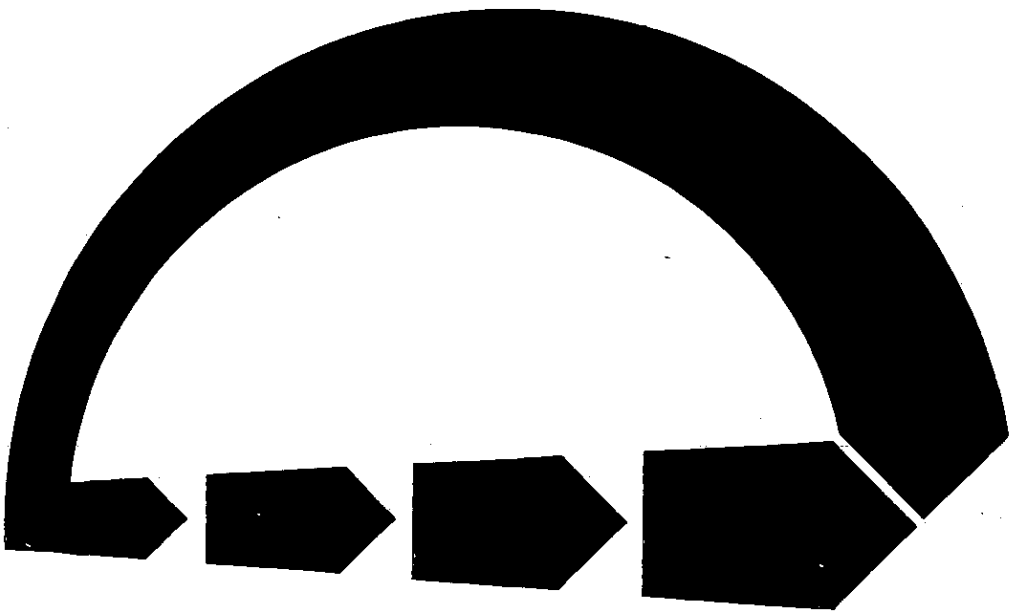
Bibl/Mag

Technische Hogeschool
Eindhoven

Dictaatnummer 2.269
Prijs f. 5,50

Onderafdeling der Wiskunde en Informatica

Inleiding tot de Informatica



TECHNISCHE HOGESCHOOL EINDHOVEN
Afdeling Algemene Wetenschappen
Onderafdeling der Wiskunde

INLEIDING tot de INFORMATICA

Auteur Onbekend

Najaarssemester 1980

TECHNISCHE HOGESCHOOL EINDHOVEN

Onderafdeling der Wiskunde

Inleiding tot de Informatica

Najaarssemester 1980

Inhoudsopgave

1. Inleiding.

Programmeren is het opstellen van een rij instructies; de uitvoering van de rij instructies moet leiden tot een gewenst effect.

Rijen instructies, zoals hierboven bedoeld, zijn ons uit het dagelijks leven overbekend. Denk bijvoorbeeld maar aan breipatronen, recepten en montagevoorschriften. Wij willen ons beperken tot instructies, die (in principe) uitgevoerd kunnen worden door een *rekenmachine*. Een rij van dergelijke instructies wordt een *algoritme* genoemd. Een algoritme wordt ook wel een procesbeschrijving genoemd, omdat het als het ware een beschrijving is van het proces, dat - door uitvoering van de instructies - plaatsvindt.

Bij een algoritme spelen twee partijen een rol. Aan de ene kant is dat de opsteller van het algoritme en aan de andere kant is dat de uitvoerder van het algoritme. Tussen opsteller en uitvoerder moet overeenstemming bestaan ten aanzien van de notatie die gebruikt wordt en ten aanzien van het effect van de uitvoering van de opdracht. De uitvoerder moet de instructies kunnen lezen (en daarna uitvoeren). De opsteller moet er zeker van zijn dat uitvoering van de opdracht leidt tot het gewenste effect. Een breipatroon bijvoorbeeld zal niet door iedereen gelezen kunnen worden. Maar zelfs als je de taal der breipatronen verstaat, is nog niet gegarandeerd dat je het betreffende werkstuk ook kunt maken. Om dit te kunnen moet je kunnen breien.

De rekenmachine heeft een bepaald opdrachtenrepertoire: de verzameling van alle opdrachten die de rekenmachine kan uitvoeren. Bovendien schrijft de rekenmachine voor hoe wij de opdrachten moeten noteren. Deze notatiewijze wordt de *programmeertaal* genoemd (hetgeen een wat wijdse betiteling is voor een notatiewijze, maar hierboven was ook sprake van "de taal der breipatronen").

Als een algoritme geschreven is in een programmeertaal, wordt het een *programma* genoemd. Een programma moet dan nog op een speciale wijze (bijvoorbeeld door middel van gaatjes in ponskaarten) worden vastgelegd, omdat de rekenmachine geen geschreven tekst kan interpreteren.

Het gaat ons hier vooral om het ontwerpen van algoritmen: Hoe kom je tot een algoritme dat een gewenst effect realiseert? We zullen ons daarom wat losmaken van machine-eigenschappen en -eigenaardigheden en ons beperken tot de essentie. Ook zullen we voor de algoritmen een taal hanteren die niet een echte programmeertaal is (het wel zou kunnen worden als we de rekenmachine zouden aanpassen).

Voor het beschrijven van een complex geheel staan ons twee beschrijvingsvormen ter beschikking die we de *procesbeschrijving* en de *toestandsbeschrijving* kunnen noemen. Stel bijvoorbeeld dat we een cirkel moeten beschrijven. Dit kan met de procesbeschrijving: Om een cirkel te construeren draait men een, met één been vast staande, passer zolang rond tot het andere been het uitgangspunt weer bereikt. Of met de toestandsbeschrijving: Een cirkel is de verzameling van alle punten die een gelijke afstand hebben tot een gegeven punt. De eerste beschrijving geeft aan hoe we een cirkel kunnen construeren, de tweede geeft aan wat we willen realiseren.
In veel beschrijvingen worden beide vormen door elkaar gebruikt. Zo vind je in een bestek van een architect vaak zowel wat er moet gebeuren als hoe bepaalde constructies gerealiseerd moeten worden. Vaak zien we dat in een procesbeschrijving als commentaar toestandsbeschrijvingen worden opgenomen. Bij een breipatroon staat bijvoorbeeld dat na een bepaald aantal toeren er 10 cm van het achterpand klaar moet zijn. Wij zullen, omdat de rekenmachine dat vraagt, procesbeschrijvingen moeten geven. We zullen echter ook van toestandsbeschrijvingen gebruik maken als hulpmiddel bij het opstellen van de procesbeschrijving.

In beschrijvingen zoals recepten, breipatronen en montagevoorschriften zie je vaak dat men een bepaald onderdeel van het proces apart beschrijft, deze beschrijving een naam geeft en in de totale beschrijving dit deel opneemt door de naam te noemen. Neem bijvoorbeeld dat in het werkstuk dat gebreid moet worden een bepaald patroon of figuur voorkomt. Men geeft dan een aparte beschrijving van dit patroon.

Bij beschrijvingen als recepten en breipatronen laat de exactheid vaak veel te wensen over. De ervaring van de uitvoerder van de procesbeschrijving staat er borg voor dat het werkstuk toch tot een goed einde wordt gebracht. Er wordt echter wel een beroep gedaan op de welwillendheid van de uitvoerder. Deze menselijke eigenschap ontbeert de machine ten enenmale. Een algoritme moet dan ook zeer precies zijn, zowel wat betreft de opdrachten om een bepaald effect te bereiken als wat betreft de notatie.

2. Introductie van begrippen.

Programmeren is een onderdeel van het vakgebied *Informatica*. In de informatica houdt men zich bezig met de verwerking van informatie (information processing). Juister gezegd houdt de informatica zich bezig met *gegevensverwerking* (data processing). Informatie is de betekenis die aan de data wordt toegekend.

Voor de dataverwerking kan gebruik worden gemaakt van een *rekenautomaat*. Deze automaat is in staat operaties op data uit te voeren aan de hand van een gegeven rij opdrachten. Deze rij opdrachten wordt voor de uitvoering van de opdrachten in de automaat opgeslagen, ze zijn te beschouwen als speciale data, waar de automaat de betekenis van kent. Dit maakt de rekenautomaat bijzonder ten opzichte van andere automaten (vergelijk met sigarettenautomaat, roltrap die steeds een slag "afrolt" en dergelijke).

Een basisbegrip bij de uitvoering van operaties op data is het begrip *actie*, ook wel handeling genoemd. *Onder een actie verstaan we een gebeurtenis, (met een eindige tijdsduur), die een wel gedefinieerd effect realiseert.* Teneinde het effect van een actie te kunnen omschrijven, gaan we er van uit dat een actie zich afspeelt in een bepaalde *omgeving*. Deze omgeving verkeert in een bepaalde *toestand*. Het effect van de actie kunnen we dan waarnemen aan het verschil tussen de toestand vóór en de toestand ná de actie. De actie is de oorzaak van deze *toestandstransformatie*.

De toestand voor de uitvoering van het montagevoorschrift kan omschreven worden als: Alle onderdelen zitten aan het frame. Na de uitvoering beschikken we over het modelvliegtuig. De toestand vóór de actie "het bepalen van de wortels van de vierkantsvergelijking $x^2 - 3x + 2 = 0$" wordt omschreven door:

- de coëfficiënten van de vierkantsvergelijking hebben respectievelijk de waarden 1, -3 en 2;
- de wortels zijn onbepaald.

De toestand ná de actie wordt omschreven door:

- de coëfficiënten van de vierkantsvergelijking hebben respectievelijk de waarden 1, -3 en 2;
- de waarden van de wortels zijn 1 en 2.

De omgeving, waarin een actie plaatsvindt, wordt gekenmerkt door een aantal, voor de actie *relevante grootheden*. Voor het montagevoorschrift zijn dit bijvoorbeeld de onderdelen van het modelvliegtuig. Voor de vierkantsvergelijking zijn dat de coëfficiënten en de wortels. We beperken ons verder tot kwantificeerbare grootheden, grootheden die waarden kunnen bezitten. Een toestand op een bepaald moment wordt dan bepaald door de op dat moment geldende waarden van de relevante grootheden.

Met een grootheid is verbonden een voor deze grootheid karakteristieke *waardenverzameling* met daarop gedefinieerde *operaties*. Zo'n waardenverzameling met zijn operaties noemen we een *type*.

We beperken ons voorlopig tot de volgende typen:

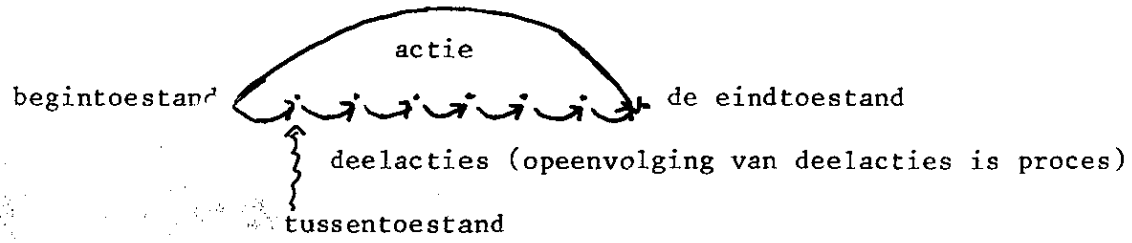
- integer, met als waardenverzameling de verzameling van de gehele getallen;
- real , met als waardenverzameling de verzameling van de reële getallen;
- boolean, met als waardenverzameling de verzameling van de logische waarden {true, false}.

Op de bijbehorende operaties komen we later terug (bij de eerste twee typen hebben we natuurlijk wel enig idee).

We noemen de grootheden in onze algoritmen voortaan *variabelen*. De functie van een variabele kunnen we omschrijven als het onthouden van een (deel van een) toestand; een variabele is te beschouwen als een *geheugenelement*, zoals in een roltrap een geheugenelement aanwezig zal zijn dat op ieder moment aangeeft over hoeveel treden de roltrap zich nog moet afrollen. In tegenstelling tot de roltrap komen in onze algoritmen meestal meerdere variabelen voor. Om deze van elkaar te onderscheiden geven we ze een *naam*. Een variabele staat op ieder moment voor een *waarde* uit de waardenverzameling van het type van de variabele (we zeggen slordig: de variabele heeft een waarde). Zo zal het geheugenelement van de roltrap op ieder moment een waarde hebben tussen bijvoorbeeld 0 en 40 (als er 36 treden van beneden naar boven zijn); als er ie-

mand door het licht loopt wordt de waarde 40 en bij iedere afwikkeling van een trede wordt de waarde 1 kleiner. Iedere variabele in een algoritme wordt geïntroduceerd met een bepaalde bedoeling, iedere variabele heeft een *betekenis*, zoals we ook gezien hebben voor het geheugenelement van de roltrap.
We kunnen de begrippen toestand, toestandstransformatie en actie nu nader preciseren:
Een toestand wordt gekarakteriseerd door een rij waarden van variabelen. Een toestandstransformatie is de verandering van de waarde(n) van één of meer variabelen. Een actie is de oorzaak van de verandering van de waarde(n) van deze variabele(n).

Vaak kan men een actie opvatten als een rij *deelacties*; uitgaande van een begintoestand leiden de achtereenvolgende toestandstransformaties van de deelacties (via de tussentoestanden) dan tot een eindtoestand, welke gelijk is aan de toestand die bereikt wordt als we de actie als één geheel beschouwen.



Als we de actie beschouwen als een rij deelacties, dan noemen we de actie een *proces*. We zijn hier alleen geïnteresseerd in processen waarvan de deelacties strikt na elkaar plaatsvinden. Men spreekt dan wel van *sequentiële* processen.
Of we de actie als één geheel beschouwen of als een proces (= rij deelacties) hangt in het algemeen af van onze interesse. In het eerste geval zijn we slechts geïnteresseerd in de *begintoestand* en de *eindtoestand*. In het tweede geval zijn we tevens geïnteresseerd in de *tussentoestanden*, we beschouwen de actie ook inwendig. Elke actie is op zijn beurt te beschouwen als deelactie van een omvattend proces. Sommige acties willen of kunnen we niet meer opsplitsen in deelacties, we spreken dan van *elementaire acties*. Welke acties elementair zijn wordt bepaald door het laagste niveau dat wij in onze beschouwingen willen betrekken of wordt voorgeschreven door de uitvoerder van de acties.

De toekomstige eigenaar ziet het bouwen van zijn huis wellicht als één actie omdat hij alleen geïnteresseerd is in de transformatie van begin- naar eindtoestand. De bouwer van het huis zal geïnteresseerd zijn in de deelacties van deze actie. De rij deelacties zou, in een eerste stap van verfijning, kunnen luiden:

- het leggen van de fundering
- het optrekken van de muren
- het plaatsen van het dak
- het afwerken van het geheel.

Ieder van de acties kan weer verfijnd worden. Een elementaire actie, die in meerdere acties als deelactie kan optreden, zou het metselen van een steen kunnen zijn. (Als deze actie niet elementair is zouden we hem één keer kunnen uitdrukken in elementaire deelacties en de actie zelf in de verfijningen kunnen opnemen.) Het bouwen van een huis kan zelf weer deelactie zijn van de omvattende actie van het bouwen van een stad.
De actie "bepaal de wortels van de vierkantsvergelijking $x^2 - 3x + 2 = 0$" blijkt bij nader inzien bijvoorbeeld te zijn opgebouwd uit de rij acties:

- bepaal de waarde van de discriminant: $(-3)^2 - 4 * 1 * 2 = 1$
- bepaal de waarde van de eerste wortel: $(-(-3) + \sqrt{1})/(2 * 1) = 2$
- bepaal de waarde van de tweede wortel: $(-(-3) - \sqrt{1})/(2 * 1)$

We zullen later zien of dit elementaire acties zijn.

De overgang van proces naar actie geeft ons de mogelijkheid tot *abstractie*. We zijn bij de actie alleen geïnteresseerd in het uiteindelijke effect, in *wat* er gebeurt. Beschouwen we dezelfde gebeurtenis als proces, dan zijn we er ook in geïnteresseerd *hoe* de gebeurtenis zich afspeelt, via welke tussentoestanden de eindtoestand vanuit de begintoestand wordt bereikt.
Het verloop van het proces is geheel vastgelegd door de opeenvolging van de waarden van de variabelen.
Als we nu een algoritme moeten maken voor de oplossing van een probleem, kunnen we er in eerste instantie van uitgaan dat de uitvoerder bepaalde machtige acties kan uitvoeren. Later zullen we dan de opdrachten voor

deze acties herhaald opsplitsen in opdrachten voor deelacties, zodat we tenslotte uitkomen bij opdrachten die behoren tot het repertoire van de uitvoerder. In het voorbeeld van het modelvliegtuig wordt in eerste instantie volstaan met de opdracht "stel romp samen". Omdat deze opdracht (door de meeste mensen) niet zonder meer uitvoerbaar is, moet deze opdracht gesplitst worden in een rij opdrachten voor deelacties.
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 opdracht, noemen we een *statement*. De statische beschrijving van een proces, de rij van opdrachten, noemen we een *procesbeschrijving* (algoritme, programma).
Eerder is gezegd dat het verloop van een proces geheel is vastgelegd door de opeenvolging van de waarden van de variabelen. We kunnen dit nu aanvullen met de opmerking dat bij iedere procesbeschrijving een rij toestandsbeschrijvingen (zie hoofdstuk 1) hoort, namelijk de beschrijvingen van de begintoestand, de tussentoestanden en de eindtoestand.

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

< variabele >:=< waarde >

De waarde in het rechterlid van de statement kan zijn de waarde van een constante, de waarde van een variabele of de waarde van een expressie (hier komen we later op terug). Als in een begintoestand de variabele m onbepaald is of reeds een waarde heeft, dan zal na uitvoering van de statement

m := 4

de eindtoestand gekarakteriseerd worden door het feit dat de variabele m de waarde 4 heeft. Is in de begintoestand m onbepaald en heeft n de waarde 5, dan zal in de eindtoestand, na uitvoering van de rij statements

m := 4; m := m + n

n nog steeds de waarde 5 hebben en heeft m de waarde 9.

We zouden nu procesbeschrijvingen kunnen geven die elk bestaan uit een rij assignment statements. Het effect dat door uitvoering van zo'n rij assignment statements wordt bereikt is echter niet groot. We kunnen dan ook wellicht beter deze acties (op papier) uitvoeren dan dat we de procesbeschrijving maken en die laten uitvoeren door de processor. We zoeken naar mogelijkheden om een groot effect te bereiken terwijl de procesbeschrijving kort is. We willen daarom de mogelijkheid hebben om een statement al dan niet uit te laten voeren en de mogelijkheid om een statement herhaald uit te laten voeren. Juist deze laatste mogelijkheid benut bij uitstek de eigenschap van de rekenmachine: de snelheid.
De (notatie-) hulpmiddelen die ter beschikking staan worden wel *besturingsstructuren* (sequentiëringsprimitiva) genoemd. Met deze structuren wordt dus de volgorde beschreven waarin de samenstellende acties van een proces moeten plaatsvinden. We onderscheiden drie methoden van sequentiëring; hieronder volgen de betekenis en de notatie hiervan.

1. *concatenatie*

Aaneenrijging van een aantal acties tot een geordende rij van acties.

notatie: S1; S2

(hierin zijn S1 en S2 statements).

In feite hebben we deze methode al eerder gezien.

m := 4; n := m + 5

Het effect van de rij wordt bepaald door de statements en door de begintoestand. Hierboven is gezegd dat een rij assignment statements niet zinvol is, omdat we beter zelf het proces kunnen uitvoeren. We moeten dit gedeeltelijk herroepen omdat iedere procesbeschrijving, zoals hier ook blijkt, in feite de beschrijving is van een hele klasse van processen. Een element uit deze klasse (één proces) wordt bepaald door de begintoestand (in het voorbeeld is de waarde van n bepalend).

2. *selectie/conditie*

Deze methode bestaat in twee varianten:

- keuze uit twee alternatieven:

    notatie: if B then S1 else S2 fi

Hierin is B een voorwaarde (logische expressie, hierop komen we later terug). Als aan de voorwaarde wordt voldaan wordt statement S1 uitgevoerd, zoniet dan statement S2.

Stel bijvoorbeeld dat de variabelen a en b een waarde hebben en dat aan de variabele max de grootste van deze twee waarden moet worden toegekend. Dit wordt bereikt door uitvoering van

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

- voorwaardelijke actie:

        notatie: if B then S fi

Als aan de voorwaarde B wordt voldaan, dan wordt de statement S uitgevoerd; zoniet dan vindt er geen actie plaats.

Stel dat de variabele x een waarde heeft en dat gevraagd wordt aan x een nieuwe waarde toe te kennen, die gelijk is aan de absolute waarde van de huidige waarde. Dit kan door uitvoering van

        if x < 0 then x := -x fi

3. *repetitie*

Het onder een bepaalde voorwaarde herhaald uitvoeren van een statement;

    notatie: while B do S od

Als aan de voorwaarde B wordt voldaan, dan wordt S uitgevoerd, als (dan weer) aan de voorwaarde B wordt voldaan, dan wordt S (nogmaals) uitgevoerd, ..., als aan de voorwaarde niet wordt voldaan, wordt het geheel beëindigd.

Als reeds de eerste keer niet aan de voorwaarde wordt voldaan, vindt er in het geheel geen actie plaats. Een belangrijk punt bij deze constructie is de beëindiging van de herhaling. In de statement S moet er voor gezorgd worden dat op een gegeven moment niet meer aan de voorwaarde wordt voldaan. De uitvoering van

    while x > 0 do y := y + 1 od

heeft geen enkele actie tot gevolg (als $x \leq 0$) of het geheel eindigt niet.

Stel dat de variabele n een waarde heeft (geheel en > 0). Gevraagd wordt de kleinste gehele waarde van k te bepalen waarvoor geldt $2^k \geq n$.
Dit is te realiseren met:

    k := 0;
    while 2 ↑ k < n do k := k + 1 od

De herhaling eindigt omdat er bij iedere n een k te vinden is met $2^k \geq n$. Als de herhaling eindigt geldt dus $2^k \geq n$. Het is de kleinste k omdat er gestart wordt met k = 0 en k steeds met 1 opgehoogd wordt.

Stel dat de variabelen x en y een waarde hebben (geheel en > 0). Gevraagd wordt aan de variabele g als waarde toe te kennen de grootste gemene deler van x en y (g = GGD(x, y)). Of a een deler is van b kan getest worden met behulp van b mod a; de operator mod bepaalt de rest bij deling. Het gevraagde wordt gerealiseerd door

    if x < y then g := x else g := y fi;
    while ¬(x mod g = 0 ∧ y mod g = 0)
      do g := g - 1 od

In het stukje programma heeft g de betekenis: het getal dat als eerstvolgende als gemeenschappelijke deler geprobeerd zal worden. Als startwaarde wordt gekozen g = min(x, y). De repetitie eindigt omdat een ondergrens voor g bekend is, namelijk 1. Als de repetitie eindigt geldt

    x mod g = 0 ∧ y mod g = 0

dus g is een gemeenschappelijke deler. Het is de grootste gemene deler omdat voor de startwaarde van g, g_o = min (x, y), geldt : $g_o \geq$ elke gemene deler van x en y en g wordt steeds met 1 verlaagd. (Voor de betekenis van ¬ en ∧ zie blz. 35.)

