorbeeld aan het zoeken in een telefoonboek.) Dit kan op een systematische

manier door steeds het gebied in twee delen te splitsen, te kijken in welk van de twee delen x voorkomt als x voorkomt en het zoeken tot dit deel te beperken. Als het gebied nu 1 groot is geworden moet deze component van a gelijk zijn aan x of x komt niet voor. Laten we direct het algoritme opschrijven:

```
i := 1; j := n
while i < j do
   begin m := (i+j) div 2;
         if x < a[m] then j := m else i := m
   end;
if x = a[i] then k := i else k := 0
```

Het algoritme is echter fout. Waarom? Is het eenvoudig te verbeteren? Laten we het algoritme systematisch afleiden via relaties. Als invariant kiezen we:

P: Als x voorkomt in a[1..n] dan komt x voor in a[i..j-1].

Initiëel, met i = 1 en j = n+1, geldt deze invariant uiteraard.

Als we weten te bereiken j = i+1 dan weten we of x voorkomt (x = a[i]?).

Om beëindiging te garanderen moet het verschil j-i kleiner worden. Als geldt i < m < j dan kan dat door de statements i := m of j := m. We hebben dus gevonden:

```
i := 1; j := n+1 ;
while j > i+1 do
   begin m := ...; {i < m < j}
         "i := m of j := m en herstel invariant"
   end;
if x = a[i] then k := i else k := 0
```

Omdat $j > i+1$ levert $m := (i+j)$ div 2 een m met $i < m < j$. De statement $i := m$ laat P invariant als $x \geq a[m]$ (P gold en $i < m < j$); zo ook laat de statement $j := m$ P invariant als $x < a[m]$.

Het algoritme luidt:

```
i := 1; j := n; {P}
while j > i + 1 do
   begin m := (i+j) div 2; {P ∧ i < m < j}
         if x ≥ a[m] then {P ∧ x ≥ a[m]}i:= m {P}
                     else {P ∧ x < a[m]} j:=m {P}
   end;{P ∧ j = i+1}
if x = a[i] then k := i else k := 0
```

We zouden voor de gevallen $x < a[1]$ en $x > a[n]$ het zoeken kunnen vermijden door het bovenstaande algoritme alleen uit te laten voeren onder de conditie $a[1] \leq x \leq a[n]$ (if $x \geq a[1] \wedge x \leq a[n]$ then "algoritme" else k := 0)

Voorbeeld 3

We definiëren de verzameling M als volgt:

- $1 \in M$
- als $x \in M$ dan ook $2x + 1 \in M$ en $3x + 1 \in M$
- M bevat geen andere elementen.

De kleinste vijftien elementen van M zijn: 1, 3, 4, 7, 9, 10, 13, 15, 19, 21, 22, 27, 28, 31 en 39.

Gevraagd wordt een algoritme dat de kleinste honderd elementen van M genereert.

Omdat we de kleinste honderd elementen zoeken ligt het voor de hand om de elementen in volgorde van grootte te genereren. Bij het genereren van een

element van M is een vorig element nodig. We introduceren daarom

```
var m : array [1..100] of integer
```

De eindrelatie is dan: m[i] is het i-de getal uit de verzameling M voor i = 1,2,...,100.

Invariant : m[i] is het i-de getal uit de verzameling M voor $i = 1,2,\ldots,k$.

Initialisatie: m[i] := 1; k := 1 {invariant geldt}

Variant: $k \neq 100$

Eindigheid: k met 1 ophogen.

Algoritme:

```
m[1] := 1; k := 1;
while k <> 100 do
   begin k := k+1;
         "invariant herstellen"
   end
```

m[k] moet "gevuld" worden met de kleinste waarde (verkregen via 2x + 1 of 3x + 1 met $x \in$ m[1..k-1]) van M, die nog niet is opgenomen. Daarom moet "onthouden" worden welke m[i] al gebruikt zijn om via 2 * m[i] + 1 of via 3 * m[i] + 1 een element van M te berekenen. We breiden daarom de invariant uit met:

- voor i = 1,2,...,a-1 is 2 * m[i] + 1 al opgenomen in m[1..k]
- voor i = 1,2,...,b-1 is 3 * m[i] + 1 al opgenomen in m[1..k]

Initialisatie: m[1] := 1; a := 1; b := 1 {invariant geldt}

Variant: $k \neq 100$

Eindigheid: k met 1 ophogen

Invariant herstellen: m[k] := 2 * m[a] + 1; a := a + 1

m[k] := 3 * m[b] + 1; b := b + 1

afhankelijk van het feit welke de kleinste is.

Algoritme:

```
m[1] := 1; k := 1; a := 1; b := 1;
while k <> 100 do
   begin k := k+1;
         y := 2*m[a]+1; z := 3*m[b]+1;
         if y <= z then begin m[k] := y;
                              a := a+1
                        end;
         if y >= z then begin m[k] := z;
                              b := b+1
                        end
   end
```

We zouden y en z in de invariant kunnen opnemen ($y = 2 * m[a] + 1$, $z = 3 * m[b] + 1$). Verandert het algoritme? En zo ja, is het dan beter?

Voorbeeld 4

Gegeven is dat

```
var b:array [0..99] of integer
```

een zodanige waarde heeft dat $b[i] \leq b[i+1]$ voor $i = 0,1,2,\ldots,98$.

Gevraagd wordt welke waarde het meest voorkomt en hoe vaak.

Dit probleem wordt wel het plateau-probleem genoemd omdat een rijtje gelijke waarden kan worden gezien als een plateau in een verder stijgende omgeving.

De eindrelatie luidt (in de terminologie van de plateaus): p is de lengte en m de hoogte van het langste plateau van b[0..99].

Als invariant zouden we kunnen proberen:

p is de lengte en m de hoogte van het grootste plateau van b[0..i-1] en

ℓ is de lengte en h de hoogte van het "plateau" waarop b[i-1] ligt.

Initialisatie: i := 1; p := 1; m := b[0]; ℓ := 1; h := b[0] {invariant geldt}

Variant: i ≠ 100

Eindigheid: i met 1 ophogen

Algortime:

```
i := 1; p := 1; m := b[0]; ℓ := 1; h := b[0];
while i <> 100 do
   begin if b[i] <> h
            then {nieuw plateau begint}
                 begin ℓ := 1; h := b[i] end
            else  begin ℓ := ℓ + 1;
                        if ℓ > p
                           then begin p := ℓ; m := h end
                  end;
         i := i + 1
   end
```

In dit algoritme is de h niet nodig omdat we overal waar h gebruikt wordt ook b[i-1] kunnen gebruiken. Zouden we ook ℓ kunnen missen? Met andere woorden: lukt het om het algoritme te construeren als we als invariant nemen:

p is de lengte en m de hoogte van het grootste plateau van b[0..i-1]

Initialisatie: i := 1; p := 1; m := b[0] {invariant geldt}

Eindigheid: i met 1 ophogen

Algoritme:

```
i := 1; p := 1; m := b[0];
while i <> 100 do
   begin "p en m eventueel aanpassen voor b[0..i]";
         i := i + 1
   end
```

Wanneer behoeft p niet aangepast te worden? Wanneer is dus p de lengte van het grootste plateau van b[0..i]? Dit is zo dan en slechts dan als b[i-p..i] geen plateau is. En het is geen plateau dan en slechts dan als b[i] ≠ b[i-p].

