TECHNISCHE HOGESCHOOL EINDHOVEN
Afdeling Algemene Wetenschappen
Onderafdeling der Wiskunde

INFORMATICA I

Prof. Dr. R.J. Lunbeck

Najaarssemester 1974

JdG

Technische Hogeschool Eindhoven

Onderafdeling der Wiskunde

Informatica I

Dictaatnr. 2.212 Prijs ƒ 8,--

Inhoudsbeschrijving

INFORMATICA I

R.J. Lunbeck

Najaarssemester 1974

TECHNISCHE HOGESCHOOL EINDHOVEN
Onderafdeling der Wiskunde

INFORMATICA I

Prof.dr. R.J. Lunbeck

Najaarssemester 1974

1.0 ALGORITHMEN en ALGOL

1.1 Algorithmen

In het begin van de negende eeuw na Christus werkte in de bibliotheek van het toen pas gestichte Bagdad een zekere Muhammed ben Mûsâ al-Khowârizmî (Mohammed, zoon van Mozes, geboren in Khowarizm), die een Arabisch boek over algebra schreef. (Het woord "Algebra" is een verbastering van de Arabische titel!) Behalve dat schreef hij een handleiding voor het "Indisch" rekenen, d.w.z. het rekenen in het tientallig stelsel. Enige eeuwen later heeft dit boek (in Latijnse vertaling) een grote rol gespeeld bij de invoering van het tientallig stelsel in Europa; de bijnaam van de auteur, al-Khowârizmî, leeft tot op de huidige dag voort in het woord "algorithme", dat sedert die tijd in zwang is voor "rekenvoorschrift" of algemener voor "handelingsvoorschrift".

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

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

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

Vervolgens zien wij, dat een algorithme alleen dan als zodanig functioneert, wanneer tussen opsteller en uitvoerder geen misverstand of verschil van mening bestaat over het repertoire basishandelingen, waaruit de totale verrichting moet worden opgebouwd. Breipatronen en montagevoorschriften zijn in dit opzicht vrij zuivere algorithmen; recepten zijn in dit opzicht al wat troebeler ("zout naar smaak toevoegen" is niet zo duidelijk; het is dan ook geen toeval dat een gerecht eerder mislukt dan een breiwerk). Bij het vragen naar de weg is de verwarring over wat toegelaten handelingen zijn doorgaans volkomen; ze zijn als algorithme dan ook zelden zonder dubbelzinnigheid uitvoerbaar.

Het is verder noodzakelijk dat het totaal aantal uit te voeren basishandelingen eindig is (en liefst zelfs niet al te groot) opdat het resultaat van het volgen van de algorithme in een eindige tijd ter beschikking komt.

Tenslotte moet een algorithme nog een zekere innerlijke logica hebben. Als er in een recept, waarin nog niet over bouillon geschreven is, staat "Laat vervolgens de bouillon afkoelen", dan is er iets mis, ook al weet de uitvoerder op zich zelf best, hoe hij bouillon volgens de regels van de kunst moet laten afkoelen! Hetzelfde geldt voor de aanwijzing: "Rechtuit en dan bij de laatste stoplichten linksaf"!

Al hebben wiskundigen zich sinds eeuwen met algorithmen beziggehouden, in de twintigste eeuw zijn algorithmen opnieuw in het middelpunt van de wiskundige belangstelling komen te staan.

De eerste aansporing daartoe kwam van het grondslagenonderzoek, toen men zich de vraag ging stellen aan welke eisen een deugdelijk wiskundig bewijs nu eigenlijk moest voldoen. Dit was een soort "wiskundig gewetensonderzoek", dat nodig was geworden omdat men tegen allerlei paradoxen aanstootte. Dat men hierbij algorithmen tot voorwerp van studie maakte is niet zo verwonderlijk als we bedenken, dat een bewijs een speciaal soort handelingsvoorschrift is, nl. een redeneervoorschrift om tot een bepaalde conclusie te komen. De moeilijkheden bleken voort te komen uit bewijsstappen, waarbij niet gezegd werd, hoe je ze moest uitvoeren. Wij zullen deze logische schermutselingen niet verder vervolgen en vermelden slechts dat de gedachteconstructie van een van de onderzoekers op dit gebied, A.M. Turing, onder de naam "Turing Machine" bekend is geworden.

De tweede aansporing om zich op algorithmen te concentreren kwam door de ontwikkeling van de digitale electronische rekenautomaten ("digital computers"; voorlopig zal de toevoeging "digitale" worden weggelaten, omdat pas later een uiteenzetting zal worden gegeven van analoge rekenautomaten). Deze computers zijn nl. in staat om algorithmen (niet alleen "numerieke" voor het berekenen van bepaalde grootheden, al zijn dit de best bekende en oudste toepassingen) in zich op te nemen en vervolgens getrouwelijk uit te voeren. Voor algorithmen, die bestemd zijn om door een rekenautomaat uitgevoerd te worden is de naam "programma" in zwang gekomen; het opstellen van programma's heet "programmeren" en degene, die dit doet, heet "programmeur".

Door de komst van de rekenautomaten is de belangstelling voor algorithmen drastisch veranderd: van een uiterst theoretisch onderwerp is het, door de grote toepassingsmogelijkheden van rekenautomaten en hun economische importantie geworden tot een buitengewoon practische aangelegenheid. Bovendien, was aanvankelijk de belangstelling voor algorithmen in hoofdzaak analytisch, nu is de belangstelling in hoge mate synthetisch: er moeten algorithmen van allerlei soort gemaakt worden, willen wij de mogelijkheden van rekenautomaten kunnen benutten. In de afgelopen twintig jaren heeft het programmeren zich ontwikkeld tot een van de zeer creatieve loten aan de wiskundige stam.

1.2 Automaten

Alvorens verder over programmeren te spreken, moeten wij een globale indruk geven van wat een rekenautomaat kan en uit welke functionele componenten zo'n machine is opgebouwd. Uit het dagelijkse leven zijn wel diverse automatische mechanismen bekend, die ieder een of ander aspect van een rekenautomaat illustreren.

Allereerst zijn roltrappen te noemen, die indien niet gebruikt veelal stil staan. Wie een roltrap oploopt, onderbreekt daarbij een lichtstraal die op een fotocel pleegt te vallen en ten gevolge van deze onderbreking zet de trap zich in beweging en rolt hij (ruim) eenmaal zijn eigen lengte af. Wie zich dan daardoor laat transporteren en aan de andere kant de trap verlaat, merkt dat even later de trap achter hem weer tot stilstand komt, tenzij inmiddels opnieuw iemand de straal heeft onderbroken: de roltrap rolt zijn eigen lengte af na iedere onderbreking van de straal. Dit betekent, dat vanaf een bepaald ogenblik - nl. de onderbreking van de straal- het mechanisme automatisch een vaste handeling verricht, autonoom "een vast programma" afwerkt, nl. zijn eigen lengte eenmaal afrollen.

Er moet een bepaald aantal treden "verrold" worden; er zit kennelijk ergens in het mechanisme een geheugenelement, dat b.v. bijhoudt over hoeveel treden doorgerold moet worden, voordat de trap weer zal stoppen. Wij kunnen ons voorstellen, dat de roltrap is uitgerust met een zg. teller, een geheugenelement, waarin genoteerd staat het aantal treden, waarover de trap nog rollen moet. Wie de lichtstraal onderbreekt, zorgt daardoor dat dit geheugenelement gevuld wordt met het maximale bedrag (= aantal zichtbare treden van de trap + een beetje extra), als de trap loopt wordt de inhoud van dit geheugenelement per treetransport met 1 verminderd en als de inhoud van dit geheugenelement terugkeert tot nul, stopt de trap.

In de praktijk worden dergelijke geheugenelementen gerealiseerd door een mechanisme, dat zich in een groot aantal discrete toestanden kan bevinden, waarbij men met elke toestand van het geheugenelement een waarde associeert. Het element "vullen met een bepaald bedrag" betekent "het element brengen in de met dit bedrag geassocieerde toestand". Typerend voor de automatische roltrap is:

- de aanwezigheid van een dergelijk geheugenelement,
- de beinvloeding van de inhoud van dit geheugenelement door de activiteit van het mechanisme (elke tredeverschuiving gaat gepaard met een vermindering van de inhoud met 1),
- de beinvloeding van het mechanisme door de inhoud van het geheugenelement (trap stoppen als deze inhoud tot nul is teruggekeerd),
- het startsignaal (hier de onderbreking van de lichtstraal) waardoor het geheugenelement in de aanvangstoestand gebracht wordt.

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

Er is een ander, drastisch verschil tussen de moderne rekenautomaat en de roltrap. Wat van de roltrap gevergd wordt is eigenlijk altijd hetzelfde, nl. een standaard reactie op een uniek startsignaal. Van een andere bekende automaat, nl. een lift, wordt al iets meer verwacht, nl. een aantal reacties op combinaties van startsignalen. In (af)wasautomaten zijn een aantal programma's "vast" ingebouwd; welk programma uitgevoerd wordt hangt weer van het startsignaal af.

Van de moderne rekenautomaat verlangt men dat hij allerlei algorithmen in zich kan opnemen en gehoorzamen. Men drukt dit wel uit door te zeggen dat een rekenautomaat een "general purpose" machine is; het wordt nog duidelijker tot uitdrukking gebracht door te zeggen dat moderne rekenmachines "programmeerbare automaten" zijn.

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

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

Anderzijds zijn de apparaten, binnen hun toepassingsgebied, tamelijk algemeen! Voor het weefgetouw van Jacquard betekent dit, dat het allerlei patroontjes kan weven, van Franse lelies tot familiewapens toe en juist door die algemeenheid door het boek gestuurd moet worden om te bepalen, wat er deze keer gemaakt moet worden. Voor het draaiorgel betekent dit, dat het allerlei muziek kan maken, van psalmen en volksliederen tot tophits toe en juist door die algemeenheid het draaiboek nodig heeft opdat vastligt wat er deze keer voor muziek gemaakt moet worden. Zo ook voor de moderne programmeerbare rekenautomaat: hij kan allerlei algorithmen uitvoeren en het is juist

deze algemeenheid die maakt, dat hij steeds een programma nodig heeft ter beschrijving van de gedragslijn, die deze keer gevolgd moet worden.

1.3 Rekenautomaten

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

Hier houdt overigens de analogie tussen weefgetouw en draaiorgel enerzijds en rekenautomaat anderzijds op. Weefgetouw en draaiorgel beschikken niet over een groot geheugen (d.w.z. een groot aantal geheugenelementen) en elke regel van het draaiboek wordt "uitgevoerd" op het moment, dat deze regel onder de aftasters van het leesstation ligt. Bij een draaiorgel heeft dit bv. tot gevolg dat de enige manier om zestien maten muziek te laten herhalen is om deze zestien maten twee keer in het immers doordraaiende boek te laten voorkomen. De rekenautomaat beschikt echter wel over een groot geheugen, dat onder andere gebruikt wordt om eerst de hele algorithme op te nemen voordat aan de werkelijke uitvoering ervan begonnen wordt.(Deze "stored program" automaten kunnen zo nodig hun eigen programma veranderen, hetgeen de al tientallen jaren oudere ponskaartapparaten niet kunnen.) In een programma kunnen we, zoals we later zullen zien, wel aangeven, dat bv. een stukje nog een keer herhaald moet worden (zoals ook in bladmuziek!).

We zijn inmiddels dus twee componenten van de rekenautomaat tegengekomen, het leesstation ("invoerorgaan") via hetwelk het geponste programma aan de machine wordt toegevoerd, en het geheugen, waarin dit programma wordt opgenomen. Als het programma in zijn geheel is opgenomen, dan begint het eigenlijke werk: het handelingsvoorschrift wordt opgevolgd en de berekening wordt

uitgevoerd. Hierbij speelt een derde onderdeel van de machine een belangrijke rol, het zg. "besturingsorgaan". Dit zorgt er voor dat de door de programmaschrijver voorgeschreven bewerkingen in de juiste volgorde afgewerkt worden doordat de benodigde automaatorganen op de juiste tijdstippen aan het werk worden gezet. Tijdens deze afwerking speelt het geheugen een dubbele rol: ten eerste wordt het geraadpleegd omdat het uit te voeren programma er in is opgeslagen, ten tweede wordt het gebruikt als "kladpapier" om voor het proces belangrijke tussenresultaten er zolang in op te slaan. (Dit tweede gebruik van het geheugen is analoog aan het geheugenelement van de roltrap.)

De rekenautomaat bevat verder apparatuur voor een vierde functie, nl. het aan de buitenwereld terugmelden van de gevraagde uitkomsten. Deze uitvoerorganen kunnen zijn: bandponsers, kaartponsers, regeldrukkers, schrijfmachines, tekentafels, kathodestraalbuizen, etc.. (Omdat men in het Nederlands ook spreekt over het"uitvoeren" van een programma zou "import- en exportorganen" in plaats van invoer- en uitvoerorganen een prettiger terminologie geweest zijn. In het Engels gebruikt men de woorden "Input", "Output" en "Execution").

Om met name rekenprocessen snel uit te kunnen voeren bevat de rekenautomaat tenslotte meestal een "rekenorgaan" (processor), waarmee o.a. de gebruikelijke arithmetische bewerkingen worden uitgevoerd.

Uit bovenstaande beschrijving van een rekenautomaat blijkt ook de overeenkomst met de werkwijze van een menselijke rekenaar. Het leesstation correspondeert met zijn in-postbak, waarin hij zijn getallenmateriaal en opdrachten vindt; het uitvoerorgaan met de uit-postbak; het geheugenorgaan met het menselijke geheugen en de "verlengstukken" ervan in de vorm van papier, tabellenwerken, etc.; het rekenorgaan met de tafelrekenmachine en tenslotte het besturingsorgaan met de menselijke beheersing van zijn activiteiten. Dat er uiteraard ook verschillen zijn t.a.v. bijvoorbeeld de communicatiemiddelen (bij de mens woord en geschrift), werkwijze en manier van formuleren van opdrachten, enz. kan men zelf bedenken.

Door het schrijven van een programma wordt een scheiding tot stand gebracht tussen het bewerkingsschema en het gegevensmateriaal, wat niet gebruikelijk is wanneer wij zelf een berekening uitvoeren. Het bewerkingsschema passen wij dan vaak nog op het laatste moment aan bij het gegevensmateriaal en zelfs bij tussenresultaten. Bij gebruik van een rekenautomaat (trouwens ook met een menselijke assistent die wij niet te veel willen zien) moet

van tevoren overdacht worden welke situaties zich kunnen voordoen voor en tijdens de bewerking. Daar staat tegenover dat we dit (als we het goed doen) maar eenmalig hoeven uit te voeren, waarna het programma "altijd" te gebruiken is.

Het communicatieprobleem met een rekenautomaat zal het hoofdthema van dit hoofdstuk zijn, de logica van de opbouw en de wijze van gebruik van een rekenautomaat komt pas daarna aan de orde, terwijl de electronische realisering en constructie van een rekenautomaat helemaal niet besproken worden omdat dit grotendeels op het terrein van de electrotechniek ligt en omdat kennis van deze terreinen niet strikt nodig is om rekenautomaten goed te gebruiken.

1.4 Een eerste programma

Zoals gezegd mag er tussen de opsteller en de uitvoerder van een algorithme geen misverstand bestaan over het repertoire handelingen, waaruit de totale verrichting moet worden opgebouwd. Met betrekking tot een programma, dat bestemd is om door een machine te worden uitgevoerd betekent dit, dat de programmatekst en gegevens aan heel precieze regels moet voldoen: de programmatekst moet zijn geschreven in een "programmeertaal", waarvan de taalregels heel nauwkeurig gedefinieerd zijn. (Natuurlijke talen schieten daarin absoluut tekort!) Een van de elegantste talen is Algol 60 (Algol is een samentrekking van ALGOrithmic Language),een internationaal erkende programmeertaal, waarvan in dit college het belangrijkste gedeelte behandeld zal worden. Het is een taal die niet alleen buitengewoon geschikt is om er numerieke processen mee te formuleren voor een rekenautomaat (dus als communicatiemiddel tussen mens en automaat), maar die ook zeer geschikt is voor publicatiedoeleinden (dus als communicatiemiddel tussen mensen). Bij deze eerste kennismaking met Algol zal niet primair gestreefd worden naar een volledige uiteenzetting van de regels van Algol, doch zal eerder getracht worden om een inzicht te geven in het"waarom" van deze regels en in een goed gebruik van deze regels. Al in ons eerste programma zullen we zien hoe de rekenautomaatorganen een rol spelen:

- een invoerorgaan voor de invoer van programma en gegevens,
- een geheugenorgaan voor het onthouden van het programma, gegevens en tussenresultaten,
- een rekenorgaan ter uitvoering van o.a. arithmetische bewerkingen.
- een uitvoerorgaan voor het afleveren van de antwoorden,
- een besturingsorgaan voor activering van de andere organen en de coördinatie van de werking van deze organen.

Een programma, dat twee gehele getallen ("integers"), a1 en a2, van een ponsband leest en op de regeldrukker op een regel achter elkaar vier getallen afdrukt, nl. de twee gelezen getallen, hun som en hun verschil, kan als volgt luiden:

```
begin integer a1, a2, som, verschil;
      a1 := read; a2 := read; som := a1 + a2; verschil := a1 - a2;
      print(a1); print(a2); print(som); print(verschil)
end
```

Allereerst zien we hier een paar onderstreepte woorden (begin, integer en end). Zo'n onderstreept woord moet beschouwd worden als één enkel symbool (zoals wij op stations nu ook symbolen hebben voor ingang, bagage bergplaats, etc.), het fungeert als een soort van "leesteken" en de gelijkenis met bepaalde Engelse woorden is alleen een geheugensteuntje. Bij het ontwerp van programmeertalen werd namelijk alras duidelijk, dat - zoals in de meeste wiskundige teksten - er behoefte zou zijn aan meer symbolen dan normaliter op een schrijfmachine aangebracht kunnen worden. Hiervoor is in Algol 60 een algemene conventie gekozen, nl. een onderstreept woord. De consequentie hiervan is, dat in Algol-teksten onderstreping voor geen enkel ander doel gebruikt mag worden. Op deze wijze zijn ruim 20 speciale symbolen, met nauwkeurig vastgelegde betekenis, ingevoerd. (Onderstreping is op een schrijfmachine en in een geschreven manuscript heel makkelijk uit te voeren, voor een zetter is het echter moeilijk: als Algol-programma's in druk verschijnen, worden deze woorden daarom niet onderstreept maar vet gedrukt.) De onderstreping is dus niets diepzinnigs, het is louter een notatieafspraak hoe een aantal symbolen er op papier zullen uitzien (hoe deze symbolen op een ponsband of ponskaart gerepresenteerd worden, zij hier in het midden gelaten).

Een Algol-programmatekst begint altijd met het symbool begin en eindigt altijd met het symbool end. Zij fungeren als haakjespaar, begin is de openingshaak en end is de sluitingshaak. Zij worden hier gebruikt om de begrenzing van de programmatekst aan te geven. (In normale conversatie hebben we geen teken nodig om aan te geven, dat we bv. uitgesproken zijn: je gesprekspartner neemt dat aan op vage indicaties, zoals mimiek of een tijdje stilte van jouw kant! Bij radiotelefonie conversatie is het gebruik van de symbolen "over" en "uit" bekend.) Ander gebruik van het hakenpaar begin - end zullen we straks tegenkomen.

Om een overzichtelijke indeling te krijgen is het gebruikelijk om bij elkaar horende begin en end symbolen onder elkaar te schrijven en de tussenliggende regels in te laten springen. Slechts wanneer tussen begin en end heel weinig tekst staat zal men ze desnoods op één regel schrijven.

De op begin volgende rest van de eerste regel - de "declaratie" van de variabelen genaamd a1, a2, som en verschil - beschrijft het gebruik dat van het geheugen gemaakt zal worden voor het vastleggen van alle in ons programma voorkomende begin-, eind- en tussenresultaten. Dat de declaratie ingeleid is met het symbool integer geeft aan dat elk der vier geintroduceerde variabelen op elk moment een geheel getal representeert. De namen van gedeclareerde variabelen moeten door komma's van elkaar gescheiden worden; achter de laatste moet een punt-komma volgen.

Bij de roltrap, die maar een enkel geheugenelement had, kon dit onbenoemd blijven, zoals je in een monogame maatschappij er mee volstaan kunt, je echtgenote als "Vrouw" toe te spreken. Heb je een harem, dan is dat wat onduidelijk en moet je kunnen aangeven welke van je vele vrouwen je toespreekt en daartoe moeten de vrouwen benoemd zijn. In een rekenproces, waarin als regel vele variabelen voorkomen, moeten de variabelen om dezelfde reden benoemd zijn en dat is nu net, wat de declaratie voor ons uitdrukt. De eerste regel drukt dus niet alleen uit, dat er in dit programma vier (geheeltallige) variabelen voorkomen, hij geeft ook de namen op, onder welke in de programmatekst naar deze variabelen verwezen zal worden. Door de declaratie wordt aan de variabele(n) geen (bekende) waarde toegekend, ook niet de waarde nul!

1.5 Namen

In een programma hebben namen - de vakterm voor hen luidt "identifiers" - geen intrinsieke betekenis. "Identifiers have no inherent meaning, they serve for identifying purposes only". Het is als in het normale leven, waarin meneer De Groot best een dwerg zou kunnen zijn. De programmeur is (in hoge mate) vrij in de keuze van de namen die hij zelf invoert, maar hij doet er goed aan, om sprekende en liefst korte namen in te voeren en verwarrende namen te vermijden.

Ter illustratie van het feit, dat namen geen intrinsieke betekenis hebben, geven we een andere programmatekst, die dezelfde opgave als boven oplost:

```
begin integer a1, a2, integer, som;
      a1 := read; a2 := read; integer := a1 + a2; som := a1 - a2;
      print(a1); print(a2); print(integer); print(som)
end
```

Dit programma doet precies hetzelfde als het voorafgaande; het enige verschil is dat de arithmetische som de misleidende naam integer (let op dat het woord integer niet onderstreept is en dus niet een symbool, maar een naam, zij het een slechte, voorstelt), en het arithmetische verschil de misleidende naam som gekregen heeft. Aangezien de machine wel de regels van Algol, maar niet die van het Nederlands (of welke andere natuurlijke taal) zijn bijgebracht, wordt dit programma gewoon uitgevoerd. Desalniettemin wordt het introduceren van een dergelijke (voor ons!) misleidende nomenclatuur natuurlijk ten zeerste afgeraden.

Opmerking 1: Men kan Algol 60 beschouwen als een taal, waarvan de syntax (grammatica) weliswaar heel streng vastgelegd is, maar waarvan het vocabulaire min of meer open is; de declaratie fungeert dan als "woordenlijstje" voor het bijbehorende programma.

Opmerking 2: Naast de door de programmeur voor variabelen ingevoerde namen komen er ook een paar ongedeclareerde namen voor standaard functies (zoals read) en handelingen (zoals print) in dit programma voor. Hun betekenis is door conventie bepaald, zij verwijzen naar een standaard arsenaal - de zg. bibliotheek -, dat zonder meer bekend verondersteld mag worden in een bepaald Rekencentrum.

Opmerking 3: Bij afspraak worden in een Algoltekst geen subscripten als zodanig gebruikt. Later zullen wij zien hoe subscripten en exponenten aangeduid worden.

In het voorafgaande hebben wij gezegd, dat de programmeur een grote vrijheid heeft in de keuze van namen voor zijn variabelen. Helemaal vrij is hij echter niet. Als naast de namen a1 en a2 bijvoorbeeld ook de symbolenreeks a1 + a2 als naam van een derde variabele zou mogen voorkomen, dan zou niet meer duidelijk zijn of met a1 + a2 de som van de eerste twee variabelen, dan wel de waarde van de derde variabele bedoeld zou zijn!

(Beperkingen aan de toelaatbaarheid van namen, ter vermijding van misverstand, zijn bekend. Odysseus, door de Cycloop gevangen, zei dat hij Niemand heette. Toen Odysseus de Cycloop het enige oog had uitgestoken en de Cycloop, verblind en brullend van de pijn de andere cyclopen vroeg om de boosdoener te vangen en zij hem vroegen, wie dan wel zijn oog had uitgestoken, antwoordde de verminkte "Niemand heeft het gedaan". Zodat zij schouderophalend huns weegs gingen en Odysseus een kans kreeg om te ontsnappen. Onderscheid dus altijd goed een grootheid en zijn naam!)

In Algol is als naam van een variabele toegelaten een identifier, dat is een opeenvolging van symbolen bestaande uit letters en cijfers, mits niet beginnend met een cijfer. (Eventuele spaties en overgang op een nieuwe regel hebben, als overal in een Algol-programma, geen betekenis. Leestekens en andere symbolen zijn uitdrukkelijk niet toegelaten bij de vorming van namen!) Toelaatbaar als naam zijn seven up en den bosch, ontoelaatbaar zijn 7 up en 's-hertogenbosch; toelaatbaar is austin seven, toelaatbaar is ook austin 7, maar de laatste twee namen hebben betrekking op verschillende variabelen omdat het verschillende symbolenreeksen zijn.

1.6 Getallen

In Algol worden getallen in het tientallig stelsel genoteerd, maar ook hier is de schrijfwijze streng gereglementeerd. Onderscheid moet worden gemaakt tussen de twee typen die gedeclareerd worden met de symbolen integer en real.

Een geheel getal of integer (van het type integer) is een opeenvolging van de cijfers 0,1,2,3,4,5,6,7,8,9, al of niet voorafgegaan door een + of - teken. Ieder anders geschreven getal is van het type real ook al is de waarde van het getal geheel (42.0 is van het type real, al is het een integer).

De notatieregels voor getallen van het type real vereisen eerst de invoering van de begrippen: "decimal fraction", d.i. een decimale punt gevolgd door één of meer cijfers, en "exponent part", d.i. het symbool $_{10}$ gevolgd door een integer. Toegelaten schrijfwijzen voor reals zijn nu:

1) exponent part, eventueel voorafgegaan door een + of - teken,
2) decimal fraction, eventueel voorafgegaan door een + of - teken,
3) als 2) maar bovendien gevolgd door een exponent part,
4) integer, gevolgd door een exponent part,

5) integer, gevolgd door een decimal fraction,
6) als 5) maar bovendien gevolgd door een exponent part.
Met deze regels kunnen de volgende voorbeelden gecontroleerd worden:

legale getallen			illegale getallen		
.12	+ .12	- .12	.12.34		
34	+ 34	- 34	++ 34	+ -34	
10^5	- 10^{+5}	+ 10^{-5}	+ $.10^5$		
6.0	+ 6.0	- 6.0	6.	+ 6.	-6. (!)
-7_{10} - 8			-7_{10} - 8.		
-12.34_{10} - 56			-12.34_{10} - 5.6		

Of variabelen van het type integer of real zijn, volgt niet uit de schrijfwijze (als bij getallen) maar uit de declaratie

real a, b, c; integer p, q;

(let op de gebruikte scheidingstekens of separators). Integer gedeclareerde variabelen kunnen slechts gehele waarden krijgen. Real gedeclareerde variabelen krijgen reële waarden met slechts een eindige (van het type rekenautomaat afhankelijke) nauwkeurigheid.

1.7 Assignment statements

Na de declaratie van namen volgen in de programmatekst acht zg. "statements". De statements beschrijven de eigenlijke handelingen, die verricht moeten worden. Het is een regel in Algol, dat op elkaar volgende declaraties en/of statements onderling door een puntkomma gescheiden worden. Deze regel, die enige analogie vertoont met de normale interpunctieregel zinnen met een punt af te sluiten, is uitgevoerd om de tekst eenduidig interpreteerbaar te maken: zonder de "statement separator", nl. de puntkomma, was het bv. niet duidelijk waar de eerste statement zou eindigen en waar de tweede zou beginnen, was het niet duidelijk dat de letterrij "read a2" gesplitst moet worden in de naam "read" gevolgd door de naam "a2".