In bovenstaande beschrijving van de programmastructuren staan S, S1 en S2 voor statements. In feite mogen hier procesbeschrijvingen staan. We

kunnen ieder van de S, S1 en S2 weer vervangen door elk van de bovenstaande constructies, bijvoorbeeld:

```
T1; T2; T3;
if B1 then if B2 then T4 fi else T5 fi;
T6; if B3 then if B4 then T7 else T8 fi fi;
T9;
while B5
  do T10;
     if B6 then T11 fi
  od
```

Vaak worden de constructies 2 en 3 ook statements genoemd. Volgens de betekenis die wij aan het woord statement hebben gegeven, is dit niet correct.

In een procesbeschrijving komen statements en besturingsstructuren voor. Van de statements is alleen de assignment statement behandeld.
We introduceren nu:

```
read(f,x)
write(g,x)
```

De eerste statement kent aan de variabele x de eerstvolgende waarde toe uit de rij f van invoerwaarden. (De eerste read "leest" de eerste waarde uit de invoerrij, de tweede read "leest" de tweede waarde uit de invoerrij, enzovoorts.)
De tweede statement plaatst de waarde van x als volgende element in de uitvoerrij g.
De invoerrij en de uitvoerrij (ook wel files genoemd) zorgen voor de communicatie tussen het programma en de "buitenwereld". Bij een programma behoren een of meer invoerrijen en een of meer uitvoerrijen, die door namen worden geïndentificeerd. Deze rijen worden "gekoppeld" aan zogenaamde randapparaten. Zo kan een invoerrij een waarde krijgen via een kaartlezer en kunnen de waarden uit de uitvoerrij zichtbaar gemaakt worden via een regeldrukker.

Het uitvoeren van statements of procesbeschrijvingen, met andere woorden het realiseren van de door de statements (procesbeschrijvingen) bedoelde acties, wordt gerealiseerd door een *processor*. Een processor kan aan de hand van een statische beschrijving een dynamische gebeurtenis laten plaatsvinden: een actie of een procesbeschrijving. Deze actie of dit proces is volledig bepaald door de statische beschrijving en de begintoestand zoals die heerst op het moment dat de beschrijving zal worden uitgevoerd. Een procesbeschrijving beschrijft dus een klasse van processen, waarvan er een zal plaatsvinden; welke dat is, wordt bepaald door de begintoestand.

Een algoritme bestaat uit een rij opdrachten en besturingsstructuren.
In het voorgaande is de betekenis (*semantiek*) van de acties, behorende bij de opdrachten en de besturingsstructuren, beschreven door aan te geven wat de processor moet doen om de acties te laten plaatsvinden (mechanistische beschrijving).
We gaan nu de semantiek van de acties wat onafhankelijker van de processor beschrijven. Het effect van een actie kunnen we waarnemen aan het verschil tussen de toestand voor en de toestand na de actie. De toestand leggen we vast in een beschrijving. In deze beschrijving zijn niet de individuele waarden van de variabelen belangrijk, maar de onderlinge relaties tussen de variabelen. In deze beschrijvingen spelen de betekenissen van de variabelen en hun onderlinge relatie dus een grote rol.
Met deze beschrijvingen:

- kunnen we de semantiek formeel vastleggen;
- is het mogelijk om hints te krijgen uit de toestandsbeschrijvingen (van begin- en eindtoestand) voor de toe te passen constructie;
- kunnen we de correctheid van programmadelen bewijzen.

We gebruiken dus de toestandsbeschrijvingen om tot de procesbeschrijving te komen. Een toestandbeschrijving wordt gegeven in de vorm van een (ware) uitspraak. Een uitspraak kan waar zijn of niet waar; zo is de uitspraak $5 = 2 + 3$ waar en is de uitspraak $3 < 2$ niet waar.
Als P en Q uitspraken zijn over toestanden, en S is een van de constructies (of een totaal algoritme), dan is

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

een notatie voor: als P vooraf geldt (P is preconditie, de beschrijving van de begintoestand) dan geldt na afloop van de actie S de Q (Q is postconditie, de beschrijving van de eindtoestand). P en Q leggen het effect van S vast, tezamen vormen ze de specificatie van S. Als we de correctheid van een algoritme S moeten bewijzen, moeten we dus aantonen dat S vanuit de gegeven begintoestand P leidt tot de gevraagde eindtoestand Q. Als S bijvoorbeeld staat voor het algoritme voor de GGD, dat we zojuist gezien hebben, dan moeten we, om de correctheid van S aan te tonen, bewijzen dat geldt

$$\{x > 0 \text{ en } y > 0\}\ S\ \{g = GGD(x,y)\}$$

We hebben gezegd dat $\{P\}\ S\ \{Q\}$ betekent dat, als P waar is voor de uitvoering van S, dat dan Q waar is als de actie beëindigd is. Of de actie echt eindigt hangt van de constructie af ("while $3 \neq 2$ do $x := x + 1$ od" eindigt niet). We zullen dus ook altijd nog de eindigheid van een constructie moeten aantonen om de totale correctheid te kunnen bewijzen.
Bij het ontwerpen van een algoritme spelen de volgende activiteiten een rol:

- Allereerst wordt de specificatie $\{P\}\ S\ \{Q\}$ van het algoritme gegeven. Er wordt dus vastgelegd wat gegeven is (de preconditie P) en wat bereikt moet worden (de postconditie Q).
- We zoeken tussentoestanden en bijbehorende deelacties waaruit het totale algoritme opgebouwd kan worden. Voor de opdrachten van ieder van de deelacties, S_i, worden de toestandsbeschrijving gegeven.
- Het bewijs van de correctheid van het algoritme (inclusief de eindigheid) gaat hand in hand met de constructie.

We zullen nu, in algemene termen, de semantiekbeschrijvingen geven van de assignment statement en de besturingsstructuren. Deze beschrijvingen

zijn dus in de vorm van uitspraken. Voordat we de beschrijvingen geven, eerst iets over uitspraken. Als P_1 en P_2 uitspraken zijn, dan is $P_1 \wedge P_2$ (P_1 én P_2) de uitspraak die alleen dan waar is als zowel P_1 als P_2 waar zijn, in andere gevallen (P_1 waar, P_2 niet waar; P_1 niet waar, P_2 waar; P_1 niet waar, P_2 niet waar) is $P_1 \wedge P_2$ niet waar. $P_1 \vee P_2$ (P_1 óf P_2) is waar als of P_1 waar is, of P_2 waar is, of P_1 en P_2 waar zijn; $P_1 \vee P_2$ is alleen dan niet waar als zowel P_1 als P_2 niet waar zijn. Als P een uitspraak is die waar is, dan is de uitspraak $\neg$ P (niet P) niet waar; als P niet waar is, dan is $\neg$ P waar (zie ook blz. 35).

Semantiekregels.

. *Assignment statement.*

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

Als na uitvoering van x := E(x) de toestand, beschreven door Q(x) moet gelden, dan moet vooraf de toestand gelden die beschreven wordt door Q met daarin voor x gesubstitueerd E(x).

Voorbeelden:

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

$\{a = (q + 1) * b + r\}\ q := q + 1\ \{a = q * b + r\}$

$\{a = q * b + r\}\ r := r - b\ \{a = (q + 1) * b + r\}$

. *Concatenatie.*

Als S_1 en S_2 beschreven worden door $\{P\}S_1\{Q\}$ en $\{Q\}S_2\{R\}$ dan geldt:

$\{P\}S_1;\ S_2\{R\}$.

Voorbeelden:

$\{n = 7\}\ m := 4;\ m := m + n\ \{m = 11 \wedge n = 7\}$

$\{a = q * b + r\}\ r := r - b;\ q := q + 1\ \{a = q * b + r\}$

. *Selectie/conditie.*

Als voor S_1 en S_2 gelden $\{P \wedge B\}S_1\{Q\}$ en $\{P \wedge \neg B\}S_2\{Q\}$ dan geldt:

$\{P\}$ if B then S_1 else S_2 fi $\{Q\}$.

Voorbeelden:

{a = 5 ∧ b = 7} if a > b then max := a else max := b fi {max = 7}

{x = 8} if s > 0 then t := s + x else t := -s + x fi {t ≥ 8}

Als voor S geldt {P ∧ B}S{Q} en als bovendien geldt dat Q waar is als P ∧ ¬B waar is, dan geldt:

{P} if B then S fi {Q}.

- *Repetitie*

Als S beschreven wordt door {P ∧ B}S{P} dan geldt:

{P} while B do S od {P ∧ ¬B}.

P wordt de *invariante relatie* genoemd (een uitspraak die door S niet wordt beinvloed). Daarnaast hebben we de uitspraak B (ook wel de *variante relatie* genoemd). Wil de totale actie eindigen dan moet de waarde van B beinvloed worden door S. Bovenstaande beschrijving zegt dat als de repetitie eindigt, dat dan P ∧ ¬B geldt.

We zullen niet steeds de bovenstaande regels gebruiken voor de constructie van de algoritmen. De betekenissen van de variabelen (zoals die in de uitspraken over de toestanden gebruikt worden) zullen we wel gebruiken bij de constructie. Bovendien gebruiken we de regels wel bij moeilijker situaties, zoals bijvoorbeeld bij de repetitie. Het is vaak duidelijk dat een bepaalde eindtoestand niet in één stap is te realiseren.

We kunnen dan vaak een repetitie gebruiken en proberen de eindtoestand te karakteriseren door een uitspraak van de vorm P ∧ ¬B , waarin P relatief makkelijk is te realiseren en waarvan B gebruikt wordt in de repetitie.

3. Ontwerpen van een algoritme. Betekenis van variabelen.

Wij zullen eerst enkele eenvoudige voorbeelden van algoritmen bekijken.

Voorbeeld 1.

Stel dat a en b variabelen zijn, die een waarde hebben (a geheel en ≥ 0, b geheel en > 0). Verder bestaan nog de variabelen q en r die alleen gehele waarden kunnen hebben.

Gevraagd wordt aan q en r zodanige waarden toe te kennen dat geldt:

$a = q * b + r$ ...(1)

$0 \leq r < b$ ...(2)

Gevraagd wordt dus een S te construeren waarvoor geldt:

$\{a \geq 0 \wedge b > 0 \wedge$ a en b geheel$\}$ S $\{a = q * b + r \wedge 0 \leq r < b\}$.

We zullen proberen de gewenste eindtoestand, die beschreven wordt door (1) en (2), te bereiken via tussentoestanden. De tussentoestand, die beschreven wordt door (1) is eenvoudig te bereiken:

q := 0; r := a

We hebben dan bovendien al een deel van (2), $r \geq 0$, gerealiseerd. Een ongelijkheid als $r < b$ is te realiseren met behulp van een repetitie

while $r \geq b$ do...od

Aan deze repetitie moeten een aantal eisen gesteld worden:

- de repetitie mag datgene dat reeds bereikt is niet verprutsen ($a = q * b + r$ en $r \geq o$)
- de repetitie moet eindigen

($r < b$ is natuurlijk nog gemakkelijker te realiseren door bijvoorbeeld r := b - w, waarbij w een willekeurig gehele waarde is; $r \geq 0$ blijft gehandhaafd en $r < b$ als $0 < w \leq b$; de bepaling van q voor de handhaving van (1) levert echter problemen op).

De repetitie eindigt als we er bijvoorbeeld in opnemen:

r := r - b

Omdat geldt $b > 0$ wordt r in iedere slag kleiner. Omdat de voorwaarde voor de uitvoering van r := r - b is $r \geq b$ blijft gelden $r \geq 0$. Om (1) te handhaven moet, bij gelijkblijvende a en b, bij q de waarde 1 opgeteld worden (zie bij de semantiekregels in het vorige hoofdstuk).

Het algoritme (met de toestandsbeschrijvingen) luidt dus:

```
{a ≥ 0 ∧ b > 0 ∧ a en b geheel}
q := 0; r := a; {a = q * b + r ∧ r ≥ 0}
while r ≥ b
   do {a = q * b + r  ∧  0 < b ≤ r}
      r := r - b;
      q := q + 1
   od;
{a = q * b + r  ∧  0 ≤ r < b}
```

Voorbeeld 2.

In het vorige hoofdstuk is een algoritme voor de GGD gegeven. Als de gevraagde g = GGD(x, y) klein is ten opzichte van x en y zal de repetitie 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 geldt GGD(a, b) = GGD(a - b, b)
2. GGD(a, b) = GGD(b, a)
3. GGD(a, a) = a

Een algoritme dat hiervan gebruik maakt:

```
a := x; b := y;
while a ≠ b
  do if a > b then a := a - b
               else b := b - a
     fi
  od;
g := a
```

Hoe komen we aan dit algoritme?

In het algoritme worden de waarden van x en y toegekend aan a respectievelijk b. Hierdoor wordt voorkomen dat de waarden van x en y veranderen.

Er geldt:

```
{x > 0 ∧ y > 0}
a := x; b := y
{a > 0 ∧ b > 0  ∧  GGD(x, y) = GGD(a, b)}
```

Eigenschap 3 suggereert dat we a en b gelijk moeten maken.

Dit kan met de repetitie

    while a ≠ b do S od

S zal er voor moeten zorgen dat a en b inderdaad gelijk worden, maar bovendien moet S zodanig zijn dat {a > 0 ∧ b > 0 ∧ GGD(x, y) = GGD(a, b)} blijft gelden.
Het gelijk-worden van a en b kan door de grootste van de twee kleiner te maken. Eigenschap 1 suggereert hoe dit kan zonder dat GGD(x, y) = GGD(a, b) verstoord wordt. En met gebruikmaking van eigenschap 2 komen we dan tot de selectie in bovenstaand algoritme (b := b - a wordt toegepast als ¬(a > b) waar is, dus als a ≤ b waar is; voordat de selectie wordt uitgevoerd geldt echter a ≠ b, dus geldt in dit geval a < b waardoor eigenschap 1 gebruikt kan worden).

Als de repetitie eindigt geldt dus a = b = GGD(x, y). Dat de repetitie eindigt kunnen we bewijzen aan de hand van de waarde van a + b. Steeds geldt dat a + b > 0. Dit geldt namelijk in het begin (a > 0 ∧ b > 0). De assignments a := a - b en b := b - a laten a + b > 0 onveranderd waar. In iedere "slag" van de repetitie wordt a + b echter kleiner (door de aftrekking, en a en b zijn groter dan 0). Wil a + b > 0 dan blijven gelden dan moet de repetitie wel eindigen.
Het aantal herhalingen van dit algoritme is niet altijd kleiner dan het aantal herhalingen van het vorige algoritme, neem maar (x, y) = (6,81). Kan bovenstaand algoritme "versneld" worden?

Voor het bovenstaande algoritme is gebruik gemaakt van kennis over de GGD. Zo ook zouden we aan de specificatie van het probleem van voorbeeld 1 hebben kunnen zien dat het daar ging om geheel delen en rest bepalen. Dit zou ons geholpen kunnen hebben bij het algoritme (geheel delen is herhaald aftrekken). Als we nu bovendien geweten zouden hebben dat er operatoren zijn voor geheel delen (div) en rest bepalen (mod) dan hadden we als algoritme voor voorbeeld 1 kunnen geven:

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

Voorbeeld 3.

Gevraagd wordt een oplossing voor

$x = \cos(x)$ met $0 \leq x \leq \frac{\Pi}{2}$

Daar de oplossing reëel is en deze oplossing benaderd moet worden, zoeken we naar een interval [og, bg], dat voldoende klein is. We gaan ervan uit dat de grootte van dit interval wordt gegeven in de invoerrij f (wordt toegekend aan de variabele epsilon).

We zorgen ervoor dat de gezochte oplossing blijft liggen in het interval [og, bg], dat we gaan verkleinen tot het kleiner is dan de ingelezen lengte.

We kunnen de gewenste eindtoestand

oplossing in [og, bg] ∧ bg - og < epsilon

weer proberen te bereiken via een tussentoestand.

Laten we daarvoor kiezen: oplossing in [og, bg].

Dit is te realiseren door:

```
og := 0; bg := 3.14/2
```

Nu willen we ervoor zorgen dat gaat gelden bg - og < epsilon, zonder dat we "oplossing in [og, bg]" verstoren. We krijgen dan:

```
og := 0; bg := 3.14/2;
read(f, epsilon);
while bg - og ≥ epsilon
  do "verklein [og, bg], maar zorg ervoor dat de oplossing in het
     interval blijft"
  od
```

Het verkleinen van (bg - og) kunnen we als volgt realiseren.

1. Verdeel het interval in twee gelijke delen via m := (bg + og)/2: [og, m] en [m, bg].
2. Als m ≤ cos(m) dan ligt de oplossing van x = cos(x) in het interval [m, bg]; als we dan dus og gelijk maken aan m houden we "oplossing in [og, bg]" geldig. Als m ≥ cos(m) dan ligt de oplossing van x = cos(x) in het interval [og, m]; als we dan dus bg gelijk maken aan m houden we "oplossing in [og, bg]" geldig.

We krijgen dus als procesbeschrijving:

```
og := 0; bg := 3.14/2;
read(f, epsilon);
while bg - og ≥ epsilon
  do m := (bg + og)/2;
     if m ≤ cos(m) then og := m fi;
     if m ≥ cos(m) then bg := m fi
  od;
x := (bg + og)/2
```

Als de repetitie eindigt levert het algoritme de oplossing. Of het algoritme eindigt hangt af van de waarde van epsilon en van de wijze waarop reële getallen door waarden van het type real worden gepresenteerd. Algemeen geldt dat bij reële getallen opgepast moet worden (zie ook hoofdstuk 4).

In de zojuist bekeken voorbeelden is over het algoritme gesproken in termen van toestanden. Nu hebben we in hoofdstuk 1 gezien dat een toestand een rij waarden van variabelen is, maar ook dat een algoritme een klasse van processen beschrijft. We kunnen dus niet spreken over een individuele toestand, maar moeten ook spreken over een klasse toestanden. Een klasse van toestanden wordt niet gekarakteriseerd door waarden van variabelen, maar door de betekenissen van variabelen en hun onderlinge relaties. *Deze betekenissen en relaties gebruiken we ook als hulpmiddelen bij het ontwerpen van een algoritme*. Vooral ook bij de repetitie blijkt dit een goed hulpmiddel te zijn. We passen dus de regel voor de repetitie toe. We gaan hierbij als volgt te werk. We zorgen ervoor dat de betekenissen van de variabelen gelden op het moment dat de repetitie uitgevoerd gaat worden. Bovendien zorgen we ervoor dat deze betekenissen gehandhaafd blijven bij de uitvoering van de repetitie (invariante relatie). De betekenissen van de variabelen moeten zodanig zijn dat zij, samen met de ontkenning van de voorwaarde in de repetitie (variante relatie), de gewenste eindtoestand (na afloop van de repetitie) beschrijven. In feite gaan we omgekeerd te werk. Uit de gewenste eindtoestand, beschreven door uitspraken over variabelen (onder andere de betekenis) leiden we de voorwaarde voor de repetitie en de betekenis van de variabelen af. In de voorgaande voorbeelden is deze aanpak eigenlijk al toegepast. We doen het nu nog explicieter.

Stel dat berekend moet worden (bij gegeven waarde voor p, ≥ 0)

$$1 + 2 + 4 \ldots + 2^p = \sum_{k=0}^{p} 2^k$$

De berekende waarde zal in het algoritme voorgesteld moeten worden door een variabele, noem die s. Na uitvoering van het algoritme moet dus gelden

$$s = \sum_{k=0}^{p} 2^k$$

We geven de variabele s een zodanige betekenis dat deze eindtoestand een bijzonder geval is van de algemene toestandsbeschrijving. Zo kunnen we nemen

$$s = \sum_{k=0}^{n} 2^k$$

Voor iedere tussentoestand geldt dat s gelijk is aan de som van de eerste n termen. Deze uitspraak, die voor alle toestanden geldt, is de *invariante relatie*. Vaak komt het er op neer dat de betekenissen van de variabelen gehandhaafd blijven. De eindtoestand is dan die tussentoestand waarvoor geldt n = p. Ook de begintoestand kunnen we zien als een bijzonder geval van de algemene tussentoestand, als namelijk geldt n = 0. Maar dan moet ook gelden s = 1.
De repetitie zal van de vorm **while** n ≠ p **do** ... **od** zijn. Om de eindigheid te garanderen kan in de repetitie n met 1 opgehoogd worden. Daardoor is de betekenis van de variabele s verstoord. Deze betekenis is te herstellen door bij s op te tellen de tweemacht van de "huidige" n.
Het algoritme wordt dan:

```
n := 0; s := 1;
while n ≠ p
  do n := n + 1;
     s := s + 2↑n
  od
```

De hierboven geschetste werkwijze zullen we algemeen toepassen. We geven de variabelen een zodanige vaste betekenis (invariante relatie) dat de begin- en de eindtoestand bijzondere gevallen zijn van deze algemene toestand. We zorgen ervoor dat de betekenis geldt vóór de uitvoering van de repetitie. De eindtoestand vraagt een extra voorwaarde (in het voorbeeld n = p). Deze voorwaarde gebruiken we bij het stopcriterium van de repetitie (de extra voorwaarde is het stopcriterium, de ontkenning van de variante relatie B). Bovendien zorgen we in de repetitie voor de eindigheid met tegelijkertijd het herstellen van de betekenis van de variabelen.

Voor we de bovenstaande werkwijze gaan toepassen op enkele voorbeelden nog enkele afsluitende opmerkingen over het bovenstaande algoritme. In het algoritme wordt bij iedere nieuwe n berekend 2↑n, terwijl in de vorige slag van de repetitie is berekend 2↑(n - 1). Om hiervan gebruik te kunnen maken introduceren we een nieuwe variabele met de betekenis

t = 2↑n

Het algoritme wordt dan:

```
n := 0; t := 1; s := 1;
while n ≠ p
  do n := n + 1; t := t * 2;
     s := s + t
  od
```

Ook voor dit probleempje geldt dat we gebruik kunnen maken van de kennis die we van het probleemgebied hebben om tot een ander, wellicht "beter" algoritme te komen.
Voor de gevraagde som geldt:

$$\sum_{k=0}^{p} 2^k = 2^{p+1} - 1$$

en het algoritme wordt teruggebracht tot één assignment statement

s := 2↑(p + 1) - 1

Voorbeeld 4.