Als b[i-p..i] een plateau is (het grootste) moet de m bewaard worden.

Zo hebben we gevonden:

```
i := 1; p := 1; m := b[0];
while i <> 100 do
   begin if b[i] = b[i-p]
            then begin p := p + 1; m := b[i] end;
         i := i + 1
   end
```

8. Procedures

De assignment statement beschrijft een actie. Deze actie heeft een toestandstransformatie tot gevolg: een begintoestand gaat over in een eindtoestand. De eindtoestand verschilt van de begintoestand doordat een variabele een (nieuwe) waarde heeft gekregen. De wijze waarop de waarde berekend wordt is vastgelegd in de expressie. De waarde is verder afhankelijk van de waarden van de constanten en van de variabelen in de expressie (en daardoor van de begintoestand); we noemen dit de *invoerwaarden* voor de statement. De berekende waarde (voor de variabele in het linkerlid) noemen we de *uitvoerwaarde*.

Voorbeeld

```
k := m * n + 3 * p
```

De waarden van m, n, 3 en p zijn de invoerwaarden; de waarde van k is de uitvoerwaarde.

(einde voorbeeld)

Invoerwaarden worden door de assignment actie niet veranderd

$$\{m = m_0,\ n = n_0,\ p = p_0\}k := m * n + 3 * p\{k = m_0 * n_0 + 3 * p_0,\ m = m_0, n = n_0, p = p_0\}$$

tenzij een variabele uit de expressie ook in het linkerlid voorkomt.

$$\{a = a_0, b = b_0\}\ a := a * b\ \{a = a_0 * b_0, b = b_0\}$$

De assignment actie kent geen deelacties. Alle andere acties moeten wel worden opgesplitst in deelacties, als proces beschreven worden. Zo moet

$\{x = x_0,\ y = y_0\}$

"ken aan g toe als waarde: de grootste gemene deler van x en y"

$\{g = GGD(x_0, y_0),\ x = x_0,\ y = y_0\}$

uitgeschreven worden als rij

$$\{x = x_0, y = y_0\}S_1;\{T_1\}S_2;\{T_2\}\ldots\{T_{n-1}\}S_n\{g = GGD(x_0,y_0),\ x = x_0, y = y_0\}$$

De procesbeschrijving, $S_1;S_2;\ldots;S_n$, om van twee willekeurige invoerwaarden (en niet zoals in het voorbeeld alleen van x_0 en y_0) de grootste gemene deler te bepalen en deze toe te kennen aan een willekeurige uitvoervariabele wordt een *procedure* genoemd. Als we het effect van deze procedure willen realiseren voor bepaalde invoerwaarden en een bepaalde uitvoervariabele gebruiken we de bij deze procedure behorende *procedure statement* (de activering van de procedure). Bij het gebruik van de procedure statement zijn we dus alleen geïnteresseerd in het effect van de procedure, niet in de beschrijving.

De meest elementaire actie is de assignment, dit kunnen we opvatten als de activering van de "assignment procedure". Van alle procedures (uitgezonderd deze "assignment procedure") moeten we de beschrijving zelf maken en binnen ons algoritme opnemen. Deze beschrijving van de procedure wordt de *declaratie* van de procedure genoemd.

Procedure statement en procedure declaratie

In een procedure statement zijn opgenomen:

- de *naam van de procedure* om aan te geven welke procedure geactiveerd wordt (welke actie plaatsvindt);
- de *invoerwaarden* (de operanden waarop door de procedure wordt geopereerd);
- de *uitvoervariabelen* (de variabelen waarin de resultaten van de actie zijn terug te vinden).

De procedure statement wordt als volgt genoteerd:

de naam van de procedure met daarachter tussen haakjes (en onderling gescheiden door komma's) de uitvoervariabelen en de invoerwaarden, tezamen

de *actuele parameters* genoemd. Bijvoorbeeld:

ggd(g,x,y)

Syntaxis:

<procedure statement> ::= <procedure identifier>
[<actual parameter list>]

<procedure identifier> ::= <identifier>

<actual parameter list> ::= (<actual parameter> {,<actual parameter>})

Een procedure statement mag overal in het programma staan waar bijvoorbeeld ook een assignment statement mag staan.

Zoals is gezegd is de "assignment procedure" bekend en moeten alle andere procedures gedeclareerd worden. Na de declaratie kan de procedure geactiveerd worden door de bijbehorende statement.
Bij het ontwerpen en declareren van nieuwe procedures maken we gebruik van reeds (door declaratie) bekende procedures en van de besturingsstructuren.
Bij de procedure declaratie zijn van belang:

- Het *effect* van de procedure: wat bewerkstelligt de procedure?
  Om na te kunnen gaan of in een bepaalde situatie (toestand) een bepaalde procedure bruikbaar is, moet er een goede beschrijving van het effect van de procedure voorhanden zijn. Deze beschrijving legt de relatie vast tussen de invoerwaarden en de uitvoerwaarden. Ook voor de maker van de procedure is de beschrijving van belang: aan de hand hiervan wordt de procedurebeschrijving geconstrueerd.
- De *invoerwaarden* en *uitvoervariabelen*: waarop werkt de procedure? In de procedure (declaratie) worden deze grootheden gerepresenteerd door formele namen, de zogenaamde *formele parameters*. Bij uitvoering van de procedure als gevolg van de procedure statement worden de formele parameters ver-

vangen door de actuele parameters.

De parameters leggen de relatie van de procedure met de omgeving vast; het effect van de procedure wordt uitgedrukt in de formele parameters.

- De *beschrijving van het proces*: de procesbeschrijving die vastlegt op welke wijze uit de invoerwaarden de waarden van de uitvoervariabelen bepaald worden.

  Deze procesbeschrijving heeft eventueel een eigen toestandsruimte (er worden variabelen gedeclareerd) en in de acties spelen deze variabelen (*locale variabelen*) plus de formele parameters een rol.

  Men noemt deze eigenlijke procesbeschrijving wel de *body* van de procedure.

  Een procedure beschrijft een hele klasse van processen. Wordt de procedure geactiveerd met andere invoerwaarden, dan wordt (waarschijnlijk) een ander proces uitgevoerd.

In de procedure declaratie moeten worden vastgelegd:

- de naam van de procedure;
- de formele parameters en hun type;
- de body van de procedure.