Zolang niet anders is aangeduid (zie later) worden statements uitgevoerd in de volgorde, waarin ze in de tekst voorkomen. De eerste statement

a1 := read

(lees "a1 wordt read") behelst, dat op grond van de functie read onder doorschuiven van de band een getal van de band gelezen wordt en dat de gelezen

waarde aan de variabele a1 wordt toegekend. Precieser: de waarde van de (in de bibliotheek bekend verondersteIde) functie genaamd "read" is gelijk aan de volgende getalwaarde op de invoerband; tot nader order zal de variabele genaamd a1 na uitvoering van de eerste statement de zojuist gelezen waarde hebben. (Tot nader order betekent tot een nieuwe waarde is toegekend, dus hier: voor de verdere duur van dit programma). Het "wordt-symbool", weergegeven door ":=" heet toekenningsoperator of "assignment operator", en moet goed onderscheiden worden van de gelijkheidsoperator = (In de meeste wiskunde leerboeken en in andere programmeertalen wordt dit onderscheid niet gemaakt omdat de lezer verondersteld wordt uit de tekst op te kunnen maken of hij met de := dan wel = operator te maken heeft.)Door het symbool := wordt tot uitdrukking gebracht dat het hier om een uit te voeren handeling gaat, en niet zoals bij het symbool = om een relatie waaraan al dan niet voldaan kan zijn. Zo wordt door de "assignment statement"

```
i := i + 1
```

de waarde van i met 1 opgehoogd, terwijl de relatie

```
i = i + 1
```

voor geen enkele waarde van i juist is.

In het algemeen wordt door een assignment statement aan de in het linkerlid genoemde variabele (dan wel in een meervoudige assignment statement

```
x := y := z := .....
```

aan de eerst genoemde variabelen x, y en z, die allen van hetzelfde type moeten zijn) de waarde toegekend die volgt uit de uitwerking ("evaluatie") van de uitdrukking die na het laatste := teken volgt. In deze uitdrukking mogen uiteraard naast getallen alleen maar variabelen genoemd worden die in voorafgaande assignment statement(s) reeds een waarde gekregen hebben.

Ten gevolge van de eerste statement is het eerste getal van de invoerband gelezen en heeft de variabele a1 (waarvan de waarde voordien ongedefinieerd was) het gelezen getal als waarde gekregen. De volgende statement

```
a2 := read
```

behelst, dat weer een getal van de band gelezen wordt, maar dat de nu gelezen waarde aan de variabele a2 wordt toegekend. Na de eerste twee statements zijn twee getallen van de invoerband gelezen en toegekend aan de variabelen

a1 en a2; met gebruikmaking van (de namen van) die variabelen kunnen we nu in het programma zo vaak als nodig is van deze twee getallen gebruik maken. In dit programma gebeurt dit voor beide twee keer, nl. in de volgende statement, waarin de som van de getallen berekend wordt en als waarde wordt gegeven aan de variabele genaamd som en in de daarop volgende statement, waarin op analoge wijze het verschil wordt berekend en toegekend aan de variabele genaamd verschil.

Na de twee invoerstatements, gevolgd door de twee berekeningsstatements volgen in de derde regel vier uitvoerstatements. Hier wordt een beroep gedaan op de (in de bibliotheek bekend veronderstelde) handeling genaamd print, een handeling met één argument, die zoals de naam suggereert, de meegegeven waarde op papier doet verschijnen. Ten gevolge van dit programma zou bv. gedrukt kunnen worden:

```
+ 456678    + 243179    + 699857    + 213499
```

Dit programma is nog niet representatief voor het gebruik van variabelen. Na de declaratie, die de variabele heeft geintroduceerd, is zijn waarde ongedefinieerd tot aan de uitvoering van de eerste assignment statement, die een waarde aan hem toekent. We hebben gezegd, dat de variabele dan deze waarde blijft behouden "tot nader order": een variabele verliest nl. zijn oude waarde wanneer er door een assignment statement wederom een waarde aan wordt toegekend. Dit wordt geillustreerd in het volgende programma, dat precies hetzelfde effect heeft als het vorige. Merk op, dat in dit programma maar drie variabelen voorkomen.

```
begin integer a1, a2, antwoord;
      a1 := read; print(a1); a2 := read; print(a2);
      antwoord := a1 + a2; print(antwoord);
      antwoord := a1 - a2; print(antwoord)
end
```

1.8 Simple arithmetic expressions (SAE)

In het voorgaande is reeds vermeld dat in een assignment statement de waarde van de uitdrukking ("expression") moet worden berekend die rechts van het laatste := symbool staat. Een "simple arithmetic expression" is een variabele of een getal of wordt met gebruikmaking van de gewone algebraïsche regels opgebouwd uit "variables, numbers, arithmetic operators" en ronde haakjes (geen accolades of rechte haken!). De "arithmetic operators" zijn

+ voor de optelling
- voor de aftrekking
* voor de vermenigvuldiging
/ voor de deling
↑ voor de machtsverheffing
÷ voor de "gehele deling".

Een product van twee variabelen a en b moet geschreven worden als a * b (de * verschijnt op het schrijfmachinepapier als een × teken) en niet als a.b (de punt mag alleen als decimaal punt gebruikt worden) of ab (wat een nieuwe identifier zou zijn). Bij machtsverheffing kan dank zij invoering van de operator ↑ de exponent op hetzelfde niveau geschreven als het grondtal en moeten eventueel ronde haakjes gebruikt worden om misverstand uit te sluiten of te voorkomen dat twee (arithmetische) operatoren naast elkaar komen te staan.

Voorbeelden:

a^2 wordt a ↑ 2, a^{2p} wordt a ↑ (2 * p), a^{-2} wordt a ↑ (-2).

De gehele getallendeling is voor integers m en n als volgt gedefinieerd: als |m| = q * |n| + r met 0 ≤ r < |n|, en sign(m/n) := +1 als m en n hetzelfde teken hebben en -1 als ze verschillend teken hebben, dan is

m ÷ n := sign(m/n) * q .

Het gebruik van, strikt genomen, overbodige haakjes kan de duidelijkheid van een Algol-tekst verhogen, al gelden de gebruikelijke prioriteitsregels (machtsverheffen voor vermenigvuldigen of delen en tenslotte optelling en aftrekking) en worden bij gelijke prioriteit de bewerkingen van links naar rechts uitgevoerd:

Voorbeeld:

a * b/c * d is equivalent met ((a * b)/c) * d en niet met bv. (a * b)/(c * d) .

Enkele andere voorbeelden:

wisk. formule	correcte Algol weergave	incorrecte Algol weergave
$(a + b)(c + d)$	(a + b) * (c + d)	(a + b)(c + d)
$2n - 1$	2 * n - 1	2n - 1
$\frac{1}{1 + a}$	1/(1 + a)	1/1 + a
$(2^3)^4$	2 ↑ 3 ↑ 4 of (2 ↑ 3) ↑ 4	2 ↑ (3 ↑ 4)
2^{3^4}	2 ↑ (3 ↑ 4)	2 ↑ 3 ↑ 4
$p \cdot \frac{-1}{qr}$	p * (-1)/(q * r)	p * -1/q * r
$\begin{vmatrix} a & b \\ c & d \end{vmatrix}$	a * d - b * c	ad - bc .

De waarde van een simple arithmetic expression is een getal van een type dat afhangt van de operatoren en van het type getallen en variabelen in de expressie:

a + b, a - b en a * b zijn alleen integers als zowel a als b integers zijn,
a ↑ b is alleen een integer als a en b integers zijn en bovendien b ≥ 0,
a/b is altijd van het type real,
a ÷ b is alleen gedefinieerd als a en b integers zijn en is dan ook een integer.

Wanneer in een assignment statement v := SAE v van het type real is en de waarde van SAE van het type integer, dan wordt de waarde van SAE als getal van het type real aan v toegekend; is v van het type integer en SAE van het type real, dan wordt de waarde van SAE afgerond op het dichtstbijzijnde integer dat dan aan v wordt toegekend (v is niet gedefinieerd als SAE binnen de rekennauwkeurigheid gelijk is aan integer + ½!!).

Opmerking: De betekenis van de toevoeging "simple" zal later duidelijk worden.

1.9 Voorwaardelijke uitvoering van statements

Wij hebben eerder gezegd, dat de statements worden uitgevoerd in de volgorde, waarin zij in de tekst voorkomen, tenzij anders is aangegeven. Wij zullen nu een voorbeeld geven, waarin inderdaad "anders is aangegeven". Stel, dat wij een programma moeten schrijven, dat weer twee getallen van de band moet lezen en deze om te beginnen moet printen, maar daarachter de som, wanneer het eerste getal kleiner is dan het tweede, edoch het verschil, wanneer zulks niet het geval is. Bij een opgave als deze weten wij niet van te voren, welke handelingen achtereen volgens door de machine moeten worden uitgevoerd, omdat dit mede bepaald is door de getallen die van de band gelezen zullen worden. Wij moeten in het programma dus op een of andere manier kunnen aangeven, hoe de machine de uit te voeren handeling kiest. Dit kan in Algol als volgt:

```
begin integer a1, a2, antwoord;
      a1 := read; a2 := read;
      if a1 < a2 then antwoord := a1 + a2 else antwoord := a1 - a2;
      print(a1); print(a2); print(antwoord)
end
```

Mogelijke uitvoer van dit programma is

```
+ 243179     + 456678     + 699857
```

of

```
+ 456678     + 243179     + 213499 .
```

De structuur van de derde regel van dit programma is

"if voorwaarde then statement 1 else statement 2";

de betekenis hiervan is, dat eerst onderzocht wordt, of aan de voorwaarde voldaan is, zo ja dan wordt daarna statement 1 uitgevoerd en statement 2 overgeslagen, zo nee dan wordt statement 1 overgeslagen en statement 2 uitgevoerd. De if - then - else constructie geeft ons de mogelijkheid om in het programma de keuze tussen twee alternatieven voor te schrijven. De hele constructie fungeert weer als statement en wordt derhalve van voorganger en opvolger in de tekst door een puntkomma gescheiden.

Om aan de variabele, genaamd y, de absolute waarde van de variabele, genaamd x, toe te kennen, kan men in een programma dus schrijven:

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

Stel nu, dat gevraagd wordt om de waarde van de variabele, genaamd x, te vervangen door zijn absolute waarde. Naar analogie zou men kunnen schrijven:

if x < 0 then x := - x else x := x ;

Dit is op zich zelf goed in de zin, dat deze statement het gewenste effect bewerkstelligt, maar het staat een beetje onnozel. Het is ook onpractisch, want de assignment statement x := x, die aan de variabele x de waarde toekent, die hij al heeft, heeft geen enkel effect, maar kost wel rekentijd! Wat je zou willen uitdrukken is "als x negatief is, wissel dan x van teken, anders hoef je niets te doen". Dit is een zoveel voorkomende situatie, dat het in Algol toegestaan is om "else statement 2" weg te laten. Men krijgt dan de constructie

"if voorwaarde then statement 1"

die behelst, dat onderzocht wordt, of aan de voorwaarde voldaan is: zo ja, dan wordt statement 1 wel uitgevoerd, zo nee, dan wordt statement 1 overgeslagen. Weer fungeert de hele constructie als statement en wordt hij in de tekst van voorganger en opvolger door een puntkomma gescheiden. Om zo de waarde van de variabele genaamd x door zijn absolute waarde te vervangen schrijft men dus in zijn programma:

if x < 0 then x := - x ;

(Opgave. Waarom is bovengegeven statement te verkiezen boven

if x ≤ 0 then x := - x ;

hoewel deze ook goed is in de zin, dat hij het gewenste effect bewerkstelligt?)

(Terzijde zij opgemerkt dat de vrijheid om het else deel van een if - then - else constructie weg te laten, onder omstandigheden in samengestelde constructies tot dubbelzinnigheden aanleiding kan geven. Ter voorkoming hiervan zijn bij nog te behandelen constructies vaak beperkende regels gesteld.)

1.10 Compound statements, dummy statements en commentaar

Bij de if - then en bij de if - then - else constructie hebben we dus de mogelijkheid om aan te geven, dat conditioneel (d.w.z. afhankelijk van het al dan niet vervuld zijn van een voorwaarde) een statement uitgevoerd, dan wel overgeslagen moet worden. Onmiddellijk rijst de vraag, hoe we hiervan gebruik kunnen maken, wanneer de handeling, die eventueel overgeslagen

zou moeten worden, zich in eerste instantie niet door één enkele statement, maar door een rijtje statements laat beschrijven.

Beschouw bv. de opgave "wissel x en y van teken als x negatief is". De variabelen x en y inconditioneel (d.w.z. onvoorwaardelijk) van teken wisselen, laat zich beschrijven door twee statements achter elkaar:

```
x := - x; y := - y ;
```

maar we mogen voor de conditionele tekenwisseling, zoals gevraagd, niet schrijven:

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

want dit zou betekenen, dat y altijd van teken wordt gewisseld! We moeten op een of andere manier aangeven dat het duo

```
x := - x; y := - y;
```

in zoverre een onscheidbaar geheel vormt, dat ze samen wèl, of samen nìet worden uitgevoerd. Het aangeven van een dergelijke groepering is de tweede (en belangrijkste) functie van het reeds eerder vermelde paar statementhaken begin en end. Door een rijtje statements door dit hakenpaar te omvatten, zorgt men, dat dit rijtje in de omgeving, waarin het staat, fungeert als één enkele statement, een zg. compound statement. Binnen het hakenpaar worden de "interne" statements als gebruikelijk weer door een puntkomma onderling gescheiden. We schrijven voor de conditionele tekenwisseling van x en y daarom:

```
if x < 0 then begin x := - x; y := - y end;
```

Het is nu duidelijk, waarom het symbolenpaar begin en end een hakenpaar genoemd wordt. Vergelijk nl. de algebraïsche uitdrukkingen a * b en a * (b + c); in het tweede product, waarin de laatste factor niet een enkele variabele, maar weer een samengestelde uitdrukking is, gebruikt men haakjes om aan te geven over welk stuk van de formule de beschrijving van de tweede factor zich uitstrekt.

Opmerking: Veel dubbelzinnigheid in het normale Nederlands is te herleiden tot de afwezigheid van haken. Een aanbieding "slechts drie voor een kwartje" kan gelezen worden als "slechts (drie voor een kwartje)", maar ook als "(slechts drie) voor een kwartje", waarmee het als onvoordelige aanbieding functioneert!

Een dummy statement bestaat uit een lege rij van Algol symbolen; bij uitvoering ervan is het effect nihil. Als S1 en S2 statements voorstellen, dan staat in de constructie S1; ; S2 tussen de twee puntkomma's een dummy statement. Evenzo staat in begin S1;.....Sn; end tussen de laatste puntkomma en end een dummy statement, zodat deze laatste puntkomma meestal weggelaten wordt.

Het is veelal gewenst om aan een programma ter verduidelijking verklarende tekst toe te voegen. Dit kan bv. gebeuren op de volgende manieren:

```
begin comment berekening; S1; S2;..... of
..... end berekening;
```

Het zo toegevoegde commentaar, altijd af te sluiten met een puntkomma, heeft geen effect bij uitvoering van het programma.

1.11 Herhaalde uitvoering van statements

Met de if - then - constructie hebben we de mogelijkheid om uit te drukken dat iets eventueel gebeuren moet; met de if - then - else - constructie dat of het een, of het ander gebeuren moet. We hebben echter nog niet de mogelijkheid om uit te drukken, dat iets onder een bepaalde voorwaarde herhaald moet worden uitgevoerd.

Laat ons ter toelichting aannemen dat er twee positieve getallen van de band gelezen moeten worden, zeg a en d, en dat de rest bepaald moet worden die overblijft bij deling van a door d. Om der wille van het voorbeeld leggen wij ons de beperking op, dat we niet van vermenigvuldiging en deling gebruik mogen maken en dat daarom de rest bepaald moet worden met de algorithme: d "zovaak als het gaat" van a af te trekken. Met de term "zovaak als" is al aangegeven, dat het hier om een herhaling van iets gaat. Wij geven eerst het programma, dat daarna besproken zal worden.

```
begin integer a, d, r;
      a := read; d := read; r := a;
      while r ≥ d do r := r - d;
      print(a); print(d); print(r)
end restbepaling;
```

In de eerste regel van dit programma worden drie integers gedeclareerd, a (voor het deeltal), d (voor de deler) en r (voor de rest). In de volgende regel worden twee getallen van de band gelezen, in volgorde het deeltal en

de deler. De grootheid, waarvan we steeds zoals eerder gezegd zullen proberen d af te trekken noemen we r (de "rest in wording"); de eerste keer is r gelijk aan het deeltal a. Daarna uiteraard achtereenvolgens a - d, a - 2d, etc. totdat uiteindelijk r < d is en dan niet meer met d verminderd kan worden.
Er zijn nu twee mogelijkheden voor r na uitvoering van de derde statement:

a) r < d of

b) r ≥ d .

In geval a) zijn we klaar omdat r niet meer met d verminderd kan worden. In geval b) kunnen we r nog tenminste één keer met d verminderen, overeenkomstig de statement

while r ≥ d do r := r - d;

Met het while statement wordt tot uitdrukking gebracht dat hierna het spel opnieuw moet worden uitgevoerd, dat wederom geïnspecteerd moet worden, of nu r soms klein genoeg is.

Opgemerkt moet worden dat volgens het officiële Algol rapport in plaats van het while statement het volgende gebruikt moet worden:

m: if r ≥ d then begin r := r - d; goto m end;

waarin goto m aangeeft, dat het rekenproces moet worden voortgezet bij het begin van de statement die door de "label" m: gemarkeerd of "gelabeld" is. Let op dat de goto statement de volgorde van uitvoering onvoorwaardelijk verandert ("onvoorwaardelijke sprong"). Omdat zowel de verlaging van r als de goto m alleen uitgevoerd moet worden als r ≥ d moeten beide statements tussen begin en end geplaatst worden.

In plaats van "m" hadden we elke willekeurige identifier mogen gebruiken, uitgezonderd "a", "d" en "r", die in dit programma al voorkomen. Hadden we tot nog toe identifiers alleen gebruikt om variabelen te benoemen, nu zien we, hoe identifiers ook als labels gebruikt worden om punten in het programma te markeren. Voor de duidelijkheid schrijven we een label naar links uitspringend.

Opmerking: Met het element van herhaling hebben we de mogelijkheid geïntroduceerd van quasi-algorithmen, d.w.z. iets dat er als algorithme uitziet, maar bij nader inzien een nimmer eindigende taak beschrijft. Ga in het voorbeeld na, wat er gebeurt, wanneer a positief, maar d abusievelijk negatief is!

Opmerking: We zullen meestal de modernere while vorm gebruiken, omdat het programma dan eenvoudiger te overzien is en labels niet nodig zijn.

Opgave: Verifieer dat met het volgende programma de som $\sum_{n=1}^{1000} (\frac{1}{n})^2$ wordt berekend:

```
begin integer n; real som;
      som := 0; n := 0;
      while n < 1000 do begin n := n + 1; som := som + 1/n ↑ 2 end;
      print(som)
end   sommatiereeks;
```

Het zal duidelijk zijn dat in dit geval waar de bovengrens van tevoren bekend is, het resultaat - ten koste van veel schrijfwerk - ook verkregen kan worden door de sommatie volledig uit te schrijven. De hier gegeven programmering met een "lus" is niet alleen eleganter, maar ook aangewezen wanneer bv.

- de bovengrens ten tijde van het schrijven van het programma (nog) niet bekend is,
- de bovengrens bepaald wordt in een programma dat aan het hier gegeven programma vooraf gaat,
- het aantal keren dat de lus doorlopen moet worden, afhangt van een of ander (convergentie)-criterium dat ook in de lus uitgewerkt wordt.

1.11.1 Een snellere algorithme

Tegen ons programma, dat de rest bepaalt, kan het bezwaar worden aangevoerd, dat het, hoewel in theorie goed, in de praktijk wel eens erg tijdrovend zou kunnen zijn, nl. als a vele malen groter is dan d. In de volgende versie zullen we aan dit bezwaar tegemoet komen; we nemen daarbij aan, dat verdubbelen en halveren tot het arsenaal toegestane operaties behoren (en delen nog steeds niet).

Ons versnellingsplan berust daarop, dat we om in grotere stappen te kunnen werken, eerst de deler zovaak verdubbelen, totdat hij groter is dan het deeltal, vervolgens deze verdubbelingen door halveringen weer te niet doen, maar en passant r ("de rest in wording") steeds, indien mogelijk, met het heersende veelvoud van de deler verminderen. We introduceren een nieuwe variabele dd (voor "dubbele deler").

```
begin integer a, d, r, dd;
      a := read; d := read; r := a; dd := d;
      while r ≥ dd do dd := 2 * dd;
      while dd ≠ d do begin dd := dd ÷ 2;
                            if r ≥ dd then r := r - dd
                      end;
      print(a); print(d); print(r)
end
```

In de eerste herhalingscyclus wordt dd (die begint met de waarde van de deler d) verdubbeld zolang $r \geq dd$ geldt; na afloop van de eerste herhalingscyclus geldt dat dus niet meer en dus

$$r < dd \ .$$

In de tweede herhalingscyclus wordt tussen begin en end eerst dd gehalveerd. Geldt aanvankelijk

$$r < dd \qquad (1)$$

na halvering van dd geldt

$$r < 2 * dd \ . \qquad (2)$$

In de volgende regel geldt of $r < dd$ of $r \geq dd$; in het laatste geval wordt r met dd verlaagd, zodat op grond van (2) weer (1), d.w.z. $r < dd$ geldt. Dit stuk wordt herhaald zolang $dd \neq d$ is; na afloop van de tweede herhalingscyclus moet dus $dd = d$ zijn. Uit (1) volgt daarmee

$$r < dd \ .$$

Aangezien r alleen met veelvouden van d is verlaagd en deze verlagingen met dd alleen maar plaatsvinden onder de voorwaarde ($r \geq dd$ nl.), dat hierdoor geen negatieve waarde van r ontstaat, volgt dat eveneens aan de relatie

$$0 \leq r$$

voldaan is en blijft.

Opgave: Voeg aan beide programma's een declaratie voor een variabele q en de statements toe, die aan q de waarde geven van het quotient, dat deling van a door d oplevert. Voor het eerste programma is dit het makkelijkst, voor het tweede is het het leukst!

1.12 Arrays

Een van de manieren, waarop men in de analytische meetkunde coördinaten kan benoemen is met x, y en z. Een andere manier om dit te doen is met x_1, x_2 en x_3. Het voordeel van de laatste notatie is, dat men dan ook een algemene naam heeft voor een coördinaat, nl. x_i. Het is duidelijk, dat deze naam alleen maar een van de drie coördinaten aanduidt als i = 1, i = 2 of i = 3.

Een analoge mogelijkheid om een rij geïndiceerde variabelen (bv. een vector of matrix) aan te duiden bestaat in Algol. Om te beginnen is er een notatietechnische kwestie: een Algol-tekst bestaat uit een opeenvolging van karakters, waarin het onaantrekkelijk is om een index (een "subscript") aan te geven door hem een beetje lager te schrijven: we willen er geen betekenis aan toekennen, wanneer karakters een beetje hobbelig op de lijn lijken te staan. In plaats van x_1, x_2, x_3, en algemeen x_i schrijven we daarom met gebruikmaking van de "rechte subscriptiehaken" [en]:

x[1], x[2], x[3] en algemeen x[i]

waarin x een "array identifier" heet, wat tussen de vierkante haken staat een subscriptlijst, en de verzameling geïndiceerde variabelen een "array". Als subscripts zijn willekeurige arithmetic expressions toegestaan; de waarde van een subscript is dan de integer die verkregen wordt door de waarde van deze expression af te ronden naar het dichtstbijzijnde gehele getal. Alle subscripted variables van een array moeten van hetzelfde type zijn en kunnen als zodanig in willekeurige expressions voorkomen.

Voorbeelden:

```
p[1]          p[2 * n - 1]
a[1,2]        a[i,j + 1]
b[5,3,-2]     b[1.2,12 - 1, p + q + r]
```

In de verplichte declaratie van een array moet men aangeven de identifier van het array, het aantal subscripts en per subscript onder- en bovengrens van de subscript. (In de volgende toepassingen worden alleen zg. ééndimensionale arrays gebruikt, d.w.z. arrays met één subscript, het analogon van vectoren. Er zijn in Algol ook meer-dimensionale arrays toegestaan: het twee-dimensionale array, waarin elk element door twee subscripts geïdentificeerd wordt, is het analogon van de matrix.) In de declaratie vermeldt men eerst het type van de subscripted variables, dan de array identifier en dan tussen de vierkante haken de subscript grenzen. De ondergrens en boven-

grens worden gescheiden door een dubbele punt (ondergrens ≤ bovengrens!); de grenzen voor verschillende subscripts door een komma.

Voorbeelden:

```
real array    p[0 : 4], q[1 : 10], r[-1 : 0, 1 : 8]
integer array a[-4 : 7, 2 : 13]
real array    b[1 : 10, 3 : 3, -7 : -1]
integer array c[10 : 3, 4 : 7] is niet toegestaan. Waarom?
```

Een subscripted variable is alleen gedefinieerd als het aantal subscripts overeenstemt met het aantal grenzenparen in de array declaration en als bovendien de waarde van het subscript niet buiten de opgegeven grenzen ligt. De arraygrenzen mogen getallen of variabelen zijn, mits de laatsten weer reeds eerder gedeclareerd zijn en in vorige assignment statements een waarde hebben gekregen. Onder dezelfde condities zijn zelfs arithmetic expressions toegelaten als array grenzen.

Opmerking: Het is toegestaan om de volgende meervoudige assignments op te schrijven

```
a[i] := i := 4;
```

De afspraak is in zo'n geval dat de oude waarde van i (bijv. 3) gebruikt wordt voor de index van a, zodat zowel a[3] als i dus de waarde 4 krijgen.

1.12.1 Toepassingen

In het volgende voorbeeld worden 100 gehele getallen van de band gelezen en in omgekeerde volgorde geprint. In de simpelste versie luidt dit programma bv. als volgt:

```
begin integer i; integer array a[0 : 99];
      i := 0;
      while i < 100 do begin a[i] := read; i := i + 1 end
      i := 100;
      while i > 0 do begin  i := i - 1; print(a[i]) end
end
```

In de eerste regel hebben we behalve de integer i de honderd integer variabelen a[0] t/m a[99] gedeclareerd. In de lees-cyclus is i gelijk aan "het aantal gelezen getallen" en als we afspreken, dat we het array in volgorde van opklimmende subscripten zullen vullen, tevens gelijk aan "de subscriptwaarde van het array-element, dat aan de beurt is om de volgende waarde te krijgen". In de printcyclus is i gelijk aan "het aantal getallen, dat nog geprint moet worden". Dat voor deze printcyclus i op 100 gesteld wordt, lijkt in dit voorbeeld misschien overbodig maar is toch in het algemeen veiliger (bij latere wijzigingen van het programma zal men misschien tussen de twee while-statements andere statements toevoegen die de waarde van i beïnvloeden).
Tenslotte zij nog opgemerkt dat aan het eind van het programma tweemaal end voorkomt om de aantallen open- en sluithaken weer met elkaar in overeenstemming te brengen.

We gaan dit programma nu op twee manieren wijzigen: we willen dat er bij het uitprinten slechts 7 getallen op een regel komen. (Omdat we 100 getallen hebben, geeft dat 14 volle regels plus een 15-de regel met nog 2 getallen.) We mogen voor de overgang op een nieuwe regel gebruik maken van de statement nlcr (d.w.z. "new line carriage return"), welke handeling in de programmabibliotheek onder die naam bekend verondersteld wordt. Bovendien willen we op de 16-de regel maximum- en minimumwaarde van de gelezen getallen printen.

We voeren daartoe drie extra variabelen in:
j: het aantal getallen dat op de onderhavige regel nog geprint kan worden;
max: de (tot nog toe gevonden) grootste waarde;
min: de (tot nog toe gevonden) kleinste waarde.