Gevraagd wordt het maximum te bepalen van een rij positieve getallen, die stuk voor stuk uit de invoerrij f "ingelezen" kunnen worden (door middel van read(f,x)). De invoerrij wordt afgesloten door een 0.
We zullen een variabele nodig hebben voor het maximum en ook een voor de gelezen waarde. Als de 0 gelezen is, moet het maximum van de voorgaande rij bekend zijn. We geven de variabelen daarom de volgende betekenissen:

> g is de laatst gelezen waarde uit de invoerrij; van alle getallen die voorafgaan aan de huidige waarde van g is het maximum bepaald in m.

De eindtoestand wordt gekarakteriseerd door g = 0.
Als we er voor zorgen dat voor de repetitie één waarde is gelezen en m de waarde 0 heeft (alle invoerwaarden zijn groter dan 0), geldt de betekenis ook voor de repetitie (met als definitie dat het maximum van een lege rij positieve getallen 0 is). De repetitie zal eindigen als in iedere slag van de repetitie een volgende waarde uit de invoerrij wordt gelzen. Maar om de betekenissen van g en m te handhaven, moet dan, voordat g een nieuwe waarde krijgt, eerst de huidige waarde van g in m verwerkt worden; de nieuwe waarde voor m is: maximum(m,g).

Het algoritme wordt:

```
m := 0; read(f, g);
while g ≠ 0
  do if g > m then m := g fi;
     read(f, g)
  od
```

Voorbeeld 5.

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

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

Als we de getallen aangeven met f_i, dan geldt:

$f_0 = 0$

$f_1 = 1$

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

Gevraagd wordt bij gegeven waarde van n (n geheel en ≥ 0) f_n te bepalen.

Om de termen te bepalen hebben we als variabelen nodig:

t: de laatst bepaalde term.

Voor n = 0 of n = 1 is t eenvoudig te bepalen. Voor $n \geq 2$ zullen we met behulp van een repetitie de betreffende t moeten bepalen. We krijgen dus als opzet

```
if n ≤ 1 then t := n
         else "t bepalen via repetitie"
fi
```

We zullen de termen moeten aftellen en introduceren daarom:

k: het rangnummer van de als laatste bepaalde term (dus $t = f_k$).

Omdat iedere nieuwe waarde voor term gevonden wordt als de som van de vorige term en zijn voorganger introduceren we voor de repetitie:

ℓ: de (k-1)-ste term (f_{k-1}).

Voor de repetitie geldt dan

Eindtoestand : k = n

Begintoestand: t = 1, k = 1, ℓ = 0

Repetitie : De nieuwe waarde van t wordt gevonden als som van de huidige waarde van t en de waarde van ℓ. De nieuwe waarde van ℓ moet gelijk worden aan de huidige waarde van t. We hebben dus naast elkaar zowel de huidige als de nieuwe waarde van de term nodig. Daarom introduceren we binnen de repetitie een hulpvariabele

v: de (k-2)-de term (f_{k-2}).

Het algoritme luidt:

```
if n ≤ 1
then term := n
else k := 1; ℓ := 0; t := 1; {t = f_k, ℓ = f_{k-1}}
     while k ≠ n
       do {t = f_k, ℓ = f_{k-1}}
          k := k + 1; {t = f_{k-1}, ℓ = f_{k-2}}
          v := ℓ; {v = f_{k-2}}
          ℓ := t; {v = f_{k-2}, ℓ = f_{k-1}}
          t := v + ℓ {v = f_{k-2}, ℓ = f_{k-1}, t = f_k}
       od
fi
```

We hebben in de repetitie v geïntroduceerd omdat door het sequentiële karakter van de processen noch

```
t := t + ℓ; ℓ := t
```

noch

```
ℓ := t; t := t + ℓ
```

de gevraagde toestand ($t = f_k$ en $\ell = f_{k-1}$) opleveren. Kunnen we v dan niet vermijden? Laten we nog eens naar de repetitie kijken.

```
k := k + 1; {t = f_{k-1}, ℓ = f_{k-2}}
```

Om f_k te berekenen moeten we f_{k-1} en f_{k-2} optellen:

```
k := k + 1; {t = f_{k-1}, ℓ = f_{k-2}}
t := t + ℓ; {t = f_k, ℓ = f_{k-2}}
```

Nu moet ℓ nog de waarde van f_{k-1} krijgen. Uit de definitie volgt dat $f_{k-1} = f_k - f_{k-2}$. We krijgen dan voor de repetitie:

```
while k ≠ n
   do k := k + 1; t := t + ℓ; ℓ := t - ℓ od
```

Zo zien we hoe de sematiekregels ons kunnen helpen bij het vinden van de procesbeschrijving.

Voorbeeld 6.

Cosinus-benadering.

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots \qquad (x \text{ in radialen, } 0 \leq x \leq \frac{\Pi}{2})$$

Wordt deze reeks na een eindig aantal termen afgebroken, dan is de afbreekfout ten hoogste gelijk aan de (absolute waarde van de) eerstvolgende term in de reeks.

Gevraagd wordt cos(x) te benaderen bij gegeven waarde van x en bij gegeven afbreekfout (in de variabele eps).

We introduceren de variabelen:

t: eerstvolgende nog niet verwerkte term (met + of - teken)

c: de benadering voor cos(x), waarin t nog niet is opgenomen.

We krijgen dan als globale opzet voor het algoritme:

```
"geef c en t beginwaarden", {betekenis!}
while abs(t) > eps
   do c := c + t;
      "bepaal volgende term"
   od
```

(abs(e) is een functie die als waarde oplevert de absolute waarde van e)

Als we een nieuwe term willen berekenen, krijgen we

```
t := - t * x↑2/ ...
```

Wat moeten we invullen?

We introduceren nieuwe variabelen:

n : voor de berekening van n! in te maken term

noemer: de noemer van de te maken term

teller: de teller van de te maken term

Bij gegeven t (en de daarbij behorende n, noemer, teller), die reeds verwerkt is in c, vinden we de nieuwe t uit:

```
n := n + 2;
noemer := noemer * (n - 1) * n;
teller := - x * x * teller;
t   := teller / noemer
```

Het algoritme wordt hiermee:

```
c := 1;
n := 2; noemer := 2; teller := - x * x;
t := teller/noemer; {t is nog in c op te nemen term}
while abs(t) > eps
   do c := c + t;
      n := n + 2; noemer := noemer * (n - 1) * n;
      teller := - x * x * teller;
      t := teller / noemer
   od
```

Het kan ook zonder teller en noemer:

```
t := - t * x * x / (n * (n - 1))
```

4. Typen

Iedere waarde in een algoritme is van een bepaald type. Een waarde wordt o.a. gegeven door een constante of de waarde van een variabele. Variabelen moeten dan ook van een bepaald type zijn; de variabele kan alleen waarden aannemen uit de bij het type behorende waardenverzameling.
Een toestand van het proces wordt bepaald door de waarden van de in het proces optredende variabelen. De typen van deze variabelen leggen tezamen vast welke toestanden in het proces mogelijk zijn. De verzameling van alle mogelijke toestanden noemt men de *toestandsruimte* van het proces. De toestandsruimte van een proces wordt bepaald door de *declaratie van de variabelen*. In de declaratie van een variabele worden van de variabele de naam en het type vastgelegd; de waarde van de variabele is op dat moment nog ongedefinieerd. De declaraties van de variabelen worden in het begin van de betreffende procesbeschrijving (of deelprocesbeschrijving) opgenomen, vóór de statements. Na afloop van een proces bestaat de toestandsruimte van dat proces niet meer.
Wij zijn als typen reeds tegengekomen: *integer*, *real* en *boolean*.
Voorbeelden van declaraties zijn:

```
integer k, m, n;
real x, y, z;
boolean p
```

De declaraties worden onderling gescheiden door puntkomma's en ook wordt het declaratiegedeelte van de eigenlijke procesbeschrijving gescheiden door een puntkomma.

```
begin integer k; real getal, som, gemiddelde;
      som := 0; k := 0;
      read(f, getal);
      while getal ≠ 0
         do k := k + 1;
            som := som + getal;
            read(f, getal)
         od;
      if k ≠ 0 then gemiddelde := som/k; write(g, gemiddelde)
         else write(g, "geen gemiddelde")
      fi
end
```

begin en end treden op als een soort openingshaakje en sluithaakje voor een totale procesbeschrijving.

De toestand wordt bepaald door de waarden van de relevante variabelen. Voor de interpretatie van de toestand is de *betekenis* van de variabelen belangrijk. Een variabele wordt niet zomaar gedeclareerd, hij speelt een bepaalde rol in het proces.
Als we op ieder moment een eenduidige uitspraak wensen te doen over de heersende toestand, dan moet de betekenis van een variabele gedurende het hele proces dezelfde zijn. De betekenis van de variabele kunnen we tot uitdrukking te laten komen in de naam; het is echter niet mogelijk deze betekenis eenduidig vast te leggen in de naam. Een nadere omschrijving voor onszelf is daarom noodzakelijk.

Voor ieder type zijn als operaties gedefinieerd:
- de assignment: de toekenning van een waarde aan een variabele;
- de test op gelijkheid van twee waarden.

Daarnaast zijn voor ieder type een aantal bij dat type behorende operaties gedefinieerd. Zijn V en W waardenverzamelingen van twee typen, dan kunnen we de operaties onderscheiden in:
- unaire operaties : afbeeldingen van de vorm $V \rightarrow V$ of $V \rightarrow W$;
- binaire operaties: afbeeldingen van de vorm $V \times V \rightarrow V$ of $V \times V \rightarrow W$.

Een unaire operatie $V \rightarrow V$ kent aan een element van V, de operand, eenduiding een element van V, het resultaat, toe. Een voorbeeld is het tegengestelde nemen bij het type integer ($a \rightarrow -a$). Een binaire operatie $V \times V \rightarrow V$ kent aan een tweetal elementen van V, de operanden, eenduidig een element van V, het resultaat, toe. Een voorbeeld is de optelling voor het type real. Omdat de processor met eindige verzamelingen werkt, kan het voorkomen dat niet voor ieder tweetal operanden (of iedere operand) de operaties gedefiniëerd zijn. (Stel bijvoorbeeld dat V = {1, 2, 3, 4} en dat de operatie optellen is; alleen de optellingen 1 + 1, 1 + 2, 1 + 3 en 2 + 2 zijn dan gedefinieerd.)
De operaties $V \rightarrow W$ en $V \times V \rightarrow W$ zijn zogenaamde type transferoperaties. Zij voegen aan een waarde (of een tweetal waarden) van een bepaald type een waarde van een ander type toe. De operaties $V \rightarrow W$ zijn vaak impliciet. Als a een real variabele is, dan vindt bij de assignment a := 0 een type transferplaats op de waarde 0 (van integer naar real). Een klasse van type transferoperaties van de vorm $V \times V \rightarrow W$ zijn de relaties (bijvoorbeeld $p > 3$). Deze zijn gedefinieerd voor de typen real en integer, omdat deze typen geordend zijn, dat wil zeggen dat er een volgorde is gedefinieerd op de waardenverzameling.

De typen integer, real en boolean.

Het type integer.

De *waardenverzameling* is, door de beperking van de processor, een deelverzameling van de verzameling $\mathbb{Z}$ van de gehele getallen, te weten het gesloten interval [min(integer), max(integer)]; min(integer) en max(integer) zijn grenzen die door de processor bepaald worden.

Operaties:

- unaire operaties:

  inverteren , met als notatie: -i

  identiteit, met als notatie : +i

- binaire operaties:

notatie	betekenis	opmerkingen
i + j	som	
i - j	verschil	
i * j	produkt	
i div j	geheel quotiënt	opmerking 1
i mod j	rest	opmerking 1
i↑j	i^j	opmerking 2

opmerking 1: Deze operatie is niet gedefinieerd voor j = 0

opmerking 2: Deze operatie is niet gedefinieerd voor j < 0 en voor (i = 0 en j = 0).

Voor de definities van het geheel quotiënt en de rest verwijzen we naar hoofdstuk 3 (waarbij nog vermeld moet worden dat tekenregels zijn zoals bij het produkt).

De operaties zijn alleen gedefinieerd als het resultaat in het interval [min (integer), max (integer)] ligt. Een operatie als 4↑(-2) is niet gedefinieerd binnen de integers, het resultaat is van het type real.

De *ordening* welke bestaat op de verzameling der gehele getallen bestaat ook op het type integer. Deze ordening geeft de mogelijkheid tot de relatie-operaties op twee elementen.

De relatie-operatoren zijn $<$, $\leq$, $=$, $\geq$, $>$, $\neq$.

Naast de bovengenoemde operaties, kennen we nog een tweetal operaties, die we noteren als functie:

abs(i) , met als resultaat $|i|$

sign(i), met als resultaat + 1 als i > 0
0 als i = 0
- 1 als i < 0

Constanten van het type integer worden geschreven als een rijtje cijfers, eventueel voorafgegaan door een plus- of een minteken.

Het type real.

De *waardenverzameling* van het type real is, door de beperking van de processor, een deelverzameling van de verzameling $\mathbb{R}$ der reële getallen, bepaald door twee restricties:

- Ieder element van het type real ligt in het gesloten interval [min(real), max(real)], waarbij min(real) en max(real) bepaald worden door de processor.
- Het interval [min(real), max(real)] bevat slechts een eindig aantal elementen van het type real; ieder element is een rationaal getal.

Een gevolg van de tweede restrictie is dat de meeste reële getallen, die liggen in het interval [min(real), max(real)] niet exact kunnen worden gerepresenteerd door een waarde van het type real; zij moeten door een waarde van het type real benaderd worden. De restrictie ontstaat door de representatie van de waarden binnen de processor.

Operaties:

- unaire operaties:
  inverteren, met als notatie: - r
  identiteit, met als notatie: + r
- binaire operaties:

notatie	betekenis	opmerkingen
r + s	som	
r - s	verschil	
r * s	produkt	
r / s	quotiënt	opmerking 1
r ↑ s	r^s	opmerking 2

opmerking 1: Deze operatie is niet gedefinieerd voor s = 0

opmerking 2: Deze operatie is niet gedefinieerd voor r < 0 en voor (r = 0 en s ≤ 0).

De genoemde operaties zijn de bekende operaties op reële getallen, met dien verstande dat ze als resultaat een benadering van het exacte resultaat geven. De operaties zijn alleen gedefinieerd als het resultaat in het interval [min(real), max(real)] ligt.
De *ordening* in de verzameling der reële getallen bestaat eveneens in het type real, waardoor de relatie-operatoren (real * real → boolean) gedefinieerd zijn.
Naast bovengenoemde operaties, die om voor de hand liggende redenen met speciale symbolen worden genoteerd, zijn er vaak nog een aantal operaties, die als functie genoteerd worden, zoals bijvoorbeeld:

notatie	waarde
sign(r)	+1 als r > 0 0 als r = 0 -1 als r < 0
entier(r)	de grootste gehele waarde niet groter dan r
abs(r)	$\|r\|$
sqrt(r)	$\sqrt[2]{r}$
exp(r)	e^r
ln(r)	$^e\log r$
sin(r)	sinus van r (r in radialen)
cos(r)	cosinus van r (r in radialen)

De eerste twee operaties zijn van het soort real → integer, de andere zijn van het soort real → real.

Constanten van het type real worden geschreven als $p.q_{10}r$ (met als waarde: $p.q*10^r$). Hierin is p een rijtje cijfers (lengte ≥ 0) eventueel voorafgegaan door een plus- of een minteken, q is een rijtje cijfers (lengte ≥ 0; is de lengte 0 dan blijft ook de punt er voor weg) en r is een rijtje cijfers (lengte ≥ 0; is de lengte 0 dan blijft ook het lage tientje er voor weg). Met deze notatie zijn getallen te noteren die in de processor eventueel niet exact zijn te representeren. In de algorithmen zullen door deze benaderingen niet-exacte resultaten worden berekend.

Voorbeeld 1 (ter illustratie van mogelijke rekenfouten).
Stel dat de variabele m een waarde heeft (m geheel en groter dan 0). Een wiel met straal 1 rolt vanaf de oorsprong in positieve richting. Na hoeveel volledige omwentelingen is punt m gepasseerd (is de afgelegde afstand ≥ m)?
We voeren als variabelen in:

omw : aantal gemaakte omwentelingen

afstand: de overbrugde afstand bij omw omwentelingen.

Er geldt : afstand = omw * 2Π. Voor Π nemen we als constante in het programma op 3.14.
Het programma wordt:

```
afstand := 0; omw := 0; {de betekenis van afstand en omw geldt}
while afstand < m
  do afstand := afstand + 2 * 3.14;
     {betekenis geldt niet meer}
      omw := omw + 1
     {betekenis hersteld}
  od
```

Als m groot is, is het mogelijk dat dit programma niet goed is door de fout die in de waarde van Π gemaakt is. Ook als we Π nauwkeuriger opgeven dan de nu gebruikte 3.14 is er een m te vinden zodanig dat de fout die we maken in de waarde van Π kan leiden tot een foute waarde voor 'omw'.

Voorbeeld 2 (ook in verband met rekenfouten).
In het vorige hoofdstuk is een programma ontwikkeld voor de berekening van

$$\sum_{k=0}^{p} 2^k$$

Is dit algoritme ook te gebruiken voor

$$\sum_{k=1}^{p} \tfrac{1}{2}^k ?$$

Stel dat de processor waarden van het type real representeert als $(x_1x_2x_3, y_1y_2y_2)$, waarin x_i en y_i decimale cijfers zijn, waarvan de waarde is $0.x_1x_2x_3 * 10^{y_1y_2y_3}$ (eventueel met tekens erbij). Stel ook dat, als een getal uit meer dan drie cijfers bestaat, het vierde cijfer en de volgende cijfers gewoon worden weggelaten. Het exacte resultaat voor bovenstaande som is 0.9990234375 als p = 10. De optelling tot en met de term met k = 9 is 0.994. De exacte waarde van de term met k = 10 is 0.0009765625. De waarde van deze term in de processor is 0.000976. Deze term geeft dan geen bijdrage meer tot de som. Het resultaat is dus 0.994. Er treedt een groot verschil op met de exacte som.
Als de som bepaald wordt in "omgekeerde volgorde" (beginnen met k = 10, dan k = 9, enzovoorts) dan krijgen we als resultaat 0.998. Doordat bij deze werkwijze steeds waarden bij elkaar worden opgeteld die van dezelfde grootte-orde zijn is het resultaat nu beter.
Om het verschil met de exacte berekening te illustreren hebben we aangenomen dat de processor afkapt om tot drie cijfers te komen; veel processoren ronden af.
Ook met

$$\sum_{k=0}^{p} 2^k$$

moeten we oppassen omdat bij grote p het resultaat niet in het interval van het type integer past.

Het type boolean.

De *waardenverzameling* bestaat uit twee elementen, die we noteren als true en false (de elementen van het type boolean zijn de zogenaamde logische waarden, waarden van "uitspraken").
Operaties:

- unaire operatie:
  ontkenning, met als notatie $\neg a$
- binaire operaties:
  conjuctie , met als notatie $a \wedge b$
  disjunctie, met als notatie $a \vee b$

De genoemde operaties kunnen we definiëren met behulp van tabellen (a en b staan voor vormen die van het type boolean zijn).

a	¬a
true	false
false	true

a	b	a ∧ b	a ∨ b
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

Het type boolean is niet geordend.

Expressies.

Een expressie is een notatie voor een samengestelde operatie: een rij operanden onderling gescheiden door operatoren (eventueel met gebruik van ronde haakjes).

De samenstellende operaties zijn de genoemde unaire en binaire operaties van de typen. Iedere operator verwacht operanden van één bepaald type en levert een resultaat af van een bepaald type. Als hetzelfde symbool wordt gebruikt voor operaties in verschillende typen (zoals + voor de optelling in zowel het type real als het type integer), dan moeten we dit eigenlijk opvatten als verschillende operatoren. We zullen echter expressies toelaten waarin operanden en operatoren van de typen real en integer naast elkaar voorkomen. Als het type integer en het type real gemengd in expressies optreden spreken we van *aritmetische expressies*. In bijvoorbeeld i + r (i van het type integer en r van het type real) zal een impliciete type transfer optreden voor i, zo ook voor i1/i2. Niet toegestaan zijn r div s en r mod s (r en/of s van het type real); er treden dus geen impliciete type transfers op van real naar integer.

Een expressie met operanden en operatoren van het type boolean wordt een *boolean expressie* genoemd. In hoofdstuk 2 hebben we bij de programmastructuren gesproken van voorwaarden waar al dan niet aan voldaan wordt. We kunnen nu spreken van boolean expressies die true of false zijn.

De operanden in een aritmetische expressie zijn variabelen of constanten (dit zijn notaties voor de elementen van de typen). De operanden in een boolean expressie zijn variabelen, constanten of relaties (uitdrukkingen van bijvoorbeeld de vorm a1 < a2, waarin a1 en a2 aritmetische expressies zijn).

Zo is

$$a + b > c \wedge \neg (d * e > 0)$$

een boolean expressie.

Een expressie heeft een waarde, die berekend kan worden als de variabelen een waarde hebben. De waarde van de expressie is eenduidig bepaald doordat er een volgorde afgesproken is, waarin de operaties worden uitgevoerd (de operatoren hebben prioriteiten ten opzichte van elkaar). Zo worden in een aritmetische expressie eerst de machtsverheffingen uitgevoerd, daarna de vermeningvuldigingen, helingen (div), restbepalingen (mod) en delingen (met gelijke prioriteit) en dan de optellingen en aftrekkingen (met gelijke prioriteit). Bij gelijke prioriteit worden de operaties van links naar rechts uitgevoerd. De volgorde, die zo is vastgelegd, kan doorbroken worden door het gebruik van ronde haakjes; hetgeen tussen haakjes staat wordt eerst uitgewerkt volgens de gegeven regels.
Denk er dus om dat bijvoorbeeld vermenigvuldigen "niet gaat voor" delen. Zo is het resultaat van $4 * 4/2 * 2$ gelijk aan 16.
Bij de waardebepaling van een boolean expressie gelden ook prioriteitsregels. Eerst wordt (eventueel herhaaldelijk) de waarde bepaald van hetgeen tussen haakjes staat. Voor het stuk tussen haakjes en uiteindelijk voor de totale expressie geldt als volgorde van uitwerking: aritmetische expressies in relaties, relaties, ontkenning, conjunctie en disjunctie. Bij gelijke prioriteit is de volgorde van uitwerking van links naar rechts. Als we niet gewend zijn met boolean expressies te werken is het omwille van de duidelijkheid aan te bevelen om met haakjes zelf de volgorde van uitwerking vast te leggen.
Denk er om dat bijvoorbeeld $0 < x < 3$ geen correcte boolean expressie is, wel $0 < x \wedge x < 3$.
Bij de assignment statement moet het type van de expressie hetzelfde zijn als het type van de variabele (in het linker lid), met dien verstande dat bij aritmetische expressies het type van de variabele zowel real als integer mag zijn. Bij de toekenning r := i vindt een automatische typetransfer plaats voor de waarde van de expressie; bij de toekenning i := r wordt als waarde aan i toegekend: entier (r + 0.5).