Syntaxis:

```
<procedure declaration> ::= <procedure heading>; <procedure body>
<procedure heading> ::= procedure <procedure identifier>
                                  <formal parameter list>
<formal parameter list> ::= (<formal parameter section>
                            {;<formal parameter section>})
<formal parameter section> ::= <value parameter section>|
                               <variable parameter section>
```

<value parameter section> ::= <parameter group>

<variable parameter section> ::= var <parameter group>

<parameter group> ::= <identifier list> : <type identifier>

<procedure body> ::= <block>

<block> ::= <declaration part> <statement part>

Voorbeeld procedure statement en procedure declaratie

procedure declaratie:

```
procedure kwadsom (var s: integer; m,n: integer);
{preconditie: m ≤ n; postconditie: s = Σ_{i=m}^{n} i^2}
    var i: integer;
    begin s := 0;
          for i := m to n do s := s + i * i
    end
```

procedure statement:

```
kwadsom(p,1,100).
```

formele parameters voor invoer: m en n

voor uitvoer: s (door var aangegeven)

actuele parameters voor invoer: 1 en 100

voor uitvoer: p

locale variabele in procedure body: i

(einde voorbeeld)

Opmerking: De som van de kwadraten kan efficiënter bepaald worden.

Op het einde van hoofdstuk 4 (p.61) is de algemene opbouw van een Pascal programma beschreven. In het daar beschreven declaratiegedeelte moet als

laatste (mogelijke) component worden opgenomen:

<procedure and function declaration part> ::=

{(procedure declaration | function declaration);}

Op functies komen we later terug.

Een gedeclareerde procedure kan meermalen door procedure statements in het programma geactiveerd worden. Activering van een procedure door middel van de uitvoering van een procedure statement bestaat uit twee fasen:

(1) de formele invoerparameters (value parameters) worden als het ware in de procedure body gedeclareerd en krijgen als beginwaarde de waarde van de overeenkomende actuele parameter; de formele uitvoerparameters (variabele parameters) worden in de body vervangen door de overeenkomstige actuele parameters;

(2) de door de onder (1) genoemde activiteiten veranderde body wordt uitgevoerd.

Voorbeeld

procedure declaratie:

```
procedure ggd(var k: integer; a,b: integer);
  begin while a <> b do
            if a > b then a := a - b
                     else b := b - a;
        k := a
  end
```

procedure statement:

ggd(g,x,y).

Uitvoering van deze procedure statement leidt tot uitvoering van de procesbeschrijving:

```
var a,b: integer;
begin a := x; b := y;
      while a <> b do
         if a > b then a := a - b
                  else b := b - a;
      g := a
end
```

(einde voorbeeld)

Bij iedere procedure behoort een effectbeschrijving in de vorm van een preconditie en een postconditie. Stel dat de volgende procedure is gedeclareerd zonder verdere beschrijving:

```
procedure vkw(var x1,x2: real; a,b,c: real);
   var d: real;
   begin d := sqrt(b * b - 4 * a * c);
         x1 := (-b - d)/(2 * a);
         x2 := (-b + d)/(2 * a)
   end
```

De gebruiker van de procedure wordt er niet op attent gemaakt dat procedure statements als vkw(y,z,0,2,4) en vkw(x1,x2,4,2,4) aanleiding geven tot fouten. De preconditie is: $a \neq 0 \wedge b^2 - 4ac \geq 0$; de postconditie is $a(x - x1)(x - x2) = ax^2 + bx + c$. Deze moeten vermeld worden.
(Voor het berekenen van x1 en x2 zal in het algemeen een andere methode worden gebruikt.)

Opmerking

Een procedure heeft niet altijd parameters. De volgende procedure zet in de uitvoerrij de kwadraten van de gehele getallen die in de invoerrij staan.

```
procedure kopenkwad;
   var n: integer;
   begin while not eof do begin read(n); write(n * n) end end
```

(einde opmerking)

Voorbeeld van een programma met procedure.

Dit voorbeeld laat enkele aanroepen zien van een procedure voor machtsverheffen (in Pascal niet als operator aanwezig).

```
program machtsverheffen (input,output);
   var a,b,c,d,e: integer;
       p,q: real;
   procedure macht(var z: real; x: real; y: integer);
      {preconditie: ¬(x = 0 ∧ y = 0); postconditie: z = x^y}
      var i: integer;
      begin if x = 0
               then z := 0
               else begin if y < 0
                             then begin x := 1/x;
                                        y := -y
                                  end;
                    {y ≥ 0}
                    z := 1; i := 0; {z = x^i}
```

```
                while i<> y do
                   begin i := i + 1;
                         z := z * x {z = x^i}
                   end {z = x^i ∧ i = y}
             end
    end;
begin read(a,b,c,d,e);
      if (a<>0) or ((b-c) <> 0)
         then begin macht(p,b-c,a);  writeln(b-c,a,p) end
         else writeln(0,0);
      if (a+b) <> 0
         then begin macht(q,a+b,c-d*e);
                    writeln(a+b,c-d*e,q)
              end
end.
```

Als actuele parameters zijn expressies toegestaan; actuele uitvoerparameters moeten variabelen zijn.

(einde voorbeeld)

Voorbeeld van een programma met procedures

In de invoerrij staat een geheel getal (≥ 1) : n. H(n) is gedefinieerd als $H(n) = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$. Het programma plaatst alle partiële sommen van H(n) ($\sum_{k=1}^{i} \frac{1}{k}$ voor i = 1,2,...,n) in de uitvoerrij als twee gehele getallen (de teller en noemer van een breuk), afgezien van de som 1 die als één getal in de uitvoerrij wordt geplaatst.

```
program harmonisch (input,output);
    var n, i, teller, noemer, hulp: integer;
    procedure ggd(var k: integer; a,b: integer);
       {preconditie: a,b > 0; postconditie k = GGD(a,b)}
       begin while a<>b do
                if a > b then a := a-b
                         else b := b-a;
             k := a
       end;
    procedure   telbreukop(var q, r: integer; p: integer);
         {preconditie: p ≠ 0; postconditie: q/r = q0/r0 + 1/p}
         begin q := p * q + r; r := r * p end;
    begin   read(n);
            i := 1; teller := 1; noemer := 1;
            {teller/noemer = Σ(k=1..i) 1/k}
            writeln(1);
            while i <>n do
               begin i := i+1;
                     telbreukop(teller, noemer, i);
                     {teller/noemer=Σ(k=1..i) 1/k}
                     ggd(hulp, teller, noemer);
                     if hulp > 1
                        then begin teller := teller div hulp;
                                   noemer := noemer div hulp
                             end;
                     {teller/noemer = Σ(k=1..i) 1/k }
                     writeln(teller, noemer)
               end
    end
```

In de procedure telbreukop zijn teller en noemer zowel invoer- als uitvoerparamater (vergelijk met x in x := x + 1). Omdat de nieuwe waarden voor teller en noemer bekend moeten worden moeten het variabele parameters (var) zijn.
(einde voorbeeld).