Zoals gezegd zou j de betekenis hebben van het aantal getallen, dat nog op de onderhavige regel geprint kan worden. Aanvankelijk is j = 7; elke keer, dat een getal geprint wordt, wordt j daarom met 1 verlaagd. Komt daarbij j op nul, dan is de onderhavige regel dus vol, wordt dus de overgang op een nieuwe regel ingelast en wordt j, in overeenstemming met zijn betekenis weer op 7 gezet. (We hebben er in dit programma duidelijk gebruik van gemaakt, dat als we niets over nieuwe regels voorschrijven, het eerste getal, dat door het programma geprint wordt, op het begin van een (nieuwe) regel geprint wordt.)

```
begin integer i, j, max, min; integer array a[0 : 99]; i := 0;
            while i < 100 do begin a[i] := read; i := i + 1 end;
            i := 100; j := 7;
            while i > 0 do begin i := i - 1; print(a[i]);
                                 j := j - 1; if j = 0 then begin nlcr; j := 7 end
                        end;
            max := min := a[0]; i := 0;
            while i < 99 do begin i := i + 1;
                                 if a[i] > max then max := a[i] else
                                 if a[i] < min then min := a[i]
                        end;
            nlcr; print(max); print(min)
end
```

Na het printen van de 100 getallen worden in de volgende lus maximum en minimum gezocht. De integer i heeft nu de betekenis van "wijzer naar de plaats in het array a waar het laatst onderzochte getal staat". Steeds is max = "het maximum van de eerste i + 1 getallen" en min = "het minimum van de eerste i + 1 getallen". Zolang i < 99 is, zijn nog niet alle getallen onderzocht en moet het volgende getal onderzocht worden om te kijken of dat soms groter is dan max of kleiner dan min; zo ja, dan moeten de waarden van max of min worden aangepast. Tenslotte worden de laatste twee uitkomsten op een nieuwe regel geprint.

Als we de getallen niet in omgekeerde volgorde hadden hoeven te printen en alleen maar gevraagd was om maximum en minimum te bepalen, dan hadden we het hele array niet nodig gehad. In plaats daarvan hadden we kunnen volstaan met een enkele variabele (die we gemakshalve maar weer "a" noemen) voor de waarde van het laatstelijk van de band gelezen getal.

```
begin integer i, max, min, a;
      a := read; i := 1; max := a; min := a;
      while i < 100 do begin a := read; i := i + 1;
                              if a > max then max := a else
                              if a < min then min := a
                        end;
      print(max); print(min)
end
```

Hier is:

a = het laatst gelezen getal van de band;

i = het aantal van de band gelezen getallen;

max = het tot nog toe grootste gevonden getal;

min = het tot nog toe kleinste gevonden getal.

1.12.2 Switch

In 1.9 is er op gewezen dat de if - then en de if - then - else constructies een keuze mogelijk maken tussen twee mogelijkheden. De if - then constructie kiest tussen wèl of niet, de if - then - else tussen het een óf het ander.

Willen we nu een keuze maken uit bv. 4 statements, dan kunnen we dat als volgt noteren:

```
     if voorwaarde 1 then S1
else if voorwaarde 2 then S2
else if voorwaarde 3 then S3 else S4;
```

Opmerking: Er wordt altijd precies een (eventueel compound) statement uitgevoerd. Als in deze constructie het 4^e statement zal worden uitgevoerd, dan worden ook de drie voorgaande voorwaarden getest. In het algemeen heeft Algol 60 hier geen kortere oplossing voor (nieuwere talen, bv. Algol 68, wel), maar in sommige gevallen is een andere constructie wel mogelijk. Als de uitvoering van de statements S_1 t/m S_4 gekoppeld is aan een rij opeenvolgende integerwaarden, dus

```
if i = 1 then S1 else if i = 2 then S2 else ..... else S4;
```

dan kunnen we gebruik maken van een switch.

Een switch is a.h.w. een array van labels, bv. als volgt gedeclareerd:

```
switch s := a, b, c, d;
```

Na het symbool switch komt een willekeurige identifier, de switch identifier, gevolgd door := en een rij van labels of switch list. Door dit "statement" is a.h.w. het "switch array" s gedefinieerd:

s[1] := a, s[2] := b, s[3] := c en s[4] := d.

Het "switch array" mag niet apart als array gedeclareerd worden; er worden ook geen grenzen aangegeven, want die volgen vanzelf uit de switch declaraties. Na deze declaratie is het statement: goto s[1] equivalent met: goto a. In het algemeen mag tussen de vierkante haken een arithmetische expressie staan; is de waarde ervan ≤ 0 of groter dan het aantal labels in de switch list, dan werkt de goto s[ ] statement als een dummy statement. Het gebruik van een switch zou nu bv. als volgt kunnen zijn:

```
begin integer i; switch s := a, b, c, d;
      i := read;
      if i ≥ 1 then
         begin if i ≤ 4 then
               begin goto s[i];
               a : statement 1; goto e;
               b : statement 2; goto e;
               c : statement 3; goto e;
               d : statement 4;
               e : statement 5;
               end
         end
end switch voorbeeld;
```

Na de declaratie van i en s lezen we i van de band. Alleen als $1 \leq i \leq 4$ is, willen we statement i gevolgd door statement 5 uitvoeren. Na ieder van de statements 1, 2 en 3 is de verwijzing goto nodig, omdat anders het volgende statement (resp. 2, 3 en 4) zou worden uitgevoerd, en dat was ons doel niet.

Opmerking: In het voorbeeld is niet sprake van een keuze uit vier, maar uit vijf mogelijkheden; het is ook mogelijk dat er niets gebeurt.

1.13 Logische expressies, variabelen en operatoren

Tussen if en then zijn wij tegengekomen wat wij "voorwaarden" genoemd hebben. Voorbeelden van dergelijke voorwaarden waren: $x < 0$, $i < 100$, $i > 0$, $j = 0$, etc., en we hebben gesteld, dat onderzocht moet worden of aan een dergelijke voorwaarde voldaan is. In deze sectie gaan we dit onderzoek beschrijven als "berekening", maar onmiddellijk rijst de vraag: "Berekening van wat?". Het volgende is reeds bekend: als x en y gegeven numerieke waarden zijn, dan weten we wat met x + y bedoeld is, nl. weer een numerieke waarde en wel de som. Omdat de som weer een numerieke waarde is, heet + een zg. "arithmetische operator"; andere voorbeelden van arithmetische operatoren zijn −, * en :.

Er zijn echter ook andere operatoren, die wel op arithmetische operanden opereren maar niet een numeriek resultaat afleveren. Dit zijn de zg. "relational operators": =, ≠, >, <, ≥, ≤ in bijvoorbeeld:

x = y	"x gelijk y"
x ≠ y	"x ongelijk y"
x > y	"x groter dan y"
x ≥ y	"x tenminste y"
x < y	"x kleiner dan y"
x ≤ y	"x tenhoogste y" .

Deze relational operators verbinden twee numerieke operanden en beschrijven daarmee een voorwaarde, waaraan al dan niet voldaan kan zijn. We kunnen ze ook interpreteren als bewering, die al of niet waar is. Het onderzoek of de "relatie" x = y geldt, kunnen we ook beschrijven als het vaststellen van de waarheid, of nog pregnanter het berekenen van de waarheidswaarde (truth value) van de bewering x = y. Hiermee zijn de relational operators geworden tot operatoren, die als domein hebben de (geordende) getallenparen en als bereik de mogelijke waarheidswaarden. Dit laatste bereik bevat slechts twee verschillende waarden, die we als true resp. false noteren.

We kenden arithmetische expressies (nl. arithmetische variabelen, getallen of arithmetische expressies verbonden door arithmetische operatoren); hiernaast onderkennen we nu de zg. logische of boolean expressies (voorshands arithmetische expressies verbonden door een relational operator of een waarheidswaarde). Wat de getallen zijn voor de arithmetische expressies zijn de logische waarden true en false voor de logische expressies, nl. een opsomming van het waardebereik.

Voorbeelden:

boolean expression	value
x = x	true
x = x + 1	false
1 ≥ 2	false
a = 2	true als a = 2, anders false
x ≥ x + 10	false
(x + 3) * (x - 2) ≥ 0	false als -3 < x < 2, anders true

Opmerking: De betekenis van de if - then - else - constructie kunnen we nu als volgt herformuleren:

"if true then S1 else S2" is equivalent met "S1" en
"if false then S1 else S2" is equivalent met "S2".

De analogie tussen arithmetische en logische expressies gaat verder. Elke waarde van een arithmetische expressie kan aan een arithmetische variabele worden toegekend. Na de declaratie

```
integer x, y, som, verschil;
```

mogen (nadat x en y een waarde hebben gekregen) de assignment statements

```
som := x + y; verschil := x - y;
```

voorkomen. Geheel analoog mogen na de declaraties

```
integer x, y; boolean x groter, x kleiner;
```

nadat x en y een waarde hebben gekregen de assignment statements

```
x groter := x > y; x kleiner := x < y;
```

voorkomen (ook hier zijn de namen van de "boolean variables" x groter en x kleiner willekeurig te kiezen, maar liefst zo dat ze een geheugensteuntje vormen).

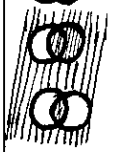
Zoals de assignment statement som := x + y in de variabele genaamd som de som van de nu heersende waarden van x en y fixeert (waarna x en/of y van waarde mogen veranderen), zo legt de assignment statement x groter := x > y (links een boolean variabele, rechts een boolean expression) vast of nu de heersende waarde van x groter is dan die van y (waarna x en/of y van waarde mogen veranderen).

In de declaratie zijn we het nieuwe symbool boolean tegengekomen. De declaratie van variabelen heeft nl. een dubbele functie: als we een variabele declareren wordt niet alleen de hem toegevoegde identifier vastgelegd, maar met behulp van de zg. declarator (integer, real of boolean) ook het bijbehorende waardebereik van de variabele. Het werken met boolean variabelen kan tijdsbesparing opleveren, maar vooral ook een korte en duidelijke programmatekst.

Naast de relational operators - die telkens twee arithmetische operanden versmelten tot een logisch resultaat - kent Algol bovendien enkele zg. logische operatoren die in het algemeen tussen twee logische operanden geplaatst weer een logisch resultaat opleveren. Bij het samenstellen van deze logische expressies mogen niet twee logische operatoren direct achter elkaar geplaatst worden met de uitzondering dat de not operator mag volgen op een van not ver-

schillende operator.

De betekenis van de logische operatoren volgt uit de volgende tabel:

operator \ b_1 / b_2	f f	f t	t f	t t	alternatief symbool	naam	Venn diagram	dagelijkse formulering
$\neg b_1$	t	t	f	f	not	negatie		niet
$b_1 \wedge b_2$	f	f	f	t	and	conjunctie		èn
$b_1 \vee b_2$	f	t	t	t	or	disjunctie		b_1 of b_2
$b_1 \supset b_2$	t	t	f	t	$\rightarrow$	implicatie		als b_1 dan b_2
$b_1 \equiv b_2$	t	f	f	t	$\leftrightarrow$	equivalentie		b_1 alleen dan, indien b_2
$b_1 - b_2$	f	f	t	f		differentie		b_1 wel, b_2 niet
$b_1 \downarrow b_2$	f	t	t	f	eor	antivalentie		ôf b_1, ôf b_2 (niet beide)
$b_1 \updownarrow b_2$	t	f	f	f	nor	Peirce functie		noch b_1, noch b_2
$b_1 \leftrightarrow b_2$	t	t	t	f	nand	Sheffer functie		b_1 niet, of b_2 niet

Opmerking: In de onderste helft van deze tabel zijn nog enkele minder vaak gebruikte logische operatoren vermeld, die dan ook niet in Algol voorkomen. Het aardige van de twee laatste operatoren is o.a. dat de andere operatoren tot één van deze twee alleen herleid kunnen worden, bv.

$b_1 \vee b_2$ is equivalent met $(b_1 \updownarrow b_2) \updownarrow (b_1 \updownarrow b_2)$

$b_1 \wedge b_2$ is equivalent met $(b_1 \updownarrow b_1) \updownarrow (b_2 \updownarrow b_2)$

Let verder op het verschil tussen de disjunctie (inclusive or) en antivalentie (exclusive or).

De volgorde van uitwerken van logische expressies is van links naar rechts met inachtneming van de volgende prioriteitsvolgorde (van hoog tot laag): relaties, not, and, or, $\supset$ en $\equiv$.

Let er bij het schrijven van expressies op dat tussen logische operanden geen = teken mag worden geplaatst; meestal is dan de fout te herstellen door het $\equiv$ teken te gebruiken. Zo is de dubbele voorwaarde $0 < x < 1$ te schrijven als $0 < x \equiv x < 1$.

Immers is

$x \leq 0$ dan is $0 < x$ false en $x < 1$ true, het geheel false,
$0 < x < 1$ dan is $0 < x$ true en $x < 1$ true, het geheel true,
$x \geq 1$ dan is $0 < x$ true en $x < 1$ false, het geheel false.

Een natuurlijker formulering van $0 < x < 1$ in Algol is overigens

$$0 < x \wedge x < 1$$

Opgave: Verifieer zelf de volgende expressies:

$(\underline{\text{not}}\ x = y) \equiv x \neq y$ $\qquad$ $x \leq y \wedge x \geq y \equiv x = y$
$(\underline{\text{not}}\ x < y) \equiv x \geq y$ $\qquad$ $x < y \vee x > y \equiv x \neq y$
$(\underline{\text{not}}\ x > y) \equiv x \leq y$ $\qquad$ $\underline{\text{not}}\ (B_1\ \underline{\text{and}}\ B_2) \equiv (\underline{\text{not}}\ B_1)\ \underline{\text{or}}\ (\underline{\text{not}}\ B_2)$

1.13.1 Toepassingen

We beschouwen de volgende opgave. Op een band staan voor $0 \leq n \leq 100$ in volgorde $2n + 1$ getallen, nl.

$$n, x_1, \ldots, x_n, y_1, \ldots, y_n .$$

(Als $n = 0$, dan bevat de band alleen het getal 0; als $n = 1$, dan bevat de band 1, x_1, y_1, etc.) Deze band moet worden gelezen en ons programma moet vaststellen of het rijtje x-en elementsgewijs aan het rijtje y's gelijk is. (Hierbij gelden "lege rijtjes", d.w.z. $n = 0$, als gelijk: ze bevatten nl. geen overeenkomstige elementen, die verschillen!) Zo ja, dan moet het programma "gelijk" printen, zo nee, dan moet het "ongelijk" printen. We geven eerst het programma.

```
begin integer i, n; integer array x, y[1 : 100]; boolean equal;
      n := read;
      i := 0;
      while i < n do begin i := i + 1; x[i] := read end;
      i := 0;
      while i < n do begin i := i + 1; y[i] := read end;
      i := 0; equal := true;
      while i < n and equal do
                  begin i := i + 1; equal := (x[i] = y[i]) end;
      if equal then printtext(‡gelijk‡) else printtext(‡ongelijk‡)
end
```

In de eerste regel worden o.a. twee arrays x en y gedeclareerd, beide ter lengte 100 (dat is in elk geval lang genoeg). In de eerste lus wordt het rijtje x-en elementsgewijs gelezen en in volgorde in het array x opgeslagen; bij de test heeft i de betekenis "het aantal inmiddels gelezen x-en". De volgende lus leest op analoge wijze de y's van de band. In de test van de laatste cyclus heeft i de betekenis "het aantal inmiddels vergeleken x-y-paren" en "equal" heeft de betekenis "bij de inmiddels vergeleken x-y-paren is geen verschil geconstateerd". De voorwaarde

```
i < n and equal
```

drukt uit dat, om door te gaan met vergelijken van het volgende paar, er aan twee voorwaarden voldaan moet zijn: de rijtjes moeten nog niet zijn uitgeput en bovendien moet nog geen verschil geconstateerd zijn. Zodra nl. wel verschil geconstateerd is, is de uitslag bekend en kan de elementsgewijze vergelijking gestaakt worden.

In de laatste regel wordt een beroep gedaan op de wederom in de bibliotheek bekend veronderstelde handeling genaamd printtext, die zijn (ene) argument zoals gebruikelijk meekrijgt tussen ronde haken. Dit argument is voor printtext een zg. "string"; de karakters ‡ en ‡, "string open" en "string close", worden gebruikt om begin en einde van de string aan te geven; zij verschijnen zelf niet op papier, de karakters er tussen wel.

1.14 De opbouw van een groter programma

Als volgende voorbeeld behandelen we de opgave om de eerste 1000 priemgetallen (te beginnen bij 2) uit te rekenen en te printen. De structuur van dit programma zal de volgende gedaante hebben:

```
begin integer array p[1 : 1000];
      "vul het array p met de opklimmende priemgetallen";
      "print het array p elementsgewijs uit"
end
```

en vervolgens gaan we deze twee tussen aanhalingstekens benoemde handelingen nader uitwerken. Voor de eerste bedenken we, dat na het eerste priemgetal (= 2) alle volgende priemgetallen oneven zijn. Het zal blijken, dat we dit gegeven goed kunnen gebruiken en in de eerste uitwerking van "vul het array p met de opklimmende priemgetallen" zullen we het enige even priemgetal apart behandelen.

Onder aanname van de aanvullende declaratie

```
integer k, j
```

luidt de uitwerking:

```
p[1] := 2; k := 1; j := 1;
while k < 1000 do
   begin "verhoog oneven j tot het volgende oneven priemgetal";
         k := k + 1; p[k] := j
   end
```

Hier heeft k de betekenis "het aantal ingevulde priemgetallen", j is altijd oneven en (na de eerste slag) gelijk aan het laatst ingevulde priemgetal.

Nu zitten we nog met de handeling "verhoog oneven j tot het volgende oneven priemgetal". Met invoering van de declaratie

```
boolean j deelbaar
```

kunnen we dit de uitwerking geven:

```
b: j := j + 2;
   "geef aan j deelbaar de waarde: j is deelbaar";
   if j deelbaar then goto b
```

Dat we hier j meteen met stapjes van 2 kunnen laten oplopen danken we aan het feit, dat we ons door de aparte behandeling van het enige even priemgetal verder tot oneven priemgetallen kunnen beperken.

Pas voor de uitwerking van de handeling "geef aan j deelbaar de waarde: j is deelbaar" komt enige algebra om de hoek kijken. We gaan kijken of we van de (oneven) j een (eveneens oneven) priemfactor kunnen vinden: de priemfactoren, die we willen proberen staan echter al in het inmiddels gevulde stuk van het array p! Omdat we alleen oneven waarden van j onderzoeken hoeven we p[1] = 2 als mogelijke factor niet te proberen en is p[2] de kleinste factor, die mogelijk in aanmerking komt. Verder heeft elk deelbaar getal tenminste 1 priemfactor, waarvan het kwadraat niet groter is dan dat getal en op grond van deze twee overwegingen kan het zoeken naar factoren van j zich beperken tot het proberen van

```
p[2].............p[ord - 1]
```

als ord de kleinste index is, en dus p[ord] het kleinste priemgetal is, zodat

$$p[ord]^2 > j \ .$$

Na de declaratie

```
integer ord, m
```

en de veronderstelling, dat aanvankelijk, toen j = 1 was, ook ord = 1 gemaakt is, kunnen we "geef aan j deelbaar de betekenis: j is deelbaar" de volgende uitwerking geven:

```
while p[ord] * p[ord] ≤ j do ord := ord + 1;
m := 2; j deelbaar := false;
while m < ord and not j deelbaar do
begin "geef aan j deelbaar de betekenis: p[m] is een factor van j";
      m := m + 1
end
```

In de eerste lus wordt (zo nodig) ord opgehoogd, daarna worden (zo nodig, d.w.z. voor j ≥ 9; ga dat na!) priemgetallen uit de tabel als factor geprobeerd, maar dit zoekproces strekt zich ten hoogste uit over de priemgetallen

```
p[2].....p[ord - 1]
```

en wordt onmiddellijk afgebroken, zodra een factor van j gevonden is. (In bovenstaande uitwerking worden de priemgetallen in opklimmende volgorde geprobeerd omdat de kans dat een willekeurige oneven j door een priemgetal deelbaar is, groter is naarmate het priemgetal kleiner is.)

Ten leste hebben we de uitwerking van "geef aan j deelbaar de betekenis: p[m] is een factor van j". Dit is het geval, wanneer het op gehelen afgeronde quotiënt gelijk is aan het quotiënt zelf. Het op gehelen afgeronde quotiënt kunnen we na de declaratie

```
integer q
```

aan de variabele q toekennen door de assignment statement

```
q := j ÷ p[m]
```

De uitwerking van "geef aan jdeelbaar de betekenis: j is deelbaar door p[m]" wordt daarmee tot

```
q := j ÷ p[m]; jdeelbaar := (j = q * p[m]) .
```

Rest ons de uiteindelijke samenstelling van het programma, d.w.z. het verzamelen van de declaraties en het substitueren van de opeenvolgende uitwerkingen; het printen van de priemgetallen gebeurt hier met 8 getallen per regel

```
begin integer array p[1 : 1000]; integer k, j, ord, m, q; boolean jdeelbaar;
      p[1] := 2; k := j := ord := 1;
      while k < 1000 do
            begin b: j := j + 2;
                  while p[ord] * p[ord] ≤ j do ord := ord + 1;
                  m := 2; jdeelbaar := false;
                  while m < ord and not jdeelbaar do
                        begin q := j ÷ p[m];
                              jdeelbaar := (j = q * p[m]);
                              m := m + 1
                        end;
                  if jdeelbaar then goto b;
                  k := k + 1; p[k] := j
            end;
      k := 0; j := 8;
      while k < 1000 do begin k := k + 1; print(p[k]);
                              j := j - 1; if j = 0 then begin j := 8; nlcr end
                        end
end priemgetallen;
```

Opgave: Onderzoek hoe de goto b vervangen kan worden door een while.

1.15 Simple expressions, expressions en conditional expressions

In het voorgaande is reeds vermeld dat zg. simple (arithmetic en boolean) expressions opgebouwd worden uit values, variables, operators en ronde haakjes. Met invoering van de zg. if clause:

```
if Boolean Expression then
```

zijn deze simple expressions uit te bouwen tot "conditional" expressions. Hiermee kunnen we de waardetoekenning aan een variabele (van het type real, integer of boolean) afhankelijk maken van een boolean expression.

Voor een real of integer variable x ziet dit er uit als

x := if BE then SAE else AE

(waarin de afkortingen BE, SAE en AE hopelijk voor zichzelf spreken) en voor een boolean variable als

b := if BE then SBE else BE

Voorbeelden zijn y := if x ≥ -2 then x + 2 else -x - 2 als equivalent van y := |x + 2| en c max := if a > b then c > a else c > b wanneer c = max(a,b,c). In beide gevallen is achter het then symbool slechts een simple expression geoorloofd en moeten de expressions achter then en else van hetzelfde type zijn (beide arithmetic of beide boolean). In conditionele expressions mag het else gedeelte niet weggelaten worden zoals wel mag bij conditionele statements:

if B then S_1 else S_2;
if B then S_1;

(In het laatste geval is met een voorbeeld eenvoudig te demonstreren dat S_1 niet weer een voorwaardelijke constructie mag zijn; immers

if B_1 then if B_2 then S_1 else S_2;

kan worden uitgelegd als

if B_1 then (if B_2 then S_1) else S_2;

of als

if B_1 then (if B_2 then S_1 else S_2);

welke uitdrukkingen - zoals eenvoudig te verifiëren is - niet equivalent zijn.)

Dat achter then simple expressions moeten staan, is overigens geen ernstige beperking omdat conditional expressions door ze tussen ronde haakjes te plaatsen simple expressions worden.

Voor en achter conditional expressions mogen geen arithmetische of logische operatoren staan, wel achter simple expressions!

1.16 For-statement

In het eerder behandelde voorbeeld $\sum_n (1/n)^2$ hebben we reeds kennis gemaakt met een programmalus, gekenmerkt door het feit dat een aantal statements een vooraf bekend aantal malen doorlopen moet worden en dat bovendien n hier direct in de berekening betrokken wordt. Bij $\sum_n a_n$ treedt daarentegen n alleen als index op.

Willen we de algebraïsche uitdrukking **f(x) voor een gegeven aantal,** overigens willekeurige,waarden van x berekenen, dan ligt het voor de hand om dit ook met een programmalus te doen, die een aantal malen doorlopen wordt. In andere gevallen is het aantal malen, dat een lus doorlopen moet worden, niet van te voren vastgelegd maar wordt tijdens de berekening bepaald door een of ander convergentiekenmerk. Dit gebeurt bijvoorbeeld bij de iteratieve bepaling van de wortel van een polynoom f(x) met de methode van Newton:

$$x_{i+1} := x_i - f(x_i)/f'(x_i)$$

uitgaande van een of andere startwaarde x_0. We houden dan bijvoorbeeld met het iteratieproces op wanneer het verschil tussen x_{i+1} en x_i kleiner is geworden dan een gegeven ε.

Al deze cyclische processen zijn met een if - constructie te programmeren, maar iets eenvoudiger en overzichtelijker gaat het met de zg. for-statement, waarvan hier de drie varianten besproken zullen worden.
De for-statement wordt opgebouwd uit één enkel statement (eventueel een compound statement), voorafgegaan door een "for-clause". Deze bestaat uit het symbool for, gevolgd door een variabele van het type real of integer, gevolgd door het symbool := en dan een aantal zg. "elementen", onderling door komma's gescheiden, en tenslotte het symbool do. De elementen die kunnen voorkomen (en ook met elkaar gecombineerd mogen worden) zijn:

arithmetic element: arithmetic expression
step-until-element: p step q until r
} handig voor lussen die een vast aantal keren doorlopen worden

while element : a while b — handig voor een variabel aantal doorlopen

Hierin zijn p, q en r arithmetic expressions, bij voorkeur (om afrondingsproblemen te vermijden) van het type integer; a is een arithmetic expression en b een boolean expression. Bij de uitvoering van de for-statement worden de elementen in de neergeschreven volgorde afgewerkt volgens de volgende regels:

a) de statement for v := p,q,... do S is gelijkwaardig met de reeks statements (p,q,... zijn in het algemeen arithmetische expressies)

```
v := p; S; v := q; S; .....
```

(die meestal alleen maar zinvol is als S de variabele v bevat).

b) de statement for v := a while b do S is gelijkwaardig met de reeks statements

```
m : v := a; if b then begin S; goto m end
```

(meestal slechts zinvol als v, dan wel de (compound) statement S ook de waarde van de boolean expression kunnen veranderen omdat we anders nooit in of uit S komen. Ook v zal in de regel door S van keer tot keer veranderen.).

c) de statement for v := p step q until r do S is gelijkwaardig met de reeks statements

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

(zelfde opmerking als hierboven). De waarde van v moet na afloop als onbekend worden beschouwd!

Het is duidelijk dat het step-until element altijd te vervangen is door het while element (en omgekeerd); in het algemene geval bv. door

```
if (r-p) * q ≥ 0 then begin for v := p, v + q while(r-v) * q ≥ 0 do S end
```

of in het eenvoudiger geval

```
for v := 1 step 1 until 10 do S  door
for v := 1, v + 1 while v ≤ 10 do S.
```

De enkele statement S mag in al deze gevallen ook weer een for statement zijn, maar een for statement mag niet op een if clause volgen (omdat dit soms dubbelzinnigheden geeft, bv. bij

```
if b1 then for ..... do if b2 then S1 else S2
```

waar hoort nu else S_2 bij?).

Opmerking: De afwerking van de for statement kan door een goto statement in S afgebroken worden! Deze constructie moet echter vermeden worden omdat dit aanleiding geeft tot moeilijk verifieerbare programma's.

Voorbeelden:

1) Bereken $y = x^2 - x + 1$ voor

$x = 0.1, 0.2, 0.30\ (0.01)\ 0.40\ (0.1)\ 0.7, 1.0.$

Oplossing:

```
for x := 0.1, 0.2, 0.30 step 0.01 until 0.40, 0.5 step 0.1 until
0.7, 1.0 do y := x ↑ 2 - x + 1.
```

2) Bereken

$$\ell n(1 + x) = \sum_{n} (-1)^{n+1} \frac{x^n}{n} \qquad (0 < x < 1)$$

wanneer we zouden mogen aannemen (zoals bekend is dit een slechte methode!) dat

a) m termen voldoende zijn, of dat

b) afbreken is toegestaan zodra een $|\text{term}| \leq 10^{-3}$ wordt.