Arrays.

Het komt vaak voor dat een rij waarden gerelateerd moet worden aan andere waarden. Zo kan gevraagd worden naar de aantallen mannen bij de lengtes 150, 151, 152, ..., 200. We geven de relatie tussen de lengtes en de aantallen een naam, bijvoorbeeld door de declaratie

```
integer array aantal [150:200]
```

Voor iedere i, 150 ≤ i ≤ 200, staat **aantal**[i] voor een waarde van het type integer; het treedt op als een variabele. We kunnen bijvoorbeeld waarden toekennen

aantal[175] := 542

en aantal[182] := 739

Ook toegestaan is **aantal**[i], als i van te voren maar een waarde heeft gekregen tussen 150 en 200. We noemen i de index. In feite is als index een expressie toegestaan.

Men spreekt van de *componenten* van het array, die door middel van een *index* uit het totale array worden geselecteerd.

De waarde van een array is een afbeelding van de mogelijke indices (bijvoorbeeld: [150:200]) in het type van de componenten (bijvoorbeeld: integer).

De array-structuur kan ook gebruikt worden voor het voorstellen van vectoren en matrices. De vector $v = (v_1, v_2, \ldots, v_{40})$ kan gerepresenteerd worden door

real array v[1:40]

en de matrix

$$A=\begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,10} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,10} \\ \cdots & \cdots & \cdots & \cdots \\ a_{7,1} & a_{7,2} & \cdots & a_{7,10} \end{pmatrix}$$

door

real array A[1:7, 1:10]

Men zegt dat het array v *ééndimensionaal* is en het array A tweedimensionaal; dit laatste houdt in dat de componenten van A geselecteerd worden met behulp van twee indices, bijvoorbeeld: A[3,9], A[7,1] en A[i,j]. Ook meer dan twee dimensies zijn toegestaan.

Voorbeeld 3.

Gegeven: integer array A[1:100]

de componenten zijn gedefinieerd (hebben een waarde).

Gevraagd: bepaal het maximum van de componenten.

1e oplossing.

We introduceren twee variabelen, max en i, met als betekenissen:

max = maximum(A[1], A[2], ..., A[i-1]).

Het algoritme wordt dan:

```
max := A[1]; i := 2; {betekenis geldig}
while i ≤ 100
   do if A[i] > max then max := A[i] fi; {betekenis verstoord}
      i := i + 1 {betekenis geldig}
   od
 {max = maximum(A[1], A[2], ..., A[i-1]) ∧ i = 101}
```

2e oplossing.

We kunnen aan max en i ook de volgende betekenis geven:

max = maximum(A[1], A[2], ..., A[i]).

Dan wordt het algoritme:

```
max := A[1]; i := 1;
while i ≠ 100
   do i := i + 1;
      if A[i] > max then max := A[i] fi
   od
 {max = maximum(A[1], A[2], ..., A[i]) ∧ i = 100}
```

N.B.

Voor 1e oplossingsmethode bestaat ook een andere notatie:

```
max := A[1];
for i := 2 step 1 until 100
   do if A[i] > max then max := A[i] fi od
```

De besturingsstructuur

for v := A step B until C do S od

is dus niet anders dan een andere schrijfwijze voor sommige gevallen van het gebruik van een reeds bekende besturingsstructuur: de while. In feite komt bovenstaande constructie overeen met:

```
if A ≤ C then v := A;
              while v ≤ C do S; v := v + B od
fi
```

Deze beschrijving geldt voor positieve A, B en C. Een zelfde beschrijving is te geven als A, B en C niet alle drie positief zijn

Als de 10 * 15 matrix A en de 15 * 20 matrix B in het programma gerepresenteerd worden door

```
real array a[1:10, 1:15], b[1:15, 1:20]
```

en deze arrays hebben een waarde, dan kan het matrix-produkt

$$c_{ij} = \sum_{k=1}^{15} a_{ik} * b_{kj} \quad (i = 1,2,\ldots,10;\ j = 1,2,\ldots,20),$$

gerepresenteerd in

```
real array c[1:10, 1:20]
```

berekend worden door middel van

```
for i := 1 step 1 until 10
    do for j := 1 step 1 until 20
           do c[i, j] := 0;
              for k := 1 step 1 until 15
                  do c[i,j] := c[i,j] + a[i,k] * b[k,j] od
           od
    od
```

Voorbeeld 4.

De Nederlandse vlag.

Het array

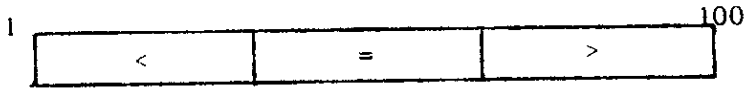
integer array x[1:100]

en de variabele

integer getal

hebben een waarde.

Gevraagd wordt een programmadeel dat de waarden van de elementen zodanig herrangschikt (door het verwisselen van waarden van elementen) dat, gezien vanaf index 1, eerst alle waarden komen die kleiner zijn dan getal, dan de waarden die gelijk zijn aan getal en tenslotte alle waarden die groter zijn dan getal.



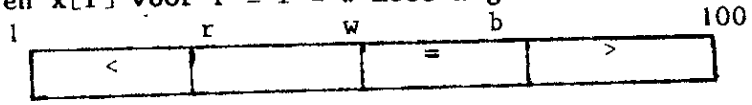
We kunnen de toestanden karakteriseren door:

x[i] < getal voor 1 ≤ i < r,

x[i] = getal voor w < i ≤ b

x[i] > getal voor b < i ≤ 100,

en x[i] voor r ≤ i ≤ w moet nog onderzocht worden.



We proberen het probleem op te lossen door alle waarden (van het oorspronkelijke array) slechts één keer te inspecteren.

We hebben dan wat de inspectie betreft, het beste algoritme gevonden.

De begintoestand is een speciaal geval van bovenstaande algemene tussentoestand, namelijk r = 1, b = 100 en w = 100.

De eindtoestand is bereikt als r = w + 1. De eindigheid is gegarandeerd als in iedere slag van de repetitie w - r kleiner wordt (door w af te lagen of r op te hogen). Om w - r kleiner te maken kan bijvoorbeeld x[w] op zijn plaats gezet worden. Er kunnen zich dan drie gevallen voordoen:

- x[w] = getal:
  Door een verlaging van w met 1 blijven de betekenissen van de variabelen gehandhaafd en wordt w - r kleiner
- x[w] < getal

Door een verwisseling van de waarden van x[r] en x[w] en een verhoging met 1 van r blijven de betekenissen van de variabelen gehandhaafd en wordt w - r kleiner.

- x[w] > getal:

Door een verwisseling van de waarden van x[b] en x[w] en een verlaging met 1 van b en w blijven de betekenissen van de variabelen gehandhaafd en wordt w - r kleiner.

Het stukje programma wordt dus:

```
r := 1; w := 100; b := 100;
while r ≠ w + 1
   do h := x[w];
      if h = getal then w := w - 1
                   else
      if h < getal then p := x[r]; x[r] := x[w]; x[w] := p;
                        r := r + 1
                   else p := x[b]; x[b] := x[w]; x[w] := p;
                        b := b - 1; w := w - 1
      fi
      fi
   od
```

In plaats van x[w] zou ook het element x[r] geïnspecteerd kunnen worden. Maak dit enig verschil voor het aantal verwisselingen?

Het traject van gelijke waarden zouden we direct achter het traject van de kleinere waarden kunnen leggen. Maakt dit enig verschil voor het aantal verwisselingen?

We zouden het traject van nog niet geïnspecteerde waarden helemaal "achteraan" kunnen leggen. Maakt dit enig verschil voor het aantal verwisselingen?

We zouden er voor kunnen zorgen dat x[w] alleen met x[r] verwisseld wordt als niet geldt dat x[r] < getal. Maakt dit enig verschil voor het aantal inspecties en of verwisselingen?

De bovenstaande vragen hebben betrekking op de efficiëntie van het algoritme, uitgedrukt in aantallen verwisselingen en/of inspecties.

Voorbeeld 5.

Genereer de eerste (de kleinste) honderd elementen van de (geordende) verzameling M, die als volgt is gedefinieerd:

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

De eerste vijftien elementen van M zijn:

1, 3, 4, 7, 9, 10, 13, 15, 19, 21, 22, 27, 28, 31, 39.

We introduceren

integer array M[1:100]

In M komen de elementen (M[i] < M[i+1] voor i = 1, 2, ..., 99).

Tussentoestand: M[1], M[2], ..., M[k] bevatten de eerste k elementen van de verzameling M.

- begin: M[1] = 1, k = 1;
- einde: k = 100;
- stap maken door verhoging van k met 1; dan moet M[k] gevuld worden met de kleinste waarde (uit de formules 2x + 1 en 3x + 1) die nog niet is opgenomen.

Maar dan moeten we bijhouden welke elementen al in de formules 2x + 1 en 3x + 1 gebruikt zijn; daarom gebruiken we naast de k om de tussentoestanden vast te leggen ook een a en een b met de betekenissen:

a: voor i = 1, 2, ..., a-1 zijn de waarden 2M[i] + 1 al wel en voor i = a, a+1, ..., k zijn de waarden 2M[i] + 1 nog niet verwerkt in M

b: evenals bij a, maar dan met de formule 3M[i] + 1.

Zo krijgen we:

```
k := 1; M[1] := a := 1; b := 1;
while k ≠ 100
    do k := k + 1;
       y := 2 * M[a] + 1; z := 3 * M[b] + 1;
       if y ≤ z then M[k] := y; a := a + 1 fi;
       if y ≥ z then M[k] := z; b := b + 1 fi
    od
```

Een opmerking over efficiëntie.

Als we y en z niet alleen als hulpvariabelen gebruiken, maar ook in de beschrijvingen van de tussentoestanden opnemen (en y(z) dus veranderen als a(b) verandert), dan worden y en z minder vaak aangepast.

5. Procedures

5.1. Inleiding

Een actie is een toestandstransformatie, die in een eindige tijd plaatsvindt en een welgedefinieerd effect heeft. Deze definitie zegt alleen dat er een transformatie plaatsvindt (*wat* er gebeurt) en niet *hoe* het effect tot stand komt. Het effect is de transformatie van een begintoestand in een eindtoestand. De begintoestand wordt gekerakteriseerd door de *invoerwaarden* voor de actie. De actie heeft als resultaat *uitvoerwaarden* die de eindtoestand vastleggen. De actie vindt plaats door uitvoering van een statement.

Zo geldt dat voor de actie, beschreven door de assignment statement

```
k := m*n + 3↑p
```

de waarden van de variabelen m, n en p en van de constante 3 de invoerwaarden zijn en dat de berekende uitvoerwaarde aan de variabele k wordt toegekend. We zijn er nu in geïnteresseerd hoe, voor andere acties dan de assignment actie, de transformatie plaatsvindt.
We zijn dus geïnteresseerd in het proces. De beschrijving van dit proces noemen we een *procedure*. De activering van de procedure vindt plaats door middel van de uitvoering van de *procedure statement*. Bij het gebruik van de statement zijn wij slechts geïnteresseerd in (het effect van) de actie. In een (procedure) statement moeten altijd de volgende drie componenten aanwezig zijn:

- de *identificatie* van de procedure; welke procedure wordt actief, dat wil zeggen: welke actie vindt plaats;
- de *invoerwaarden*; op welke operanden wordt door de procedure geopereerd, dat wil zeggen: wat is de begintoestand;
- de *uitvoervariabelen*; waar kunnen we na afloop van de actie de resultaten terugvinden, in welke variabelen.

Een 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. De invoerwaarden zijn waarden van expressies (constanten, variabelen).

De meest elementaire actie is de assignment, dit is de activering van de "assignment procedure". Deze activering vindt plaats door middel van de *assignment statement*. De assignment procedure wordt niet geïdentificeerd door een naam, alle andere procedures wel. Van de assignment procedure is de procesbeschrijving, de beschrijving van het "inwendige", in de processor aanwezig. Van alle andere procedures moeten we de beschrijving zelf maken en binnen onze procesbeschrijving opnemen. Deze beschrijving binnen de totale procesbeschrijving wordt de *declaratie* van de procedure genoemd.

5.2. Constructie en gebruik van procedures

Constructie en gebruik van procedures.
Zoals gezegd is in de processor de assignment procedure aanwezig en moeten alle andere procedures gedeclareerd worden. Na de declaratie kent de processor de naam en kan de procedure geactiveerd worden door de bijbehorende statement. Het kan zijn dat naast de assignment ook al andere procedures voor de processor bekend zijn, men spreekt dan van de *procedurebibliotheek*.
Bij het ontwerpen (en declareren) van een nieuwe procedure maken we gebruik van voor de processor reeds bekende procedures en de volgordebepalers. Bij de procedure spelen drie aspecten een rol:

(1) Het *effect* van de procedure: wat bewerkstelligt de procedure.
Diegene die een reeds bekende procedure wil gebruiken zal een goede beschrijving van het effect wensen om na te kunnen gaan of de procedure voor hem bruikbaar is. Deze beschrijving moet de relatie aangeven tussen de invoer en uitvoer.
Ook voor de constructeur is deze beschrijving van belang; aan de hand hiervan wordt de procedure geconstrueerd.

(2) De *invoerwaarden* en *uitvoervariabelen*: waarop werkt de procedure.
In de procedure (declaratie) worden deze grootheden gerepresenteerd door formele namen; we noemen deze de *formele parameters*.
Bij de activering van de procedure door middel van de procedure statement moeten actuele waarden en actuele uitvoervariabelen, samen de zogenaamde *actuele parameters*, bekend gemaakt worden.
De actuele parameters zijn bij uitvoering van de procedure de vervangers voor de formele namen die in de procedurebeschrijving gebruikt worden. De parameters leggen de relatie van de procedure met de omgeving vast.

(3) De beschrijving van het proces, dit is *het patroon van acties dat, uitgaande van de begintoestand (bepaald door de invoerwaarden) de eindtoestand (vastgelegd in de uitvoervariabelen) produceert*. Deze procesbeschrijving kent een eigen toestandsruimte en in de acties spelen de variabelen uit deze toestandsruimte (*lokale variabelen*) plus de formele parameters een rol. Deze eigenlijke procesbeschrijving noemt men wel *de body van de procedure*. Een procedure beschrijft een hele klasse van problemen. Wordt de procedure geactiveerd met andere invoerwaarden, dan wordt er (waarschijnlijk) een ander proces uitgevoerd. De body beschrijft dus een klasse van processen.

In de proceduredeclaratie moeten worden vastgelegd:
- de naam van de procedure;
- de formele namen voor de uitvoervariabelen en de invoerwaarden; bovendien moeten hiervan de typen worden opgegeven;
- de body van de procedure.

Voor de gebruiker van de procedure zijn alleen "het wat" (het effect) en "het waarop" (de parameters) belangrijk. De procedure kan door de gebruiker verder beschouwd worden als een "black box", die een bepaalde invoer transformeert tot een bepaalde uitvoer. Dit is analoog aan de assignment, waarbij we er ook niet in geïnteresseerd zijn hoe de waardetoekenning plaatsvindt. Van de constructeur van een procedure (kan dezelfde persoon zijn als de gebruiker) wordt gevraagd om bij een gegeven formulering van het gewenste eindeffect de bijbehorende body te construeren en de garantie, het bewijs, te leveren dat deze body ook inderdaad het gewenste effect bewerkstelligt.

Voorbeeld.

Procedure declaratie:

```
procedure kwadsom (s,m,n); value m,n; integer s,m,n;
   begin integer i;                          }
         s := 0;                             }
         for i := m step 1 until n           } procedure body
             do s := s + i↑2 od              }
   end                                       }
```

Procedure statement:

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

Formele parameters voor invoer : m en n
voor uitvoer : s
Actuele parameters voor invoer : 1 en 1000
voor uitvoer : p
Locale variabele in procedure body : i

De actie die aan g als waarde geeft de grootste gemene deler van x en y kunnen we bijvoorbeeld realiseren met behulp van de statement

{x en y geheel, > 0} GGD(g,x,y) {g = ggd(x,y)}

Dit kan alleen als de bijbehorende procedure al bekend is voor de processor of door onszelf gedeclareerd is in het begin van de procesbeschrijving. Deze declaratie kan er bijvoorbeeld uitzien als:

```
procedure GGD(k,a,b); value a,b; integer k,a,b;
   begin while a ≠ b
               do if a > b then a := a - b
                           else b := b - a
                  fi
               od;
         k := a
   end
```

Deze procedure heeft geen lokale variabelen. De formele namen (formele parameters) zijn k, a b, waarvan door value a, b wordt opgegeven dat a en b de invoerparameters zijn (waarbij het alleen om de waarde gaat). Activering van de procedure door middel van de uitvoering van de procedurestatement bestaat uit twee fasen:

(1) parameterinitialisering:
Dit houdt in dat bij het begin van de uitvoering van de body de parameters a en b de waarden krijgen van x respectievelijk y en dat k staat voor g.

(2) uitvoering van de body:
De body van de procedure wordt uitgevoerd onder de onder (1) genoemde intitiële omstandigheden.

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

```
a := x; y := b;
begin while a ≠ b
            do if a > b then a := a - b
                        else b := b - a
               fi
            od;
      g := a
end
```

Denk er wel om dat de actuele uitvoerparameters (in bovenstaand voorbeeld: g) letterlijk gesubstitueerd worden voor de formele uitvoerparameters (in bovenstaand voorbeeld: k) in de procedure body. Zo leidt de aanroep

```
arraysom (s,a[i],i,1,100)
```

van de procedure

```
procedure arraysom (as,p,j,o,b); value o,b; integer as,p,j,o,b;
   begin as := 0;
         for j := 0 step 1 until b do as := as + p od
   end
```

inderdaad tot het resultaat dat s gelijk is aan a[1] + a[2] + ... + a[100] (ga na!).

Stel dat de gevraagde actie is: Bepaal quotiënt en rest van de waarden van a (geheel) en b (geheel en ≠ 0) en leg de resultaten vast in q en r. De actie kunnen we weer opvatten als de activering van een procedure, bijvoorbeeld divmod genaamd. De actie noteren we dan als

```
divmod (q, r, a, b)
```

We gaan nu de procedure divmod creëren. In de body van de procedure noemen we de invoerparameters deeltal en deler, de uitvoerparameters quotënt en rest. Deeltal en deler krijgen bij activering door bovenstaande statement dus de waarden van a en b. Aan q en r worden waarden toegekend via quot en rest. Voor de constructie maken we gebruik van voorbeeld 1 (en de opmerking na voorbeeld 2) van hoofdstuk 3. De procedure-declaratie ziet er nu als volgt uit:

```
procedure divmod (quot, rest, deeltal, deler);
          value deeltal, deler; integer quot, rest, deeltal, deler;
   begin integer A, B, Q, R;
         if deeltal ≥ 0 then A := deeltal else A := - deeltal fi;
         if deler > 0 then B := deler else B := - deler fi;
         Q := A div B; R := A mod B;
         if deeltal * deler ≥ 0
            then quot := Q else quot := - Q
         fi;
         if deeltal ≥ 0 then rest := R else rest := - R fi
   end
```

In de body treden A, B, Q en R als lokale variabelen op (voorstellende de absolute waarden van respectievelijk quot, rest, deeltal en deler). In feite hadden we de operatoren mod en div direct kunnen toepassen omdat deze ook gedefinieerd zijn voor negatieve operanden.
Een procedure kunnen we opvatten als de beschrijving van een klasse van gelijksoortige processen. (We spraken ook van het patroon van acties.) Ieder proces is een speciaal geval van zo'n klasse, namelijk het geval bepaald door de actuele parameters. Voor de statements

```
divmod(x, y, - 27, 8)
```

en

```
divmod(p, q, 14, 5)
```

wordt wel hetzelfde patroon gevolgd, de processen zijn echter verschillend.
In het proces, behorende bij een procedure, spelen zowel parameters als de lokale variabelen een rol. Het relevante deel van de omgevende toestandsruimte is via de parameters aan het proces meegegeven en is dus in zekere zin intern gemaakt. De toestand van de omgeving van het proces speelt, buiten de parameters, verder geen rol meer. De procedure is dus afgeschermd van de omgeving. Dit houdt ook in dat een procedure in iedere omgeving gebruikt kan worden. Vandaar dat het mogelijk is om, via een procedurebibliotheek, het actierepertoire van de processor uit te breiden.

Een procedure voor het berekenen van de wortels van de vierkantsvergelijking

$$ax^2 + bx + c = 0$$

kan luiden:

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

Bij iedere procedure behoort een effectenbeschrijving. De procedure is alleen correct als hij voldoet aan deze beschrijving. In de beschrijving moeten eventuele beperkingen, die aan de invoerwaarden worden gesteld, zijn vastgelegd (in het voorbeeld: $a \neq 0$ en $b^2 \geq 4ac$) en moet ook de betekenis van de uitvoervariabelen zijn beschreven (in het voorbeeld: x1 en x2 bevatten na afloop de waarden van de wortels). Voor de volgende procedure behoeven aan de invoerwaarden geen eisen te worden gesteld. In de effectbeschrijving dient echter te worden opgenomen:

- als a = 0 dan zijn na afloop de uitvoerparameters x2, y1 en y2 ongedefinieerd (hebben geen waarde);
- als $a \neq 0$ dan zijn de wortels van de vierkantsvergelijking x1 + iy1 en x2 + iy2.

De procedure luidt:

```
procedure vkw2(x1, x2, y1, y2, a, b, c); value a, b, c;
          real x1, x2, y1, y2, a, b, c;
    begin real d;
          if a = 0
             then x1 := -c/b
             else d := b↑2 - 4 * a * c;
                  if d ≥ 0
                     then if b ≥ 0
                             then x1 := (-b - sqrt(d))/(2 * a)
                             else x1 := (-b + sqrt(d))/(2 * a)
                          fi;
                          x2 := c/(a * x1); y1 := 0; y2 := 0
                     else y1 := sqrt(-d)/(2 * a);
                          x1 := -b/(2 * a);
                          x2 := x1; y2 := -y1
                  fi
          fi
    end
```

Als b^2 erg veel groter is dan 4ac, dan geldt $d \approx b^2$; dit houdt in dat in x1(x2) een behoorlijke fout (reals!) kan optreden. Daarom is in de procedure een andere methode voor de berekening van x2 (voor het geval van reële wortels) toegepast.