We kunnen de oplossing van een probleem vaak splitsen in een rij oplossingen van deelproblemen, waarbij de oplossing van een deelprobleem een semantische eenheid vormt (een bepaald effect bereikt) en de oplossingen van de deelprobelemen onderling zo onafhankelijk mogelijk zijn. Juist zo'n opsplitsing van een probleem in deelproblemen geeft ons de mogelijkheid om een complex probleem aan te pakken, waarbij we ons niet op ieder moment in elk detail behoeven te verdiepen. Het procedure mechanisme geeft ons de mogelijkheid om een deelprobleem op te lossen los van de omgeving. We zorgen er daartoe voor dat de enige relaties die een procedure met zijn omgeving heeft, verlopen via de parameters. De invoerparameters ontlenen hun waarde aan de omgeving waarin de procedure geactiveerd wordt (zonder deze omgeving te veranderen). Het resultaat van de procedure wordt aan de omgeving overgedragen via de uitvoervariabelen (waardoor de toestand van de omgeving verandert). De procedure is dus onafhankelijk van de omgeving te construeren en is dan in elke omgeving inzetbaar.

We kunnen de oplossing van een deelprobleem natuurlijk ook direct als een stuk procesbeschrijving in ons programma opnemen. We moeten er dan wel voor zorgen dat het een semantische eenheid blijft en geen onbedoelde effecten op de omgeving realiseert.

Voorbeeld

Stel dat een tabel van de volgende vorm moet worden gegenereerd.

```
1
2     4
3     6     9
4     8    12    16
5    10    15    20    25
```

In de invoerrij is een gehele waarde gegeven die aangeeft uit hoeveel regels de tabel moet bestaan (< 1000). Het programmafragment dat het gevraagde realiseert zou kunnen luiden:

```
var i, aantal: integer;
begin read(aantal);
      i := 0; {i is het aantal gegenereerde rijen}
      while i <> aantal do
         begin i := i + 1;
               genereerrij(i);
               writeln
         end
end
```

Hierin is de genereerrij(i) een procedure statement; de bijbehorende procedure moet nog gedeclareerd worden. We kunnen het genereren van de i-de regel ook uitwerken op (in) de plaats van de procedure statement:

```
var i, aantal, j: integer;
begin read(aantal);
      i := 0;
      while i <> aantal do
```

```
            begin i := i + 1;
                  j := 0; {aantal uitgevoerde getallen}
                  while j <> i do
                     begin j := j + 1;
                           write(j * i : 6)
                     end;
                  writeln
            end
      end
```

We dienen ons nu goed te realiseren dat voor de oplossing van het deelprobleem i een invoerwaarde is, die dus niet mag veranderen van waarde (bij het procedure mechanisme zorgt de value-parameter hier automatisch voor).

9. Functies

Een procedure voor het berekenen van de faculteit bij een gegeven n (geheel, ≥ 0) kan, op grond van de definitie

$$0! = 1$$

$$n! = 1 * 2 * \ldots * (n-1) * n \qquad \text{voor } n \geq 1$$

luiden:

```
procedure fac1(var nfac: integer; n: integer);
   {preconditie: n ≥ 0; postconditie: nfac = n!}
   var i: integer;
   begin i := 0; nfac := 1; {nfac = i!}
         while i<> n do
            begin i := i + 1;
                  nfac := nfac * i {nfac = i!}
            end
         {nfac = i! ∧ i = n}
   end
```

Als nu de binomiaalcoëfficiënt $\binom{p}{q}$ uitgerekend moet worden kan dat als volgt($\binom{p}{q} = \frac{p!}{q!(p-q)!}$):

```
...; fac1(pf,p); fac1(qf,q); fac1(pminqf,p-q);
pbovenq := pf/(qf * pminqf);...
```

(het berekenen van $\binom{p}{q}$ kan efficiënter dan hier wordt gedaan.)

Deze situatie, waarin een procedure een uitvoerresultaat heeft dat direct daarna in een expressie gebruikt wordt, komt vaak voor. Voor deze soort procedures bestaat, wat de declaratie betreft, een andere vorm: de *functie*. De activering van een functie gebeurt ook anders: direct in een expressie (*function designator*). Het berekenen van de faculteit als functie:

```
function fac2(n: integer): integer;
   {preconditie : n ≥ 0; postconditie: fac2 = n!}
   var  i,h: integer;
   begin i := 0; h := 1; {h=i!}
         while i <> n do
            begin i := i + 1;
                  h := h * i {h = i!}
            end;
         {h = i! ∧ i = n}
         fac2 := h
   end
```

Een functie levert een waarde op die direct in een expressie gebruikt kan worden. In de declaratie van de functie moet het type van deze waarde worden opgegeven. In de body van de functie moet tenminste eenmaal een waardetoekenning plaatsvinden aan de naam van de functie.

Voor het probleempje van $\binom{p}{q}$ krijgen we nu:

```
...; pbovenq := fac2(p)/(fac2(q) * fac2(p - q));...
```

Bij de behandeling van de typen hebben we al enige (standaard) functies gezien.

Syntaxis functies:

```
<function declaration> ::= <function heading>;<function body>
<function heading> ::= function <function identifier>
                       [<formal parameter list>]
                       :<result type>
<result type> ::= <type identifier>
<function body> ::= <block>
```

De type identifier is de naam van een eerder gedefinieerd type. Verder is alles als bij procedures, afgezien dus van de waardetoekenning aan de naam van de functie in de body van de functie.

Syntaxis designator:

<function designator> ::= <function identifier>
[<actual parameter list>]

Voorbeeld

```
function max(x,y: real): real;
 begin if x > y then max := x else max := y end
```

Deze functie kan bijvoorbeeld geactiveerd worden door:

```
a := 25 * max(c,d)
```

of

```
t := max(p,max(q,r))
```

(einde voorbeeld)

Voorbeeld

Als tweede voorbeeld voor functies kijken we naar het volgende probleempje. In de invoerrij zijn twee getallen gegeven. De getallen zijn positief en geheel en het eerste getal is niet groter dan het tweede.
Gevraagd wordt alle perfecte getallen te bepalen tussen de twee gegeven getallen (inclusief deze getallen zelf). 1 is perfect en een getal >1 is perfect als het gelijk is aan de som van zijn delers (1 inclusief, het getal zelf, uiteraard, exclusief). Zo is 6 perfect, want 6 = 1 + 2 + 3.

```
program perfectegetallen (input,output);
    var i, n, m: integer;
    function perfect(j: integer): boolean;
      {preconditie: j ≥ 1; postconditie: perfect ; "j is perfect"}
        var i, som : integer;
       begin som := 1; i := 1;
             {som = "som van de delers van j die ten hoogste i zijn"}
             while i < j div 2 do
                begin i := i + 1;
                      if j mod i = 0 then som := som + i
                end;
             perfect := (som = j)
        end;
    begin read(n,m);
          writeln('lijst van alle perfecte getallen tussen',n,'en',m,':');
          for i := n to m do
              if perfect(i) then writeln(i)
    end.
```

Voorbeeld

De onderstaande functie zorgt voor machtsverheffen. De methode die in de body gebruikt wordt verschilt van die die in de procedure voor machtsverheffen in het vorige hoofdstuk is gebruikt.

```
function machts(x: real; y: integer): real;
  {preconditie: ¬(x = 0 ∧ y = 0); postconditie: machts = x^y}
    var z: real;
```

```
begin if x = 0
        then machts := 0
        else begin if y < 0
                then begin x := 1/x;
                           y := -y
                     end;
             {y ≥ 0}
             z := 1; {z = x^(y0-y)}
             while y > 0 do
               begin while not odd(y)
                       begin y := y div 2;
                             x := sqr(x)
                       end;
                     y := y - 1; z := x * z
               end; {z = x^(y0-y) ∧ y = 0}
             machts := z
        end
end
```

(einde voorbeeld)

In de body van de functie mogen meerdere toekenningen aan de naam van de functie voorkomen. Maar de naam van de functie is geen locale variabele, er kunnen dus geen waarden in bewaard worden. Zo kunnen we in de procedure machts niet schrijven:

machts := x * machts; machts in het linkerlid is wel toegestaan maar machts in het rechterlid niet. Deze laatste wordt nl. als aanroep van een functie opgevat (zoals odd en sqr) maar heeft dan een foutief aantal actuele parameters (namelijk geen in plaats van twee zoals de declaratie aangeeft).