Oplossing:

a)
```
ℓn := factor := x
for n := 2 step 1 until m do begin factor := factor * (-x);
                                   ℓn := ℓn + factor/n
                              end
```

b)
```
ℓn := factor := absterm := x; eps := 0.001; n := 1
for n := n + 1 while absterm > eps do
    begin factor := factor * (-x);
              ℓn := ℓn + factor/n;
          absterm:= if factor < 0 then - factor/n else factor/n
    end
```

In uitbreiding op de officiële Algol-60 taal zijn bij sommige rekencentra twee extra statements toegestaan, te weten de zg. repeat-statement

```
do S until B;
```

en de reeds eerder behandelde while-statement

```
while B do S;
```

waarin zoals in het voorgaande S een (compound) statement is en B een Boolean expression. De repeat statement is equivalent met

```
m: S;        (deze statement wordt dus tenminste één keer uitgevoerd!)
   if not B then goto m;
```

en de while statement met

```
m: if B then begin S; goto m end
```

Deze twee statements zijn juist ingevoerd om een herhalingscyclus eleganter te beschrijven dan met if- of for-statements mogelijk is, vooral wanneer niet a priori bekend is hoe vaak een cyclus doorlopen moet worden.

Wanneer bijvoorbeeld de opgave luidt om in een integer array a[i] (i = 1,...,n) te bepalen of er een index i is (en zo ja wat de kleinste is) van een array element dat gelijk is aan een gegeven integer e, dan kan dit met een while statement

```
found := true; i := 0;
while i < n and found do begin i := i + 1; found := a[i] ≠ e end;
```

of met een repeat statement

```
do begin i := i + 1; found := a[i] ≠ e end until i ≥ n or found;
```

of met een for statement

```
for k := 1, k + 1 while k ≤ n and found do
    begin found := a[k] ≠ e; i := k end;
```

Om de verschillende statements ondubbelzinnig te maken, gelden de volgende restricties:

- tussen bij elkaar horende then en else mag geen for- of while-statement staan (tenzij ingekapseld in een compound statement);
- labels in een compound statement S achter een do mogen niet van buiten af met behulp van een goto statement bereikt worden.

1.17 Block structuur

Een block is een compound statement waarin tussen het eerste symbool begin en de eerste statement één of meer declaraties staan, die betrekking hebben op een aantal in het compound statement voorkomende grootheden (bv. variabelen). Grootheden in het block, die niet in het block gedeclareerd zijn, heten globaal t.o.v. dit block; zij dienen reeds gedeclareerd te zijn in een omvattend block.

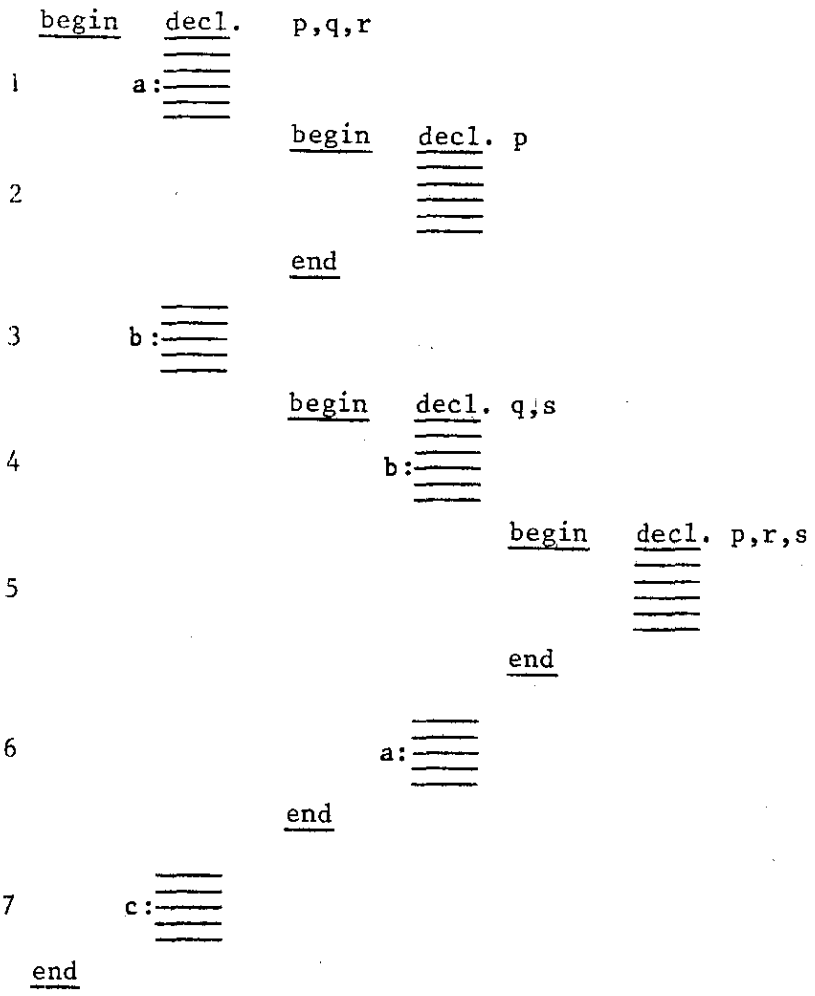
Grootheden die wel in het block gedeclareerd zijn, heten lokaal t.o.v. het block, hetgeen inhoudt dat deze grootheden buiten het block niet bekend zijn (men zegt ook wel dat de scope van een in een block gedeclareerde grootheid beperkt is tot dat block). Omgekeerd zijn buiten het block met dezelfde namen gedeclareerde grootheden binnen het block niet bekend. Hieruit volgt dat lokale identifiers vrij gekozen mogen worden, zodat wanneer twee personen samen een programma schrijven zij onafhankelijk van elkaar hun lokale identifiers kunnen gebruiken.

Bij het verlaten van een block verliezen lokale variabelen hun betekenis en hun waarde; ook bij het opnieuw binnenkomen van datzelfde block is de waarde niet opnieuw bekend! Niet-lokale variabelen daarentegen behouden hun betekenis en waarde zowel bij het binnenkomen als bij het verlaten van het block.

De grenzen in een array declaration in het begin van een block moeten direct bij binnenkomst in het block uitgerekend kunnen worden zodat eventueel in de grenzen gebruikte variabelen nooit lokaal mogen zijn. In het buitenste block mogen arraygrenzen daarom geen variabelen bevatten!

Een label is altijd lokaal t.o.v. het kleinste omvattende block waarin hij voorkomt en is daardoor buiten het block niet bekend; een zo gelabelde instructie is dus ook niet bereikbaar met een goto statement vanuit een omvattend block. (Dit verbod geldt niet voor een compound statement) Met andere woorden een block heeft maar één ingang, nl. via zijn begin symbool, zodat statements in een block pas uitgevoerd kunnen worden nadat de declaraties zijn afgewerkt. Omgekeerd is het wel mogelijk om met een goto statement naar een gelabelde instructie in een omvattend block te springen.

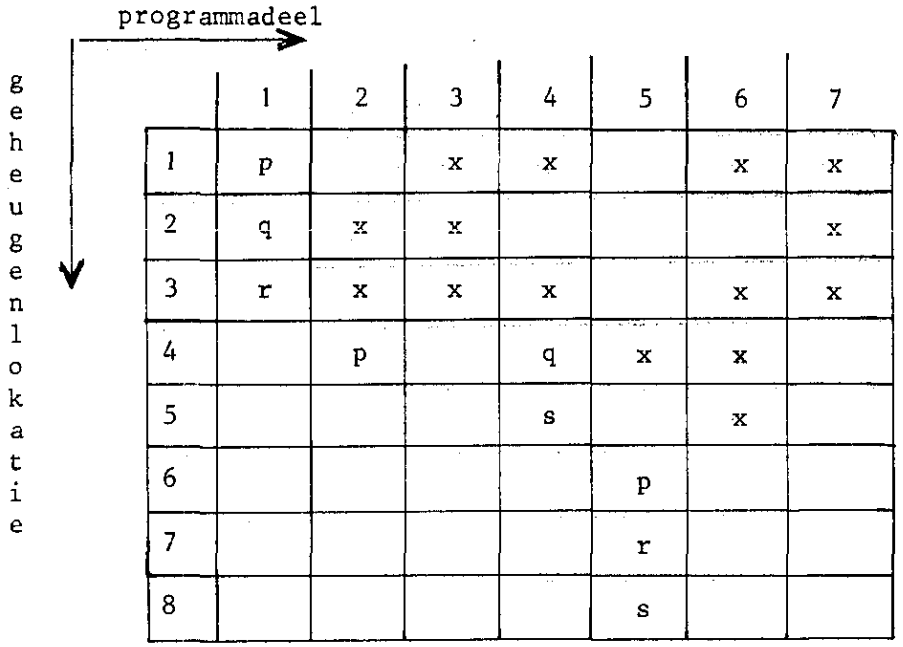
In de navolgende figuur is een mogelijke "nesting" van blocks geschetst.



Gezien het "lokale" karakter van labels is

de eerste label a als zodanig bekend in de programmadelen 1,2,3,7
de eerste label b als zodanig bekend in de programmadelen 1,2,3,7
de tweede label a als zodanig bekend in de programmadelen 4,5,6
de tweede label b als zodanig bekend in de programmadelen 4,5,6
de label c als zodanig bekend in alle programmadelen.

In de volgende tabel is verder aangeduid hoe de geheugenplaatsen 1 t/m 8 gebruikt zullen worden bij declaraties (x geeft aan dat de betreffende links ervan staande grootheid ook bekend is binnen het block). We zien hieruit dat de waarde van q na het verlaten van het derde programmadeel bij binnenkomst in 7 onveranderd is.



	1	2	3	4	5	6	7
1	p		x	x		x	x
2	q	x	x				x
3	r	x	x	x		x	x
4		p		q	x	x	
5				s		x	
6					p		
7					r		
8					s		

De blockstructuur van Algol opent de weg voor een "dynamisch" geheugengebruik: geheugenplaatsen worden afhankelijk van de voortgang in het programma (tijdelijk) gereserveerd voor verschillende grootheden.

Opgemerkt moet worden dat een for statement als een block is op te vatten omdat het niet is toegestaan om van buiten met een goto statement over de for clause te springen. (Dit mag wel naar één van de delen van een conditional statement in welk geval het andere deel als een dummy statement moet worden opgevat.)

Voorbeelden:

1) Een block dat een willekeurige vierkante matrix a[1 : k,1 : k] transponeert:

```
begin integer m, n; real w;
      for m := 1 step 1 until k - 1 do
      for n := m + 1 step 1 until k do
      begin w := a[m,n]; a[m,n] := a[n,m]; a[n,m] := w end
end
```

2) Verifieer dat het volgende (niet zinvolle) programma eindigt met x = y = 4

```
begin real x, y; boolean b;
      x := 4; y := 1.25;
   p: b := x > y;
      if b then begin real a, b, x;
                      x := y; b := 0;
                   p: b := b + 1;
                      a := x + y + b;
                      if x ≥ b then goto p;
                      y := b;
                      goto q
                end;
   q: if x > y then goto p
end
```

1.18 Procedures

1.18.1 Procedure identifiers, statements en declaraties

In het voorgaande hebben we gezien dat een block een min of meer afgerond stuk programma is. Wanneer zo'n block op veel plaatsen in een omvattend programma gebruikt moet worden, zou het plezierig zijn om op eenvoudige wijze het steeds weer opschrijven van een block te voorkomen. Dit kan gebeuren door zo'n block met een verkorte schrijfwijze aan te geven; deze bestaat uit het symbool procedure gevolgd door een zg. "procedure identifier", een willekeurige (geschikt gekozen) naam zoals we die reeds bij identifiers tegenkwamen. Op plaatsen in een programma waar zo'n block uitgevoerd moet worden, kunnen we dan volstaan met het vermelden van de procedure identifier, die zodoende fungeert als statement (een "procedure statement") en dus van voorafgaande en volgende statement door een puntkomma gescheiden wordt. Een procedure statement is toegestaan op elke plaats waar een compound statement is toegestaan.

Voor de uitvoering van een procedure statement is het nodig de procedure en zijn werking als zodanig te declareren. Net als de declaratie van identifiers gebeurt dit in het declaratiedeel van een programma direct na het begin symbool. In het hele stuk programma omsloten door dit begin symbool en het bijbehorende end symbool is dan de procedure als zodanig bekend.

De eenvoudigste vorm van een procedure declaratie bestaat uit het symbool procedure, direct gevolgd door de procedure identifier, gevolgd door een puntkomma, gevolgd door de "procedure body". De procedure body is het block waarvan de uitvoering op de plaats van de procedure statement gewenst wordt.

Voorbeeld:

Wanneer op verschillende plaatsen in een programma de waarde van twee variabelen a en b verwisseld moet worden, dan is dit te doen met de declaratie

```
procedure wissel; begin real s; s := a; a := b; b := s end
```

In het programma kunnen we dan de volgende constructies hebben:

```
....; a := 4; b := 2; ....; wissel; ....; if a < b then wissel;....
```

1.18.2 Function designators

In andere gevallen kan het gebeuren dat we in een programma met het resultaat van een standaard bewerking, bv. die van de lengte van een vector of de (3/2)-de machtswortel van een variabele willen manipuleren alsof dit resultaat ook een variabele was. Dit kan met gebruikmaking van een "function designator", die dezelfde gedaante als een procedure statement heeft en die overal toegelaten is waar een arithmetic of boolean variable is toegestaan.

Bij de declaratie van een function designator moet nu echter
- voor het symbool procedure ook het type (real, integer of boolean) van de function designator worden opgegeven,
- in de procedure body een assignment statement voorkomen, waardoor aan de procedure identifier de vereiste waarde wordt toegekend.

Voorbeeld:

In een programma moet de waarde van een polynoom

$$a_0x^n + a_1x^{n-1} + \ldots + a_n$$

voor verschillende waarden van x en n worden berekend. De benodigde declaraties zijn dan:

```
real x; integer n; real array a[0 : n];
real procedure pol;
    begin integer i; real s; s := a[0]; i := 1;
        while i ≤ n do begin s := s * x + a[i]; i := i + 1 end;
        pol := s
    end
```

In het programma kunnen dan de volgende constructies voorkomen:

```
...; n := ...; ...; x := 0; y := pol + 4; x := 2; y := y + pol; ...;
```

In de declaratie moet de hulpgrootheid s ingevoerd worden omdat s in de procedure body ook rechts van := voorkomt; bij vervanging van s door pol zou de procedure zichzelf aanroepen (hetgeen hier niet in de bedoeling ligt!)

1.18.3 Parameters bij procedure statements en function designators

In de tot dusver gegeven voorbeelden moet voor ieder gebruik van procedure statement of function designator aan de in de procedure body voorkomende globale variabelen via een voorafgaande assignment statement een waarde zijn toegekend. Bovendien moeten schrijvers van procedure body en van de rest van het programma dan overeenkomen welke namen zij voor deze globale variabelen zullen gebruiken.

Om dit te ondervangen is het ook mogelijk om bij de declaratie achter de procedure identifier tussen ronde haakjes de benodigde argumenten (parameters), onderling gescheiden door komma's, te vermelden. In de procedure declaratie heten ze dan "formele parameters"; deze worden bij het gebruik van de procedure (de "procedure call") vervangen door de "actuele parameters" uit de procedure statement of de function designator. Omdat formele parameters vervangen worden, zou men denken dat ze strikt genomen niet benoemd hoeven te worden, maar bij de meeste installaties - ook die van de THE - is toch "specificatie" verplicht! Deze specificatie wordt op dezelfde wijze als de declaratie opgeschreven, maar is verschillend wat uitwerking betreft. Bij een declaratie worden geheugenplaatsen benoemd en gereserveerd, bij een specificatie wordt slechts aangegeven van welke soort de later te gebruiken parameters zijn. Het aantal formele parameters in de procedure declaratie moet uiteraard gelijk zijn aan het aantal actuele parameters in de procedure aanroep; de vervanging van formele door actuele parameters, geschiedt in de volgorde van voorkomen "in de parameterslijst". (Als bij deze vervanging locale variabelen in

de procedure body in naam overeenstemmen met in de actuele parameters voorkomende identifiers worden de namen van de lokale variabelen automatisch veranderd.)

Met gebruik van parameters wordt het laatste voorbeeld

```
real array a[0 : n];
real procedure pol (x); real x;
    begin integer i; real s; s := a[0]; i := 1;
        while i ≤ n do begin s := s * x + a[i]; i := i + 1 end;
        pol := s
    end
```

en het eerder gegeven stukje programma eenvoudig

```
...; n := ...; ...; y := pol (0) + 4 + pol (2); ... .
```

Ook de constructie `.....; z := 0.7; y := pol (z ↑ 2 + 1); ...` is toegestaan, m.a.w. actuele parameters mogen ook expressies zijn (en trouwens ook procedure identifiers of labels). Deze vrijheid roept echter de vraag op of bij de vervanging van formele door actuele parameters:

- in de procedure body een formele parameter vervangen wordt door de expressie op de corresponderende actuele parameter plaats,
- of dat de formele parameter vervangen wordt door de waarde die de corresponderende actuele parameter bezit op het moment van uitvoering van de procedure.

Het antwoord op deze vraag luidt: een formele parameter wordt vervangen door de waarde van een actuele parameter ("call by value") wanneer deze formele parameter in de procedure declaratie achter de procedure identifier en parameterlijst vermeld wordt in een "value-part", bestaande uit het symbool value gevolgd door de zo te behandelen formele parameters (onderling door komma's gescheiden). Het effect is dan dus alsof aan de procedure body eerst een programma voorafgaat dat deze formele parameters uitrekent en ze als het ware non-local maakt voor de procedure body. Formele parameters, die niet in een "value-part" zijn opgenomen, heten "called by name"; zij worden in de procedure body vervangen door de corresponderende actuele parameter (nadat deze zo nodig tussen haakjes is geplaatst).

Voorbeelden:

```
real procedure f(x,y,z); value x, y; real x,y,z;
                begin if x > 0 then f := x ↑ 2 + y ↑ 2
                               else begin z := (x-4) * (y+1) ↑ 2;
                                          f := z/(z+1)
                                    end
                end
```

Deze procedure is equivalent aan

```
real procedure f(x,y,z); real x,y,z;
     begin real xf, yf; xf := x; yf := y;
          if xf > 0 then f := xf ↑ 2 + yf ↑ 2
                    else begin z := (xf - 4) * (yf + 1) ↑ 2;
                               f := z/(z+1)
                         end
     end
```

De procedure aanroep q := f(0,0,u) levert nu aan q de waarde 4/3 en aan u de waarde -4. Merk op dat de procedure aanroep q := f(4,2,1) niet is toegestaan, omdat de parameter 1 geen identifier is waaraan de procedure een waarde zou kunnen toekennen (al zou dit met deze waarden van x en y niet gebeuren). Ook q := f(4,2,p-q) is om dezelfde reden niet toegestaan. In het algemeen moet men er voor zorgen dat formele parameters zo gespecificeerd en gebruikt worden in de procedure declaratie en dat voor de corresponderende actuele parameters dusdanige expressies worden genoemd dat bij de procedure aanroep zinnige bewerkingen worden uitgevoerd.

Merk voorts op dat het via de parameters mogelijk is om voor een procedure statement die meerdere resultaten oplevert, deze resultaten naar buiten te voeren, bv.

```
real procedure f(a,x,y,z); value a; real a,x,y,z;
                begin x := a * a;
                      y := a * x;
                      z := a * y;
                      f := a * z;
                end
```

levert via x,y,z en f de waarden a^2, a^3, a^4 en a^5.

Vergelijk de volgende procedures:

```
procedure subst (a,b,c,d);          procedure subst (a,b,c,d);
value b,d; real a,b,c,d;                    real a,b,c,d;
begin a := b; c := d end            begin a := b; c := d end
```

Met het "programma" ...; p := 6; subst (p,3,q,p-1); ... krijgt met de eerste declaratie p achtereenvolgens de waarden 6 en dan 3, terwijl q de waarde 6 - 1 = 5 krijgt. Met de tweede declaratie gebeurt met p hetzelfde, maar q krijgt nu de waarde 3 - 1 = 2! Tenslotte zij opgemerkt dat globale variabelen in een procedure van waarde kunnen veranderen, wat een neveneffect heet. Uiteraard moet men oppassen dat een procedure geen ongewenste neveneffecten heeft, hetgeen voorkomen kan worden door in een procedure bij voorkeur geen globale variabelen te gebruiken.

1.18.4 De Jensen device

Aan de andere kant kan men soms handig gebruik maken van een neveneffect, zoals bij een bepaald soort gebruik van de call-by-name faciliteit die Jensen-device (of methode van de gebonden variabele) genoemd wordt. We zullen deze toelichten aan de hand van het probleem om het inproduct van twee vectoren te berekenen.

Dit kan gebeuren met

```
real procedure inprod (a,b,n); value n;
                integer n; real array a,b;
    begin integer i; real h; h := 0;
          for i := 1 step 1 until n do h := h + a[i] * b[i];
              inprod := h
    end
```

(Merk op dat bij de specificatie van arrays a en b geen grenzen worden opgegeven omdat het geen declaratie is.).

Gebruikmakend van de call-by-name faciliteit kunnen we ook de volgende procedure construeren:

```
real procedure inprod (a,b,i,n); value n;
              integer i,n; real a,b;
    begin real h; h := 0;
          for i := 1 step 1 until n do h := h + a * b;
          inprod := h
    end
```

Deze procedure geeft op het eerste gezicht n × a × b. Onderstellen we nu gegeven de declaraties

```
real array p,q [1 : m]; integer j,m;
```

dan is inprod(p[j],q[j],j,m) een geldige procedure aanroep, want p[j] en q[j] zijn van de soort real, j en m van de soort integer. In de procedure body wordt eerst h = 0 gemaakt en i, d.w.z. j (call-by-name!) wordt 1. De statement h := h + a * b moeten we dan lezen als h := h + p[j] * q[j] en heeft dus het resultaat h := p[1] * q[1]. Zo doorgaande in de cyclus zal h uiteindelijk de waarde van het gewenste inproduct krijgen, dat dan via inprod naar buiten wordt doorgegeven.

Op analoge wijze is te verifiëren dat inprod(v[i,j],w[j,k],j,n) het i,k element van het product van twee matrices v en w is. Dit gebruik van een parameter (j in de laatste voorbeelden) als dummy variabele van andere parameters heet de "Jensen device".

Een ander voorbeeld van het gebruik van de Jensen device is:

```
real procedure som(term, start, eind, var);
    value eind; integer start, eind, var; real term;
    begin real h; h := 0;
          for var := start step 1 until eind do h := h + term;
          som := h
    end
```

Verifieer zelf dat

som(4,1,5,n) de waarde 20 krijgt,
som(n,1,5,n) de waarde 1+2+3+4+5 krijgt,
som(n↑2,1,5,n) de waarde $1^2 + 2^2 + 3^2 + 4^2 + 5^2$ krijgt,
som(a[n],1,5,n) de waarde a[1] + a[2] + a[3] + a[4] + a[5] krijgt.

1.18.5 Samenvatting

procedure declaration

```
a)   <type> procedure procedure identifier (formele parameters);   } procedure
                value de nodige formele parameters;                } heading
c)              specificatie van alle formele parameters;          }
          begin declarations van alle lokale parameters;           }
d)              statements;                                        } procedure
b)              procedure identifier := expression                 } body
          end
```

Opmerking:

a) <type>, d.w.z. integer of real of boolean wordt weggelaten bij gebruik van de procedure als procedure statement

b) wordt weggelaten bij gebruik als procedure statement

c) specificatie geschiedt door opschrijven van alle formele parameters, telkens voorafgegaan door een van de symbolen <type> of <type> array of label of procedure of <type> procedure

d) in de procedure body kunnen zodoende de volgende parameters voorkomen:
   - locale, d.w.z. identifiers die in de body gedeclareerd zijn (alsmede labels),
   - formele, d.w.z. identifiers die niet in de body gedeclareerd, maar wel in de heading gespecificeerd zijn (komt een identifier op beide plaatsen voor, dan is hij locaal in de body!)
   - globale, d.w.z. identifiers die noch locaal, noch formeel zijn en dus in een de procedure declaration omvattend block gedeclareerd moeten zijn. Het gebruik van globale parameters moet zoveel mogelijk vermeden worden; vergelijk

```
real procedure pol (x); value x; real x;
```

en

```
real procedure pol (a,n,x); value n,x; integer n; real x; real array a;
```

beiden met de body

```
begin real h; integer i; h := a[0];
      for i := 1 step 1 until n do h := a[i] + h * x;
      pol := h
end
```

procedure_call:

procedure identifier (actuele parameters)

Opmerking:

1) actuele parameters kunnen zijn:
   - al of niet subscripted variables,
   - identifiers van een array, een procedure of een label,
   - expressies.
2) op het moment van de procedure call moeten alle identifiers, voorkomend in actuele parameters (met uitzondering van een label), gedeclareerd zijn in overeenstemming met de specificatie van de corresponderende formele parameter.
3) procedures vormen hèt hulpmiddel om een ingewikkeld programma overzichtelijk te houden. Het goed onderkennen van de punten waar het schijven van een procedure geboden is, is een gave van de goede programmeur!

1.18.6 Standaardfuncties

Een aantal standaard procedures hoeft niet gedeclareerd te worden. Hiertoe horen de internationaal erkende standaardfuncties:

abs (E)	voor de absolute waarde van een arithmetische expressie E,
entier (E)	voor de grootste integer ≤ waarde van E,
sign (E)	+1 als E > 0, 0 als E = 0 en -1 als E < 0,
sqrt (E)	voor de vierkantswortel van de waarde van E,
sin (E)	voor de sinus van de waarde van E,
cos (E)	voor de cosinus van de waarde van E,
arctan (E)	voor de arctangens van de waarde van E,
ℓn (E)	voor de natuurlijke logarithme van de waarde van E,
exp (E)	voor de exponentiaal functie van de waarde van E,

alsmede voor de betreffende installatie geldende invoer/uitvoer conventies, zoals de reeds besproken procedure statements

print(x), nlcr, printtext(‹commentaar›), space(n)

en de function designator als read. Voor een gedetailleerde beschrijving van de THE invoer/uitvoer conventies zij verwezen naar appendix 1 bij dit hoofdstuk.

1.19 Recursie

Het is toegestaan om in de body van een procedure gebruik te maken van deze procedure zelf; de procedure heet dan recursief. Dit naar analogie van een recursieve functie-definitie, zoals bijvoorbeeld voor de faculteitsfunctie n! ($n \geq 0$):

$$0! = 1 \qquad n! = n(n-1)!$$

met andere woorden als n = 0 dan is n! = 1, anders wordt n! berekend met gebruikmaking van dezelfde functie voor een lagere argumentswaarde, enz..

In Algol kan de met de wiskundige formulering corresponderende procedure geschreven worden als

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

(begin en end mogen in de body worden weggelaten omdat de body slechts 1 statement bevat). Het bewijs dat deze procedure correct is, is op dezelfde manier te geven als het bewijs voor een recurrente formule: voor n = 0 is de procedure goed; neem aan dat hij dan goed is voor n = k en bewijs dan de juistheid voor n = k + 1.

Het is natuurlijk ook mogelijk om een niet-recursief, nl. een iteratief programma voor n! op te schrijven:

```
integer procedure fac(n); value n; integer n;
        begin integer h,i; h := 1; if n = 0 then else
              for i := 1 step 1 until n do h := h * i; fac := h
        end
```

Een recursief programma is voor recurrente functie definities gemakkelijk op te schrijven, maar kan aanleiding geven tot een langer lopend en veel geheugen gebruikend programma, omdat zoveel tussenresultaten (hier n, n-1, n-2, ,..., 2,1) vastgelegd moeten worden. De iteratieve methode vraagt in deze gevallen een "wiskundig minder natuurlijk" programma (bij processen uit de numerieke wiskunde van het type $x_{n+1} = f(x_n)$ is het daarentegen eenvoudiger om een iteratief werkend programma te schrijven), maar geeft meestal een sneller verlopend en minder geheugen vergend programma. Dit voordeel wordt echter klein, wanneer gebruik gemaakt wordt van rekenautomaten, die speciaal met het oog op recursie gebouwd zijn.