Procedures zijn zeer geschikt om deelproblemen van een totaal probleem op te lossen. In eerste instantie noteren we de oplossing van het deelprobleem als procedure-statement. Later werken we deze oplossing uit in de vorm van de declaratie van de bijbehorende procedure. Stel bijvoorbeeld dat we als deelprobleem hebben dat bij een gegeven n de n! berekend moet worden (en evenzo bij m de m!), omdat $\binom{n}{m}$ berekend moet worden. We krijgen dan als programmadeel:

```
n := .....;
fac(nfac, n);
m := .....;
fac(mfac, m);
fac(minmfac, n-m);
nbovenm := nfac/( mfac * nminfac)
```

We zijn er hierbij van uit gegegaan dat het mechanisme voor de faculteitberekening aanwezig is. Als dit niet het geval is, maken we zelf de constructie hiervoor en leggen die vast in een declaratie:

```
procedure fac(kf, k); value k; integer kf, k;
    begin integer i;
          i := 0; kf := 1; {kf = 1 * 2 * ... * i}
          while i ≠ k
             do i := i + 1; kf := kf * i od
    end
```

Bij de effectbeschrijving van deze procedure dient vermeld te worden dat moet gelden: $k \geq 0$.
Het probleem van de berekening van $\binom{n}{m}$ kan uiteraard anders (en beter?) opgelost worden. Geef zelf een andere oplossing.

5.3. Het gebruik van (binnen) blokken

In plaats van het declareren van de procedure, kunnen we ook de procedure-statement vervangen door de uitwerking zelf.
Stel dat we een tabel van de volgende vorm moeten genereren.

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

In de invoerrij is gegeven een waarde die aangeeft uit hoeveel regels de tabel moet bestaan.
Als de opdracht nr bij uitvoering leidt tot de overgang op een nieuwe regel (bij koppeling van de uitvoerrij f aan een regeldrukker), wordt het programma in eerste instantie

```
begin integer i, aantal;
      read(g, aantal);
      i := 0; {i regels gegenereerd}
      while i ≠ aantal
         do i := i + 1;
            genereer(i); {voor het genereren van de i-de regel}
            nr
         od
end
```

We kunnen het genereren van de i-de regel uitwerken op (in) de plaats van de procedure statement:

```
begin integer i, aantal;
      read(g, aantal);
      i := 0;
      while i ≠ aantal
         do i := i + 1;
            begin integer j;
                  j := 0; {aantal uitgevoerde getallen}
                  while j ≠ i
                     do j := j + 1;
                        write(f, j * i)
                     od
            end;
            nr
         od
end
```

We hebben hier voor de oplossing van het deelprobleem gebruik gemaakt van een zogenaamd *binnenblok*. Een binnenblok dient om - voor de oplossing van een deelprobleem - een lokale toestandsruimte te creëren.

In het binnenblok bestaan zowel de variabelen van deze lokale toestandsruimte als de variabelen van de omvattende toestandsruimte (van het programma). Een binnenblok is toegestaan op alle plaatsen waar een statement is toegestaan.
Een blok bestaat dus uit een aantal declaraties en een aantal statements. Een blok heeft een locale toestandsruimte.

Opmerkingen.

(1) We zouden in het bovenstaande programma de variabele j ook in de toestandsruimte van het hele programma hebben kunnen opnemen. De variabele j is echter alleen nodig in de oplossing van het deelprobleem.

(2) Op het moment van declaratie van

real array x[1:k]

moet k een waarde hebben omdat geëist wordt dat bij de declaratie de waarden, die de indices kunnen aannemen, bekend zijn. Stel nu dat in de invoerrij een aantal getallen staan, waarvan het eerste getal aangeeft hoeveel getallen er daarna nog volgen. Gevraagd wordt de getallen, afgezien van het eerste, in een array te plaatsen. Omdat in het programma de declaraties voorafgaan aan de statements komen we dan in de hierboven genoemde moeilijkheid, die we echter kunnen oplossen door gebruik te maken van een binnenblok:

```
begin integer k;
      read(f, k);
      begin real array x[1:k]; integer i;
            for i := 1 step 1 until k do read(f, x[i]) od;
            .....
      end
end
```

(Einde opmerkingen).

De enige relaties die een procedure heeft met zijn omgeving verlopen via de parameters. De invoerparameters ontlenen hun waarde aan de omgeving. Het resultaat van de procedure wordt aan de omgeving meegegeven via de uitvoervariabelen. Afgezien van deze actuele uitvoervariabelen verandert de toestandsruimte van de omgeving dus niet.

Bij een binnenblok moet de programmeur zelf de discipline opbrengen om alleen die variabelen uit een omgevend blok te veranderen die optreden als een soort uitvoervariabelen. In het voorbeeld van de vermenigvuldigingstabel moet in het binnenblok i niet veranderd worden. Zouden we voor het deelprobleem van het genereren van een regel een procedure hebben gebruikt, dan zouden in de body van deze procedure een lokale variabele (overeenkomend met de j uit het voorbeeld) en een formele invoerparameter (die bij aanroep van de procedure staat voor i) voorkomen en we zouden de omgeving niet kunnen veranderen. Het procedure-mechanisme levert dus een automatische bescherming die bij een binnenblok door de programmeur aangebracht moet worden. Dit is hèt voordeel van het gebruik van procedures.

5.4. Functies

Het komt nogal eens voor dat een procedure één waarde berekent die aan de uitvoervariabele wordt toegekend. Er bestaat voor deze soort procedure een aparte vorm, de *functie*, en de activering van de functie kan direct in een expressie worden opgenomen. De procedure voor het berekenen van de faculteit kunnen we ook als functie schrijven:

```
integer procedure fac(k); value k; integer k;
 begin integer i, h;
       i := 0; h := 1;
       while i ≠ k
          do i := i + 1; h := h * i od;
      fac := h
 end
```

Een functie levert een waarde af, het type van deze waarde moet opgegeven worden. Het berekende resultaat wordt in de functie toegekend aan de naam van de functie.

De functie kunnen we in een expressie activeren. Voor het probleempje van $\binom{n}{m}$ krijgen we dan:

```
nbovenm := fac(n)/(fac(m) * fac(n-m))
```

Bij de behandeling van de typen hebben we al een aantal functies gezien.

Als tweede voorbeeld voor het gebruik van functies kijken we naar het volgende probleempje. In de invoerrij f zijn twee positieve gehele getallen gegeven, die niet gelijk zijn. Gevraagd wordt alle perfecte getallen te bepalen tussen de twee gegeven getallen. Een getal is perfect als het getal gelijk is aan de som van zijn delers (1 inclusief). Zo is 6 perfect, want 6 = 1 + 2 + 3.
Oplossing:

```
read(f,n); read (f,m);
for i := n step 1 until m
    do if perfect (i) then write (g,i) fi od
```

De declaratie van de procedure perfect luidt:

```
boolean procedure perfect (j); value j; integer j;
    begin integer k, som;
          som := 1;
          for k := 2 step 1 until j div 2
              do if j mod k = 0
                    then som := som + k
                 fi
              od
          perfect := som = j
    end
```

5.5. Voorbeelden gebruik procedures

Voorbeeld 1.
Stel dat het array

integer array A[1:100]

een waarde heeft.
Gevraagd wordt om door herrangschikking van de waarden van de componenten te bereiken dat

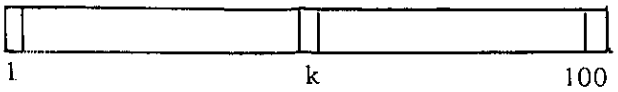
$\forall i : 1 \leq i \leq 100 \ (A[i] \leq A[i+1])$

(De rij moet niet-dalend gesorteerd worden.)
Er staat bij dat het gevraagde gerealiseerd moet worden door herrangschikking om aan te geven dat het na afloop om dezelfde waarden gaat als in het begin (anders zou bijvoorbeeld

```
for i := 1 step 1 until 100 do A[i] := i od
```

een oplossing zijn).

We introduceren een variabele k



met de betekenis:

1. A[i] ≤ A[i+1] voor i = 1, 2, 3, ..., k-1
2. De elementen A[i] (voor i = 1, 2, ..., k) hebben hun uiteindelijke waarde.

We kunnen deze betekenis van k in het begin waarmaken door te nemen k = 0. Het proces kan beëindigd worden als k = 100. We krijgen dan als stukje programma:

```
k := 0;
while k ≠ 100
   do k := k + 1;
      "betekenis van k herstellen"
   od
```

De betekenis van k kan hersteld worden door van A[k], A[k+1], ..., A[100] het minimum te zoeken en deze waarde te verwisselen met A[k]. We krijgen dus

```
k := 0;
while k ≠ 100
   do k := k + 1;
      minimum(A, k, n, index);
      wissel(A[k], A[index])
   od
```

De proceduredeclaraties zijn (we hebben deze probleempjes al vaker opgelost, vandaar dat geen afleiding wordt gegeven):

```
procedures minimum(x, on, bo, i); value on, bo;
           integer array x; integer on, bo, i;
     begin integer min, j;
           min := x[on]; j := on; i := on;
           while j ≠ bo
              do j := j + 1;
                 if x[j] < min then min := x[j];
                                    i := j
                 fi
              od
     end;

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

Let er op dat van de formele parameter x van de procedure minimum niet het index-bereik wordt opgegeven. Dit wordt bepaald door de actuele parameter (in dit geval A).
Voor het bepalen van het minimum zou ook een functie in plaats van een procedure gebruikt kunnen worden.
We kunnen het zoeken van het minimum en het verwisselen ook direct uitwerken. We krijgen dan (als we zonder binnenblok werken):

```
k := 0;
while k ≠ 100
   do k := k + 1;
      min := A[k]; index := k; j := k;
      while j ≠ 100
         do j := j + 1;
            if A[j] < min then min := A[j];
                               index := j
            fi
         od;
      A[index] := A[k]; A[k] := min
   od
```

Voorbeeld 2.

Een palindroom is een rij karakters die van links naar rechts en van rechts naar links gelezen dezelfde is; zo is "lepel" een palindroom. We bekijken palindromen bestaande uit cijfers, zoals "1" en "121", waarvan de getalwaarde een kwadraat is.

Gevraagd wordt bij gegeven waarde van n (n > 0) in de invoerrij f aan te geven van welke getallen tussen 1 en n het kwadraat een palindroom is.

Een eerste (informele) versie van het stukje programma kan luiden:

```
read(f,n);
k := 0;
while k ≠ n
   do k := k + 1;
      kw := k * k;
      "bepaal cijfers van kw";
      p := "kw is palindroom";
      if p then write(g, k) fi
   od
```

Hierin geeft k aan van welke getallen reeds nagegaan is of het kwadraat een palindroom is.

Het vastleggen van de cijfers van kw kan in het

```
integer array r[1:m]
```

zodanig dat $kw = \sum_{p=1}^{m} r[p] * 10^{p-1}$. De representatie van kw is dus r[m]r[m-1]...r[2]r[1]. De waarde van m kan bepaald worden uit de grootte van kw:

```
hulp := kw; m := 0;
while hulp ≠ 0
   do hulp := hulp div 10; m := m + 1 od
```

Het programma wordt dan:

```
read(f, n);
k := 0;
while k ≠ n
   do k := k + 1;
      kw := k * k;
      hulp := kw; m := 0;
      while hulp ≠ 0
         do hulp := hulp div 10; m := m + 1 od;
      begin integer array r[1:m];
            representatie(r, m, kw);
            p := palindroom(r, m)
      end;
      if p then write(g, k) fi
   od
```

De procedure representatie en de functie palindroom moeten nog uitgewerkt en gedeclareerd worden.

De procedure representatie bepaalt, aan de hand van de invoerwaarde g (een kwadraat), de waarde van dec[1:bg], zodanig dat

$$g = \sum_{p=1}^{bg} dec[p] * 10^{p-1} \text{ met } 0 \leq dec[p] \leq 9 \text{ voor } p = 1, 2, \ldots, bg$$

Voor de repetitie geldt steeds (invariant) dat

$$g * 10^{p} + \sum_{k=1}^{p} dec[k] * 10^{k-1}$$

gelijk is aan de oorspronkelijke waarde van g.

```
procedure representatie (dec, bg, g); value bg, g;
        integer array dec; integer bg, g;
   begin integer p;
        p := 0; {p is index van laatst gevulde plaats van dec}
        while p ≠ bg
           do p := p + 1;
              dec[p] := g mod 10;
              g := g div 10
           cd
    end
```

Denk er wel om dat het parametermechanisme zodanig werkt dat een invoerwaarde niet verandert. Na afloop van de aanroep representatie(r, m, kw) heeft kw dus dezelfde waarde als daarvoor.
Bij beëindiging van de repititie geldt p = bg, maar ook g = 0, en dus hebben we dan inderdaad de representatie gevonden (zie invariant).

```
boolean procedure palindroom(dec, bg); value bg;
                  integer array dec; integer bg;
   begin integer i, j; boolean p;
        p := true; i := 1; j := bg;
        while i < j ∧ p do  p := (dec[i] = dec[j]); i := i + 1;
                            j := j - 1
                        od;
        palindroom := p
   end
```

Voor i en j geldt steeds dat d[i] en d[j] de volgende te onderzoeken elementen zijn en p geeft aan dat dec[1] = dec[bg], dec[2] = dec[bg-1], ..., dec[i-1] = dec[j+1].
Bij beëindiging geldt $i \geq j \vee \neg p$. Als $i \geq j$ dan zijn alle elementen vergeleken (als $i = j$ blijft alleen dec[i] over, dus de waarde van p bepaalt of we te maken hebben met een palindroom).

5.6. Recursie

De body van een procedure (functie) bevat activeringen van bestaande procedures (functies); in veel gevallen zal dit de assignment zijn. 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 de beschrijving van de functie "cijfersom", die de som bepaalt van de cijfers van de decimale representatie van een positief geheel getal. Deze functie ziet er als volgt uit:

```
integer procedure cijfersom (arg); value arg; integer arg;
        begin integer kop, staart;
              if arg < 10 then cijfersom := arg
                          else divmod(kop, staart, arg, 10);
                               cijfersom := cijfersom(kop) + staart
              fi
        end
```

Deze functiebeschrijving berust op de volgende overwegingen.
Wat betreft de representatie van een getal met behulp van cijfers kunnen we een positief geheel getal definiëren als ofwel "een cijfer" of "een geheel getal gevolgd door een cijfer", anders gezegd: "een staart" ofwel "een kop gevolgd door een staart".
De definitie van cijfersom volgt hier direct uit: als het getal uit één cijfer bestaat dan is de cijfersom het getal zelf, anders is cijfersom gelijk aan de som van de "cijfersom van de kop" en de staart.
Deze definitie van cijfersom vinden we letterlijk terug in de functiebeschrijving. Voor de splitsing van een getal in kop en staart zorgt de procedure divmod.
We hebben de definitie van cijfersom recursief gegeven en op grond hiervan is er ook een recursieve functie gemaakt. We kunnen echter eenvoudig een definitie geven op grond waarvan we in de body van de functie voor cijfersom het resultaat bepalen met behulp van een repetitie. Geef zelf de body.

In de bovenstaande functiedeclaratie staat de statement

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

cijfersom in het linkerlid staat daar op grond van de afspraak dat de berekende waarde wordt toegekend aan de functienaam; cijfersom in het rechterlid is de recursieve aanroep van de functie en heeft dan ook een (actuele) parameter.

n! kunnen we op twee manieren definiëren:

$$- n! = \begin{cases} 1 \text{ voor } n = 0 \\ 1 * 2 * 3 * \ldots * n \text{ voor } n \geq 1 \end{cases}$$

$$- n! = \begin{cases} 1 \text{ voor } n = 0 \\ n * (n-1)! \text{ voor } n \geq 1 \end{cases}$$

De eerste definitie leidt tot de reeds eerder gegeven niet-recursieve functie; de tweede definitie leidt tot:

```
integer procedure fac2(n); value n; integer n;
        if n = 0 then fac2 := 1
                 else fac2 := n * fac2(n-1)
        fi
```

Een partitie van een positief geheel getal krijgen we door m te schrijven als de som van een aantal (>0) positieve gehele getallen (termen), waarbij de volgorde van deze termen niet van belang is. De partities van 6 zijn:

```
1 + 1 + 1 + 1 + 1 + 1
2 + 1 + 1 + 1 + 1
2 + 2 + 1 + 1
2 + 2 + 2
3 + 1 + 1 + 1
3 + 2 + 1
```

3 + 3
4 + 1 + 1
4 + 2
5 + 1
6

Voor 6 zijn er dus 11 partities. Stel dat Q(m, n) het aantal partities is van een getal m, waarbij alle termen kleiner of gelijk n zijn (het aantal partities voor m is dan Q(m, m)). Voor Q gelden de volgende eigenschappen:

(1) Q(m, 1) = 1
Voor iedere m geldt dat er slechts één partitie is die bestaat uit de som van een aantal enen.

(2) Q(1, n) = 1
Er is slechts één partitie van het getal 1, ongeacht de waarde van n.

(3) Q(m, n) = Q(m, m) als m < n
Er kunnen geen partities van m zijn die een grotere term in de sommatie hebben dan m.

(4) Q(m, m) = 1 + Q(m, m-1)
Er is één partitie van m waarin m zelf optreedt; alle andere partities van m hebben termen die ten hoogste m - 1 zijn.

(5) Q(m, n) = Q(m, n-1) + Q(m-n, n)
De partities van m (met n als grootste term) kunnen we in twee groepen verdelen. De eerste groep bestaat uit de partities waarin n niet voorkomt (aantal hiervan is Q(m, n-1)).
De tweede groep bestaat uit alle partities waarin n wel voorkomt. Als we in deze partities steeds één keer n weglaten, dan houden we alle partities van m - n over waarvan de grootste term n is (aantal is Q(m-n, n)).

Deze eigenschappen definiëren Q(m, n). De functiedeclaratie luidt dan ook:

```
integer procedure Q(m, n); value m, n; integer m, n;
        if m = 1 ∨ n = 1
           then Q := 1
           else if m ≤ n then Q := 1 + Q(m, m-1)
                         else Q := Q(m, n-1) + Q(m-n, n)
                fi
        fi
```

De eis voor de invoerparameters is: m ≥ 1 en n ≥ 1.

Recursie lijkt moeilijk doordat men zich het proces tracht voor te stellen dat het gevolg is van uitvoering van een recursief algoritme. Steeds zal echter aan een recursief algoritme een redenering ten grondslag liggen die op de een of andere wijze gebaseerd is op volledige inductie. Zo wordt het algoritme geconstrueerd en zo moeten we het ook trachten te begrijpen. We moeten wel nagaan of de aanroepen toegestaan zijn en of het algoritme eindigt.

6. Grotere voorbeelden.

6.1. Een loket probleem.

Van een loket wordt gebruik gemaakt door klanten die genummerd zijn van 1 tot en met n. De klanten komen in volgorde van hun nummer bij het loket aan en worden ook in deze volgorde van aankomst bediend zodra het loket vrij is.
Aankomsttijd en bedieningsduur van klant i worden gegeven door a_i (geheel en $0 < a_1 < a_2 < \ldots < a_n$) en b_i (geheel en > 0), waarvoor geldt dat deze zodanige waarden hebben dat geen enkele aankomsttijd samenvalt met een vertrektijd.
Het loket gaat open op tijdstip 0 en sluit na vertrek van klant n. Geef een programma dat het sluitingstijdstip bepaalt en de lengte van de langste rij die voor het loket gestaan heeft. De waarden van n en van de a_i en b_i staan in de invoerrij f ($n, a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_n$.
De invoerwaarden moeten intern gemaakt worden. We krijgen dan als eerste aanzet:

```
begin integer n;
      read(f, n);
      begin integer array a, b[1:n];
            integer i;
            for i := 1 step 1 until n do read(f, a[i]) od;
            for i := 1 step 1 until n do read(f, b[i]) od;
            "oplossing probleem"
      end
end
```

We zullen ons nu verder concentreren op het eigenlijke probleem.
De vraag naar het sluitingstijdstip leidt tot de gedachte om te letten op het vertrektijdstip van de klanten, want voor het vertrektijdstip van klant n is het sluitingstijdstip. Voor klant 1 is het vertrektijdstip $v_i = a_1 + b_1$, doch iedere volgende klant hangt het er van af of hij arriveert als de rij leeg is (vertrektijdstip is dan $v_i = a_i + b_i$), dan wel als er nog mensen voor het loket staan (vertrektijdstip is dan $v_i = v_{i-1} + b_i$).

Er geldt dus

$$v_i = \max(a_i, v_{i-1}) + b_i \text{ voor } 1 \leq i \leq n$$

met $v_o = 0$.

Voor de klanten kunnen nu de vertrektijdstippen berekend worden. Blijft over de lengte van de langste rij te berekenen. Als klant i vertrekt wordt de lengte van de rij 1 kleiner. Na het vertrektijdstip v_{i-1} kan de rij echter langer zijn geworden. In de rij zijn die klanten j bijgekomen waarvoor geldt

$$v_{i-1} < a_j < v_i \quad (\text{voor } j \leq n).$$

Om de toestanden vast te leggen gebruiken we de volgende variabelen:

i : $0 \leq i \leq n$

v : vertrektijdstip van klant i

r : lengte van de rij na vertrek van klant i

max: maximale rijlengte die opgetreden is tot op het moment v

j : nummer van de laatste klant waarvoor geldt dat zijn aankomsttijd ligt voor de vertrektijd van klant i.

We krijgen dan:

```
i := 0; v := 0; r := 0; max := 0; j := 0;
while i ≠ n
   do i := i + 1; {eindigheid van de repetitie}
      if a[i] > v then v := a[i] + b[i]
                  else v := v + b[i]
      fi;
      while "klant binnegekomen voor v"
         do j := j + 1; r := r + 1 od;
      if r > max then max := r fi;
      r := r - 1
   od
```

Voor de klanten, die na het vorige vertrektijdstip in de rij zijn komen staan, geldt dat hun nummer groter is dan het nummer van de laatste klant die voor dat vorige vertrektijdstip in de rij is komen staan. Bovendien moet hun aankomsttijd kleiner zijn dan v. En tenslotte moet het nummer van de laatst binnengekomen klant voor het vorige vertrektijdstip kleiner zijn dan n. We zouden dus kunnen schrijven:

```
while j < n  ∧  a[j+1] < v
   do j := j + 1; r := r + 1 od
```

Als echter alle klanten binnen zijn, j = n, zal in de boolean expressie gerefereerd worden aan het niet bestaande element a[n+1]. We kunnen dit voorkomen door gebruik te maken van een hulpvariabele nogj:

```
if j < n then nogj := a[j+1] < v;
              while nogj
                 do j := j + 1; r := r + 1;
                    if j < n then nogj := a[j+1] < v
                             else nogj := false
                    fi
                 od
fi
```

Nu het aanpassen van j en r onder de voorwaarde j < n is gebracht, kunnen we de eventuele aanpassing van max ook onder deze voorwaarde brengen daar de aanpassing alleen nodig is als r veranderd is.
We krijgen dus als algoritme:

```
i := 0; v := 0; r := 0; max := 0; j := 0;
while i ≠ n
   do i := i + 1;
      if a[i] > v then v := a[i] + b[i]
                  else v := v + b[i]
      fi;
      if j < n then nogj := a[j+1] < v;
                    while nogj
                       do j := j + 1; r := r + 1;
                          if j < n then nogj := a[j+1] < v
                                   else nogj := false
                          fi
                       od;
                    if r > max then max := r fi
      fi;
      r := r - 1
   od
```

6.2 De mediaan

Gevraagd wordt van een rij gehele getallen $a_1, a_2, \ldots, a_n$ de mediaan te bepalen. De mediaan is ruwweg dat getal x uit de rij waarvoor het aantal getallen uit de rij die niet groter zijn dan x ongeveer gelijk is aan het aantal getallen uit de rij die niet kleiner zijn dan x.

Een betere definitie voor de mediaan dan de bovenstaande is:

> De mediaan x is het getal a'_m waarbij geldt dat $m = (n + 1)$ div 2 en dat de rij $a'_1, a'_2, \ldots a'_n$ een permutatie is van de rij $a_1, a_2, \ldots, a_n$ en wel zodanig dat $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

Deze definitie van de mediaan suggereert al direct een oplossing. We kunnen namelijk de rij $a_1, a_2, \ldots, a_n$ sorteren en dan het m-de element kiezen. Als we sorteren, doen we echter te veel. We kunnen dan ook een "zwakkere" definitie geven:

> De mediaan x is het getal a'_m waarbij geldt dat $m = (n + 1)$ div 2 en dat de rij $a'_1, a'_2, \ldots, a'_n$ een zodanige permutatie is van de rij $a_1, a_2, \ldots, a_n$ dat $a'_i \leq a'_m$ voor $1 \leq i < m$ en $a'_i \geq a'_m$ voor $m < i \leq n$.

Het blijkt nu dat de deelrijen $a'_1, a'_2, \ldots, a'_{m-1}$ en $a'_{m+1}, \ldots, a'_n$ niet gesorteerd behoeven te zijn en dat kan in het algoritme besparend werken ten opzichte van een algoritme dat op de eerste definitie is gebaseerd.

Uit bovenstaande blijkt dat we de rij $a'_1, a'_2, \ldots, a'_n$ in drie delen gesplitst kunnen denken, namelijk een deelrij waarvan de elementen kleiner zijn dan a'_m, een deelrij waarvan de elementen gelijk zijn aan a'_m en een deelrij waarvan de elementen groter zijn dan a'_m.

Hoe bepalen we de a'_m (= x)? We zullen, zoals steeds, proberen het aantal getallen waaruit gekozen moet worden stapje voor stapje kleiner te maken. We kunnen dit doen (suggestie uit bovenstaande eigenschap van de rij $a'_1, a'_2, \ldots, a'_n$) door de rij in drie delen te verdelen, waarbij we kunnen bepalen in welk van de drie delen te verdelen, waarbij we kunnen bepalen in welk van de drie delen x moet liggen.

We gaan nu naar het algoritme kijken, waarbij gegeven is dat het array

    integer array a[1:n]

een waarde heeft.

Uit het bovenstaande leiden we de volgende invariant af:

    P: (1 ≤ p ≤ q ≤ n) ∧ (p ≤ m ≤ q) ∧ de elementen in
       a[1:p-1] zijn kleiner en de elementen in a[q+1:n] zijn
       groter dan de elementen in a[p:q].

Uit de invariant blijkt dat de mediaan x in a[p:q] moet liggen.

De eerste opzet van het programma wordt:

    "initialiseer p en q zodanig dat P geldt";
    while p ≠ q
       do "verklein q - p onder invariantie van P" od

De initialisatie voor p en q (bij m = (n + 1) div 2) kan zijn:

    p := 1; q := n

De verkleining van q - p bereiken we met behulp van de eerder reeds genoemde driedeling. Daarbij gebruiken we een algoritme uit 6.2. (Nederlandse Vlag).

Laat s een willekeurig getal uit de rij a_p, a_{p+1}, ... a_q zijn.

We kunnen nu de getallen a_p, a_{p+1}, ..., a_q zodanig rangschikken dat er een i en j bestaan waarvoor geldt

a[p:i-1] zijn kleiner dan s
a[i:j] zijn gelijk aan s
a[j+1:q] zijn groter dan s

Een programma voor zo'n soort driedeling hebben we al in 6.2. gemaakt. Als de deze driedeling bereikt hebben, kunnen zich drie gevallen voordoen ten aanzien van m:

- m < i(i > (n + 1) div 2):
  In dit geval geldt dat s > x en voor het herstellen van de invariant moeten we q de waarde i - 1 geven (ga na.).

- $i \leq m \leq j$:

  In dit geval geldt s = x en de mediaan is gevonden. We zullen in dit geval zowel p en q de waarde m geven (zodat er direct gestopt wordt).

- $m > j$:

  In dit geval geldt dat s < x en voor het herstellen van de invariant moeten we p de waarde j + 1 geven (ga na.).

Voor s kiezen we (we weten niets beters): a[p].

De verkleining van p - q wordt dus als volgt bereikt:

```
if m < i then q := i - 1
         else
if m ≤ j then p := m; q := m
         else p := j + 1
fi
fi
```

Alle ingrediënten samengevoegd wordt het algoritme (zie volgende bladzijde):

```
m := (n + 1) div 2;
p := 1; q := n;
while p ≠ q
   do s := a[p];
      i := p; j := q; w := q;
      while i ≠ w + 1
         do h := a[w];
            if h = s then w := w - 1
                     else
            if h < s then x := a[i]; a[i] := a[w]; a[w] := x;
                          i := i + 1
                     else x := a[j]; a[j] := a[w]; a[w] := x;
                          j := j - 1; w := w - 1
            fi
            fi
         od;
      if m < i then q := i - 1
               else

      if m ≤ j then p := m; q := m
               else p := j + 1
      fi
      fi
   od;
x := a[m]
```

Het bovenstaande algoritme is door C.A.R. Hoare gepubliceerd en staat bekend onder de naam FIND.

6.3. De eerste duizend priemgetallen

Een priemgetal is een natuurlijk getal dat alleen door 1 en door zichzelf deelbaar is. We stellen dat 2 het kleinste priemgetal is en gaan er vanuit dat er ten minste duizend priemgetallen bestaan. Gevraagd wordt een programma dat de eerste duizend priemgetallen genereert.

De priemgetallen kunnen we laten opbergen in een array.

We gebruiken de variabele k die het aantal reeds gevonden priemgetallen aangeeft en j die het laatst gevonden priemgetal aangeeft; j wordt op 1 gezet omdat we alleen oneven waarden door ophoging met 2 voor j zullen proberen. We krijgen dan:

```
begin integer array pr[1:1000];
      integer k, j;
      pr[1] := 2; j := 1; k := 1;
      while k ≠ 1000
         do "verhoog j tot het volgende priemgetal";
            k := k + 1; pr[k] := j
         od;
      for k := 1 step 1 until 1000 do write(pr[k]) od
end
```

Het algoritme is eindig omdat in iedere slag van de repetitie een nieuw priemgetal wordt gevonden en dus k met 1 wordt opgehoogd.

Het verhogen van j tot het volgende priemgetal kan in eerste instantie door

```
j := j + 2;
while "j niet priem" do j := j + 2 od
```

Om na te gaan of j priem is, moet er nagegaan worden of j een deler heeft. De kleinste deler die geprobeerd moet worden is pr[2]. De grootste deler die geprobeerd moet worden is pr[m-1] als pr[m] het kleinste priemgetal is waarvoor geldt pr[m]↑2 > j. Als namelijk een priemgetal, dat groter of gelijk pr[m] is, deelbaar is op j, dan is er ook een priemgetal kleiner dan pr[m] deelbaar op j. Uitwerking van het bovenstaande levert dan:

```
j := j + 2; "geef m een waarde";
n := 2; jpriem := true;
while n < m ∧ jpriem
   do jpriem := j mod pr[n] ≠ 0; n := n + 1 od;
while ¬jpriem
   do j := j + 2;
      "geef m een waarde";
      n := 2; jpriem := true;
      while n < m ∧ jpriem
         do jpriem := j mod pr[n] ≠ 0; n := n + 1 od
   od
```

In deze uitwerking zijn we er van uitgegaan dat de waarde van m bekend is en steeds wordt aangepast (bij een nieuwe waarde van j hoort eventueel een nieuwe waarde voor m). Steeds als j een (nieuwe) waarde krijgt, kan m bepaald worden door:

```
m := 1; while pr[m]↑2 ≤ j do m := m + 1 od
```

In feite kunnen we met de waarde m = 1 beginnen bij j = 1 en m aanpassen als j een nieuwe waarde krijgt, omdat m een niet-dalende functie van j is.

Het totale programma wordt nu:

```
begin integer array pr[1:1000];
      integer k, j, m, n; boolean jpriem;
      pr[1] := 2; k := 1; j := 1; m := 1;
      while k ≠ 1000
         do j := j + 2; while pr[m]↑2 ≤ j do m := m + 1 od;
            n := 2; jpriem := true;
            while n < m ∧ jpriem
               do jpriem := j mod pr[n] ≠ 0; n := n + 1 od;
                  while ¬jpriem
                     do j := j + 2; while pr[m]↑2 ≤ j do m := m + 1 od
                        n := 2; jpriem := true;
                        while n < m ∧ jpriem
                           do jpriem := j mod pr[n] ≠ 0; n := n + 1 od
                     od;
            k := k + 1; pr[k] := j
         od;
      for k := 1 step 1 until 1000 do write (pr[k]) od
end
```

6.4. Worteltrekken

Gevraagd wordt een programma te maken dat voor een gegeven niet-negatief geheel getal a de naar beneden afgeronde vierkantswortel b bepaalt, i.e. entier($\sqrt{a}$). (Er geldt dus dat b geheel is.)

De eindrelatie voor het programma luidt:

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

Laten we eens als invariant proberen:

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

Deze P volgt uit R door één operand in de conjunctie weg te laten. Hieruit volgt voor de variant: $(b+1)^2 \leq a$.

In elk geval is deze relatie P initieel eenvoudig waar te maken, namelijk door b := 0, want dan staat er: $a \geq 0$, en dat is gegeven.

Het eindresultaat R volgt uit P indien bovendien nog zou gelden $a < (b+1)^2$. We moeten b ophogen om vanuit P ooit op R terecht te komen.

We krijgen dus als programma:

```
b := 0;
while (b + 1)↑2 ≤ a
   do {P ∧ (b + 1)^2 ≤ a} "verhoog b" {Q} od
```

De preconditie voor "verhoog b" luidt:

$$b^2 \leq a \wedge b \geq 0 \wedge (b+1)^2 \leq a$$

Dit is equivalent met

$$b \geq 0 \wedge (b+1)^2 \leq a$$

Neem nu voor "verhoog b": b := b + n (met $n \geq 1$).

Dan geldt voor Q (zie hoofdstuk 2):

$$b - n \geq 0 \wedge (b - n + 1)^2 \leq a$$

Er moet gelden $Q \Rightarrow P$; dit kan alleen als n = 1 (en dan geldt $P \equiv Q$).

Hiermee komen we tot het programma:

```
b := 0;
while (b + 1)↑2 ≤ a do b := b + 1 od
```

Het ontworpen programma is inefficiënt: voor bijvoorbeeld $a = 10^6$ vergt het liefst 10^3 repetities. Deze inefficiëntie is het gevolg van de keuze van de invariant P, die zoals we gezien hebben, geen grotere ophogingen van b toelaat dan een ophoging met 1.
We zullen daarom proberen een andere invariant te vinden. Een andere manier om uit een eindrelatie een invariant te houden is door vervanging van een constante door een variabele. We vervangen b + 1 in R door c (variabele + constante vervangen door variabele); dit levert als invariant:

P2: $b^2 \leq a \wedge a < c^2 \wedge c > b \geq 0$

Deze is initieel gemakkelijk waar te maken, namelijk door b := 0; c := a + 1, aangezien voor elke $a \geq 0$ geldt dat $(a + 1)^2 > a$.
Als we een eindrelatie afzwakken tot een invariante relatie door een constante te vervangen door een variabele, is de variante relatie constante = variabele. Dat wordt hier dus: $b + 1 \neq c$. De variante functie zal dan een verkleining van het verschil tussen c en b moeten opleveren (c > b).
Hiermee komen we dan tot een programma van de vorm:

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

De verkleining van c - b kan geschieden door een verlaging van c en/of een verhoging van b. Dat beide veranderingen moeten optreden volgt uit de initialisatie en de invariant; b moet de gevraagde waarde krijgen en is in het begin 0, c hebben we groot gekozen en deze moet kleiner worden; er geldt c > a, maar als a > 1 zal altijd b < a blijven. Per slag van de repetitie zullen we echter of c of b veranderen. Hoe groot de vermindering van c - b mag zijn is iets moeilijker te vinden dan de verhoging van b in het eerste programma. De vermindering van het verschil noemen we d.
We gaan uit van:

```
b := 0; c := a + 1;
while b + 1 ≠ c
   do d := .....; {d > 0}
      if .....then b := b + d
              else c := c - d
      fi {P2}
   od
```

Om de beginrelatie te vinden voor de selectie, zoeken we {A1} en {A2} uit:

{A1} b = b + d {P2}

{A2} c := c - d {P2}

Voor {A1} vinden we:

$\{(b + d)^2 \leq a \wedge a < c^2 \wedge c > (b + d) \geq 0\}$

En voor {A2}:

$\{b^2 \leq a \wedge a < (c - d)^2 \wedge c - d > b \geq 0\}$

De termen $a < c^2$ en $b^2 \leq a$ gelden op grond van het feit dat P2 geldt (misschien meer) na d := ..... De laatste termen kunnen we schrijven als $d < c - b$ (en $b + d \geq 0$ en $b \geq 0$, die uit de invariant volgen als $d > 0$). Deze laatste kunnen we waarmaken door een goede keus te doen voor de toekenning aan d. Voor de beginrelatie voor de selectie vinden we:

$$(((b + d)^2 \leq a \wedge a < c^2 \wedge d < c - b \wedge B) \vee$$
$$(b^2 \leq a \wedge a < (c - d)^2 \wedge d < c - b \wedge \neg B))$$

Als we nu voor B kiezen $(b + d)^2 \leq a$, zouden we graag zien dat $\neg B$ overeenkomt met $a < (c - d)^2$ (of deze impliceert). Is dit zo?
Als geldt dat $(b + d)^2 \leq (c - d)^2$, dan klopt dit. Stel namelijk dat $(b + d)^2 > a$ ($\neg B$), dan geldt zeker $(c - d)^2 > a$. We eisen dus dat

$$(b + d)^2 \leq (c - d)^2$$
$$b + d \leq c - d$$
$$2d \leq c - b$$

Als we dit opnemen in de beginrelatie, geldt dus:

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

```
if (b + d)↑2 ≤ a then b := b + d
               else c := c - d
fi {P2}
```

Van de beginrelatie blijft (als we zorgen dat de toekenning aan d de invariant niet verstoort) alleen over $2d < c - b$, want we hebben gezien dat $(b + d)^2 \leq a$ en $a < (c - d)^2$ niet beide false kunnen zijn.
We hebben nu, afgezien van de keuze voor d, gevonden:

```
b := 0; c := a + 1;
while b + 1 ≠ c
   do d := .....; {d > 0}
      if (b + d)↑2 ≤ a then b := b + d
                     else c := c - d
      fi
od
```

Het is duidelijk dat hoe groter de keuze van d, hoe sneller het programma. We kunnen d = 1 kiezen, want uit het feit dat b + 1 = c is, volgt samen met de invariant dat c - b > 1. Maar bij de keuze d = 1 zouden we een programma krijgen dat net zo langzaam is als het allereerste uit deze paragraaf. De maximale d die is toegestaan, rekening houdend met de beperking $2d \leq c - b$, wordt verkregen door d = (c - b) div 2.
Daarmee wordt het totale programma:

```
b := 0; c := a + 1;
while b + 1 ≠ c
   do d := (c - b) div 2;
      if (b + d)↑2 ≤ a then b := b + d
                       else c := c - d
      fi
   od
```

OPMERKINGEN

Bovenstaand programma vergt voor bijvoorbeeld $a = 10^6$ circa 20 repetities (het aantal repetities is $^2\log a$) en dat is een aanzienlijke winst vergeleken met de 10^3 repetities uit het eerste programma. Naast de zorg voor de correctheid speelt bij programmeren ook de efficiëntie van de programma's een grote rol.
We hebben gesteld dat uit $(b + d)^2 \leq (c - d)^2$ volgt dat beide voorwaarden false kunnen opleveren.
Kunnen echter ook beide voorwaarden $(b + d)^2 \leq a$ en $(c - d)^2 > a$ true zijn?
Als dat zo is, zouden we

```
if (b + d)↑2 ≤ a then b := b + d
             else c := c - d
fi
```

kunnen vervangen door:

```
if (b + d)↑2 ≤ a then b := b + d fi;
if (c - d)↑2 > a then c := c - d fi;
```

We moeten dan wel bewijzen dat als b met d verhoogd is, nog steeds geldt $c - b > d$ (de voorwaarde waaronder de tweede selectie uitgevoerd mocht worden).

Wanneer is aan beide voorwaarden te voldoen?

Als je $d = (c - b)/2$ neemt is maar één van de voorwaarden te voldoen (elimineren van c in $(c - d)^2$ met behulp van $d = (c - b)/2$ levert $(b + d)^2$). Maar onze d is niet precies $(c - b)/2$; als namelijk $c - b$ oneven is, is onze d iets kleiner en dus kan dan aan beide voorwaarden worden voldaan.

Nu moeten we nog wel bewijzen dat na afloop van:

    if $(b + d)\uparrow 2 \leq a$ then b := b + d fi

$c - b > d$ geldt (als $c - b$ oneven is).

Bij oneven $c - b$ geldt na d := (c - b) div 2:

    $2d + 1 = c - b$

(Dit is een sterkere uitspraak dan de oorspronkelijke $c - b \geq 2d$.)

En uit $\{2d + 1 = c - b\}$ b := b + d $\{T\}$ vinden we voor $\{T\}$:

$c - b = d + 1$ (en dus $c - b > d$).

(Einde opmerkingen.)

6.5. Regeldrukker als plotter

Gegeven een regeldrukker met twee opdrachten: NR en PRINTCHAR(n), de laatste print op huidige positie een karakter overeenkomend met n en verhoogt de printpositie met 1.
Voor ons zijn slechts twee waarden van n interessant: een marker en een spatie.

Verder zijn er twee functies fx en fy gegeven die beide een geheel resultaat opleveren bij een geheel argument:

$i: 0 \leq i < 1000 (0 \leq fx(i) < 100 \text{ en } 0 \leq fy(i) < 50)$

Gevraagd wordt een programma dat 50 regels drukt, die van boven naar beneden genummerd zijn van 49 tot 0, waarbij de posities op de regel van links naar rechts genummerd zijn van 0 tot 99 (regels → y coördinaat, posities → x coördinaat). Er is een figuur gegeven door 1000 of minder punten ($0 \leq i < 1000$) met $x = fx(i)$ en $y = fy(i)$. Op deze plaatsen moet een marker komen, verder alleen spaties (geen karakters).
Er is dus een figuur gegeven in discrete parametervoorstelling en we willen de regeldrukker gebruiken als een plotter (tekenapparaat).

We willen voor iedere i-waarde de functies fx(i) en fy(i) slechts één keer berekenen.
Omdat we op een regel niet terug kunnen en ook niet naar een oude regel terug kunnen, zouden we eerst alle functiewaarden kunnen berekenen. We krijgen dan

```
var image;
build(image); print(image)
```

We zullen image, build en print verder moeten uitwerken, want dit programma kan niet verwerkt worden. Het uitwerken van image zal zijn invloed hebben op de andere twee. Het uitwerken van de print kan echter nauwelijks zonder dat we weten hoe image eruit ziet. We beginnen daarom met build.
Bij het opbouwen zullen we wel moeten uitgaan van een "leeg" figuur en dan duizend markers stuk voor stuk moeten toevoegen. Voor build krijgen we dan

```
clear(image); setmarks(image)
```

clear(image) maakt 50 lege regels, setmarks zet de 1000 markers.

We gaan nu kijken hoe de 1000 punten kunnen worden toegevoegd, we werken dus setmarks uit:

```
begin integer i;
      i := 0; {i is het volgend toe te voegen punt}
      while i < 1000 do add mark; i := i + 1 od
end
```

Voor add mark kunnen we allereerst een uitwerking geven:

```
begin integer x, y;
      x := fx(i); y := fy(i); mark pos(x, y)
end
```

waarin mark pos de waarde van image zal veranderen doordat het punt (x, y) wordt toegevoegd.

We kunnen nu echter niet verder zonder dat we iets over image gezegd hebben. We moeten dus een representatie vinden voor de mogelijke waarden. We moeten er echter ook aan denken dat we de image moeten afdrukken op een regeldrukker met zijn beperkingen. Dit suggereert voor image:

```
array line [0:49]
```

waardoor print(image) wordt:

```
j := 49; {volgende regel die geprint wordt}
while j ≥ 0 do lineprint(line[j]); j := j - 1 od
```

en clear(image):

```
j := 49;
while j ≥ 0 do lineclear(line[j]); j := j - 1 od
```

En mark pos wordt nu

```
linemark(x, line[y])
```

We hebben tot nu toe gevonden:

```
array line[0:49];
integer j;
j := 49;
while j ≥ 0 do lineclear(line[j]); j := j - 1 od;
begin integer i;
      i := 0;
      while i < 1000
```

```
            do begin integer x, y;
                     x := fx(i); y := fy(i);
                     linemark(x, line[y])
               end;
               i := i + 1
            od
    end;
    j := 49;
    while j ≥ 0 do lineprint(line[j]); j := j - 1 od
```

We hebben nu dus het type image "vertaald" in het type line en de operaties op images "vertaald" in operaties op lines. De lines en de operaties daarop (lineclear, linemark en lineprint) moeten nu verder uitgewerkt worden. We hebben echter niets meer van doen met images.
We kunnen een line vastleggen door de x-coördinaten die een marker moeten krijgen, bijvoorbeeld een boolean array van 100 elementen waarbij de waarde van het element aangeeft of er op die positie een marker moet komen of niet; of een integer array van 100 elementen die ieder de waarde spatie of marker hebben. Deze representatie is algemener: we zouden ook andere tekens kunnen zetten. We kiezen dan ook voor de laatste representatie.
We krijgen dan voor line:

```
    integer array sym[0:99]
```

En voor de operaties achtereenvolgens (lineprint, lineclear, linemark):

```
lineprint:
    k := 0;
    while k < 100 do PRINTCHAR(sym[k]); k := k + 1 od;
    NR
lineclear:
    k := 0;
    while k < 100 do sym[k] := spatie; k := k + 1 od
linemark:
    sym[x] := mark
```

Als we dit invullen, zijn we klaar:

```
integer array symb[0:49, 0:99];
integer j, k;
j := 49;
while j ≥ 0 do k := 0;
                while k < 100 do sym[j,k] := spatie; k := k + 1 od;
                j := j - 1
              od;
begin integer i;
      i := 0;
      while i < 1000
         do begin integer x, y;
                  x := fx(i); y := fy(i);
                  symb[y,x] := mark
            end;
            i := i + 1
         od
end;
j := 49;
while j ≥ 0 do k := 0;
                while k < 100 do PRINTCHAR(symb[j,k]);
                                  k := k + 1
                               od;
                NR;
                j := j - 1
              od
```

Nadat op een regel de laatste marker is gezet, zouden we direct met de NR kunnen overgaan op een nieuwe regel (de spaties na de laatste marker doen niet ter zake).
We zouden per regel kunnen bijhouden tot en met welke positie er PRINTCHAR opdrachten gegeven moeten worden:

```
integer array plaats[0:49]
```

plaats[i] geeft de eerstvolgende te vullen positie op regel i (tot en met plaats[i] - 1 zijn de posities gevuld).

We krijgen dan:

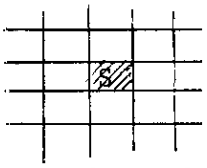
```
integer array symb[0:49,0:99];
integer array plaats[0:49];
integer j, k;
j := 49;
while j ≥ 0 do plaats[j] := 0; j := j - 1 od;
begin integer i;
      i := 0;
      while i < 1000
         do begin integer x, y;
                  x := fx(i); y := fy(i)
                  while plaats[y] < x
                     do symb[y,plaats[y]] := spatie;
                        plaats[y] := plaats[y] + 1
                     od;
                     symb[y,x] := mark; plaats[y] := plaats[y] + 1
            end;
            i := i + 1
         od
end;
j := 49;
while j ≥ 0 do k := 0;
               while k < plaats[j]
                  do PRINTCHAR(symb[j,k]);
                     k := k + 1
                  od;
               NR;
               J := j - 1
            od
```

6.6. The Game of Life.

Dit probleem is afkomstig van John Horton Conway (gepubliceerd in 1970 in de Scientific American).

Het "spel" wordt gespeeld op een bord met 24 * 24 velden. Het spel begint met een bepaalde configuratie van zogenaamde bezette velden en het gaat er nu om te zien hoe deze configuratie verandert onder invloed van de volgende regels:

1. als veld S bezet is dan blijft S bezet als S ten minste twee, maar niet meer dan drie bezette buren heeft; zo niet dan wordt S onbezet.



(ieder veld, afgezien van de randvelden, heeft 8 buren)

2. als veld S onbezet is, dan wordt S bezet als S precies drie bezette buren heeft; zo niet dan blijft S onbezet.

Een zet in het spel bestaat uit het simultaan toepassen van deze twee regels op alle velden.

De invoer voor het programma bestaat uit een rij waarin achtereenvolgens staan:

- een positief geheel getal (het aantal zetten dat gespeeld moet worden);
- paren (rij, kolom) voor de beginsituatie;
- het paar (0, 0) als afsluiting van de invoerrij.

De gevraagde uitvoer bestaat uit:

- de invoer (afgezien van het paar (0, 0)), waarbij we er van uitgaan dat ieder paar een correcte veldaanduiding is;
- de zetten.

Het programma heeft dus de vorm

```
"lees invoer en voer beginsituatie uit";
"zet het gevraagde aantal zetten"
```

Een eerste uitwerking levert op:

```
read (f, aantal);
"lees beginsituatie in en voer die uit";
stap := 0;
while stap ≠ aantal
   do "doe een zet";
      stap := stap + 1
   od
```

Omdat de regels simultaan moeten worden toegepast zullen we twee speelborden nodig hebben: "oud" en "nieuw". Uitvoering van een zet houdt dan in dat de regels toegepast worden op de configuratie van het oude bord om daaruit de configuratie van het nieuwe bord te bepalen.

Het inlezen van de beginsituatie kunnen we nu uitwerken:

```
"alle velden onbezet";
read(f,rij); read(f,kolom);
while ¬ (rij = 0 ∧ kolom = 0)
   do "noteer in oud (rij, kolom) als bezet veld";
      write(g,rij); write(g,kolom);
      read(f,rij); read(f,kolom)
   od;
"teken het bord met de configuratie"
```

Ook het uitvoeren van de zetten kunnen we nader detailleren:

```
stap := 0;
while stap ≠ aantal
   do "een zet van oud naar nieuw";
      "teken het bord met de configuratie";
      stap := stap + 1;
      "maak oud gelijk aan nieuw"
   od
```

De opdracht "teken het bord ....", waarvoor we een procedure kunnen maken, zullen we niet verder uitwerken, daar deze uitwerking nogal afhangt van de te gebruiken programmeertaal en het te gebruiken uitvoerapparaat.

We zouden ook steeds het nieuwe bord moeten initialiseren op een beginwaarde voordat we de zet van oud naar nieuw doen. Dit is niet nodig als we bij de zet ieder veld van nieuw een nieuwe waarde geven.

We moeten nu nog uitwerken:

```
"alle velden onbezet"
"noteer in oud (rij, kolom) als bezet veld"
"een zet van oud naar nieuw"
```

Het doen van een zet zouden we voor een deel kunnen uitwerken doordat we de regels gaan toepassen, verdere uitwerking is afhankelijk van de representatie van het bord binnen ons programma. De keuze voor de representatie wordt bepaald op grond van de operaties die moeten worden uitgevoerd:

- aangeven dat een veld bezet of onbezet is;
- het aantal buren van een veld tellen;
- oud en nieuw verwisselen.

Op grond van de eerste operatie zouden we een tweedimensionaal boolean array kunnen nemen. Omdat we bij de tweede operatie moeten tellen is het handiger om een integer array te nemen met voor ieder element de waarde 1 of 0 (1 → bezet, 0 → onbezet). Een kleine moeilijkheid schuilt nog in de "randvelden" (eerste of tweede index is gelijk aan 1 of 24). Deze velden hebben niet, zoals de andere velden, 8 buren. Dit betekent dat er bij de tweede operatie steeds nagegaan moet worden of een veld een randveld is of niet. Om alle velden van het eigenlijke speelbord uniform te kunnen behandelen met betrekking tot de tweede operatie voeren we een extra rand in van altijd lege velden. De tweede operatie wordt echter alleen op het oude bord toegepast. We komen zo tot:

```
integer array oud[0:25,0:25]
integer array nieuw[1:24,1:24]
```

Uitwerking van "alle velden onbezet" levert dan op:

```
for i := 1 step 1 until 24
    do for j := 1 step 1 until 24
           do oud[i,j] := 0; nieuw[i,j] := 0 od
    od;
for i := 0 step 1 until 25 do oud[0,i] := 0; oud[25,i] := 0 od;
for i := 1 step 1 until 24 do oud[i,0] := 0; oud[i,25] := 0 0d
```

En "maak oud gelijk aan nieuw":

```
for i := 1 step 1 until 24
    do for j := 1 step 1 until 24
           do oud[i,j] := nieuw[i,j]; od
    od
```

De operatie "noteer in oud (rij, kolom) als bezet veld" wordt

```
oud[rij,kolom] := 1
```

Blijft over de zet van oud naar nieuw. Voor ieder veld (rij, kolom) van oud tellen we de buren en passen de regels toe. We doen dit door de som te bepalen van de 9 velden (rij - 1, kolom - 1), (rij - 1, kolom), (rij - 1, kolom + 1), (rij, kolom - 1), ..., (rij + 1, kolom + 1). We krijgen dan:

```
for rij := 1 step 1 until 24
    do for kolom := 1 step 1 until 24
          do som := 0;
             br := rij - 1; er := rij + 1; bk := kolom - 1;
             ek := kolom + 1;
             for i := br step 1 until er
                 do for j := bk step 1 until ek
                        do som := som + oud[i,j] od
                 od;
                 som := som - oud[rij,kolom];
             if (oud[rij,kolom] = 1 ∧ (som = 2 ∨ som = 3)) ∨
                (oud[rij,kolom] = 0 ∧ som = 3)
                then nieuw[rij,kolom] := 1
                else nieuw[rij,kolom] := 0
             fi
          od
    od
```

Nu blijkt inderdaad dat ieder veld van nieuw een waarde krijgt. Dat houdt dus in dat we nieuw inderdaad niet hoeven te initialiseren.
We krijgen dan als totaal programma:

```
begin integer array oud[0:25,0:25], nieuw[1:24,1:24];
      integer i, j, rij, kolom, br, er, bk, ek, aantal, stap, som;
      read (f, aantal);
      for i := 0 step 1 until 25
          do for j := 0 step 1 until 25
             do oud[i,j] := 0 od
          od;
```

```
read(f,rij); read(f,kolom);
while ¬ (rij = 0 ∧ kolom = 0)
   do oud[rij,kolom] := 1;
      write(g,rij); write(g,kolom);
      read(f,rij); read(f,kolom)
   od;
"teken het bord en de configuratie";
stap := 0;
while stap ≠ aantal
   do for rij := 1 step 1 until 24
         do for kolom := 1 step 1 until 24
               do som := 0;
                  br := rij - 1; er := rij + 1; bk := kolom - 1;
                  ek := kolom + 1;
                  for i := br step 1 until er
                     do for j := bk step 1 until ek
                           do som := som + oud[i,j] od
                     od;
                  som := som - oud[rij,kolom];
                  if (oud[rij,kolom] = 1 ∧ (som= 2 ∨ som= 3)) ∨
                     (oud[rij,kolom] = 0 ∧ som = 3)
                     then nieuw[rij,kolom] := 1
                     else nieuw[rij,kolom] := 0
                  fi
               od
         od;
      "teken het bord en de configuratie";
      stap := stap + 1;
      for i := 1 step 1 until 24
          do for j := 1 step 1 until 24
                do oud[i,j] := nieuw[i,j] od
          od
   od
end
```

Voor het uitvoeren van een zet worden voor alle 576 velden negen optellingen uitgevoerd. In veel gevallen zullen deze nul opleveren. We zouden een andere representatie voor het bord kunnen kiezen, waarbij voor ieder paar (i,j), $1 \leq i \leq 24$ en $1 \leq j \leq 24$, niet alleen het bezet/onbezet wordt bijgehouden maar ook het aantal bezette buren.
Ook als er maar weinig bezette velden zijn worden alle velden bekeken terwijl de velden die niet "naast" een patroon liggen toch niet veranderen. Is hieraan iets te doen door een andere representatie voor het bord te kiezen?

7. De processor.

In het voorgaande hebben we ons beziggehouden met programmeren: het opstellen van een rij opdrachten die door een processor kan worden uitgevoerd. Die processor zal vaak de digitale rekenmachine zijn. We zullen hier de functionele eigenschappen van deze machine bekijken.

Het centrale deel.

Eerder is gesproken over processen, waarvan de elementaire handelingen door de processor worden gedefinieerd.
Het ontwerpen van een machine die dergelijke processen kan uitvoeren brengt het definiëren van elementaire machine-handelingen met zich mee.
Aan een machine, die de door ons besproken processen moet kunnen uitvoeren, stellen we de volgende eisen:

(eis 1) de machine moet de lijst van uit te voeren opdrachten lezen en bewaren;

(eis 2) de machine moet elk van de opdrachten kunnen uitvoeren (bijbehorende acties kunnen laten plaatsvinden);

(eis 3) tijdens de uitvoering van een opdracht moet de machine bijhouden welke opdracht daarna aan de beurt is;
de machine moet na een actie automatisch met de aan-de-beurt-zijnde actie doorgaan;

(eis 4) de machine moet van elke variabele de heersende waarde bewaren.

Eisen 1 en 4.
De machine zal beschikken over een centraal geheugen, dat bestaat uit genummerde posities, zo'n positie kan een opdracht uit de lijst van uit te voeren opdrachten of de heersende waarde van een variabele bevatten.
Terminologie: het nummer van een positie noemen we "adres"; de inhoud van (een positie met) een adres noemen we een "woord" en we geven deze aan met "inh[<adres>]".
Voorbeeld: Is adres 403 toegewezen aan een variabele, waarvan de heersende waarde -87 is, dan is inh[403] = -87;
is adres 26 gebruikt voor opslag van het programma dan in inh[26] = "code van een elementaire handeling".

Eisen 2 en 3.
De machine zal beschikken over een aantal registers, speciale geheugenplaatsen, waarvan de inhoud snel en in overeenstemming met vaste regels gewijzigd kan worden.

In het bijzonder treffen we de volgende registers aan:
- opdrachtregister (instructieregister, uitvoeringsregister);
- opdrachtteller (instructieteller);
- rekenregisters.

Deze registers zijn opgenomen in de *centrale verwerkingseenheid* van de machine, die uit twee delen bestaat, het *rekenorgaan* met de rekenregisters en het *besturingsorgaan* waarin het opdrachtregister en de opdrachtteller zijn ondergebracht.
Het *opdrachtregister* bevat de heersende uit te voeren opdracht; de code voor deze opdracht is gehaald uit het centraal geheugen; executie van de opdracht kan activering van rekenregisters of opdrachtteller betekenen.
De *opdrachtteller* bevat een getal dat, na executie van de heersende opdracht, gelijk is aan het adres van de opdracht die dan aan de beurt is.
De inhoud van een *rekenregister* kan - als effect van een instructie -
(a) optreden als operand in een rekenkundige bewerking
(b) worden overgebracht naar een geheugenpositie
(c) gelijk worden gemaakt aan de inhoud van een geheugenwoord
(d) worden onderzocht op positief-, nul- of negatief-zijn.
De rekenregisters maken, zoals gezegd, deel uit van het rekenorgaan, waar de berekeningen worden uitgevoerd.
De inhouden van de zojuist besproken registers noemen we OR (opdrachtregister) en OT (opdrachtteller).
Werking van de machine betekent dat het volgende cyclische proces (de basiscyclus) uitgevoerd wordt:

```
while true do begin OR := inh[OT];
                    OT := OT + 1;
                    executie van instructie "OR"
               end
```

Machinetaal; binaire code.

De machine kent zijn eigen, door de fabrikant ingebouwde, repertoire van elementaire handelingen. Iedere opdracht voor zo'n handeling wordt in de machine gerepresenteerd door een bepaald patroon van nullen en enen ter lengte van een woord. Voor veel machines is het zo, dat het woord dat de opdracht voorstelt uit twee delen bestaat: functiedeel en adresdeel. Het eerste deel, het functiedeel, specificeert de eigenlijke opdracht of operatie (bijvoorbeeld optellen, aftrekken of een transport van of naar het geheugen); het tweede deel, het adresdeel, specificeert het adres waarvan de inhoud bij de opdracht betrokken is. We noemen het tweede deel ook wel het operanddeel omdat de inhoud van het genoemde adres als operand in de opdracht optreedt. We beperken ons hier tot 1-adres opdrachten; er zijn ook machines die een 2-adres opdrachtcode of een 0-adres opdrachtcode hebben, of een nog andere opdrachtcode.
Het repertoire van al deze opdrachten voor elementaire handelingen, geschreven in nullen en enen, noemen we de *machinetaal*. De machine kan alleen opdrachten die in deze machinetaal gegeven zijn uitvoeren. De machinetaal is dus een binaire code en zo zou de opdracht "maak de inhoud van het tweede rekenregister gelijk aan de inhoud van adres 47" in machinetaal (bij een woordlengte van 12 bits) kunnen luiden: 000100101111, waarbij 0001 het functiedeel is en 00101111 het adresdeel.

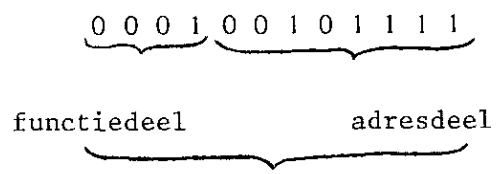


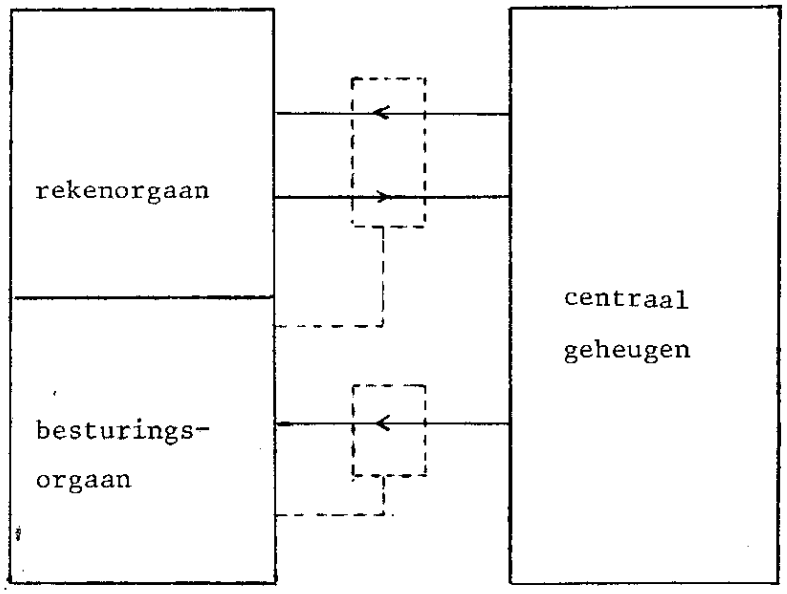
fig. 1. Samenstelling opdrachtcode.

De machinetaalopdrachten van een programma staan op achtereenvolgende adressen in het geheugen.

Bij de eerste computers moest men het machinetaalprogramma, met opdrachten in binaire code, opdracht na opdracht met behulp van schakelaars in het geheugen brengen. Als het programma en de gegevens waarop het programma moest werken dan in het geheugen stonden, kon men met weer andere schakelaars de machine dit programma laten verwerken.
Het schrijven van programma's in machinetaal en het verwerken van deze programma's op de zojuist genoemde wijze en werkt fouten in de hand.

De structuur; het centrale geheugen.

De structuur van de machine is als volgt:



Het rekenorgaan en het besturingsorgaan samen noemen we de centrale verwerkingseenheid, de CVE, (men trekt hier ook wel eens het centraal geheugen bij). De gewone lijnen zijn transportlijnen; de stippellijnen geven aan dat transport van een woord gebeurt onder controle van het besturingsorgaan en dat de selectie van de geheugenpositie, waar het woord naar toe gebracht of vandaan gehaald moet worden, door het besturingsorgaan wordt bepaald.

Op vele wijzen zijn dergelijke machines geconstrueerd. Bij al deze uitvoeringen is een geheugenplaats (genummerde positie) van het centraal geheugen opgebouwd uit een aantal elementen, bits genaamd, die ieder een nul of een één kunnen voorstellen. Een woord, de inhoud van een geheugenplaats, is dus een rij nullen en enen en deze rij kan de representatie zijn van een opdracht of van de heersende waarde van grootheid. De begrippen geheugenplaats en woord worden vaak door elkaar gehaald en ook noemt men vaak zowel een binair cijfer als de voorstelling hiervan in een geheugenplaats een bit. Het aantal bits in een woord is de woordlengte en dit is een vaste karakteristiek van de machine (voorbeelden van woordlengtes: 8, 12, 24, 27, 32, 36, 60). Er zijn ook machines die per bit adresseerbare geheugens hebben.
Een groep van 1024 (bij sommige fabrikanten van computers ook wel eens 1000) woorden noemt men K woorden; de geheugenomvang wordt in deze eenheid uitgedrukt (bijvoorbeeld: 4K, 512K). Vaak is het geheugen opgebouwd uit modules met vaste grootte, bijvoorbeeld 4K, 16K of 64K.
Soms komen de begrippen geheugenplaats en woord niet overeen. De geheugenplaatsen (genummerde posities) bestaan dan uit een kleiner aantal bits; een gangbaar aantal is 8 en men noemt zo'n groep van 8 bits dan een byte, die dus adresseerbaar is. De inhoud van een aantal aaneengesloten bytes noemt men dan een woord; een woord kan dan bijvoorbeeld 4 bytes beslaan. Naast het begrip woord kent men dan ook nog de begrippen halfwoord en dubbelwoord.
Ook de registers bestaan uit een aantal binaire plaatsen (ook hier bits genoemd); dit aantal is voor opdrachtregister en rekenregister meestal gelijk aan de woordlengte, voor de opdrachtteller zodanig dat elk voorkomend adres gerepresenteerd kan worden.

Omdat vermenigvuldiging van twee getallen ter lengte n een getal ter lengte 2 * n of 2 * n - 1 oplevert, is er ook een rekenregister met dubbele woordlengte, meestal gerealiseerd door een serieschakeling van twee enkellengte-registers.
Er zijn machines waarbij men niet werkt met een vast aantal bytes of bits per woord, maar waar men werkt met geheugentrajecten die gekenmerkt worden door het adres van de begineenheid (byte of bit) en de lengte ervan (in bytes of bits) of een eindmarkering (vlagbit). Deze machines hebben een zogenaamde variabele woordlengte. De opbouw van de machine wat bijvoorbeeld de lengte van de registers betreft wordt hier natuurlijk door beinvloed.

Opmerking.

Het overbrengen van een geheugenwoord naar opdrachtregister of rekenregister, het overbrengen van de inhoud van een rekenregister naar een geheugenpositie, het transporteren van waarden naar en van het centraal geheugen evenals de besturingssignalen geschieden in de vorm van elektrische pulsen.
De registers en het geheugen bestaan uit elektronische schakelingen.

Randapparatuur.

Er blijven nog twee vragen:

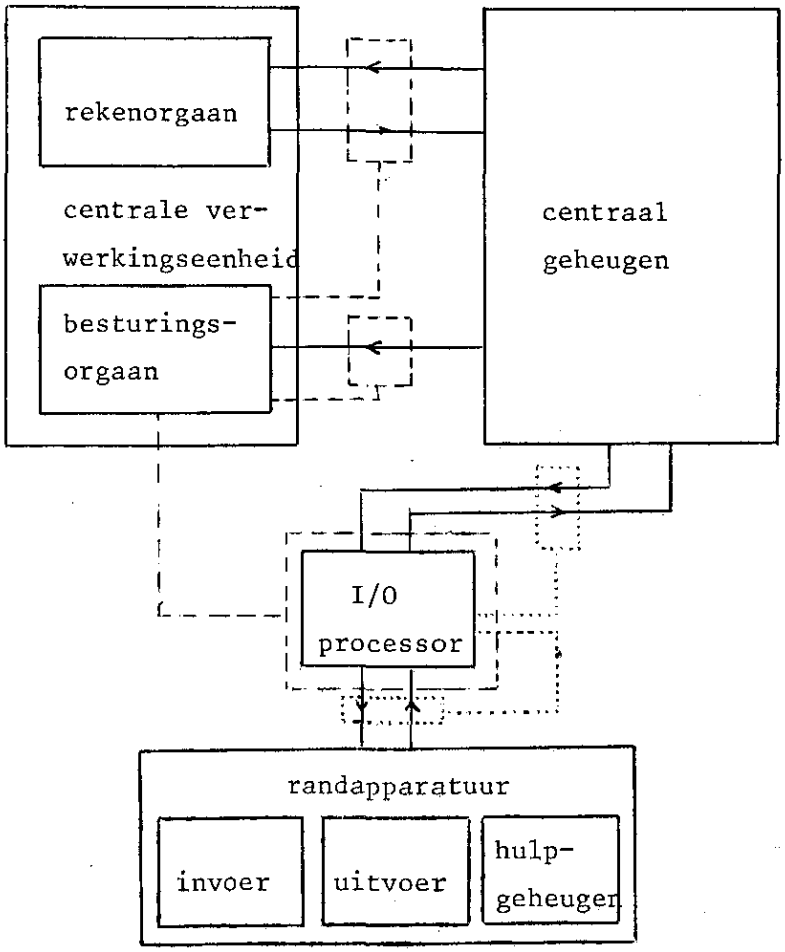
(a) Hoe komt het programma en hoe komen de door een programma gevraagde gegevens in het centraal geheugen?
(b) Hoe worden de resultaten van de programmaverwerking aan de buitenwereld medegedeeld?

Het in bedrijf stellen van de computer start de verwerking van een vast in de computer aangebracht programma. Het effect van dit programma is het importeren en in het geheugen opbergen van de aangeboden informatie. Importeren en opbergen betekent hierbij dat elektrische pulsen van "buiten" in staat worden gesteld de bits van een aantal geheugenwoorden in een toestand te brengen die overeenkomt met het patroon van die elektrische pulsen.

De machine is ook in staat om de informatie die in de geheugenwoorden staat in de vorm van elektrische pulsen naar "buiten" te brengen.
Er vindt steeds omzetting plaats van informatie naar elektrische pulsen en omgekeerd. De pulsen die in de buitenwereld terecht komen, moeten daar natuurlijk weer omgezet en/of vastgelegd worden.
De elektrische pulsen van en naar het centraal geheugen lopen naar en van de randapparaten. We onderscheiden invoer- en uitvoer-apparaten.
Invoer gebeurt door middel van: ponskaartlezers, ponsbandlezers, typemachine-toetsenbord, documentlezers, magneetbandeenheden, schijven, trommel en signalen van fysische processen.
Uitvoer gebeurt door middel van: ponskaartponsers, ponsbandponsers, typemachine-drukwerk, regeldrukker, plotter, magneetbandeenheden, schijven, trommel, displays (beeldbuizen) en signalen voor fysische processen.
Schijven, magneetbanden en trommel noemt men wel secundaire geheugens omdat ze gebruikt worden voor opslag van grote hoeveelheden informatie. Ze kunnen meer informatie bevatten dan het centraal geheugen en zijn ook goedkoper. De wijze waarop ze binnen de machine zijn opgenomen in het informatieverkeer, rechtvaardigt het beschouwen van deze apparaten als invoer- en uitvoer-apparaten en de informatie is dan ook minder snel bereikbaar dan de informatie in het centraal geheugen.
Tussen het centraal geheugen en de randapparaten staat de I/O-processor. De I/O-processor, ook wel kanaal (channel) genoemd, is een specialpurpose computer, die beschikt over een zeer beperkte opdrachtenset; deze opdrachtenset bevat opdrachten die het transport regelen tussen de randapparatuur en het centrale geheugen. Het besturingsorgaan geeft de I/O processor opdracht tot transport; hierbij moet de I/O-processor enige gegevens ter beschikking worden gesteld zoals: welk randapparaat is bij het transport betrokken, wáár in het geheugen moeten de te transporteren woorden geplaatst of gevonden worden, en om hoeveel woorden gaat het.
Het besturingsorgaan start dan het transport, maar de verdere afwikkeling hiervan gebeurt door de I/O-processor.

Nadat het transport is beëindigd moet de I/O-processor dit melden aan het besturingsorgaan.
Er kunnen meerdere I/O-processoren aanwezig zijn. Meerdere apparaten kunnen met de I/O-processor verbonden zijn, doch de I/O-processor kan slechts één apparaat tegelijk bedienen. Men kan echter een I/O-processor laten switchen van het ene randapparaat naar het andere, voordat het totale transport voor het eerste randapparaat is beëindigd. Dit kan bijvoorbeeld nadat één woord is overgebracht. I/O-processoren die deze mogelijkheid hebben noemt men multiplexors en zij worden toegepast bij langzame randapparaten zoals kaartlezers en regeldrukkers. Als een I/O-processor eerst het totale transport van het eerste apparaat afhandelt en daarna pas overgaat tot het transport voor het tweede apparaat spreekt men van een selector.
Het is niet noodzakelijk dat een randapparaat slechts met één I/O-processor is verbonden; het is mogelijk dat een randapparaat via een aantal I/O-processoren transporten kan verzorgen. Bij een transportopdracht in het besturingsorgaan voor een bepaald randapparaat zal dan naar een vrije, met het randapparaat te verbinden, I/O-processor worden gezocht. Men spreekt in deze gevallen van drijvende I/O-processoren (floating channels).
De I/O-processor heeft toegang tot het centrale geheugen, niet alleen voor het vinden of plaatsen van de bij het transport betrokken woorden, maar ook omdat het programma voor de I/O-processor in het geheugen kan staan. Nadat het besturingsorgaan de I/O-processor de opdracht tot het transport heeft gegeven, gaat de centrale verwerkingseenheid door met het verwerken van het programma en kan daarbij ook toegang tot het geheugen willen hebben.

De structuur van de machine is als volgt:



—— transport

– – besturing door besturingsorgaan

---- besturing door I/O processor

Programmatuur

De uitvoering van een programma, het proces, vindt plaats in het computersysteem.
Voordat het proces kan plaatsvinden moet de procesbeschrijving in één of andere vorm aan het computersysteem worden aangeboden. Tijdens het proces vindt een aantal activiteiten plaats die we in principe bij ieder proces terugvinden:

invoer van gegevens,
verwerking van de gegevens,
opslag van berekende tussenresultaten,
raadpleging van deze tussenresultaten,
uitvoer van uitkomsten.

De in te voeren gegevens moeten in één of andere vorm aan het systeem worden aangeboden.
Na afloop van het proces moeten de uitgevoerde resutaten uit het systeem worden verwijderd.
Het aanbieden van het programma en de in te voeren gegevens, het starten van de uitvoering (het proces) en het verwijderen van de uitvoer werd bij de eerste generatie computers door één en dezelfde persoon gedaan namelijk de persoon die ook de programmering van het programma voor z'n rekening had genomen: de programmeur. De programmeur was in die tijd namelijk tegelijk computerexpert, hij kende alle typische eigenschappen van de computer, en was dientengevolge het beste in staat om, in geval er iets mis ging met de uitvoering van zijn programma, deze uitvoering weer op gang te helpen. Bovendien wist hij precies wanneer en hoe de gegevens voor het programma moesten worden ingevoerd.
De invoering van standaardprogramma's en procedures voor het oplossen van veel voorkomende deelproblemen (zoals het uitrekenen van wiskundige functies) bracht in deze situatie enige verandering. Deze standaardprocedures konden door iedere programmeur, die er later behoefte aan had, gebruikt worden; ze werden daarom dichtbij, en later in het computersysteem bewaard, ze vormden samen een bibliotheek van standaardprocedures.

Invoering van deze bibliotheekprocedures had aan de ene kant tot gevolg dat het programmeerwerk gemakkelijker werd, althans minder omvangrijk, aan de andere kant nam het operateurswerk toe; de bibliotheekprocedures moesten op het juiste moment en op de juiste plaats in het hoofdprogramma worden tussengevoegd. Hierdoor nam het aantal *routine*handelingen van de programmeur tijdens de verwerking toe; deze routinehandelingen leenden zich uitstekend voor standaardisering en hierdoor werd het aantrekkelijk om de verwerking van programma's aan één persoon op te dragen: de *operateur* deed z'n intrede. De operateur zorgde in het vervolg voor een efficiënte verwerking van gebruikersprogramma's en kreeg daarbij verder als taak bij te houden door wie, hoe lang en waarvoor de computer gebruikt werd. Hij en de naakte hardware (centrale computer plus randapparatuur) vormden tezamen een "systeem". Aan dit systeem konden complete jobs (karweien) worden opgedragen waarbij een job in dit geval inhield: de uitvoering van een programma, verzorging van invoer en uitvoer en registratie van het verloop.

De volgende stap in de ontwikkeling van computersystemen was de intrede van de tweede generatie computer. Kenmerk van deze generatie: enorme snelheidsvergroting ten opzichte van de eerste generatie. De verwerkingssnelheid van programma's werd ongeveer 100 maal zo groot als voorheen. Deze toename in verwerkingssnelheid was enerzijds het gevolg van de versnelling van de centrale verwerkingseenheid (gebruik van transistoren in plaats van buizen), anderzijds het gevolg van een andere organisatie van de invoer/uitvoer: het datatransport tussen randapparaten en kerngeheugen werd nu door een aparte processor (de I/O processor, het kanaal) uitgevoerd, gelijktijdig met de verwerking van het programma door de CVE (overlapping van datatransport en CVE-activiteit).

De toename in de verwerkingssnelheid leidde tot het probleem van "de operateur als bottleneck" in het systeem. Als het vijf minuten duurt om een job van twee uur op gang te helpen, dan wordt er iets meer dan 4% van de nuttige computertijd niet gebruikt; als het echter ook vijf minuten duurt om een job van dertig seconden te starten dan gaat praktisch alle nuttige computertijd ongebruikt voorbij. De oplossing van dit porbleem was: de initialiseringstijd van een job verkorten.
Aangezien dit niet kon door de operateur te versnellen moest dit worden bereikt door een aantal initialiseringshandelingen van de operateur door de snelle computer zelf te laten uitvoeren (automatisering van de operateurshandelingen).
Voor dit doel werd een standaardprogramma (*jobmanager*) geconstrueerd; de computer werd aangekleed met een stuk software dat als een soort verlengstuk van de hardware kan worden opgevat; het geeft van buitenaf, door de operateur bezien, de computer een ander gezicht. Voor de gebruiker verandert er niets aan het systeem, er kunnen nog steeds complete jobs worden aangeboden, het enige wat de gebruiker zal merken is dat z'n programma veel sneller verwerkt wordt. De jobmanager zorgt voor het starten en, na uitvoering, voor de beëindiging van het ene programma na het andere uit een programmastroom. Deze stroom wordt gevoed door de operateur; deze hoeft alleen maar programma's en invoergegevens in randapparaten te deponeren en uitvoer van randapparaten te verwijderen.
Het systeem bestaat van nu af aan uit: hardware, een stuk software en operateur. Het stuk software alleen wordt het bedrijfssysteem (operating system) van de computer genoemd.

Het gebruik van een kanaal als, van de CVE-onafhankelijk opererende, transportprocessor leidde, naast de toename van de verwerkingssnelheid, tot twee problemen:

buffering

synchronisatie.

Het kanaal wordt door CVE gestart, voert daarna zelfstandig het datatransport uit en meldt de voltooiing van het transport door middel van een interrupt aan de CVE.
Om het aantal CVE-activiteiten ten behoeve van een transport door het kanaal te beperken en de mogelijkheden tot parallelvorming van CVE en kanaal goed te benutten, moeten de porties, die het kanaal per opdracht te transporteren krijgt, niet al te klein zijn. Dit houdt in dat de data (bijvoorbeeld getallen) niet stuk voor stuk getransporteerd moeten worden op het moment dat ze nodig zijn of geproduceerd zijn in het uitgevoerde programma (zoals dat bij de eerste generatie computer het geval was). De data moeten portiegewijs worden verzameld en in het geheugen worden opgeslagen, in buffers, voor toekomstig gebruik in het proces (invoer) of voor toekomstig transport naar een randapparaat (uitvoer).
Synchronisatie (= afstemming op elkaar van, onafhankelijk van elkaar verlopende, activiteiten in de tijd) is vereist op het moment dat in het proces de date gebruikt gaan worden. Er moet dan namelijk vastgesteld worden of de data reeds aanwezig zijn (in een buffer) of dat het transport nog niet is voltooid; in het laatste geval moet op de voltooiing van het transport gewacht worden. Deze synchronisatie wordt mogelijk gemaakt door invoering van de zongenaamde interruptfaciliteit in de computer, waarmee, zoals hiervoor reeds gezegd, het kanaal een voltooiing van een datatransport aan de CVE kan melden.

Realisering van gebufferde invoer en uitvoer en synchronisatie van datatransport en gebruikersproces, is de volgende taak die het bedrijfssysteem er bij krijgt; het stuk software dat deze taak uitvoert zullen we *I/O- manager* noemen.

Naast de twee bovengenoemde problemen, buffering en synchronisatie, bracht het kanaal nog een probleem met zich mee namelijk leegloop van de CVE. Als op het moment, dat de data in een proces gebruikt moeten worden, deze data nog niet aanwezig zijn, moet het proces wachten, hetgeen voor de CVE wil zeggen: niets doen. In deze, qua nuttig gebruik van de CVE, ongezonde situatie kan slechts op één manier verbetering worden gebracht: zorgen dat de CVE in de wachttijd iet anders (zinvols!) te doen heeft, bijvoorbeeld het uitvoeren van een ander,

niet op data wachtend, programma.
Het systeem wordt zodanig gemodificeerd dat een aantal programma's tegelijk door de CVE kan worden uitgevoerd; op ieder moment is de CVE naar net één programma bezig; over een langere tijd bekeken verdeelt de CVE z'n aandacht over een aantal programma's; we spreken nu van een *multiprogrammeringssysteem*.
Verwezenlijking van de multiprogrammeringsidee is de volgende taak die het bedrijfssysteem krijgt opgedragen, de software die er voor zorgt noemen we de *procesmanager*.
Invoering van multiprogrammering leidt op zijn beurt weer tot een ander probleem, namelijk dat van de geheugenbezetting.
Een programma, of op zijn minst een deel ervan, moet in het werkgeheugen aanwezig zijn op het moment dat het door de CVE wordt uitgevoerd. Bij mulitprogrammering (= uitvoering van een aantal programma's tegelijk) moet dus een aantal programma's (of delen ervan) in het werkgeheugen aanwezig zijn.
De volgende taak die we voor het bedrijfssysteem zien opdoemen is dus de geheugenverdeling over de verschillende processen, het stuk software hiervoor noemen we *memory manager*.

Het voorgaande samenvattend komen we tot de volgende taken die door het bedrijfssysteem moet worden vervuld:

- jobmanagement (karweibeheer): het accepteren, starten en beëindigen van jobs (karweien); een job kan zijn de uitvoering van één programma of van een aantal programma's na elkaar;
- I/O-management (het beheer van de invoer/uitvoerhandelingen): het starten van datatransporten door een kanaal, het reageren op gereedmeldingen van een kanaal, het reageren op foutsituaties tijdens transporten;
- procesmanagement (procesbeheer): het realiseren van multiprogrammering, hetgeen neerkomt op het verdelen van de CVE-tijd over de in het systeem plaatsvindende processen;
- memorymanagement (geheugenbeheer): het verdelen van werk- en achtergrondgeheugen over de verschillende processen.

De drie laatstgenoemde taken komen neer op: verdelen van de aanwezige hardwarefaciliteiten (CVE, geheugen, randapparaten) over de processen, daarbij zorgend voor een zo eerlijk en efficiënt mogelijk gemeenschappelijk gebruik van deze faciliteiten (resource sharing). Alle bovengenoemde taken worden vaak samengevat onder de naam: operating support; de verschillende stukken software die zorgen voor de realisering van deze taken worden vaak samengevat onder de naam: monitor (ook wel genoemd supervisor).
Aan de taken van operating support wordt vaak nog een taak toegevoegd speciaal ten behoeve van de instantie die de computer beheert, te weten het bijhouden van statistische gegevens betreffende: bezetting van CVE, geheugen en randapparaten, opgetreden fouten in hardware. Deze taak, die vaak logging wordt genoemd, zullen wij zien als een deeltaak van de jobmanager.

Parallel aan de ontwikkeling in bedrijfssystemen heeft zich ook een ontwikkeling in programmeertalen voorgedaan.
Men kwam namelijk al vrij snel tot de ontdekking dat programmeren in machinecode (hetgeen neerkomt op het genereren van rijen nullen en enen), vooral voor wat grotere programma's, een vrijwel ondoenlijke zaak is. Om de taak van de programmeur wat te verlichten werden al gauw symbolische machinetalen (waarin iets meer aansprekende symbolen dan alleen maar nullen en enen gebruikt worden) en later ook hogere programmeertalen (waarmee de programmeur zich geheel kan wijden aan de oplossing van zijn probleem zonder veel te moeten weten van de machine-eigenschappen) ontwikkeld.
De computer en met name de CVE, iest echter nog steeds programma's uitgedrukt in machinecode (rijen nullen en enen). De programma's geschreven in andere talen dan machinetaal moeten dus eerst worden omgezet (vertaald, gecompileerd) in machinecode, alvorens ze kunnen worden uitgevoerd.
Het spreekt vanzelf dat voor deze omzetting, hetgeen voor iedere computer een standaardactiviteit is, stukken software werden geconstrueerd: *assemblers* en *compilers*.
Ook deze stukken software worden tot het bedrijfssysteem gerekend.

Naast de ontwikkeling van programmeertalen moet nog worden genoemd de ontwikkeling van standaardgebruikersprogramma's en procedures; programma's en procedures voor de oplossing van veel voorkomende problemen (bijvoorbeeld het sorteren van gegevens, het uitrekenen van mathematische functies), die voor iedere gebruiker ter beschikking staan.
Om deze standaardprocedures voor iedere gebruiker snel en gemakkelijk beschikbaar te maken worden ze opgenomen in het bedrijfssysteem, ze vormen tezamen de standaardbibliotheek.
Het beheer van deze bibliotheek (*Library management*) is ook één van de taken van het bedrijfssysteem, het stuk software zullen we library manager noemen.
Vertalen en library management worden vaak samengevat onder de naam "programming support".

Tenslotte signaleren we nog een probleem, waarbij het bedrijfssysteem de gebruiker de helpende hand kan toesteken, namelijk de organisatie en manipulatie van grote hoeveelheden bij elkaar behorende data (files, bestanden), in aan het gebruikersprobleem aangepaste representatie.
Een goed bedrijfssysteem is niet compleet zonder een stuk software dat de gebruiker steun verleent bij het opbouwen, het bijhouden, het opslaan en het raadplegen van gegevensbestanden: *datamanager*.
Deze taak van het bedrijfssysteem wordt vaak "datasupport" genoemd.
De taken vallend onder programming support en data support komen neer op: regelen van het gemeenschappelijk gebruik van de aanwezige softwarefaciliteiten (vertalers, bibliotheek, bestanden).

Al het voorgaande in overweging nemend komen we uiteindelijk tot de volgende beschrijvingspoging van een bedrijfssysteem:
de software, die, gebruikmakend van de hardware, in staat is een aantal programma's, geschreven in andere talen dan de machinetaal, gelijktijdig uit te voeren, er daarbij voor zorgdragend dat in die programma's een zo efficiënt mogelijk gebruik gemaakt kan worden van

alle hardwarefaciliteiten (randapparaten, geheugen, CVE) en alle door het systeem zelf gecreëerde softwarefaciliteiten (vertalers, bestanden, bibliotheek).

Jobmanagement

Onder een job (karwei) verstaan we een afgeronde hoeveelheid werk die door het bedrijfssysteem moet worden verricht om een bepaalde gebruikerstaak uit te voeren.
Het eenvoudigste voorbeeld van een job is de uitvoering van één programma, een job komt dan overeen met één proces.
Vaak komt het voor dat, om een bepaalde taak volledig te verrichten, een aantal programma's achter elkaar moet worden uitgevoerd. We zeggen dan dat de job bestaat uit een aantal jobsteps (ook wel tasks genoemd).
Voorbeelden van jobs:

de vertaling van een programma geschreven in hogere programmeertaal (step 1), gevolgd door de uitvoering van het vertaalde programma (step 2); geponste orders inlezen en sorteren (step 1), gevolgd door bijwerken van voorraadbestand aan de hand van gesorteerde orders (step 2), gevolgd door sorteren van orders in andere volgorde (step 3), gevolgd door bijwerken van debiteurenbestand aan de hand van gesorteerde orders (step 4).

Een job bestaat uit één of meer jobsteps die achter elkaar (sequentieel) worden uitgevoerd; meestal zijn de jobsteps van elkaar afhankelijk, de uitvoer van de ene step dient als invoer voor de andere step.
In het systeem bestaat een job uit de uitvoering van een aantal processen na elkaar.

De jobmanager is het stuk systeem software dat moet zorgen voor de realisering van jobs door het systeem. De taak van de jobmanager bestaat uit:

1 accepteren van jobs
2 per job sequentieel uitvoeren van jobsteps
3 beëindigen van jobs.

Lettend op punt 2 is de jobmanager in zekere zin te vergelijken met een processor; ook een processor voert een rij opdrachten sequentieel uit. De rij opdrachten, die door een processor moet worden uitgevoerd, noemen we een programma; het ligt voor de hand om de rij jobstepbeschrijvingen, die tezamen de job beschrijven, ook een programma te noemen; we spreken dan van jobcontrolprogram (systemcontrolprogram, karweibesturingsprogramma, workflowprogram).
De taal, waarin dit programma geschreven wordt, heet jobcontrollanguage (systemcontrollanguage, karweibesturingstaal, workflowlanguage). De jobcontrollanguage is dus de taal waarin de gebruiker het systeem moet aanspreken (een programmeertaal is de taal waarin de gebruiker een compiler aanspreekt).
Enkele voorbeelden van opdrachten uit de jobcontrollanguage (jobcontrolstatements of jobstepbeschrijvingen):

"compile A tot B in library"
(compileer het programma dat zich onder de naam A in het systeem bevindt en berg het resultaat van de compilatie onder de naam B op in de programmabibliotheek);
"execute B"
(voer het programma, dat zich onder de naam B in de programmabibliotheek bevindt, uit).

Een voorbeeld van een jobcontrolprogram is:

"compile A to B"
"execute B".

De hier beschreven job bestaat uit 2 jobsteps, te weten de vertaling van het programma A, geschreven in brontaal, naar de vertaalde versie genaamd B, gevolgd door de uitvoering van dit vertaalde programma.
Deze jobbeschrijving is verre van volledig. De informatie over de organisatie van de invoer en uitvoer (de z.g. file informatie) in de verschillende jobsteps ontbreekt. We komen op de jobcontrolstatements die te maken hebben met benodigde data in de jobsteps terug bij de beschrijving van de jobstepselectie.

Op het gebied van de jobcontrollanguage is nog weinig uniformiteit te ontdekken, ieder systeem heeft z'n eigen jobcontrollanguage.

Index