10. Procedures en functies als parameters

In hoofdstuk 8 is gesproken over twee soorten parameters: value parameters (voor de invoerwaarden) en variable parameters (voor de uitvoervariabelen). Pascal staat ook toe procedures en functies als parameters op te nemen:

<formal parameter section> ::= <value parameter section>|
<variable parameter section>|
<procedure parameter section>|
<function parameter section>

<procedure parameter section> ::= procedure <identifier list>

<function parameter section> ::= function <parameter group>

Uit deze syntaxis blijkt dat van procedures en functies als parameter alleen een formele naam wordt opgegeven en dat niet wordt aangegeven hoeveel parameters deze formele procedures en functies hebben en wat het type is van deze parameters. In de body wordt hiervan wel gebruik gemaakt en het is de verantwoordelijkheid van de programmeur om actueel procedures en functies op te nemen die het goede aantal parameters hebben. De maker van de procedure (of functie) zal in de effectbeschrijving moeten aangeven wat de parameters van zijn procedure-parameters en functie-parameters moeten zijn. Deze parameters mogen alleen value parameters (dus geen variable param .) zijn. Van functies moet het type van het resultaat worden opgegeven. De heading van een procedure kan dus van de vorm

procedure voorb1(var a: integer; p: real; procedure z)

zijn en van een functie bijvoorbeeld

function p(procedure q; function r: integer; s: integer)

Voorbeeld van declaratie

```
function som(function f: integer; a,b: integer): integer;
  {preconditie: f is functie met 1 parameter (integer);
   postconditie: som = sum_{i=a}^{b} f(i)}
   var i,s: integer;
   begin s := 0;
         for i := a to b do s := s + f(i);
         som := s
   end
```

(einde voorbeeld)

De bovenstaande functie kan bijvoorbeeld geactiveerd worden door:

```
p := som(sqr, 1, 100)
```

waardoor p als waarde krijgt de som van de kwadraten van 1 tot en met 100.

Voorbeeld

```
function zero(function f: real; a,b,e: real): real;
   {preconditie: teken van f(a) ≠ teken van f(b); postconditie:
    zero = "nulpunt van f in interval [a,b] met nauwkeurigheid e"}
    var s: boolean;
        x,z: real;
   begin s := f(a) < 0;
         {nulpunt in [a,b]}
         while abs(a-b) >= e do
            begin x := (a+b)/2;
                  z := f(x);
                  if (z < 0) = s then a := x else b := x
                  {nulpunt in [a,b]}
```

```
            end;
        zero := x
    end
```

(einde voorbeeld)

Voorbeeld

```
procedure tabelleer(function f: real; a,b,c: real);
   {preconditie: f is functie met één parameter(real), c ≥ a ≥ 0, b > 0;
    postconditie: in uitvoerrij wordt getabelleerd f(a), f(a+b),
    f(a+2b),... met als laatste f(d), met d = a+kb en d ≤ c}
   var h: real;
       j, aantal: integer;
begin h := a; aantal := trunc((a-c)/b);
      j := 0;
      while j <= aantal do
         begin writeln(h: 12, f(h): 20);
               h := h+b; j := j+1
         end
end
```

Deze procedure zou bijvoorbeeld geactiveerd kunnen worden door:

```
tabelleer(sin,0,0.01,1)
```

(De standaard functie trunc(x) levert voor x ≥ 0 de grootste gehele waarde die ten hoogste gelijk is aan x, en voor x < 0 de kleinste gehele waarde die tenminste gelijk is aan x).

11. Recursie

De body van een procedure (functie) kan activeringen bevatten van bestaande procedures (functies). Een bijzonder geval van zo'n in de body geactiveerde procedure (functie) is de procedure (functie) zelf; we spreken in zo'n geval van een *recursief* gedefinieerde procedure (functie).
Als eerste voorbeeld nemen we een functie die als resultaat heeft: de som van (de decimale representatie van) de waarde van de invoerparameter, die een niet-negatieve gehele waarde heeft. Als we zo'n som van n even cs noemen dan kunnen we cs op de volgende manier definiëren:

cs(n) = cs(n div 10) + n mod 10 voor n ≥ 10

cs(n) = n voor n < 10

Om de cijfersom te bepalen kunnen we nu de volgende functie gebruiken die direct gebruik maakt van de bovenstaande definitie :

```
function cijfersom(arg: nonnegint): nonnegint;
   {preconditie: het type nonnegint (0..maxint) bestaat;
    postconditie: zie boven (cijfersom = cs(arg))}
    var kop, staart: nonnegint;
  begin if arg < 10
          then cijfersom := arg
          else begin kop := arg div 10;
                     staart := arg mod 10;
                     cijfersom := cijfersom(kop) + staart
               end
  end
```

In deze declaratie staat de statement

```
cijfersom := cijfersom(kop) + staart
```

in de body; cijfersom in het linkerlid staat daar op grond van de afspraak dat in de body van een functie een waarde wordt toegekend aan de functienaam; cijfersom in het rechterlid is de recursieve aanroep van de functie en is dan ook voorzien van een (actuele) parameter.
Het effect van de procedure cijfersom kan natuurlijk ook bereikt worden met een niet-recursieve body. Zo hebben we voor de faculteit al een niet-recursieve functie gezien. Op grond van de defintitie

$$0! = 1$$

$$n! = n * (n-1)! \quad \text{voor } n > 0$$

kunnen we ook een recursieve functie declareren:

```
function fac(n: nonnegint): nonnegint;
   {preconditie: het type nonnegint bestaat;
    postconditie: fac = n!}
  begin if n = 0
          then fac := 1
          else fac := n * fac(n - 1)
  end
```

Voor de correctheid van recursieve functies (procedures) moet gekeken worden of het geheel van aanroepen eindigt en als dit het geval is of het juiste effect wordt bereikt. In een recursieve functie (procedure) zullen de recursieve aanroepen steeds gebeuren onder een of andere conditie. In deze conditie zullen een of meer parameters van de functie voorkomen en bij iedere aanroep van de functie zullen deze parameters een zodanige waarde hebben dat het niet meer voldoen aan de conditie, waaronder de recursieve aanroep plaatsvindt, "een stapje dichterbij is gekomen". In het voorbeeld van de functie fac is de actuele parameter bij iedere volgende

aanroep 1 kleiner dan de actuele parameter bij de vorige aanroep, zodat de situatie dat deze actuele parameter de waarde 0 heeft (de conditie) bereikt zal worden.