Een ander voorbeeld, waarbij recursieve programmering een simpeler programma geeft dan de iteratieve methode, wordt gegeven door de berekening van de grootste gemene deler van twee positieve getallen m en n met de zg. algorithme van Euclides: als n een exacte deler van m is, dan is n de g.g.d. van m en n, anders is de g.g.d. van m en n ook de g.g.d. van n en de rest die overblijft na deling van m door n. (Bij een recursief proces als dit, is niet van tevoren te voorspellen wat de "diepte" van de recursie is, d.i. het aantal keren dat het proces uitgevoerd moet worden.)
Recursieve en iteratieve procedures zijn:

```
integer procedure ggd (m,n); value m,n; integer m,n;
        ggd := if n > m then ggd (n,m)
                       else if n = 0 then m else ggd(n, rest (m,n));

integer procedure ggd(m,n); value m,n; integer m,n;
        begin integer h;
              if n > m then begin h := n; n := m; m := h end;
              while rest(m,n) ≠ 0 do
                    begin h := n, n := rest(m,n); m := h end
              ggd := n
        end
```

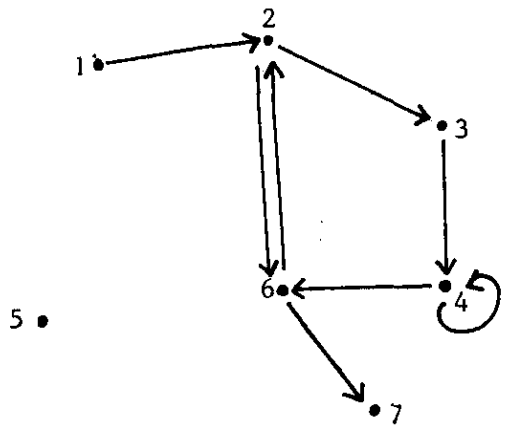
In beide gevallen is de functie rest(m,n) bekend verondersteld, bv. door

```
integer procedure rest(m,n); value m,n; integer m,n;
        rest := m - (m ÷ n) * n;
```

Een andere vorm van recursie krijgen we als onder de actuele parameters van een procedure statement weer dezelfde procedure voorkomt. Met de reeds eerder behandelde procedure som(term, start, eind, var) wordt met som(a[i],1,n,i) de som $\sum_{i=1}^{n} a_i$ berekend en met som(som(a[i,j],1,m,j),1,n,i) de dubbelsom $\sum_{i=1}^{m}\sum_{j=1}^{n} a_{ij}$. Een soortgelijke situatie treedt ook op bij de berekening van dubbelintegralen.

Recursie vindt niet alleen in de numerieke wiskunde veel toepassingen, maar ook in deelgebieden van de informatica, vooral die waar de problemen te karakteriseren zijn met een zg. boomstructuur.

Een gerichte graaf wordt gevormd door een verzameling genummerde knooppunten, waarvan een aantal door middel van pijlen met elkaar verbonden is. We zullen aannemen dat twee of meer pijlen in dezelfde richting tussen twee knooppunten als een enkele pijl beschouwd kunnen worden.



Voorbeeld van een gerichte graaf

Een rij knooppunten $k_1,\ldots,k_p$ vormt een pad als voldaan is aan:

i) er is een pijl van k_i naar k_{i+1} voor $1 \leq i < p$, én

ii) $k_i \neq k_j$ voor $i \neq j$;

een pad heet actueel als bovendien nog geldt:

iii) het nummer van k_1 is kleiner dan het nummer van de overige knooppunten in het pad.

Een rij knooppunten $k_1,\ldots,k_s,k_1$ vormt een cycle als voldaan is aan:

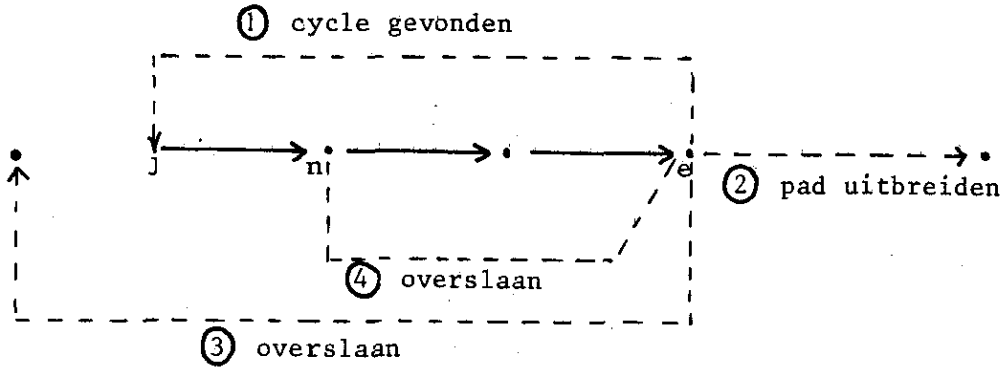
i) $k_1,\ldots,k_s$ is een pad, én

ii) er is een pijl van k_s naar k_1.

Gevraagd wordt nu een programma te maken dat alle cycles van een gerichte graaf precies één keer zal uitprinten.

Aan de eis dat iedere cycle niet vaker dan één keer mag worden uitgeprint, kunnen we voldoen door ervoor te zorgen dat eerst alle cycles met knooppunt nummer 1 precies één keer worden uitgeprint; daarna alle cycles die knooppunt nummer 2 wél, maar knooppunt nummer 1 niet bevatten; vervolgens alle cycles die knooppunt nummer 3 maar niet de knooppunt nummers 1 en 2 bevatten enz.

Het probleem alle cycles te vinden die knooppunt nummer j bevatten, maar geen knooppunten met lagere nummers, kunnen we oplossen door uitgaande van knooppunt met nummer j actuele paden te vormen en te onderzoeken of die paden tot cycles (die aan genoemde eisen voldoen) uitgebreid kunnen worden.



Als pad j,...,e is gevormd, dan kunnen we afhankelijk van de uitgaande pijlen van e 5 situaties onderscheiden:

1) een pijl wijst naar j; dan hebben we een cycle gevonden (en kunnen we de cycle uitprinten);
2) een pijl wijst naar een knooppunt dat nog niet in het pad zit en met een nummer dat groter is dan j; in dit geval kan het pad met het betreffende knooppunt "uitgebreid" worden en wordt opnieuw bekeken of het pad verder is uit te breiden tot een cycle;
3) een pijl wijst naar een knooppunt met een lager nummer dan j; een dergelijke pijl wordt genegeerd omdat we alleen actuele paden willen vormen;
4) een pijl wijst naar een knooppunt dat al in het pad zit; in dit geval hebben we weliswaar een cycle gevonden, maar toch negeren we deze pijl, omdat alle cycles maar één keer mogen worden uitgeprint; de hier gevonden cycle wordt later weer opnieuw gevonden en pas dan geprint;
5) er is geen uitgaande pijl.

Bij het uitbreiden van het pad met een nieuw knooppunt moeten we wel op de een of andere manier onthouden welke uitbreidingen al geprobeerd zijn en welke nog niet. We zullen zien dat dit, door gebruik te maken van een recursieve procedure, op een heel natuurlijke manier is op te lossen, terwijl als men zich wil beperken tot een iteratieve procedure de oplossing ingewikkelder en gekunstelder wordt.

Voor beide algorithmen nemen we aan dat de gerichte graaf (bestaande uit n knooppunten) beschreven wordt met het 2-dimensionale array graph op de volgende manier:

Stel van knooppunt i gaan p pijlen uit naar resp. de knooppunten $k_1,\ldots,k_p$, dan geldt:

$$\text{graph}[i,j] = k_j \qquad \text{voor } 1 \leq j \leq p$$

$$\text{graph}[i,j] = 0 \qquad \text{voor } p < j \leq n \ .$$

Het 1-dimensionale array path bevat het actuele pad; met de integer nip (Number In Path) wordt bijgehouden tot hoever het array path gevuld is (geeft dus het aantal knooppunten in het pad aan).

We zullen verder een integer cand invoeren, waarvoor geldt dat cand = 0 wanneer van path[nip] geen te "beschouwen" pijlen meer uitgaan. Als van path[nip] een wel te beschouwen pijl uitgaat naar knooppunt i, dan krijgt cand de waarde i.

Voor de in te voeren boolean procedure inpath geldt:

$$\text{inpath(cand)} \equiv \underline{\text{true}} \Leftrightarrow \exists_{1\leq i\leq \text{nip}} [\text{path}[i] = \text{cand}]$$

De procedure printcycle behoeft geen nadere toelichting.

De in te voeren procedures extendre en entendit zullen bij hun aanroepen bewerkstelligen dat al díe cycles precies één keer gevonden worden, die gevormd kunnen worden door het actuele pad verder uit te breiden, met uitzondering echter van cycles, die een knooppunt bevatten dat een lager nummer heeft dan het eerste knooppunt van het actuele pad. De procedures extendre en extendit zijn resp. recursief en iteratief.

Als vanuit path[nip] alle uitgaande pijlen onderzocht zijn, dan moet het laatste knooppunt van het actuele pad afgehaald worden om vervolgens die pijlen, die van path[nip - 1] uitgaan en nog niet onderzocht zijn, te onderzoeken.

De werking van de procedure extendre kan nu in grote lijnen als volgt worden beschreven:

```
initialiseer cand;
while cand ≠ 0 do
        het actuele pad
   begin if mag uitgebreid      then begin breid pad uit met cand
         worden met cand                    extendre
         (d.i. situatie 2)                  haak cand af
                                       end
                                  else if cand = path[1] (d.i. situatie 1)
                                       then printcycle;
      cand := next cand
   end
```

De procedure extendit zit (uiteraard) wat ingewikkelder in elkaar. Bij deze procedure wordt gebruik gemaakt van het 2-dimensionale boolean array closed:

closed[i,j] ≡ true betekent de verbinding van knooppunt i naar knooppunt j mag niet gebruikt worden, is als het ware opgeheven.

Een opgeheven verbinding kan weer hersteld worden door het betreffende element van het array closed de waarde false toe te kennen.

De procedure extendit ziet er ruwweg als volgt uit:

```
initialiseer cand;
while nip > 0 do begin if cand ≠ 0 then
        verbinding van
   begin if path[nip] naar      then
         cand niet opgeheven
              het actuele pad
         begin if mag uitgebreid      then breid pad uit met cand
               worden met cand
               (d.i. situatie 2)
                                  else if cand = path[1]
                                          (d.i. situatie 1)
                                       then
                                       begin printcycle;
                                             hef verbinding van
                                             path[nip] naar cand op
                                       end
         end
   end else
```

```
        begin herstel alle van path[nip] uitgaande verbroken verbindingen;
              hef verbinding van voorlaatste naar laatste knooppunt van
              pad op;
              haak laatste knooppunt van actuele pad af
        end; cand := next cand
    end
```

Onderstaande algorithmen zullen nu dus van een gerichte graaf alle cycles precies één keer uitprinten:

1) recursief algorithme:

```
Lees graaf;
for s := 1 step 1 until n do
begin path[1] := s; nip := 1; extendre end
```

2) iteratief algorithme:

```
Lees graaf;
initialiseer het array closed;
for s := 1 step 1 until n do
begin path[1] := s; nip := 1; extendit end
```

Volledig uitgeschreven in Algol 60 zien de programma's er als volgt uit:

1) recursief algorithme:

```
begin integer n; n := read;
  begin integer array graph[1 : n, 1 : n], path[1 : n];
        integer i, j, nip, s;
        procedure printcycle;
           begin integer i;
             nlcr; for i := 1 step 1 until nip do
             absfixt(6, 0, path[i])
           end printcycle;
```

```
        boolean procedure inpath(cand);
            value cand; integer cand;
            begin integer i; i := 1;
                  while path[i] ≠ cand and i < nip do i := i + 1;
                  inpath := path[i] = cand
            end inpath;

        procedure extendre;
            begin integer j, cand; j := 1; cand := graph[path[nip],j];
              while cand ≠ 0 do
              begin if cand > path[1] and not inpath(cand) then
                begin nip := nip + 1; path[nip] := cand;
                  extendre; nip := nip - 1
                end else
                if cand = path[1] then printcycle;
                j := j + 1; cand := if j ≤ n then graph[path[nip],j] else 0
              end
            end extendre;

        for i := 1 step 1 until n do
        for j := 1 step 1 until n do graph[i,j] := read;
        for s := 1 step 1 until n do
        begin path[1] := s; nip := 1; extendre end
    end
end
```

2) iteratief algorithme:

```
begin integer n; n := read;
  begin integer array graph[1 : n, 1 : n], path[1 : n];
        integer i, j, nip, s; boolean array closed[1 : n, 1 : n];

        procedure printcycle;
          begin integer i;
```

```
      nlcr; for i := 1 step 1 until nip do
      absfixt(6, 0, path[i])
    end printcycle;
  boolean procedure inpath(cand);
    value cand; integer cand;
    begin integer i; i := 1;
          while path[i] ≠ cand and i < nip do i := i + 1;
          inpath := path[i] = cand
    end inpath;
  procedure extendit;
    begin integer i, j, can, last;
      j := 1; last := path[nip]; cand := graph[last,j];
      while nip > 0 do begin
      if cand ≠ 0 then
      begin if not closed[last, cand] then
        begin if cand > path[1] and not inpath(cand) then
          begin nip := nip + 1;
                last := path[nip] := cand; j := 0
          end else
          if cand = path[1] then
          begin printcycle;
                closed[last, cand] := true
          end
        end
      end else
      begin for i := 1 step 1 until n do
        closed[last, i] := false;
        if nip > 1 then
        begin closed[path[nip - 1], last] := true;
          last := path[nip - 1]
        end;
        nip := nip - 1; j := 0
      end;
      j := j + 1;
      cand := if j ≤ n then graph[last,j] else 0
```

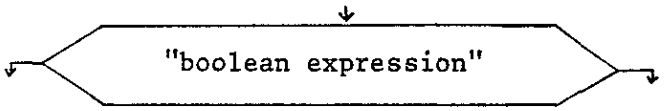
```
            end
          end extendit;
        for i := 1 step 1 until n do
        for j := 1 step 1 until n do
        begin graph[i,j] := read;
              closed[i,j] := false
        end;
        for s := 1 step 1 until n do
        begin path[1] := s; nip := 1; extendit end
    end
end
```

1.20 Stroomdiagrammen ("flowcharts")

Al in de inleiding is opgemerkt, dat de huidige rekenautomaten sequentiële automaten zijn, m.a.w. dat de door ons in de (statische) programmatekst neergeschreven statements bij de (dynamische) uitvoering in dezelfde volgorde, enkele uitzonderingen daargelaten, worden afgewerkt. Zouden we de afzonderlijke statements willen visualiseren met bv. rechthoekjes, dan bestaat zo'n rechttoerechtaan programma uit onder elkaar getekende rechthoekjes, die we door pijlen met elkaar verbinden om de sequentiële volgorde tot uitdrukking te brengen. Erg verhelderend is deze detaillering niet, zodat we veel eerder een groep bij elkaar horende statements, bv. een hele compound statement zullen representeren met één enkele rechthoek. Hierin kunnen we dan ter verduidelijking schrijven wat deze compound statement beoogt, bv. met aanduidingen als:

"bereken lengte van vector"
"transformeer matrix kolom"
"bepaal priemgetal".

Interessant wordt zo'n visueel hulpmiddel, het programma stroomdiagram, vooral wanneer we de statements beschouwen die de sequentiële uitvoering van statements beïnvloeden (bv. de statements goto, if - then, if - then - else, for ..... do, enz.), alsmede de statements die betrekking hebben op de communicatie met bv. de buitenwereld (bv. read, print, enz.). Volgens internationale normen moeten deze typen statements worden aangeduid met resp. een ruit en met symbolen voor kaartlezers, ponsbandlezers, regeldrukkers, enz. (zie een uitgave van het Nederlandse Normalisatie Instituut voor de volledige afspraken). Voor het tekenen van deze symbolen zijn plastic mallen ("templates") alom verkrijgbaar, maar wij zullen deze symbolen hier niet gebruiken omdat ze tekenproblemen geven bij het maken van een stencil. Zo zal in plaats van de ruit gebruikt worden het zeshoekige hokje



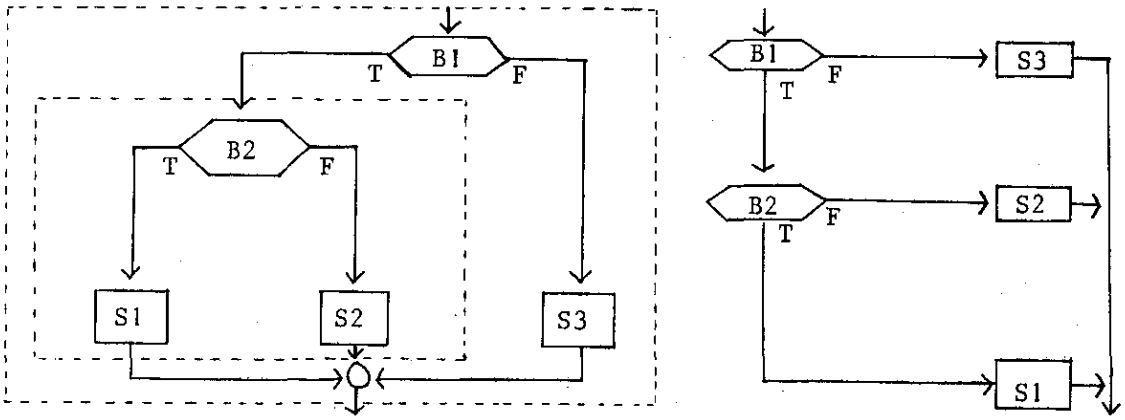
bij welks "uitgangen" symbolen als T(rue), F(alse), Y(es), N(o), >, =, <, enz. geplaatst worden (voor een vergelijking van twee grootheden wordt het symbool ↔ gebruikt). Voor invoer/uitvoer statements zal het symbool

(met daarin geschreven welke statement bedoeld is) worden gebruikt.

Het maken van een duidelijk stroomdiagram is een kunst, die enige keren beoefend moet worden, maar waarbij men enigszins geholpen wordt door de volgende richtlijnen in acht te nemen:

- verbindt de hokjes alleen door horizontaal of verticaal lopende lijnen; zet in die lijnen in ieder geval richtingsaanduidingen wanneer ze van beneden naar boven of van rechts naar links doorlopen moeten worden;

- herarrangeer zo nodig en mogelijk de hokjes opdat zo min mogelijk overkruisingen optreden; zijn kruisingen niet te voorkomen geef ze dan zo duidelijk mogelijk aan of gebruik de zg. "connectoren". Dit zijn cirkels om een lettersymbool die men bij de uiteinden van de onderbreking van een lijn plaatst (of bij het samenkomen van meer dan twee lijnen in één punt);
- teken horizontale lijnen of hokjes bij voorkeur alleen dan op verschillend niveau (en verticale lijnen of hokjes op verschillende verticalen), wanneer daarmee beslist iets tot uitdrukking moet worden gebracht;
- probeer niet in één stroomdiagram (bij voorkeur op één vel papier A4-formaat) alle gewenste (en zeker niet de "fijnste" programmerings-) detaillering op te nemen, maar teken eerst een stroomdiagram met een beperkte detaillering en "vergroot" zo nodig bepaalde hokjes hiervan in aparte stroomdiagrammen;
- de typische procedure en blockstructuur van Algol zou eigenlijk tot uitdrukking moeten worden gebracht door rechthoeken binnen rechthoeken te tekenen, bv. voor <u>if</u> B_1 <u>then begin if</u> B_2 <u>then</u> S_1 <u>else</u> S_2 <u>end else</u> S_3

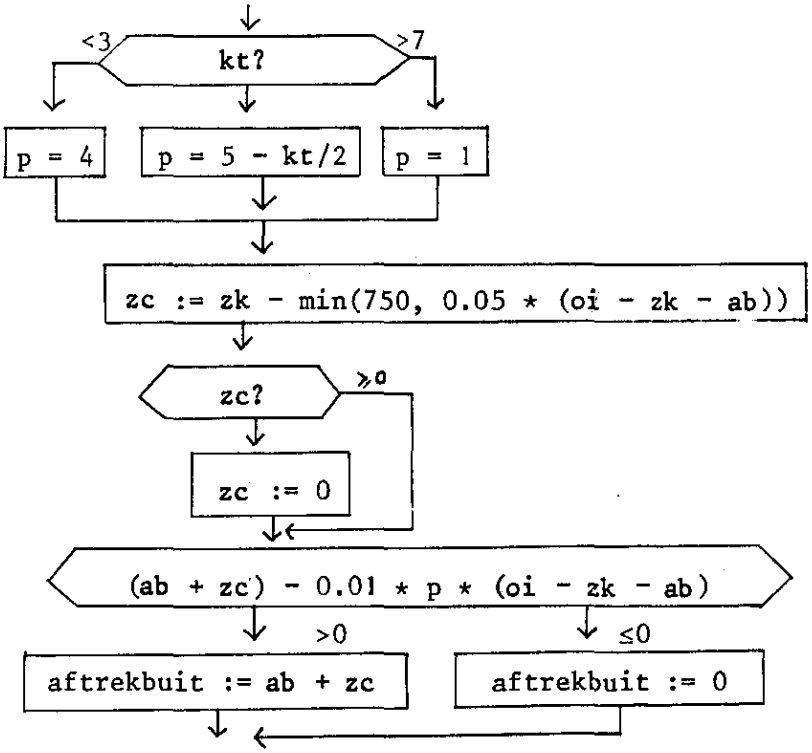


maar met enige oefening zal men de rechts getekende "gebruikelijke" voorstelling gemakkelijk in een Algol tekst kunnen "vertalen" en omgekeerd. Kleine veranderingen zullen af en toe nodig zijn, zoals we kunnen zien bij het volgende voorbeeld dat ontleend is aan onze belastingwetgeving (1969). Voor de berekening van de aftrekpost buitengewone lasten is de volgende procedure te gebruiken:

```
real procedure aftrekbuit (0i, zk, ab, kb);
    real oi, zk, ab; integer kt;
    begin comment deze variabelen stellen resp. onzuiver inkomen, ziektekos-
          ten, andere buitengewone lasten en kindertal voor;
          real zc, p;
          p := if kt < 3 then 4 else if kt > 7 then 1 else 5 - kt/2;
          zc := if zk ≤ min (750, 0.05 * (oi - zk - ab)) then 0 else
                                          zk - min(750, 0.05 * (oi - ak - ab));
          aftrekbuit := if ab + zc ≤ 0.01 * p * (oi - ab - zk) then 0 else ab + zc
    end
```

(dat met het oog op veranderingen in de belastingwetgeving de constanten 750, 0.04 en 0.05 misschien ook beter als parameters kunnen worden geschreven, zij hier in het midden gelaten).

Een corresponderend stroomdiagram is



- probeer niet recursief gebruik van procedures op een bepaalde manier in een stroomdiagram aan te geven (omdat dit gauw tot het "plaatje van de Droste zuster" problemen aanleiding geeft).

In veel computercentra worden t.a.v. het tekenen van stroomdiagrammen en de te gebruiken notaties eisen gesteld, die verder gaan dan de internationale standaards; bv. wordt verlangd dat hokjes genummerd worden in overeenstemming met de nummering van de geschreven pagina's van de programmatekst. Deze eisen zijn zeker zinvol omdat alleen zo een uniforme documentatie van in een computercentrum aanwezige programma's te bereiken is. Deze uniformiteit wordt tegenwoordig in de hand gewerkt door het feit dat er nu ook standaardprogramma's zijn, die bij, in bepaalde programmeringstalen geschreven programma's, de corresponderende stroomdiagrammen automatisch tekenen.

In weerwil van de grote aandacht die soms aan stroomdiagrammen wordt geschonken, moet hun betekenis toch niet overschat worden. Voor de gedetailleerde programmering zijn zij misschien soms een belemmering, omdat de sequentiële ordening, die er in wordt aangebracht, een herbezinning over een efficiënter programma-organisatie in de weg kan staan. Het grootste nut hebben stroomdiagrammen daarom als het er om gaat om uit een lang verhaal (zie het voorbeeld van de belastingwetgeving en derg.) de essentiële facetten te halen. Vooral als dat verhaal niet op papier staat, is het dan een nuttig hulpmiddel om "gaten in het verhaal" op te sporen (een alternatieve techniek, nl. beslissingstabellen, zij hier alleen genoemd).

1.21 Formele taalbeschrijving

In de voorgaande inleiding tot de Algol 60 programmeertaal is voor de beschrijving gebruik gemaakt van onze gebruikelijke omgangstaal. Reeds enige malen is opgemerkt dat een natuurlijke taal (primair tot stand gekomen om de communicatie tussen mensen over alle mogelijke onderwerpen mogelijk te maken, daarbij ondersteund door gebaren, mimiek, klemtoon, ritme, enz.) door onduidelijkheden of dubbelzinnigheden eigenlijk niet geschikt voor een exacte behandeling van een programmeertaal. Soms zijn dubbelzinnigheden door het gebruik van leestekens nog wel te vermijden, zoals in constructies als "de meester zei de leerling is een ezel", maar niet in een zinsdeel als "het bedrog van de leerling". Wanneer een woord meer betekenissen kan hebben (bv. kool, zin, enz.) of wanneer de betekenis van een woord pas blijkt uit de combinatie met andere woorden, geeft dat uiteraard ook aanleiding tot onduidelijkheden. Er is zodoende behoefte aan een geformaliseerde beschrijvingswijze van een programmeertaal, waarbij uitgaande van een zo klein mogelijk aantal grondbegrippen en symbolen eenduidig vastgesteld kan worden welke constructies in die taal mogelijk zijn (de grammatica of syntax van die taal)

en welke betekenis (semantiek) aan die constructies moet worden toegekend. Met zo'n geformaliseerde beschrijving kan een duidelijk antwoord worden gegeven op de vraag hoe iets precies in elkaar zit en of niet genoemde constructies ook mogelijk zijn.

De leer van de relaties tussen symbolen, symboolgroeperingen en de daaraan verbonden betekenis wordt semiotica genoemd; zij omvat

- syntax (of grammatica), de formele leer van de structuren die men met een gegeven aantal elementaire symbolen kan opbouwen, zonder zich er om te bekommeren wat deze structuren betekenen;
- semantiek, de leer van de betrekkingen tussen deze structuren en hun "werkelijke" of "theoretische" betekenis;
- pragmatiek, d.i. de leer van de betrekkingen tussen deze structuren en de interpretatie die een persoon of een machine er aan geeft.

Ter toelichting: bij getallen beschrijft de syntax hoe getallen opgebouwd kunnen worden met de cijfers 0 t/m 9, de tekensymbolen + en - en de decimale punt (of komma), enz.. De semantiek beschrijft de relatie tussen deze samengestelde symbolen en de getallentheorie, terwijl we bij de pragmatiek moeten denken aan de wijze waarop getallen binnen een rekenautomaat gerepresenteerd worden. Soms wordt echter in twijfel getrokken of we semantiek en pragmatiek wel scherp van elkaar kunnen onderscheiden en spreekt men eenvoudigheidshalve alleen van semantiek.

1.22 BNF notatie

Voor de beschrijving van Algol wordt meestal gebruik gemaakt van de zg. BNF notatie (Backus Normal Form of Backus - Naur Form). Algol kan syntactisch namelijk geheel beschreven worden met een "metataal", bestaande uit slechts drie "metasymbolen" die een precies (in onze omgangstaal) vastgelegde betekenis hebben en die in de te beschrijven taal verder niet voorkomen. Deze metasymbolen zijn

het "of" symbool	|	(bv. een teken is + | -)
het definitiesymbool	::=	(te lezen als "wordt gedefinieerd door" of "bestaat uit")
de "metahaken"	< >	(bv. <digit> is de aanduiding van de verzameling van grootheden die wij cijfers noemen).

Zo wordt met

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

uitgedrukt dat een element van de verzameling genaamd <digit> bestaat uit een van de elementaire symbolen 0 t/m 9 (elementaire symbolen hoeven niet verder gedefinieerd te worden). Met

<digit duo> ::= <digit>, <digit>

definiëren we een verzameling genaamd <digit duo> en hun structuur (de komma is wederom een elementair symbool). Combinaties als 1,4 en 7,5 behoren tot de verzameling digit duo's, maar 14 en 75 niet! Met:

<unsigned integer> ::= <digit>|<unsigned integer> <digit>

wordt een recursieve definitie gegeven van een unsigned integer (UIN). Het is nl. òf een digit òf (nu aannemend dat we reeds weten wat een UIN is) het is een UIN gevolgd door een digit. In onze natuurlijke taal uitgedrukt, is een UIN dus een willekeurig lange reeks van digits, maar het onbevredigende van de dagelijkse definitie zit in "willekeurig lange" reeks. (Ook voor een rekenautomaat zijn begrippen als "willekeurig lang", enz. niet hanteerbaar, een aftelbare constructieve definitie daarentegen wel.) Een recursieve definitie is uiteraard alleen dan zinvol, wanneer tenminste één van de alternatieven niet de te definiëren grootheid bevat en bovendien opgebouwd is uit reeds bekende begrippen.