Dat het juiste effect wordt bereikt kunnen we laten zien door gebruik te maken van inductie (zo wordt het algoritme ook gevonden als het niet direct gebaseerd is op een recursieve definitie).

Neem weer de functie voor de faculteit. De functie levert het correcte resultaat voor de actuele parameter 0. Uitgaande van het gegeven dat de functie het correcte resultaat oplevert voor een willekeurige actuele parameter k (geheel, ≥ 0) is het duidelijk dat de functie ook het correcte resultaat oplevert voor k + 1.

Door de aanroep van een recursieve functie (procedure) ontstaat een geneste structuur van aanroepen. Iedere aanroep creëert een locale toestandsruimte, die volledig onafhankelijk is van de vorige toestandsruimte van waaruit de aanroep plaatsvond, ook al hebben de locale variabelen dezelfde naam. Zo heeft de functie cijfersom twee locale variabelen. Door de aanroep van cijfersom in de body van de functie ontstaat een nieuwe toestandsruimte (en dan weer een nieuwe, enzovoort). Pas als de aanroep van cijfersom in de body volledig is verwerkt (en dus de cijfersom van de kop heeft opgeleverd), wordt teruggekeerd naar de eerste toestandsruimte en kan bij het verkregen resultaat staart worden opgeteld.

De tot nu toe gegeven voorbeelden waren recursieve functies. Ook recursieve procedures zijn mogelijk:

```
procedure keerom(n: nonnegint);
    {preconditie: het type nonnegint bestaat;
     postconditie: de cijfers van n zijn in omgekeerde volgorde in
     de uitvoerrij geplaatst}
```

```
begin write(n mod 10);
      if(n div 10) <> 0 then keerom (n div 10)
end
```

Recursie wordt vaak moeilijk gevonden omdat men zich het proces tracht voor te stellen dat plaatsvindt als gevolg van de aanroep van een recursieve functie of procedure. Als men echter recursieve functies en procedures bekijkt in termen van het effect, is het gebruik en de constructie ervan niet zo moeilijk en geven recursieve algoritmen voor sommige problemen juist duidelijke en begrijpelijke oplossingen. Dit geldt zeker als het probleem zelf "recursief gesteld" is, zoals bijvoorbeeld de wiskundige Ackermann functie die gedefinieerd is als

$$A(m,n) = \begin{cases} n + 1 & \text{voor } m = 0 \\ A(m-1,1) & \text{voor } n = 0 \\ A(m-1,A(m,n-1)) & \text{voor } m,n > 0 \text{ en geheel} \end{cases}$$

en die direct als functie te schrijven is:

```
function A(m,n: nonnegint): nonnegint;
   begin if m = 0 then A := n + 1
                  else if n = 0 then A := A(m - 1,1)
                                else A := A(m - 1,A(m,n - 1))
   end
```

Maar het geldt ook in andere gevallen. Als voorbeeld nemen we het "klassieke" probleem van de torens van Hanoi. Gegeven zijn drie "pinnen" (die we links, midden en rechts zullen noemen) waarover schijven geschoven kunnen worden. Er zijn n schijven, die allemaal een verschillende diameter hebben. In de beginsituatie zitten alle schijven op de linker pin en wel zodanig dat (afgezien van de onderste schijf) iedere schijf ligt op een schijf met een

grotere diameter. Gevraagd wordt de n schijven van de linker pin over te brengen naar de rechter pin waarbij de volgende regels in acht moeten worden gehouden:

- er mag slechts een schijf tegelijk verplaatst worden;
- steeds moet elke schijf liggen op een schijf met een grotere diameter (of het is de onderste schijf op een pin);
- steeds moet elke schijf op één van de drie pinnen liggen.

Het programma moet aangeven wat de verplaatsingen van de schijven zijn.

We kunnen het probleem van het verplaatsen van n(> 1) schijven uitdrukken in het verplaatsen van n - 1 schijven:

- verplaats de bovenste n - 1 schijven van de linker pin naar de middelste pin;
- verplaats de overgebleven schijf van de linker pin naar de rechter pin;
- verplaats de stapel van n - 1 schijven van de middelste pin naar de rechter pin.

Het patroon van deze oplossing voldoet aan de recursieve aanpak zoals we gezien hebben. We hebben ook een conditie, n = 1, waaronder de recursiviteit stopt. Bij deze aanpak wordt voldaan aan de gestelde regels.

Het programma luidt:

```
program Hanoi(input,output);
    type pin = (links,midden,rechts);
         pos = 1..maxint;
    var aantal: pos;
    procedure verplaatstoren(n: pos; een,twee,drie: pin);
        procedure verplaatsschijf(van,naar: pin);
            begin write('verplaats een schijf van');
```

```
                write(van); write('naar');
                writeln(naar)
            end;
        begin if n = 1
                then verplaatsschijf(een,drie)
                else begin verplaatstoren(n - 1,een,drie,twee);
                           verplaatsschijf(een,drie);
                           verplaatstoren(n - 1,twee,een,drie)
                     end
        end;
    begin read(aantal);
          writeln('voor',aantal: 3,'schijven','zijn de verplaatsingen:');
          writeln;
          verplaatstoren(aantal,links,midden,rechts)
    end.
```

Stel dat we het aantal verplaatsingen willen berekenen. We noemen het aantal verplaatsingen bij n schijven H(n). Er geldt:

$H(1) = 1$

$H(n) = 1 + 2H(n - 1)$

Hieruit kan worden vastgesteld: $H(n) = 2^n - 1$.

Bij vier schijven is het aantal verplaatsingen 15, bij 64 schijven $2^{64} - 1$ ($\approx 10^{19}$).

Er blijken veel verplaatsingen nodig te zijn; dit is geen eigenschap van het algoritme maar van het probleem.

Bij recursieve oplossingen van problemen blijkt er echter vaak wel veel meer gedaan te worden dan nodig is. Als voorbeeld nemen we de rij van Fibonacci

waarvoor we in hoofdstuk 3 een algoritme hebben afgeleid. De rij is gedefinieerd als:

$$f_0 = 0$$

$$f_1 = 1$$

$$f_i = f_{i-1} + f_{i-2} \quad \text{voor } i \geq 2$$

Het algoritme uit hoofdstuk 3 voert $n-1$ "slagen" van de repetitie uit als f_n bepaald moet worden.

De bovenstaande definitie kunnen we "direct" vertalen in een recursieve functie:

```
function fib(i: integer): integer;
  begin if i = 0 or i = 1
        then fib := i
        else fib := fib(i - 1) + fib(i - 2)
  end
```

Hoe vaak wordt de functie aangeroepen om f_n te bepalen? Noem dit aantal A_n. Er geldt:

$$A_0 = 1$$

$$A_1 = 1$$

$$A_n = 1 + A_{n-1} + A_{n-2}$$

Stel $B_n = 1 + A_n$, dan geldt er

$$B_0 = 2$$

$$B_1 = 2$$

$$B_n = B_{n-1} + B_{n-2}$$

Hieruit volgt: $B_n = 2f_{n+1}$, en dus: $A_n = 2f_{n+1} - 1$.

Om f_{11} te bepalen waren in het algoritme uit hoofdstuk 3 10 slagen van de repetitie nodig. Nu zijn er 287 aanroepen van de functie nodig. We moeten

bij recursie dus oppassen met efficiëntie.

Procedures en functies mogen andere procedures en functies (en zichzelf) activeren. Pascal eist dat een procedure (of functie) die in de body van een procedure of functie geactiveerd wordt van te voren in de tekst is gedeclareerd. De declaratie moet in de tekst aan het gebruik voorafgaan. Een uitzondering vormt de recursieve procedure en functie. Stel nu echter dat procedure A de procedure B activeert. (Een zelfde verhaal geldt voor functies), en dat procedure B procedure A activeert. We spreken in dit geval van indirecte recursie. Maar hoe lossen we dit nu op in Pascal gezien de regels voor declaratie en activering. Pascal eist dat in zo'n geval een van de procedures in een zogenaamde forward declaratie wordt opgenomen. Dit komt neer op een soort splitsing van de declaratie. Voor ons geval wordt het:

```
procedure B(x: integer; var y: integer); forward;
procedure A(i: integer; var j: integer);
    var a,b: integer;
  begin  ..........
         .....; B(a,b);.....
         .....
  end;
procedure B;{verder komt hier niets}
    var c,d: integer;
  begin ............
        ....; A(c,d);.....
  end
```

12. Stepwise refinement

Een probleem of programmeeropdracht wordt veelal verbaal beschreven in termen van het vakgebied waaruit het probleem afkomstig is. Als het probleem door middel van een programma tot een oplossing moet worden gebracht moet het probleem beschreven worden in eisen die aan het programma gesteld worden. Dit betekent dat de begin- en eindrelatie (de specificatie) geformuleerd moeten worden. Het probleem wordt daardoor min of meer onafhankelijk van het desbetreffende vakgebied beschreven. We zijn steeds uitgegaan van een dergelijke beschrijving.

Uitgaande van de begin- en eindrelatie zullen we proberen een algoritmische oplossing te vinden. Daarna moet deze oplossing nog uitgedrukt worden in een of andere programmeertaal, in ons geval is dat Pascal. Op het algoritmische niveau zullen we vaak het probleem oplossen in steeds grotere mate van detail; we onderscheiden niveau's van detail of omgekeerd niveau's van abstractie. Het onderscheiden van niveau's is een van de principes van *gestructureerd programmeren*. Begrippen die hierbij een rol spelen zijn *stepwise refinement, niveau's van abstractie* en *top-down programmeren*.

De specificatie van het probleem wordt opgesplitst in een rij deelspecificaties. Om te voldoen aan een deelspecificatie (of de specificatie zelf) wordt een abstract algoritme ontworpen. Het is in die zin abstract dat er gebruik wordt gemaakt van operaties en operanden die voor het gemak van de oplossing worden ingevoerd, maar die in de later te gebruiken programmeertaal niet bestaan. Deze operaties en operanden moeten dan op een lager niveau van abstractie worden uitgedrukt in primitievere bouwstenen. De operaties en operanden zouden op het hogere niveau zelfs in natuurlijke taal beschreven kunnen zijn.

De verbinding tussen twee niveau's zou gelegd kunnen worden met behulp van procedures en functies. Op het hogere niveau volstaan we met aanroepen van (tot op dat moment) fictieve procedures en functies. Het lagere niveau bestaat uit de declaraties van de procedures en functies. Juist door het gebruik van procedures en functies is binnen een programma de structuur van de oplossing met zijn deeloplossingen te herkennen. Een bijkomend voordeel van procedures is dat we eenvoudiger wijzigingen kunnen aanbrengen als we besluiten een abstracte operatie om welke reden dan ook (bijvoorbeeld efficiëntie) op een andere wijze te realiseren in de primitievere operaties.
Ook de correctheidsbeschouwingen zullen op verschillende niveau's gegeven worden. De correctheid van de oplossing op een bepaald niveau wordt "aangetoond" onder de aanname dat de realisatie van de eenheden op het lagere niveau correct geschiedt. Zo zal ook voor de correctheidsbeschouwing voor het uiteindelijke programma gelden dat deze is opgebouwd uit een hiërarchie van correctheidsbeschouwingen.

Het volgende voorbeeld is afkomstig uit het boek "Structured Programming" van O.J. Dahl, E.W. Dijkstra en C.A.R. Hoare.
In de uitvoerrij willen we 50 regels plaatsen die tezamen een figuur voorstellen als de uitvoerrij wordt weergegeven op bijvoorbeeld een regeldrukker. Een regel (die we uit 100 posities zullen laten bestaan) bestaat uit spaties en markeringen. De markeringen van de 50 regels tezamen vormen het figuur. We gaan er vanuit dat er een p en q zijn, zodanig dat char(p) een spatie is en char(q) het gewenste merkteken (een sterretje bijvoorbeeld).
De figuur wordt vastgelegd door gegeven functies fx en fy die beide een

geheel resultaat opleveren bij een geheel argument.

$\forall i: 0 \leq i < 1000$ ($0 \leq fx(i) < 100$ en $0 \leq fy(i) < 50$)

De figuur bestaat uit 1000 of minder punten; voor iedere i, $0 \leq i < 1000$, geeft y = fy(i) de regel en x = fx(i) de positie binnen die regel waar het merkteken moet komen te staan. Voor iedere i-waarde mogen de functies fy(i) en fx(i) slechts één keer berekend worden.

De eerste aanzet van het programma zou kunnen zijn:

```
type image = ....;
var im: image;
begin build(im);
      print(im)
end
```

We zullen image, build en print verder moeten uitwerken. Het uitwerken van image zal zijn invloed hebben op de andere twee. Het uitwerken van print kan echter nauwelijks als we niet weten hoe image eruit ziet. We beginnen daarom met build.

Bij het opbouwen van de figuur zullen we wel moeten uitgaan van een "leeg" figuur en dan duizend merktekens stuk voor stuk moeten toevoegen. Dit houdt in dat build(im) zal bestaan uit:

```
clear(im); setmarks(im)
```

waarin clear(im) de vijftig lege regels maakt en setmarks(im) de duizend merktekens zet.

We gaan nu eerst kijken hoe de duizend merktekens kunnen worden toegevoegd.

```
procedure setmarks(var f: image);
    var i: inti;
  begin i := 0; {het volgende toe te voegen punt}
```

```
        while i < 1000 do
          begin addmark(f,i);
                i := i + 1
          end
    end
```

Voor addmark zouden we kunnen nemen

```
    procedure addmark(var g: image; i: inti);
        var x: intx;
            y: inty;
      begin x := fx(i); y := fy(i); markpos(g,x,y)
```

waarin markpos de waarde van g (de figuur) zal veranderen doordat het punt (x,y) wordt toegevoegd.