Kenmerkend voor o.a. Algol 60 is het ontbreken van beperkingen t.a.v. de "lengte", het aantal symbolen van de gedefinieerde grootheden (in werkelijkheid zijn er voor een machine met een beperkte geheugencapaciteit uiteraard wel praktijkgrenzen). In andere talen treden deze beperkingen vaak expliciet op, met de consequentie dat dan meer dan drie metasymbolen nodig zijn voor een formele beschrijving. Bovendien is de representatie van veel andere talen niet "format free", m.a.w. bij gebruik van bijv. ponskaarten spelen begrippen als "zoveelste kaartkolom" vaak een rol in de beschrijving van de taal, dan wel spelen blanco's een bijzondere rol.

Voorbeeld:

In Algol 60 verloopt de definitie van <identifier> via de volgende stappen (we volgen niet in alle details het Algol-rapport!)

<digit> ::= 0|1|2|3|.....|8|9
<kleine letter> ::= a|b|c|.....|x|y|z
<hoofdletter> ::= A|B|C|.....|X|Y|Z
<letter> ::= <kleine letter>|<hoofdletter>
<identifier> ::= <letter>|<identifier> <letter>|<identifier> <digit>

In Fortran, een enkele jaren oudere voorloper van Algol, is de definitie van <identifier> niet alleen (daar Fortran geen kleine letters kent)

<identifier> ::= <hoofdletter>|<identifier> <hoofdletter>|<identifier> <digit>

maar is het aantal symbolen, waaruit een identifier opgebouwd kan worden, beperkt tot zes. Eigenlijk kan men de Fortran identifier niet recursief definieren, tenzij men additionele symbolen invoert, bv.

$<\text{identifier}>_{1}^{6}$

om aan te geven dat een identifier uit tenminste één en ten hoogste 6 elementaire symbolen kan worden opgebouwd.

In Cobol, een taal die vooral voor administratieve toepassingen is ontwikkeld, is vastgelegd dat een naam, het begrip dat het meest op identifier lijkt, opgebouwd kan worden uit cijfers, hoofdletters en het - teken. Het aantal symbolen is tenminste 1 en ten hoogste 30, er moet tenminste één hoofdletter in zitten, twee of meer -symbolen direct achter elkaar zijn niet toegestaan terwijl ze ook niet op de eerste of laatste plaats mogen voorkomen, blanco's mogen niet in een naam voorkomen. Het is dan ook niet te verwonderen dat de BNF notatie voor de Cobol taal vrijwel onbruikbaar is en dat men voor de formele beschrijving van Cobol een ander systeem heeft uitgedacht waarbij gebruik wordt gemaakt van rechte haken (constructies tussen rechte haken mogen weggelaten worden), accoladen (uit wat tussen accoladen staat moet precies één keus gemaakt worden), puntjes (om aan te duiden dat de aan de puntjes voorafgaande structuur tussen rechte haken of accoladen een willekeurig aantal keren herhaald mag worden), enz.. **Een kopie van het Algol-rapport is aan deze syllabus toegevoegd.**

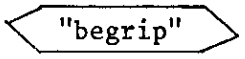
1.23 Syntaxgraph

Naast de beschrijving van Algol, o.a. in het officiële rapport, met de BNF notatie is de behoefte ontstaan aan een meer visuele representatie van een taalsyntax om bijvoorbeeld op eenvoudige wijze de volledigheid van een syntax te controleren. Dit kan gebeuren door gebruik te maken van enkele begrippen uit de graphentheorie. Een graph bestaat uit een verzameling knooppunten, waarvan een deel onderling verbonden zijn door lijnstukken of takken. We spreken van een "pad" ter lengte n tussen 2 knooppunten A en B als we n takken kunnen aanwijzen waarlangs we "zonder sprongen" van A naar B kunnen gaan.De graph is "verbonden" (connected) als tussen twee willekeurige knooppunten altijd een pad is aan te wijzen.

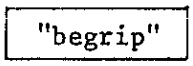
Een boom is dan een bijzondere graph in die zin dat tussen twee willekeurige knooppunten altijd één maar ook slechts één pad is aan te wijzen. Hieruit volgt dat een boom een connected graph is, maar dat hij weglating van een willekeurige tak de resulterende graph niet meer connected is (laten we de tak AB weg en zou het resultaat nog connected zijn dan was er een pad tussen A en B, maar met de tak AB er bij zouden er dan 2 paden zijn, in tegenspraak met de definitie van een boom). Op analoge wijze kunnen we ook bewijzen dat in een boom ook geen "cycles" (gesloten paden) kunnen voorkomen.

Op de theorie van graphen zullen we hier niet verder ingaan. Het is echter duidelijk dat een stroomdiagram een voorbeëld is van een georiënteerde graph d.w.z. een graph waarin de takken slechts in één richting doorlopen mogen worden (laat de stroomdiagramsymbolen maar inkrimpen tot knooppunten). Een Algol programma heeft dank zij de blockstructuur duidelijk een boomkarakter: een tak van een knooppunt naar een "lager" knooppunt correspondeert met het binnengaan van een (sub)block. Een goto statement kan uit een dieper gelegen block wel leiden naar een hoger block, maar niet omgekeerd. Evenmin kunnen blocken in cycles zitten, m.a.w. een block is nooit de voorganger van zichzelf (de recursieve aanroep van procedures is hiermee niet in strijd omdat de procedure call formeel een overschrijven van de procedure op de plaats van de call inhoudt!).

De in de volgende twee voorbeelden te behandelen syntaxgraphen vertonen overeenkomst met geöriënteerde graphen. Zij representeren de Algol constructieregels voor resp. getallen en simple arithmetic expressions. In de voorbeelden zijn de volgende afspraken verwerkt:



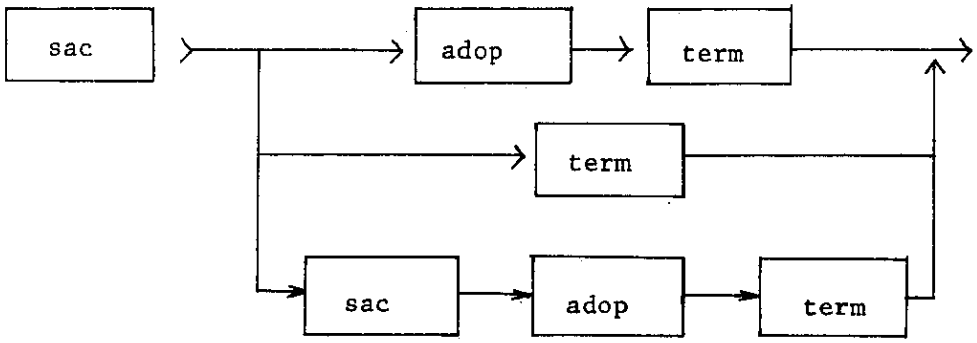
: duidt aan dat de definitie van "begrip" gevonden wordt door de van dit hokje naar beneden gaande pijlen te volgen. De gestippelde horizontale pijlen geven een verplichte opvolging van hokjes aan.



: duidt aan dat "begrip" elders gedefinieerd wordt.

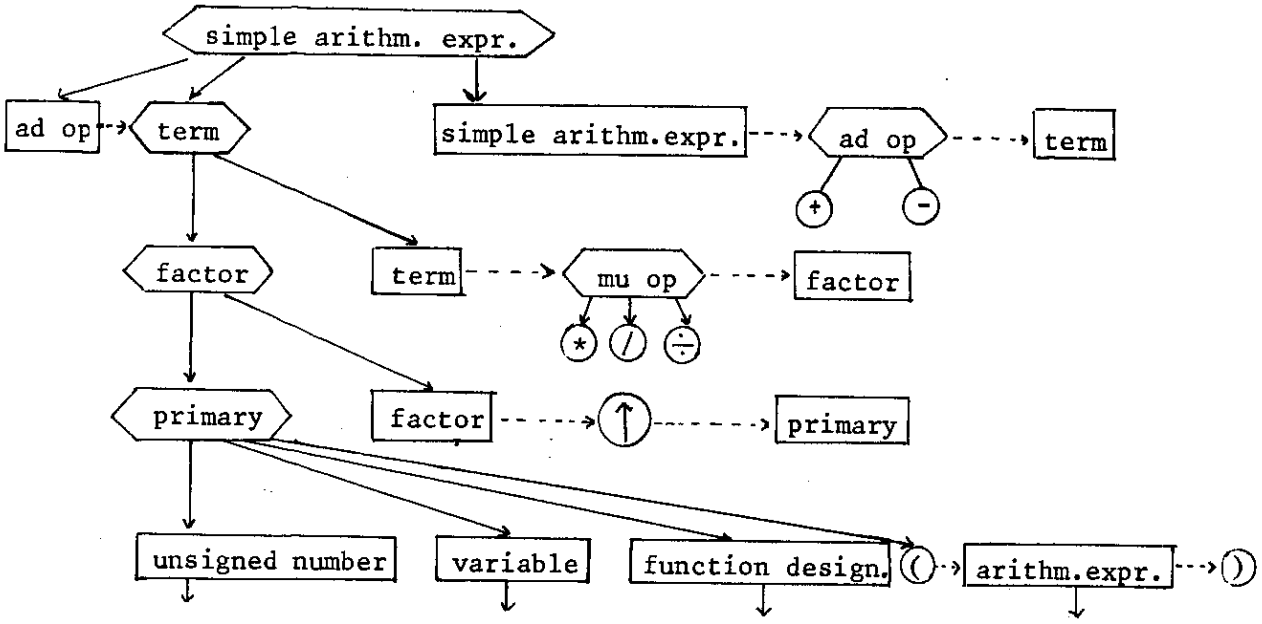
: om een symbool geeft aan dat het een elementair symbool is.

Met deze toelichting en de BNF tekst van het Algol rapport zijn de twee voorbeelden nu eenvoudig te lezen. Het zal overigens duidelijk zijn dat voor ingewikkelde constructies zelfs dit plaatje onoverzichtelijk wordt. Vaak splitst men zo'n plaatje dan op in stukken, bv. als volgt

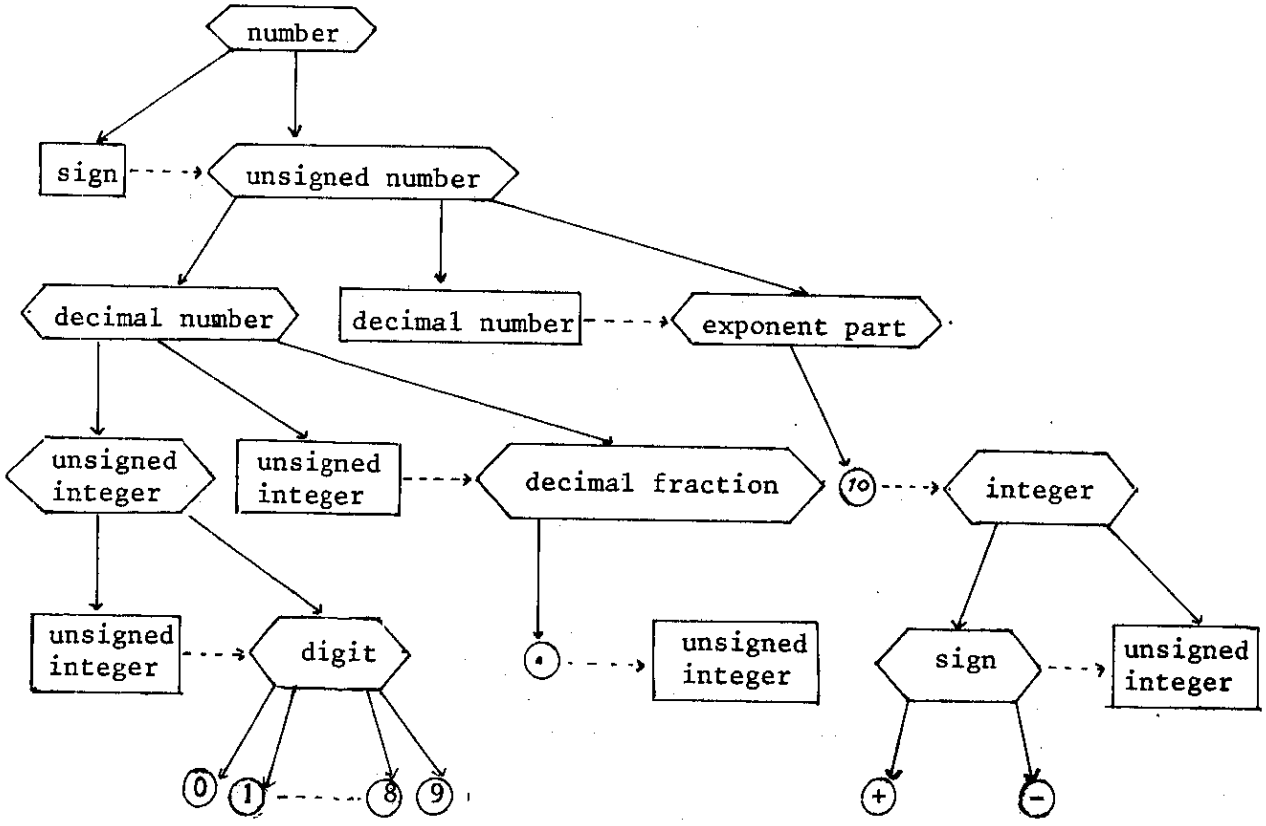


Het gemak van syntaxgraphen is ten eerste dat zij voor de lezer de vergelijking van twee talen aanzienlijk versnellen(vooral wanneer andere woorden worden gebruikt voor dezelfde begrippen). Ten tweede vereenvoudigt een syntaxgraph de taak om na te gaan of een gegeven zin in overeenstemming is met de syntax van een taal (het "parsing" proces). Ga dit zelf na aan de hand van de expressie a - (b - c * d)/(e - f) + 2, aannemend dat we al weten dat a t/m f variabelen zijn.

Het menselijk oog kan de verschillende typen hokjes en pijlen direct onderscheiden, maar een rekenautomaat niet. Als we het parsing proces door een rekenautomaat willen laten uitvoeren, zullen we voor deze hokjes en pijlen een aan een rekenautomaat aangepaste representatie moeten gebruiken. Op deze problemen gaan we in dit college echter niet in.



Syntaxgraph voor simple arithmetic expressions in Algol 60



Syntaxgraph voor getallen in Algol 60

1.24 Karakteristieken van enkele programmeertalen

Een functie van een programmeertaal is communicatie tussen een rekenautomaat en zijn gebruikers mogelijk te maken, zodat een rekenautomaat de door de gebruiker gewenste reeks van bewerkingen kan uitvoeren. Welke bewerkingen uitgevoerd en welke grootheden bewerkt kunnen worden, is in de syntax van de taal opgesloten.

Wanneer de uit te voeren bewerkingen direct corresponderen met de elementaire bewerkingen die, zoals we in het volgende hoofdstuk zullen zien, karakteristiek zijn voor een rekenautomaat, dan spreken we van een machinetaal of een assembleertaal. In andere gevallen spreekt men als regel van "hogere" talen, daarbij nog onderscheid makend tussen:

- probleem-georiënteerde talen, die ieder voor zeer bepaalde toepassingen ontwikkeld zijn, zoals bijvoorbeeld SIMULA, SIMSCRIPT, GPSS voor simulatieproblemen, STRESS voor mechanische sterkteberekeningen, SNOBOL en LISP voor tekstmanipulatie, enz.. De in deze talen gebruikte terminologie is als regel direct ontleend aan het betreffende toepassingsgebied en daardoor voor hen die niet vertrouwd zijn met dat toepassingsgebied bij een eerste kennismaking soms niet doorzichtig. Op deze talen zullen we in dit college niet nader ingaan.
- "procedure talen" als Algol, Fortran en Cobol, die zo'n grote graad van algemeenheid bezitten, dat "vrijwel alle" problemen hierin te programmeren zijn. Toch zijn deze talen in de eerste plaats voor bepaalde klassen van toepassingen ontwikkeld (de eerste twee voor mathematisch/technische, de derde voor administratieve; PL/1 verenigt weer eigenschappen van deze drie talen in zich). Vaak wordt het woord algorithmische taal als synoniem voor proceduretaal gebruikt, al moet dan eigenlijk meer gedacht worden aan een taal die bij uitstek geschikt is om er (wiskundige) algorithmen mee te beschrijven, zoals bijvoorbeeld Algol.
- "besturingstalen" (control languages), waarmee niet algorithmen worden geformuleerd, maar waarmee het verloop van de verwerking van een programma "gestuurd" wordt, d.w.z. met welke machinefaciliteiten, prioriteiten, enz..

Opmerking: Er is geen eenheid t.a.v. het gebruik van de hiervoor ingevoerde termen; deze is ook niet te verwachten omdat er nauwelijks strenge definities voor deze termen te geven zijn. De meeste talen bezitten, al zijn ze op grond van hun kernmerken tot een bepaalde groep te rekenen, bovendien enkele karakteristieken van andere talen.

Voor de drie proceduretalen Algol 60, Fortran IV, Cobol 68 is reeds een internationale standaardisatie bereikt. Andere talen (vermoedelijk tientallen) hebben om verschillende redenen dit stadium van standaardisatie (nog) niet bereikt:

- ze zijn niet ontwikkeld om er procesbeschrijvingen in een voldoende ruim toepassingsgebied mee te formuleren;
- ze liggen niet "voldoende in het gehoor", m.a.w. zijn door een onnodig ingewikkelde syntax of door onconventionele symbolen of symboolgebruik niet geschikt om die taal vlot aan te leren of in die taal geschreven programma's snel te doorzien;
- ze betekenen t.o.v. de genoemde drie talen geen essentiële verbetering;
- ze vereisen bijzondere rekenautomaatfaciliteiten of hulpapparatuur (vooral invoer- en uitvoerstations).

1.24.1. Motivering voor de proceduretalen

Algol 60 dankt zijn oorsprong (1960) aan het inzicht dat zijn primitieve voorganger, Fortran, niet geheel voldeed aan de wens dat een wiskundige algorithme op zo natuurlijk mogelijke wijze gerepresenteerd moet kunnen worden met een zo klein mogelijk aantal primitieven en regels. Kenmerkend voor Algol zijn structuren als compound statement, blocks en procedures, de uitgebreide parameter-mechanismen en de recursieve aanroepmogelijkheden. Het aantal datastructuren is echter zeer beperkt.

Fortran (samentrekking van FORmula TRANslation) was de eerste taal (1955) waarmee algebraïsche en logische uitdrukkingen op de gebruikelijke wijze geformuleerd konden worden. Op een later tijdstip werd ook het procedure mechanisme (echter zonder call-by-name en recursie-faciliteiten) ingevoerd in een vorm die het onafhankelijk van elkaar schrijven en testen van hoofdprogramma en procedure mogelijk maakte. De toenmalige kennis van programmeringstechnieken, gekoppeld met de wens om de invoer van een programma met behulp van kaarten mogelijk te maken, voerde tot het bezwaar van allerlei soms ondoorzichtige programmeringsvoorschriften. Van deze voorschriften heeft men

echter weinig last, zolang men zich niet aan "trapezewerk" waagt. Fortran is waarschijnlijk de meest gebruikte programmataal ter wereld.

Cobol (samentrekking van COmmon Business Oriented Language) is het gevolg van een door een grootgebruiker afgedwongen compromis (1960) tussen door vele fabrikanten in ontwikkeling zijnde programmeertalen voor de verwerking van administratieve problemen. Deze problemen worden gekenmerkt door datastructuren die uit vele componenten zijn samengesteld. Voor een bepaalde berekening zijn meestal echter slechts enkele componenten relevant, die bovendien ook niet-numerieke "waarden" kunnen hebben. Met Cobol wordt voor de behandeling van deze soort problemen inderdaad een oplossing gevonden; enkele bezwaren zijn echter:

- de veelheid van formuleringsmogelijkheden (gevolg van het compromis);
- de poging om de taal een deelverzameling van "gewoon" Engels te doen zijn (voorbijgaande aan het feit dat een programmeertaal en een levende taal verschillende functies hebben);
- de verwaarlozing van de verworvenheden van een algebraïsche notatie en het vrijwel ontbreken van een procedure of block mechanisme;
- het feit dat een spatie meestal een betekenis heeft en daardoor vaak aanleiding geeft tot ondoorzichtige programmeringsregels en tot verschrijvingen.

1.24.2. Elementaire symbolen (karakterset) van Algol, Fortran en Cobol

Hoe praktijkeisen kunnen doorwerken in de specificaties van een taal blijkt bijvoorbeeld uit de elementaire symbolen van de drie talen. De ontwerpers van Algol hebben zich grotendeels losgemaakt van overwegingen welke invoer- en uitvoerfaciliteiten beschikbaar waren of zouden komen en kwamen met een karakterset die bestaat uit ruim 100 symbolen, nl. alle 26 hoofdletters en 26 kleine letters, 10 cijfers, de gebruikelijke interpunctie , arithmetische en logische symbolen en tenslotte een aantal symbolen opgebouwd uit onderstreepte woorden. Met een 7 of 8 kanaalsponsband en de bijbehorende bandponser zijn deze symbolen betrekkelijk eenvoudig te representeren, ook de onderstreepte woorden. Het laatste is niet het geval bij gebruik van ponskaarten en kaartponsers zodat men de onderstreping dan vaak vervangt door het tussen apostrofen plaatsen. Met de gebruikelijke kaartponsers zijn niet meer dan circa 50-65 symbolen in een kaartkomen te representeren, zodat talen als Fortran en Cobol (en BEATHE, een Algol omvattende taal voor de Burrough 6700) een karakterset hebben die slechts bestaat uit 26 hoofdletters, 10

cijfers en de meest voorkomende "speciale" symbolen. Beperkingen t.g.v. een ponskaart en een toetsenbord maken het niet waarschijnlijk dat meer uitgebreide kaartponsers dan die welke ongeveer 65 symbolen kennen, beschikbaar zullen komen (voor regeldrukkers is de situatie overigens wel beter). Een gevolg van deze beperking is o.a. dat voor het := symbool uit Algol in Fortran en Cobol het = symbool wordt geschreven! De equivalenten van true, false, >, ≥, =, ≠, <, ≤, ¬, ∨, ∧ zijn bij gebruik van ponskaarten vaak 'TRUE', 'FALSE', 'GTR', 'GEQ', 'EQL', 'NEQ', 'LSS', 'LEQ', 'NOT', 'OR', 'AND'.

Ten aanzien van het analogon van de onderstreepte woorden uit Algol zijn in Fortran en Cobol verschillende wegen bewandeld. In Fortran is de syntax zo opgesteld dat een rekenautomaat ook zonder onderstreping wel kan uitmaken of een bepaald woord een "keyword" of een identifier is (op zichzelf een elegante oplossing). Bij het veel omslachtiger Cobol was dat niet mogelijk, zodat een lijst ontstond van ruim 200 "reserved words", welke men beslist niet als identifiers mag gebruiken. Ondanks het gemis van elegance, ziet het er niet naar uit dat Cobol gauw zal verdwijnen, omdat andere bekende talen (met uitzondering van het nieuwe PL/1 en Algol 68) in de praktijk slechts de manipulatie van "numerieke" waarden toelaten.

1.24.3. Programma besturing

Zoals we reeds weten, worden Algol statements in de opgeschreven volgorde afgewerkt, behoudens de "besturing" met de

```
if ..... then ..... else ..... ;
for ..... do ..... ;
goto ..... ;
```

constructies. In het volgende zullen we schetsmatig aangeven hoe deze constructies er in andere talen uitzien.

In Fortran hebben we in grote trekken dezelfde constructies, al zijn ze minder systematisch en algemeen. Zo kennen we de

- "arithmetische if" met syntax:

  IF(<simple arithm. expr.>) label-1, label-2, label-3

  waarbij de volgende statement door label-1, label-2 en label-3 bepaald wordt al naar gelang de waarde van de simple arithmetische expressie kleiner dan, gelijk aan of groter dan nul is.

- "logische if" met syntax:

  IF(<simple logical expr.>) <executable statement>

  waarbij de executable statement (die geen DO of andere logische IF mag zijn) wordt uitgevoerd als de waarde van de logical expression true is,
- de "lusbesturing" met syntax:

  DO <label bij laatste lusstatement> var. = begin-, eind-, incrementwaarde
- de *spronginstructies* met syntax:

  GOTO <label>
  GOTO (<label list>), <variable, waarvan de waarde de label indexeert>

  (als equivalent van de "switch" faciliteit) en nog een minder vaak gebruikte GOTO vorm.

Verder kent Fortran nog de CONTINUE als dummy instructie, de STOP en (weinig gebruikte) PAUSE statements, die als equivalent van de laatste end in Algol gebruikt moeten worden.

Dat Cobol iets jonger is dan Fortran blijkt o.a. uit de conditional statement

IF <condition>; <statement>; ELSE <statement>

In tegenstelling tot Algol mag het ELSE gedeelte als regel niet weggelaten worden; achter het then gedeelte mag dan ook (in tegenstelling tot Algol) weer een conditonal statement staan. In tegenstelling tot Algol omvat de condition verder niet alleen de <boolean term>, maar ook de zg. "class conditions" omdat "string operanden" toegelaten zijn. Zo kunnen ook vragen als <identifier> IS NUMERIC of <string> IS EQUAL TO <string> het resultaat true of false opleveren en daarmee verschillende programmawegen bewerkstelligen.

Lussen kunnen in Cobol geprogrammeerd worden met behulp van PERFORM statements. Een aantal vormen hiervan zijn:

- PERFORM <paragraafnaam> <zoveel> TIMES

  de bij paragraafnaam (lijkt op block bij Algol) behorende statements worden "zoveel" keer uitgevoerd, waarbij "zoveel" een integer of een "integer variable" is,

- PERFORM <paragraafnaam> THRU <paragraafnaam> <zoveel> TIMES

  voor het "zoveel" keer uitvoeren van opeenvolgende paragrafen inclusief de in de statement genoemde eerste en laatste,

- PERFORM <paragraafnaam> [THRU <paragraafnaam>] UNTIL <conditie>

  als analogon van de while vorm van een Algol for-statement (het stuk tussen rechte haakjes mag weggelaten worden),

- PERFORM <paragraafnaam> [THRU <paragraafnaam>]
  VARYING <identifier> FROM <variable> BY <variable> UNTIL <conditie>
  [AFTER <identifier> FROM <variable> BY <variable> UNTIL <conditie>
  [AFTER <identifier> FROM <variable> BY <variable> UNTIL <conditie>]]

als soort mengvorm van de step-until en de while vormen van de Algol for-statement. Zoals aangegeven (en in overeenstemming met de maximum array dimensie van 3) kunnen maximaal drie lussen met één PERFORM statement worden beheerst (de laatst/eerst genoemde identifier beheerst hierbij de binnenste/buitenste lus). Voorts is <variable> weer hetzij een integer variable, hetzij een integer. De Cobol

GOTO <paragraafnaam>

is weer geheel analoog aan de Algol en Fortran vormen, terwijl net als in Fortran een programma moet eindigen met een STOP statement.

Deze schets is uiteraard geen handleiding voor Fortran en Cobol, doch slechts een aanduiding hoe gelijksoortige structuren in verschillende talen ingevoerd zijn.

1.24.4. Input en output

Input en output zijn daarom ingewikkeld omdat niet alleen informatie-uitwisseling tussen buitenwereld en rekenautomaat plaats vindt, maar meestal ook iets gezegd moet worden over de manier waarop die informatie bijvoorbeeld op een kaart of op een getypte pagina is of wordt vastgelegd. Bij deze "opmaak" van informatie moeten we denken aan zaken als kantlijnmarges, tabulatorstops, aantal regels op een pagina, plaats van de eerste regel, enz.. Omdat "opmaak"-mogelijkheden sterk afhankelijk zijn van de beschikbare input/output apparatuur, heeft het vaak lang geduurd voor men hiervoor een gestandaardiseerde beschrijving had. Als er verschillende invoer/uitvoerorganen

zijn, moet bovendien vastgelegd worden welk van deze organen betrokken is in de informatie-uitwisseling.

In Fortran heeft men de opmaak van input en output vereenvoudigd door a.h.w. voor iedere "regel" (d.i. de informatie in één ponskaart of één gedrukte regel) een blauwdruk te definiëren met behulp van een zg. FORMAT statement. De grondgedachte hierbij is voor iedere positie op een regel te definiëren wat voor soort informatie daar moet verschijnen.
De volledige FORMAT behandeling vergt enkele pagina's druk (zie bv. CACM 7 (1965) 591 en 8 (1966) 287 of een Fortran handleiding); we zullen dit niet verder behandelen. In de praktijk geeft het FORMAT gebruik niet veel problemen, mits men niet een fraaie opmaak wenst. Wat dus wel veel hoofdbrekens kost, is een tabellarische opmaak, waarbij bv. getallen die toevallig nul zijn niet afgedrukt mogen worden. Ook hier zullen we echter niet op ingaan.

In Cobol is de situatie in zekere zin eenvoudiger dan in Fortran omdat de volledige opmaak van invoer en uitvoer al volledig uitgespeld moet worden in de data beschrijving.

In Algol is het aan iedere installatie overgelaten hoe zij hun invoer en uitvoer willen organiseren, maar in 1964 zijn twee voorstellen tot standaardisatie gepubliceerd (CACM 7 (1964) 273, 628). Hoewel één van deze systemen a.h.w. een uitbreiding is van het FORMAT mechanisme, heeft het noch het andere systeem, noch de vele plaatselijke conventies kunnen verdringen. Een behandeling van deze systemen zal hier daarom niet gegeven worden, doch in de appendix zijn de meest gebruikte THE-conventies samengevat.

1.24.5. Terugblik

Het is evident dat deze paragraaf 1.24 niet bedoeld is als een uittreksel uit een handleiding voor Fortran of Cobol programmering. (Hiervoor bestaan vele boeken, waarbij men er echter wel op dient te letten dat alleen na 1968 verschenen boeken relevant zijn, gezien de veranderingen aangebracht in de definities van deze talen. Handleidingen van rekenautomaatfabrikanten leggen veelal de nadruk op faciliteiten die specifiek zijn voor hun machines!) Het is daarentegen wel bedoeld om een indruk te geven van de wijze waarop men getracht heeft om op ondubbelzinnige wijze communicatie met een rekenautomaat te bewerkstelligen.

Appendix

Enkele standaard uitvoerprocedures

nlcr

- de schrijfmachinewagen wordt naar de beginpositie verplaatst en het papier wordt één regel opgevoerd. Als door het uitvoeren van nlcr ("new line carriage return") het aantal regels op de pagina 60 zou overschrijden wordt automatisch gegeven een

new page

- het papier wordt naar een nieuwe pagina opgevoerd en de wagen wordt naar de beginpositie verplaatst.

space(n)

- er worden n spaties gegeven; als daarbij de regellengte overschreden zou worden, last het systeem automatisch nlcr in.

printtext(‡string‡)

- de tussen "string quotes" geplaatste string wordt als zodanig afgedrukt; als daarbij de regellengte wordt overschreden, last het systeem automatisch een nlcr in.

flot(n,m,x)

- de waarde van de real x wordt als volgt afgedrukt: teken, dan een exact afgeronde decimale breuk met n cijfers achter de decimaalpunt, dan het scheidingssymbool $_{10}$, dan het teken van de exponent en tenslotte de exponent in m cijfers (met vervanging van niet significante nullen door spaties) gevolgd door een spatie. Het aantal gebruikte posities is n + m + 5; als het aantal symbolen op een regel meer dan de regellengte zou worden, last het systeem automatisch eerst een nlcr in. Als niet voldaan is aan $1 \leq n \leq 13$ en $1 \leq m \leq 3$ wordt automatisch flot(13,3,x) uitgevoerd.

fixt(n,m,x)

- de waarde van de real x wordt als volgt afgedrukt: teken, dan een decimaal getal met n cijfers voor de decimaalpunt (met vervanging van niet significante nullen door spaties en opschuiving van teken) en m cijfers (exact afgerond) achter de decimaalpunt en tenslotte weer een spatie. Als m = 0 vervallen decimaalpunt en breukgedeelte. Het aantal gebruikte posities is n + m + 3 als m > 0 en n + 2 als m = 0; als het aantal symbolen op een regel meer dan de regellengte zou worden, last het systeem automatisch eerst een nlcr in. Als niet voldaan is aan $n \geq 0$, $m \geq 0$ en $1 \leq n + m \leq 21$ wordt automatisch flot(13,3,x) uitgevoerd.

absfixt(n,m,x)

- als het voorgaande behoudens vervanging van het teken door een spatie.

print(x)

- als de waarde van x een geheel getal is wordt afgedrukt volgens fixt(13,0,x) gevolgd door 6 spaties. Zo niet, dan volgens flot(13,3,x) zodat in beide gevallen het geheel 21 posities beslaat. Zo nodig wordt automatisch weer een nlcr ingelast.

2.0. APPARATUUR EN MACHINETAAL

2.1. Inleiding

In het eerste hoofdstuk was uiteengezet dat een rekenautomaat (computer) over een aantal onderdelen moet beschikken om zijn taken te kunnen vervullen, nl. de taken van opdrachten en gegevens te kunnen lezen, deze te verwerken en het resultaat van de bewerking weer zichtbaar te maken. De daarbij benodigde onderdelen zijn invoerorgaan, geheugen, rekenkundig orgaan, uitvoerorgaan en het besturingsorgaan dat - de aanwijzingen van het programma volgend - voor de executie van een karwei zorgt. Het eerste hoofdstuk was echter verder gewijd aan de wijze waarop een opdrachtenreeks in een algorithmische taal als Algol geformuleerd moet worden voor verwerking door wat een Algol-automaat genoemd kan worden. De eigenschappen van zo'n taal zijn helemaal aangepast aan de menselijke manier van denken, opdat een programma zo duidelijk en eenvoudig mogelijk geschreven kan worden.

Hóe werkt echter een rekenautomaat, hóe wordt informatie daarin vastgelegd en hóe is de taal die een electronische rekenautomaat rechtstreeks "verstaat"? Het zal bekend zijn dat informatie instrumenteel vastgelegd kan worden in digitale vorm (telraam, telefoonschijf, 10-positie wielen in electriciteitsmeters of kilometertellers, enz.; dus met behulp van discrete eenheden) of analoge vorm (rekenlineaal, snelheidsmeters in auto's, vloeistofniveau's, manometers, enz.). De eerste gedachte zal daarom zijn dat informatie het eenvoudigst in analoge vorm vast te leggen is in een electronisch apparaat, bv. in de vorm van spanningen of stromen. Sommige wiskundige bewerkingen zijn dan betrekkelijk eenvoudig en bovendien zeer snel uit te voeren, bv. een tekenomkeer, een optelling en zelfs een bewerking als integreren (die immers te representeren is met het opladen van een condensator). Een vermenigvuldiging en in nog sterker mate een deling of een differentiatie zijn met electronische analoge hulpmiddelen echter niet snel en simpel te realiseren omdat er geen eenvoudige fysische processen zijn, die met deze bewerkingen corresponderen (zelfs kwadrateren gaat nog zoveel eenvoudiger, dat een vermenigvuldiging $p * q$ vaak uitgevoerd wordt door gebruik te maken van de relatie $p * q = \frac{1}{4}((p + q)^2 - (p - q)^2)$).

Zulke electronische analoge rekenautomaten zijn inderdaad ontwikkeld en in gebruik genomen, maar hun aantal is slechts een fractie van het aantal digitale rekenautomaten. Redenen hiervoor zijn o.a. de volgende:

- de exactheid van informatie vastlegging en van een elementaire bewerking wordt niet alleen beperkt door de nauwkeurigheid en reproduceerbaarheid van het fabricageproces en door veroudering van componenten, maar principeel ook door de onvermijdelijke "ruis" (veroorzaakt door de Brown beweging van electronen). In de praktijk is het daarom vrijwel onmogelijk om voor een met analoge signalen werkend electronisch apparaat meer dan ongeveer vier decimalen nauwkeurigheid te garanderen bij één enkele elementaire bewerking, zodat zonder bijzondere voorzorg voor een complex van bewerkingen niet veel meer dan twee à drie decimalen nauwkeurigheid overblijft (voor veel technisch werk is dit overigens soms voldoende);
- analoge rekenautomaten functioneren meestal in een zeer nauw werkgebied, bv. -100 tot +100 volt (of bij kleinere machines -10 tot +10 volt). Dit brengt met zich mee dat de in een probleem voorkomende grootheden meestal "geschaald" moet worden, opdat ze door spanningen in dit werkgebied gerepresenteerd kunnen worden. Schalen is ook nodig om zo min mogelijk nauwkeurigheid te verliezen bij complexe bewerkingen, maar is vrij bewerkelijk en vraagt een gedegen kennis van de numerieke wiskunde (een nauwkeurige analyse van het probleem wordt bij een digitale aanpak, vertrouwend op de dan te bereiken nauwkeurigheid van 8 tot 16 decimalen per elementaire bewerking, veelal - ten onrechte - achterwege gelaten);
- per elementaire bewerking is een machine component nodig, zodat een "groot" probleem ook een groot aantal componenten vereist, vaak meer dan in de aanwezige machine beschikbaar is. Bij een digitale machine vergt een "groot" probleem daarentegen in de eerste plaats een lange verwerkingstijd, omdat elementaire bewerkingen sequentiëel worden uitgevoerd. Bij een analoge rekenautomaat is de verwerkingstijd vrijwel onafhankelijk van de grootte van het probleem en als regel hooguit enkele minuten;
- de in electronische schakelingen soms onvermijdelijk optredende, ongewenste oscillaties beperken de toepassingsmogelijkheden van analoge rekenautomaten in de praktijk tot het gebied van gewone (niet noodzakelijk lineaire) differentiaalvergelijkingen en niet al te ingewikkelde partiële differentiaalvergelijkingen. Daar veel problemen uit de technische fysica en chemie tot dit gebied horen, moet het nut van analoge rekenautomaten hier echter overwogen worden. Voor matrixproblemen en problemen met veel tabellarische uitvoer zijn analoge automaten daarentegen weinig geschikt;

- van het programma en het met een digitale machine verrichte werk is eenvoudig automatisch een volledig "proces-verbaal" te maken, wat niet het geval is voor werk met een analoge machine.
  Het gebruik maken van vroeger of door anderen gedaan werk is bij digitale machines eenvoudig dank zij het procedure mechanisme; ieder probleem aangepakt met een analoge machine moet men eigenlijk weer opnieuw analyseren.

De laatste jaren heeft men, door het koppelen van analoge en digitale machines tot zgn. hybride rekenautomaten, getracht om de voordelen van de twee typen machines te verenigen. Ondanks de problemen die daarbij overwonnen moesten worden (vooral te wijten aan de verschillende tijdschalen waarin de rekenprocessen in de twee machinetypen verlopen), is dit vaak met succes gedaan getuige bijvoorbeeld technische prestaties in de lucht- en ruimtevaart, die mogelijk zijn geworden door o.a. simulaties met behulp van hybride automaten. In toenemende mate zien we dat digitale rekenautomaten ook direct "gekoppeld" worden aan fysische of chemische processen (via meetinstrumenten, die als regel analoge signalen afgeven, welke in digitale gegevens omgezet moet worden) en deze processen ook "sturen". De verwachting is dat de rol van "general purpose" analoge en hybride rekenautomaten (niet die van "special purpose" automaten!) over enkele jaren uitgespeeld zal zijn door de verdere ontwikkeling van digitale technieken.

Noch op analoge en hybride rekenautomaten, noch op de electronische schakelprincipes van digitale automaten zal in dit college ingegaan worden. Men zij hiervoor verwezen naar de betreffende college's in andere afdelingen. De logische structuur en enkele eigenschappen van digitale machines zullen alleen zover behandeld worden als nodig is voor een goed begrip van wat tegenwoordig programmatuur ("software" in tegenstelling tot "hardware") heet. Programmatuur omvat het geheel van standaardprogramma's ten dienste van de gebruiker om op eenvoudige wijze een rekenautomaat te benutten. Men onderscheidt in de programmatuur: de "harde" of installatie-programmatuur (meestal door de computerfabrikant geleverd) die voor vele soorten toepassingen bruikbaar is, en de "zachte" programmatuur (veelal zelf geschreven) die direct op een bepaald type toepassing gericht is. Voorbeelden van harde software zijn programma's die een Algoltekst omzetten in een taal die de machine direct begrijpt. Op het eerste gezicht is het verrassend te horen dat een machine een Algoltekst niet direct begrijpt, maar wel om kan zetten in een andere taal. Deze omzetting verloopt echter volgens een vast bewerkingsschema, waarvoor eens en

vooral voor een bepaalde machine een programma geschreven kan worden. Voor dit "vertaalprogramma" zijn de Algol statements dan de gegevens en het resultaat is de corresponderende tekst in de nieuwe taal! Voorbeelden van zachte of applicatie programmatuur zijn o.a. programma's voor optimalisatie berekeningen, voor het doorrekenen van electrische netwerken of van mechanische constructies, voor het numeriek besturen van gereedschapswerktuigen, voor het zetten van kranten, enz. De grens tussen harde en zachte programmatuur is tot op zekere hoogte natuurlijk willekeurig en zal, naarmate omstandigheden veranderen, kunnen verschuiven. Nu automatische tekenmachines ("plotters") vaak aan rekenautomaten gekoppeld worden, is bijvoorbeeld niet meer duidelijk tot welke categorie algemene plotterprogramma's gerekend moeten worden.

Wanneer over de aanschaf van rekenautomaten gedacht wordt, moet men zeker zoveel aandacht aan de programmatuur schenken als aan de apparatuur. Een moderne machine zonder of met slechte programmatuur is in de praktijk niet bruikbaar omdat het zelf ontwikkelen van programmatuur tientallen manjaren inspanning kost.

2.2. Informatiestructuren en geheugenstructuren

Wij zijn gewend om informatie, die bewaard moet blijven, op schrift vast te leggen. Meestal niet in analoge vorm, bv. met een tekening, omdat dat zelden nauwkeurig genoeg en ook niet altijd mogelijk is. Men bedient zich daarom meestal van digitale middelen, nl. van combinaties van discrete karakters, zoals letters, cijfers en nog een tien tot vijftig "speciale karakters" als +, -, :, enz.

Opmerking. In plaats van het woord karakter wordt ook wel het woord symbool gebruikt, hetgeen wij hier niet zullen doen omdat symbool reeds door ons gebezigd is in de zin van karakter plus zijn betekenis, zoals o.a. in woorden als Algolsymbool. In plaats van het woord symbool zullen wij ook niet het woord teken (bv. in "leesteken") gebruiken omdat we het woord teken reserveren voor + en - symbool.

Een karakter is zodoende een element uit een eindige verzameling, de karakterset, van onderscheidbare dingen. Wanneer in de karakterset een zekere ordening gedefinieerd is, spreken we van een geordende karakterset of alfabet. Voorbeelden van karaktersets zijn de verzamelingen van latijnse, griekse, cyrillische letters, van cijfers, van Braille symbolen, enz.

Een bijzondere rol spelen binaire karaktersets, omdat de twee karakters ("binary digits" of bits) waaruit deze bestaan op allerlei manieren fysisch gerepresenteerd kunnen worden: aan - uit, wel gemagnetiseerd - niet gemagnetiseerd, positieve spanning - negatieve spanning, enz. Op papier kunnen ze ook op allerlei manieren vastgelegd worden, bv. met het cijferpaar 0 - 1, true - false, t - f, gat - geen gat, + -, enz.

Informatie kan dan volgens bepaalde afspraken uit karakterreeksen opgebouwd worden; kan een karakterreeks volgens afspraak als een bepaalde eenheid beschouwd worden, dan spreken we meestal van een woord. Deze definitie sluit direct aan bij het dagelijkse gebruik: een woord is een karakterreeks voorafgegaan en gevolgd door een spatie. Woorden kunnen op hun beurt weer gebruikt worden om woordreeksen, doorgaans zinnen genoemd, te vormen. De verzameling zinnen is een oneindige verzameling.

Enkele bitcodes voor decimale cijfers

cijfer	direct	excess_3	1 uit 10	2 uit 5
0	0000	0011	0 ... 01	11000
1	0001	0100	0 .. 010	00011
2	0010	0101	0 . 0100	00101
3	0011	0110		00110
4	0100	0111		01001
5	0101	1000		01010
6	0110	1001		01100
7	0111	1010		10001
8	1000	1011		10010
9	1001	1100	10 ... 0	10100

Men kan karakters uit een bepaalde verzameling afbeelden op woorden gevormd uit de karakters van een andere verzameling. Beeldt men in het bijzonder een karakterset af op woorden gevormd uit de karakters van een binaire karakterset, dan spreekt men van een bitcode. Voorbeelden van zulke codes zijn de Morsecode (die niet een vaste lengte heeft) en de codes voor (schrijfmachine) letters, cijfers en symbolen in ponsband- en ponskaartvorm. (In het algemeen is een code een afbeelding van een karakterset op een andere karakterset.) Dat de afbeelding op vele manieren tot stand kan worden

gebracht blijkt uit de tabellen. De reden dat men vooral voor transmissie van informatie zoveel soorten codes ontwikkeld heeft, is meestal dat men de ontvanger van gecodeerde informatie in staat wil stellen om te constateren of zijn ontvangst goed is geweest, dan wel om verminkte informatie te reconstrueren. Hierbij zijn codes van vaste lengte eenvoudiger te behandelen dan die van wisselende lengte, maar in beide gevallen zullen zender en ontvanger wel "in de pas" moeten lopen, m.a.w. ze moeten "gesynchroniseerd" zijn. Op het ontwikkelen van codes en coderingstheorie, een onderdeel van de discrete wiskunde zullen we niet verder ingaan, evenmin op de problemen van herkennen van ontvangen gecodeerde informatie zoals in de informatietheorie behandeld worden.

Voor het één-éénduidig afbeelden van de tien cijfers op een binaire karakterset zijn 4 bits voldoende, men spreekt dan veelal van tetraden. Een getal opgebouwd met decimale cijfers kan men dan afbeelden op een reeks tetraden. Wil men behalve cijfers echter ook karakters op een binaire karakterset afbeelden, dan zijn uiteraard groepen van meer dan 4 bits nodig. Lange tijd zijn groepen van 6 bits, hexaden, gebruikelijke geweest daar hiermee $2^6 = 64$ karakters afgebeeld kunnen worden, hetgeen ruim voldoende is voor 26 (hoofd) letters, 10 cijfers en nog een aantal speciale karakters. Onder ander de wens om naast hoofdletters ook kleine letters af te kunnen beelden, heeft aanleiding gegeven tot de nu voorgestane standaardisatie van afbeelding op groepen van 8 bits of octaden. Als verzamelnaam voor tetraden, hexaden en octaden wordt het woord byte gebruikt.

Bij gebruik van 8 bits bytes vereist de afbeelding van een getal van n decimale cijfers in feite $m = 8 (n + 2)$ bits, omdat ook een byte nodig is om het teken van een getal vast te leggen en nog een byte om op de een of andere manier aan te duiden dat het einde van het getal bereikt is. Deze zgn. "binair gecodeerde decimale" representatie laat zo karakterreeksen of woorden van variabele lengte toe, maar is in aantal bits aanmerkelijk onvoordeliger dan een zgn. "directe binaire representatie" van getallen. Immers met m bits kan men een teken en een integer met een waarde van maximaal $2^{m-1} - 1$ krijgen. (Dus in totaal 2^{m-1} integers beginnend met 0 en eindigend met $2^{m-1} - 1$; dit is eenvoudig door inductie te bewijzen.) Voor deze directe representatie gebruikt men als regel woorden van vaste lengte (met woordlengten tussen 16 en 60 bits), waarbij de lengte gekozen wordt op grond van het grootste getal dat men nog exact wil voorstellen. Een nadeel van een vaste woordlengte is uiteraard dat ook kleine getallen het volledige aantal

bits van een woord in beslag nemen. Een ander nadeel is dat een reeks van letters óf een reeks van even zo vele woorden vergen óf dat bepaalde aantallen letters "samengepakt" worden in een woord, waarbij "inpakken" en "uitpakken" tijd vergt. Delen van een woord, ook weer bytes of ook wel syllaben genoemd, zijn als regel 6 of 8 bytes lang zodat de woordlengte meestal een veelvoud van 6 of 8 is.

Opmerking. Controle_bits, waarmee de correctheid van woorden of bytes gecontroleerd kan worden, zullen we steeds buiten beschouwing laten! De paritybit, waarmee bewerkstelligd wordt dat het aantal enen in een woord even (of oneven)is, is een voorbeeld van een controle bit, maar er zijn ook uitgebreidere methoden!

In overeenstemming met het voorgaande kan men in de geheugenstructuren van rekenautomaten twee typen onderscheiden: de "karakter- of byte-georienteerde" geheugens voor variabele lengte informatie en de "woord-georienteerde" geheugens voor vaste lengte informatie. Daar beide typen geheugens opgebouwd zijn uit magneetkerntjes (vandaar de naam kernengeheugen), zit het verschil tussen de twee typen eigenlijk in de manier waarop het besturingsorgaan van de automaat een reeks magneetkerntjes "uitleest" en daarmee bepaalde bewerkingen uitvoert. Vroeger liet de ontwerper van een rekenautomaat zich bij de keus van een geheugenstructuur leiden door de overweging of zijn machine primair voor rekenintensief werk (lange wiskundige berekeningen) of primair voor verwerking van alphanumeriek werk (administratieve berekeningen) gebruikt zou worden. Tegenwoordig is het verschil nauwelijks meer relevant omdat een machine nu beschikt over zowel instructies die een variabel aantal bytes (men spreekt dan wel van een veld van zoveel bytes) uitlezen als instructies die een vast aantal bytes, dus een woord, uitlezen en verwerken.

Opmerking. Eenvoudigheidshalve zullen we ons voortaan beperken tot woordgeorienteerde machines en tot het beschouwen van alleen integer getallen. De verder uiteen te zetten principes zijn betrekkelijk eenvoudig aan te passen voor karakter-georienteerde machines en reals.

Een geheugenwoord kan een "informatiestructuur" bv. een integer of een machine-instructie "bevatten". Dit concept, dat naar van Neumann genoemd wordt, dateert pas uit het einde van de veertiger jaren. Voordien dacht men aan gescheiden opslag van instructies en getallen met het gevolg dat toen nog niet gedacht werd aan de mogelijkheid om ook instructies te manipuleren,

een gedachte waarvan men echter weer teruggekomen is. De omvang van een geheugen geeft men nu meestal aan in de eenheid K = 2^{10} = 1024.

Het onderscheid tussen instructies en getallen, geplaatst in een geheugen, is bij sommige machines te constateren met aan een woord extra toegevoegde controle bits. Bij de meeste machines is dit echter niet het geval en moet de gebruiker het besturingsorgaan opgeven waar de eerste instructie in het geheugen staat. De volgende instructies staan er dan als regel achter, terwijl elders in het geheugen de getallen geplaatst worden. Door een foute programmering kan bij verwerking het programma daarom wel eens "ontsporen" (d.i. het besturingsorgaan probeert getallen als instructies te interpreteren).

Opdat woorden door ons (en door het besturingsorgaan) onderscheiden kunnen worden, adresseren wij ze met de natuurlijke getallen, beginnend bij 0. De inhoud van het woord met het "absolute" adres n duiden we voortaan aan met {n}; deze inhoud kan zoals net gezegd een getal of een instructie zijn. We moeten dan goed onderscheid maken tussen

{n + 1}, d.i. de inhoud van het woord na het woord met adres n;
{n} + 1, d.i. de met 1 vermeerderde inhoud van het woord met adres n;
{{n}}, d.i. de inhoud van het woord waarvan het adres vermeld is in het woord met adres n.

Opmerking. Voor eigenschappen van informatiedragers en hulpapparatuur en voor getalrepresentaties raadplege men de appendix bij dit hoofdstuk.

2.3. Werking van een rekenautomaat

In deze paragraaf zullen we zien hoe in principe door de wisselwerking tussen besturingsorgaan, rekenorgaan en een in een geheugen geplaatst programma dat programma verwerkt kan worden. Eenvoudigheidshalve nemen we weer aan dat ieder woord van het geheugen weer een getal of een instructie kan bevatten.

We nemen verder aan dat het rekenorgaan 4 "accumulatoren" bevat, die we aan zullen duiden met AC, AC1, AC2 en AC3. Hierin zijn AC1, AC2 en AC3 ieder weer bitgroepen met eenzelfde lengte als een geheugenwoord, terwijl AC tweemaal zo lang is als een geheugenwoord. AC, soms ook aangeduid met AC0, lijkt zodoende op het resultaattelwerk van een electrische tafelrekenmachine, waarin het "dubbele-lengte" product van twee "enkele-lengte" getallen opgebouwd

wordt. Op het nut van AC1, AC2 en AC3 komen we pas in latere paragrafen terug. Voorts bevat het rekenorgaan een aantal voorzieningen waarmee zowel arithmetische als logische bewerkingen kunnen worden uitgevoerd. De operanden (meestal 2) voor de bewerking komen uit het geheugen en/of de accumulator.

Het besturingsorgaan bevat tenminste twee "registers", wederom bitgroepen van meestal dezelfde lengte als een geheugenwoord, die als regel instructieteller (IT) en instructieregister (IR) genoemd worden. De instructieteller bevat namelijk het adres van de volgende uit te voeren instructie en moet daarom tenminste uit genoeg bits bestaan om alle geheugenplaatsen, waar instructies kunnen staan, te kunnen adresseren.

Evenzo moeten we ons bij het kernengeheugen twee registers denken die resp. adresregister (AR) en geheugenregister (GR) genoemd worden. De uitvoering van een instructie verloopt nu in grote trekken als volgt (neem aan dat een operand al in AC staat):

- de inhoud van het woord, waarvan het adres in IT staat, wordt overgebracht naar IR en daar als instructie geanalyseerd en zo nodig bewerkt;
- het resultaat van deze bewerking is nu, dat bekend is welke soort instructie uitgevoerd moet worden (bijvoorbeeld een optelling) en ook het adres van de operand die bij die instructie betrokken is; dit adres wordt in AR geplaatst;
- de inhoud van het woord waarvan het adres in AR staat wordt opgehaald en in GR geplaatst; het AR register wordt weer vrijgegeven;
- de feitelijke uitvoering van de instructie vindt plaats, gebruik makend van de operanden in AC en/of GR;
- bij de inhoud van IT wordt 1 opgeteld of, zoals bij later te bespreken "spronginstructies", een adres geplaatst (opdat straks de volgende instructie opgehaald kan worden).

Uiteraard is deze beschrijving zeer schematisch in die zin dat de volgorde van uitvoering van bovenstaande stappen anders kan zijn of dat bepaalde stappen elkaar overlappen, dat er meer registers kunnen zijn om operanden of tussenstappen vast te houden, enz. Hoe de uitvoering van een instructie precies in zijn werk gaat is trouwens niet belangrijk voor de programmeur, mits hij maar precies weet wat er gebeurt met de registers waar hij met zijn instructierepertoire toegang toe heeft (tot de inwendige registers als IR, IT, AR, GR, enz. heeft hij geen toegang). Ook de tijd nodig voor de uitvoering van een instructie kan voor hem van belang zijn; deze is voor instructies waarbij de accumulator een rol speelt veelal ongeveer tweemaal de tijd voor een

"geheugen access".

De bovenstaande schets van de uitvoering van een instructie geldt verder alleen voor een zgn. 1 adres instructietype, waarin naar slechts één operand (de andere staat al in bijv. de accumulator) wordt verwezen. Daarnaast kent men ook de volgende instructietypen:

3 adres instructies	(twee operanden en het resultaat),
3 + adres instructies	(als de vorige plus het adres van de volgende instructie),
2 adres instructies	(twee operanden waarvan één vervangen wordt door het resultaat),
2 + adres instructies	(als de vorige plus het adres van de volgende instructie),
1 + adres instructies	(één operand plus het adres van de volgende instructie),
0 adres instructies	(hierbij staan de operand(en) op voor het uitvoeringsmechanisme bekende plaatsen, zodat volstaan kan worden met het noemen van alleen de uit te voeren bewerking).