Aan het definitie- en declaratiegedeelte zullen de declaraties van de procedures setmarks en addmark moeten worden toegevoegd en daarnaast de typen

```
    type inti = 0..1000;
         intx = 0..100;
         inty = 0..50
```

We kunnen nu echter niet verder zonder dat we iets meer over image gezegd hebben. We moeten dus een representatie bedenken voor de waardenverzameling, rekening houdend met de operaties (print o.a.). Dit suggereert voor image te nemen

```
    type image =array [0..49] of line
```

Hiermee wordt print(im):

```
    procedure print(f:image);
        var j: inty;
      begin j := 50; {de regels 49,48,...,j staan in de uitvoerrij}
          while j > 0 do
            begin j := j - 1;
```

```
                    lineprint(f[j])
                end
        end
```

en clear(im):

```
    procedure clear(var f: image);
        var j: inty;
      begin j := 50; {de regels 49,48,...,j zijn leeg}
          while j > 0 do
            begin j := j - 1;
                  lineclear(f[j])
            end
      end
```

en we vervangen in de pocedure addmark de aanroep markpos(g,x,y) door linemark(x,line[y]).

We hebben tot nu toe gevonden:

```
    type inti = 0..1000;
         intx = 0..100;
         inty = 0..50;
         image = array [0..49] of line;
    var im: image;
    procedure addmark(var g: image; i: inti);
         var x: intx;
             y: inty;
       begin x := fx(i); y := fy(i); linemark(x, g[y]) end;
    procedure setmarks(var f: image);
         var i: inti;
```

```
    begin i := 0;
          while i < 1000 do
             begin addmark(f,i); i := i + 1 end
    end;
procedure print(f: image);
      var j: inty;
    begin j := 50;
          while j > 0 do
             begin j := j - 1; lineprint(f[j]) end
    end;
procedure clear(var f: image);
      var j: inty;
    begin j := 50;
          while j > 0 do
             begin j := j - 1; lineclear(f[j]) end
    end;
begin clear(im);
      setmarks(im);
      print(im)
end
```

Hierbij hadden we de procedure addmark op kunnen (voor een strikte hiërarchie: moeten) nemen in de procedure setmarks.

Uitgewerkt moeten nog worden: line, linemark, lineprint en lineclear.

Van een line moeten we vastleggen welke punten x ($0 \leq x < 100$) een merkteken moeten krijgen. Dit zou kunnen met behulp van een boolean array van 100 elementen, waarbij de waarde van het element aangeeft of er op die

positie een marker moet komen of niet. We zouden ook een karakter-array kunnen nemen met 100 elementen die elk de waarde "spatie" of "merkteken" hebben. Deze representatie is algemener: we zouden ook andere tekens kunnen afdrukken. We kiezen dan ook voor de laatste representatie. We krijgen dan voor line:

```
type line = array [0..99] of char
```

en voor de operaties:

- lineprint:

```
procedure lineprint(ℓ: line);
    var k: intx;
  begin k := 0; {volgende positie die afgedrukt wordt}
      while k < 100 do
        begin write(ℓ[k]); k := k + 1 end;
      writeln
  end
```

- lineclear:

```
procedure lineclear(var ℓ: line);
    var k: intx;
  begin k := 0; {volgende positie met spatie}
      while k < 100 do
        begin ℓ[k] := char(p); k := k + 1 end
  end
```

- linemark:

```
g[y][x] := char(q)
```

De procedure lineprint nemen we op in de procedure print, de procedure lineclear in de procedure clear en g[y][x] := char(q) vervangt linemark (x,g[y]) in de procedure addmark. Daarmee is het totale algoritme klaar.

We kijken nu nog naar de efficiëntie.

Nadat op een regel het laatste merkteken is gezet in de procedure lineprint zou er direct, via writeln, overgegaan kunnen worden op een nieuwe regel (de spaties na het laatste merkteken zijn niet van belang). We zouden per regel kunnen bijhouden tot en met welke positie er write-opdrachten uitgevoerd moeten worden. Daartoe veranderen we de representatie van line:

```
type line = record pos: array [0..99] of char;
                       vulling: 0..100
            end
```

waarin vulling de eerstvolgende vrije positie aangeeft op de regel.

We moeten nu kijken welke konsekwenties dit heeft voor lineprint, lineclear en linemark (verder kunnen er geen konsekwenties zijn!).

We beginnen met lineclear. Het feit dat een regel leeg is kunnen we vastleggen door de betreffende vulling op 0 te zetten. Dit betekent dat we voor lineclear(f[j]) in de procedure clear kunnen schrijven f[j].vulling := 0; de procedure lineclear vervalt daarmee. We moeten ons echter realiseren dat lineclear de hele regel vulde met spaties zodat, nadat de merktekens waren gezet, de totale regel een waarde had. Dit is nu niet meer het geval. We zullen er voor moeten zorgen dat, als in de procedure addmark een merkteken wordt gezet op positie k, alle posities i met vulling ≤ i < k van een spatie worden voorzien. Hiermee wordt de linemark:

```
procedure linemark(x: intx; var ℓ: line);
    begin while ℓ.vulling <= x do
            begin ℓ.pos[ℓ.vulling] := char(p);
                  ℓ.vulling := ℓ.vulling + 1
            end; {ℓ.vulling > x}
          ℓ.pos[x] := char(q)
    end
```

En, waar in eerste instantie alles om begonnen was, lineprint:

```
procedure lineprint(ℓ: line);
    var k: intx;
  begin k := 0;
      while k < ℓ.vulling do
        begin write(ℓ.pos[k]);
              k := k + 1
        end;
      writeln
  end
```

Het volledige algoritme luidt:

```
type inti = 0..1000;
     intx = 0..100;
     inty = 0..50;
     line = record pos:array [0..99] of char;
                   vulling: 0..100
            end;
   image = array [0..49] of line;
var im: image;
procedure setmarks(var f: image);
    var i: inti;
    procedure addmark(var g: image; i: inti);
        var x: intx;
            y: inty;
        procedure linemark(x: intx; var ℓ:line);
          begin while ℓ.vulling <= x do
                  begin ℓ.pos[ℓ.vulling] := char(p);
                        ℓ.vulling := ℓ.vulling + 1
                  end;
```

```
                    ℓ.pos[x] := char(q)
                end;
            begin x := fx(i); y := fy(i);
                linemark(x,g[y])
            end;
    begin i := 0;
        while i < 1000 do
            begin addmark(f,i); i := i + 1 end
    end;
procedure print(f: image);
    var j: inty;
    procedure lineprint(ℓ: line);
        var k: intx;
        begin k := 0;
            while k < ℓ.vulling do
                begin write(ℓ.pos[k]);
                    k := k + 1
                end;
            writeln
        end;
    begin j := 50;
        while j > 0 do
            begin j := j - 1;
                lineprint(f[j])
            end
    end;
```

```
procedure clear(var f: image);
      var j: inty;
    begin j := 50;
          while j > 0 do
             begin j := j - 1;
                   f[j].vulling := 0
             end
    end;
begin clear(im);
      setmarks(im);
      print(im)
end
```