De n + adres en 3 adres instructies zijn tegenwoordig vrijwel verdwenen, omdat ze slechts zin hadden in de tijd van oude geheugentypen, (waarbij de uitvoering van instructies versneld kan worden door operanden en instructies, rekening houdend met de geheugeneigenschappen, op gunstige locaties te plaatsen). Het zal duidelijk zijn dat bij de 3, 3+, 2 en 2+ adres instructies geen accumulator nodig is en dat bij de n + adres instructies geen instructieteller nodig is.

Omdat het alleen maar om de principes gaat, zullen we ons in de komende paragrafen beperken tot de 1 adres instructies, terwijl later ook nog 0 adres instructies ter sprake zullen komen.

Een woord dat door het instructieregister geinterpreteerd moet worden, bestaat uit het algemeen uit:
een aantal bits, die de soort instructie vastleggen,
een aantal bits, die een geheugenadres bepalen,
groepjes bits, die naar andere accumulatoren verwijzen.
Hoe het samenspel van de laatste groepjes is, zal nog besproken worden. Of bepaalde bits tot het operatiegedeelte gerekend worden, dan wel samen bijvoorbeeld een accumulator benoemen, is natuurlijk tot op zekere hoogte willekeurig.

Opmerking. Terwijl bij oudere machines de operatiebits er via het besturingsorgaan voor zorgen dat door het openen of sluiten van "informatietransportwegen" bepaalde bewerkingen worden uitgevoerd, gebeurt dit bij sommige moderne machines indirect. De operatiebits worden nl. door een "vast ingebouwd" programma geinterpreteerd, zodat pas dit programma de nodige machine acties initieert. Men spreekt dan van microprogrammering (bij enkele machines kan zelfs dit microprogramma **desgewenst veranderd worden, wat uiteraard niet** bevorderlijk is voor uitwisseling van programma's!). Op microprogrammering zullen we verder niet ingaan, omdat deze nauw samenhangt met de electronische schakelingen in een rekenautomaat.

Besturingsorgaan en rekenorgaan zijn vaak samengebouwd in de zgn. centrale verwerkingseenheid, met daaraan ook de communicatiemiddelen tussen machine en operateur, bestaande uit bedieningspaneel en/of bedieningsschrijfmachine. Bij oudere, langzame machines kon het voor de programmeur van belang zijn om enkele details van deze communicatie-organen te kennen. Bij moderne, snelle machines is de discrepantie tussen machinesnelheden (microseconden) en menselijke handelingssnelheden (seconden tot minuten) zo groot dat men buiten het opstarten van de machine en bijzondere situaties niet meer van deze communicatiemiddelen gebruik moet maken bij de programmering van toepassingsprogramma's. Veel belangrijker is de communicatie tussen gebruiker en machine via de (harde) programmatuur.

Opmerking. De aanduiding 1^e, 2^e of 3^e generatie computer slaat als regel op de omstandigheid of de schakelelementen van de machine opgebouwd zijn met vacuumbuizen, transistoren of "geïntegreerde schakelingen". Machine kenmerken als snelheid, geheugengrootte en universele koppelingsmogelijkheden zijn uiteraard in moderne machines als regel ook verbeterd.

2.4. Ingreepbewerkingen (interrupts)

In het voorgaande is uiteengezet hoe het besturingsorgaan in principe de instructies van een programma een voor een kan uitvoeren (totdat een instructie ontmoet wordt die de machine doet stoppen). De uitvoeringstijden van instructies kunnen echter vele factoren verschillen:

- instructies, die informatietransport tussen kernengeheugen en rekenorgaan bewerkstelligen, vereisen ongeveer 1 tot 10 microseconden per woord;

- instructies, die informatietransport tussen kernengeheugen en secundaire geheugens bewerkstelligen vergen ongeveer 10 tot 100 microseconden per woord;
- instructies, die informatietransport tussen kernengeheugen en invoer/uitvoer apparatuur bewerkstelligen, vergen ongeveer 500 tot 3000 microseconden per woord.

Uit deze cijfers blijkt dat bij strikt sequentiële uitvoering van een programma, opgebouwd met deze instructies, de totale verwerkingstijd helemaal bepaald kan worden door de langzame invoer/uitvoer bewerkingen. Er wordt daarom naar gestreefd om langzame bewerkingen zoveel mogelijk gelijktijdig te laten verlopen met snelle.

Het is duidelijk dat de eerder geschetste uitvoering van instructies uitgebreid moet worden om een langzaam proces tegelijkertijd te laten verlopen met de verwerking van een groot aantal ieder op zich kort durende processen. Nodig zijn dan tenminste:

- dat na afloop van ieder kort durend proces (bv. de uitvoering van een "snelle" instructie) door het besturingsorgaan nagegaan wordt of vòòr het volgende korte proces misschien een langzaam proces (dat de kort durende niet mag hinderen) alvast gestart kan worden (waarbij de startbewerking op zichzelf kort duurt);
- een langzaam proces na afloop aan het besturingsorgaan kan melden dat het gereed is om zo nodig en mogelijk weer een langzaam proces te beginnen.

Een langzaam proces moet daartoe een ingreepsignaal (interrupt) afgeven dat het besturingsorgaan in staat stelt om te onderzoeken of de sequentiële verwerking van korte processen onderbroken moet worden ("trapping").

Een andere aanleiding voor onderbreking van de sequentiële verwerking hangt samen met het feit dat men een rekenautomaat zodanig kan construeren, dat bepaalde fouten tijdens de werking automatisch geconstateerd kunnen worden. (Na deze automatische constatering kan echter niet altijd ook automatisch een correctieve handeling worden uitgevoerd!) Sommige fouten kunnen aan een storing van de machine toegeschreven worden: een woord is niet goed in het geheugen geschreven (te constateren met een "pariteitscontrôle"), een hoeveelheid informatie is niet goed op een magneetband geschreven, enz. Andere fouten zijn mogelijk aan de programmaschrijver te wijten: hij programmeerde een deling door nul, hij probeerde informatie weg te schrijven in een deel van het geheugen dat door middel van zgn. "geheugenbescherming" niet

voor zijn programma toegankelijk was, hij probeerde informatie te lezen uit een verzameling gegevens waartoe hij niet gerechtigd was, enz.

Nog weer een andere aanleiding voor onderbreking van de normale verwerking komt voor wanneer een rekenautomaat verbonden is met een "extern proces" (bv. een "echt" physisch of chemisch proces, of een verkeersregelingssysteem of een maanlandersysteem, enz.). Dit externe proces kan signalen uitzenden als indicatie dat de rekenautomaat het aan de gang zijnde programma moet onderbreken en een ander programma moet starten, bv. voor maatregelen bij noodsituaties.

Een faciliteit die veelal aanwezig is in grote machines, nl. een soort wekker die "afgaat" als een door de programmaschrijver opgegeven verwerkingstijd verstreken is, kan eveneens aanleiding geven tot interrupt signalen.

Voor de behandeling van een interrupt moeten alle gegevens, die nodig zijn om daarna het onderbroken programma te hervatten, tijdelijk opgeborgen worden. Hiertoe horen o.a. de stand van het IT register (om straks de volgende instructie op te halen), de inhoud van registers als accumulator, enz. (deze gegevens bij elkaar noemt men vaak de status van het onderbroken programma).

Aangezien er zo veel interrupt oorzaken kunnen zijn, heeft men een aantal zgn. interruptregisters nodig, waarvan iedere bit correspondeert met een bepaalde interrupt oorzaak. Een nul geeft bijvoorbeeld aan dat er geen bepaald interrupt signaal is, een één dat er wel een is. Of het besturingsorgaan op de aanwezigheid van een bepaald interrupt signaal moet reageren wordt vaak beheerst door de inhoud van een ander register (dat met een programma veranderd kan worden), het "maskeringsregister". Moet er wel gereageerd worden dan vindt een statuswisseling plaats, het onderbrekende programma vult eventueel een maskeringsregister of maakt daarna het reageren op verdere interrupts al dan niet mogelijk en gaat dan maatregelen nemen, passend bij de oorzaak van de interrupt. Is dit gebeurd, dan vindt wederom een statuswisseling plaats, zodat het onderbroken programma op de goede plaats hervat wordt (deze vereenvoudigde beschrijving veronderstelt dat beide programma's, althans gedeeltelijk, in het kernengeheugen aanwezig zijn). Complicaties treden echter op wanneer het onderbrekende programma op zijn beurt weer onderbroken wordt of wanneer interrupts van verschillende prioriteiten kunnen optreden (een hogere prioriteit onderbreekt de behandeling van een lagere prioriteit interrupt, maar niet omgekeerd), wanneer de oorzaak van een interrupt-

signaal niet weggewerkt kan worden, enz. Op deze problemen wordt in dit college echter niet ingegaan.

Interrupt afhandelingsprogramma's zijn in de regel vrij ingewikkeld, maar hoeven voor een bepaald machinetype slechts éénmaal geschreven te worden. Ze horen tot de harde programmatuur en zijn als regel een deel van het zgn. bedrijfsvoeringprogramma ("monitor", "operating system", "supervisor").

2.5. Programmering in een machinetaal

In het voorgaande is reeds vermeld dat een instructiewoord opgebouwd is uit groepjes bits, waarvan een groepje bepaalt welke soort instructie uitgevoerd moet worden, een ander groepje een geheugenadres bepaalt en andere groepjes afhankelijk van het machinetype diverse additionele functies hebben. Het is evident dat een binair uitschrijven van instructies uitermate vermoeiend is en tot schrijffouten aanleiding geeft. Al gauw ging men er dan ook toe over om bitgroepen van een instructiewoord in groepjes van 3 of 4 bits van links naar rechts lezend te noteren via de symbolen van een 8-tallig (symbolen 0 t/m 7) of 16-tallig (symbolen 0 t/m 9, A t/m F) getalstelsel. De lengte van de bitgroepen laat men dan corresponderen met het aantal bits in het operatiegedeelte, adresgedeelte, enz. in een woord.

Al is de omzetting van 8- of 16-tallige naar binaire patronen eenvoudig, toch is het gebruik van zo'n numerieke codering van instructies niet aantrekkelijk. Men ging dan ook al gauw in plaats van een numerieke aanduiding van een operatiecode een alfabetische notatie gebruiken. Voor zichzelf sprekende voorbeelden zijn: ADD, SUB(tract), MUL(tiply), DIV(ide), B(ranch), enz. welke min of meer altijd in dezelfde zin gebruikt worden. Ten aanzien van geheugenadressen en extra accumulatoren bleef men aanvankelijk een decimale notering aanhouden, zodat een instructie die een getal in woord 1067 bij het getal in de derde accumulator optelt er uit kan zien als ADD 3, 1067. We hebben in dit voorbeeld als separator tussen 3 en 1067 de komma gebruikt, maar vaak gebruikt men helemaal geen separatoren, maar schrijft men voor dat het eerste (of laatste) karakter van operatie-, accumulator- en geheugenadres aanduiding in een bepaalde kaartkolom geponst moeten worden. Bij gebruik van ponsband zijn separatoren uiteraard nodig (als men niet een uiterst gestandaardiseerd ponspatroon voorschrijft). In onze geschreven tekst zullen we spaties gebruiken.

De vervanging van lettercombinaties en decimale getallen door bitpatronen gebeurt eenvoudig door opzoeken in een tabel, resp. omrekening van decimale naar binaire integers.

Om de principes van machinetaalprogrammering uiteen te zetten, beschouwen we een hypothetische 1 - adres machine, waarvoor een instructie geschreven wordt als:

In m a

Hierin stelt a (≥ 0) als regel een geheugenadres voor, m en n zijn of een spatie of een van de getallen 0 t/m 3, terwijl voor I één van de letters uit de volgende tabel gekozen moet worden. In de kolom met het hoofd effect betekent ACn de inhoud van accumulator n, m.a.w. de accoladen om {ACn} worden weggelaten, omdat er toch geen verwarring mogelijk is. Voorts is in de tabel m een spatie.

afkorting		effect	opmerking
Ln	a	ACn := {a}	èn IT := IT+1
An	a	ACn := ACn + {a}	
Sn	a	ACn := ACn - {a}	
Mn	a	ACn := ACn * {a}	
Dn	a	ACn := ACn ÷ {a}	
Cn	a	{a} := ACn	
Jn	a	IT := a	niet {a}!
Pn	a	IT := if ACn ≥ 0 then a else IT + 1	niet {a}!
Gn	a	IT := IT + 1	dummy!
Hn	a	IT := IT	stop!
En	a	ACn := ACn ∧ {a} (bit voor bit)	èn IT := IT+1
On	a	ACn := ACn ∨ {a} (bit voor bit)	
Rn	a	ACn := symbool van ponsband	
Wn	a	symbool op ponsband := ACn	
Xn	a	schuif bitpatroon in Acn over a plaatsen naar rechts	
Yn	a	schuif bitpatroon in ACn over a plaatsen naar links	

(Het nut van de derde groep instructies zal pas later besproken worden. De eerste 14 vermelde letters zijn afkortingen van: laden, additie, subtractie, multiplicatie, deling, copie, jump, positive, gewoondoorgaan, halt, en, of, read, write).

Als eerste voorbeeld zullen we een machinetaalprogramma behandelen dat het getal r berekent met r := |p| - |q| en dan plaatst in geheugenwoord 203 als p en q opgeborgen zijn in geheugenwoorden 201 en 202. Het programma begint op geheugenwoord 501, terwijl we eerst aannemen dat er maar 1 accumulator is (n = spatie).

woord	{woord}		effect
501	L	202	
502	P	505	
503	S	202	{203} := |q|
504	S	202	
505	C	203	
506	L	201	
507	P	510	AC := |p|
508	S	201	
509	S	201	
510	S	203	
511	C	203	{203} := |p| - |q|
512	H		

Opmerkingen.

1. We zijn "heel slim" eerst |q| gaan bepalen omdat we anders zowel |p| als |q| "tussentijds in het geheugen hadden moeten opslaan" om |p| - |q| te kunnen berekenen.
   Het programmadeel in woorden 501 t/m 505 is het equivalent van

   if q ≥ 0 then {203} := q else {203} := q - q - q

   Voor het equivalent van het alternatieve programma

   if q ≥ 0 then {203} := q else {203} := 0 - q

   moeten we over de constante 0 beschikken. Omdat het berekenen van een absolute waarde zo vaak gebeurt, vinden we in het instructierepertoire van grote machines veelal machinetaalinstructies met het effect ACn := - {a} of ACn := - ACn, waarmee het bovenstaande programma iets korter zou worden. Tegenover het schrijven van kortere programma's staat natuurlijk het onthouden van een uitgebreidere instructieverzameling!
2. Voor de tussenopslag van |q| hebben we geheugenwoord 203 gebruikt, waar later het resultaat terechtkomt. Willen we |q| nog voor andere berekeningen gebruiken, dan hadden we tussentijds |q| op een andere "vrije" plaats moeten opbergen.
3. Maken we gebruik van meer accumulatoren, dan is een oplossing

woord	{woord}		effect
501	L	201	
502	P	505	
503	S	201	AC := |p|
504	S	201	
505	L1	202	
506	P1	509	
507	S1	202	AC1 := |q|
508	S1	202	
509	C1	203	
510	S	203	{203} := |p| - |q|
511	C	203	
512	H		

Dit programma is wel niet korter, maar de aanwezigheid van meer accumulatoren stelt ons hier in staat om de formule van links naar rechts uit te werken.

4. Bij H hebben we geen geheugenadres genoemd omdat dit toch niet relevant is (ook niet bij G, R en W!).
5. Als voor p, q en r andere geheugenplaatsen gebruikt worden of als het programma op een andere plaats moet beginnen, is een nieuw programma door wat eenvoudige veranderingen direct op te schrijven. Toch zal men daarbij veel vergissingen begaan! Men moet uiteraard ook oppassen dat "getalgebieden" niet in "instructiegebieden" terecht komen omdat men dan steeds "om de getalgebieden heen moet springen".
6. We constateren overal de vuistregels: "informatie op een bepaalde plaats blijft behouden tenzij hij overschreven wordt" en "als het niet uitdrukkelijk gezegd is, gebeurt het niet".

Als tweede voorbeeld bekijken we een programma waarmee $r = (a - b * c)/(d - e * f)$ wordt berekend als a, b, c, d, e, f zich bevinden in woorden 201, 202, 203, 204, 205, 206 terwijl r in 207 moet komen en de eerste instructie in woord 501 geplaatst wordt. Voorts zijn woorden 401-499 vrij beschikbaar, terwijl voor n weer alleen een spatie gebruikt wordt. In de linkeroplossing zijn we er van uitgegaan dat de formule van links naar rechts moest worden uitgewerkt, terwijl de rechteroplossing zo "slim en kort mogelijk is".

woord	{woord}		effect
501	L	201	
502	C	401	{401} := a
503	L	202	
504	M	203	{402} := b * c
505	C	402	
506	L	401	
507	S	402	{401} := a - b * c
508	C	401	
509	L	204	
510	C	402	{402} := d
511	L	205	
512	M	206	{403} := e * f
513	C	403	
514	L	402	
515	S	403	{402} := d - e * f
516	C	402	
517	L	401	
518	D	402	{207} := {401}/{402}
519	C	207	
520	H		

woord	{woord}		
501	L	205	
502	M	206	
503	S	204	{207} := c * f - d
504	C	207	
505	L	202	
506	M	203	AC := b * c - a
507	S	201	
508	D	207	
509	C	207	{207} := r
510	H		

Opmerkingen.

1. Bij beide oplossingen hebben we, dank zij het feit dat we een berekening indien mogelijk direct uitvoerden, een "pulserende" reeks hulpwoorden gebruikt. Zonder deze strategie zouden we een langere reeks hulpwoorden nodig gehad hebben.
2. De rechteroplossing is korter geworden door ons "vooruitzien" dat we, door de volgorde van deelbewerkingen te veranderen, een aantal "overbodige" LC bewerkingen kunnen voorkomen.
3. Op de betekenis van de linkeroplossing komen we nog later terug bij 0 - adres machines.
4. Verifieer zelf dat we door van meer accumulatoren gebruik te maken eveneens een korte oplossing kunnen realiseren.
5. Herschrijf het programma als a t/m f zich in andere geheugenwoorden bevinden.

2.6. Programmeren in een assembleertaal (symbolische machinetaal)

Om het maken van vergissingen bij het invullen van adressen te vermijden, zullen we naast numerieke adressen ook "symbolische adressen" invoeren en naast de eerder genoemde instructies ook "pseudo-instructies en labels". Het omzetten van symbolische adressen in binaire adressen, kan zoals we pas in een ander college zullen zien, aan een speciaal programma, een "assembleerprogramma" overgelaten worden. Dit assembleerprogramma verwerkt ook de pseudo-instructies en labels als aanwijzingen voor zijn taak.

Voor het eerder behandelde probleem r := |p| - |q| geven we het equivalent in symbolische vorm

	programma		commentaar
	OR	501	} pseudo-instructies
B:	DL	201	
	L	B + 1	} {B + 2} := |q|
	P	S	
	S	B + 1	
	S	B + 1	
S:	C	B + 2	
	L	B	} AC := |p|
	P	* + 3	
	S	B	
	S	B	
	S	B + 2	} {B + 2} := |p| - |q|
	C	B + 2	

Opmerkingen.

1. De pseudo-instructies worden niet vertaald, maar melden aan het assembleerprogramma dat resp. de eerste instructie ondergebracht moet worden in woord 501 en dat symbolisch adres B moet corresponderen met woord 201. (Origine en Definieer Locatie zijn de aanleiding tot deze afkortingen.) Wil men het programma op een andere plaats laten beginnen dan hoeft slechts de eerste instructie veranderd te worden. Wil men andere adressen dan 201, 202, 203 gebruiken dan hoeft slechts de tweede instructie veranderd te worden. Alle adressen zijn dus relatief t.o.v. een aantal punten.

(We willen hierbij opmerken dat de pseudo-instructie DL in assembleertalen voor echte machines niet voorkomt, omdat met de later te bespreken pseudo-instructie DA vrijwel hetzelfde te bereiken is, zonder het nadeel dat de programmeur een boekhouding van gebruikte geheugenlocaties moet bijhouden. Om de laatste reden mag de programmeur veelal achter OR een adres weglaten, omdat het assembleerprogramma dan wel een geschikt adres toewijst.)

2. In het programma komt verder de label S voor (uiteraard slechts één keer geoorloofd) waarheen verwezen wordt door de instructie P S. Voorts zien we de constructie * + 3 die we zo moeten interpreteren dat * betrekking heeft op het adres van het woord dat de in machinetaal getransformeerde instructie P * + 3 zal bevatten; * + 3 wijst dus drie plaatsen verder zoals hier nodig is. Omdat * dus zo iets als "hier" betekent, mag deze bijzondere "label" een willekeurig aantal keren herhaald worden.
3. In het adresgedeelte van een instructie mag, zoals aangegeven, veelal wat eenvoudige arithmetiek bedreven worden ("symbolische adres-arithmetiek").
4. Omdat bij spronginstructies, die gebruik maken van het (* + i) systeem, vaak i pas achteraf uitgerekend of herberekend (als er instructies in een lus toegevoegd of verwijderd worden) kan worden, maakt men vaak vergissingen. Als i dan ook groter wordt dan bijvoorbeeld 5, is het gebruik van dit systeem niet aan te raden.
   Een ander systeem gebruikt wederom een speciale label, bijvoorbeeld H(ier), die geplaatst wordt bij de instructie waarheen gesprongen moet worden. In de spronginstructie moet dan het adres AH of VH gebruikt worden, waarbij AH slaat op een achterwaartse sprong naar de dichtstbijzijnde met H gelabelde instructie en VH evenzo op een voorwaartse sprong. Door deze afspraak over de dichtstbijzijnde H kan deze constructie op meer plaatsen in een programma gebruikt worden. Het assembleerprogramma staat weer voor de taak om de symbolische adressen door de goede binaire adressen te vervangen.

Als volgende voorbeeld zullen we een programma beschouwen dat f(x) plaatst op woord F als a, b, c en x gegeven zijn in woorden A, B, C en X en als

$$f(x) := \begin{cases} 0 & \text{voor } x \leq a \text{ en } x \geq c \\ a & \text{voor } a < x < b \\ b & \text{voor } x = b \\ c & \text{voor } b < x < c \end{cases}$$

```
        OR      ....
Z:      DC      0
A:      DC      a
B:      DC      b       } pseudo-instructies
C:      DC      c
X:      DC      x
F:      DL      ....
        L       A
        S       X
        P       CKGX    } uitzoeken van de gevallen a ≥ x en x ≥ c
        L       X
        S       C
        P       CKGX
        L       X
        S       B       } uitzoeken van het geval x ≥ b
        P       BKGX
        L       A
        J       END     } het geval a < x < b
BKGX:   L       B
        S       X       } uitzoeken van het geval x = b
        P       BGX
        L       C
        J       END     } het geval b < x < c
BGX:    L       B
        J       END     } het geval x = b
CKGX:   L       Z       de gevallen a ≥ x en x ≥ c
END:    C       F       opbergen van het resultaat
        H       ....
```

Opmerkingen.

1. Voor de symbolische adressen van a, b, c, f en x hebben we ter wille van het lezen de corresponderende hoofdletters gebruikt. Evenzo zijn suggestieve labels BKGX (b kleiner gelijk x), BGX (b groter x), END ingevoerd. Al deze hulpmiddelen nemen niet weg dat het "teruglezen" en interpreteren van een dergelijk assembleertaalprogramma niet eenvoudig is.

2. Met de pseudo-instructie A : DC a wordt de waarde a geplaatst in een door het assembleerprogramma te bepalen woord met het symbolische adres A.
3. Let op de J END instructies, waarmee de verschillende takken van het programma samengeknoopt worden bij de instructie met label END. Ga na wat er gebeurt bij het vergeten van een J END!
4. Omdat de plaatsing van het programma en van f ons hier weinig kunnen schelen, hebben we het adresgedeelte niet ingevuld in de eerste en zevende pseudo-instructie.

Kenmerken van een assembleertaal en zijn gebruik zijn:

- behalve voor de pseudo-instructies is er een 1,1 duidige relatie tussen assembleertaal en binaire machinetaal (later zullen we nog zgn. macro-instructies tegenkomen, die corresponderen met meer dan één machinetaal instructie);
- de administratie van het gebruik van het geheugen blijft, ondanks de vereenvoudiging t.o.v. de binaire machinetaal, deels een taak van de programmeur;
- assembleertalen zijn specifiek voor een bepaald machinetype als gevolg van specifieke eigenschappen van die machine; evenzo zijn de assembleerprogramma's specifiek;
- sommige programmeringsfouten kunnen tijdens vertaling ontdekt worden, zoals fout gebruik van labels en niet-gedeclareerde adressen.

De betekenis van symbolische adressering is dieper dan de hier gebruikte invoering als hulpmiddel bij assembleertalen suggereert. Het vervangen van een symbolisch adres door een geheugenadres wordt van de programmeur weggenomen omdat het assembleerprogramma dit later voor hem doet (in andere colleges wordt behandeld hoe. Deze vervanging kan ook uitgesteld worden tot het moment dat het vertaalde programma voor verwerking in het geheugen komt en zelfs tot het moment dat het verwerkt zal gaan worden). De programmeur kan zodoende zijn aandacht geheel geven aan de algorithmische opbouw van zijn programma.

2.7. Programmalussen

In de vorige paragraaf hebben we kennis gemaakt met de equivalenten van de Algol assignments en conditional statements. We zullen nu het equivalent van de for ..... do constructie onderzoeken en beschouwen daartoe de berekening van n * a door herhaald optellen. Laat n (≥ 0) gegeven zijn in woord G, het getal a in woord G+1, het getal 1 in het woord ONE, het getal 0 in het woord ZERO, terwijl woord W beschikbaar is als "kladpapier". Het resultaat moet in G + 2 komen. Een mogelijk programma is dan:

```
        OR      ...
G:      DC      n
        DC      a
        DA      1                                          } pseudo-instructies
W:      DA      1
ZERO:   DC      0
ONE:    DC      1
        L       ZERO    }                                  }
        C       G + 2   } {G + 2} := 0                     }
        L       G       }                                  } initialisatie
        C       W       } {W} := n                         }
LON:    L       W       }                                  }
        S       ONE     }                                  }
        C       W       } {W} := {W} - 1                   } lusonderzoek
        P       * + 2   }                                  }
        J       END
        L       G + 2   }                                  }
        A       G + 1   } {G + 2} := {G + 2} + a           } bewerking
        C       G + 2   }                                  }
        J       LON
END     ..      ...
```

Opmerkingen.

1. Bij de pseudo-instructies komen we een nieuw type tegen, nl. DA, die het erachter genoemde aantal woorden reserveert. Voorts hoeven we de labels G + 1 en G + 2 niet te noemen na de G : DCn instructie. Het assembleerprogramma zoekt zelf wel 3 opeenvolgende plaatsen uit voor het begin van het programma (waar de constante n terechtkomt), de constante a en de vrije plaats waar de uitkomst wordt opgebouwd.

2. De strategie initialisatie → lusonderzoek → bewerking ↩ is de veiligste omdat bij verwisseling van deze programmastukken soms misschien wel instructies bespaard kunnen worden, maar de lus misschien een verkeerd aantal keren doorlopen wordt (let vooral op gevallen als n = 0!).
3. Kladpapier W is nodig omdat bij gebruik van G i.p.v. W het getal n aan het eind van het programma verloren zou zijn.
4. Bewijs dat als het programma werkt voor zekere $n \geq 0$, dat het dan ook werkt voor n + 1.
5. Schrijf dit programma in Algol en vergelijk Algol- en het assembleertaalprogramma (het is vaak handig om bij assembleertaalopgaven eerst een Algolprogramma op te stellen).

We constateren in dit voorbeeld een groot aantal L en C instructies; na enig nadenken blijkt dit het gevolg te zijn van de aanwezigheid van slechts één accumulator die gebruikt moet worden:

- als tussenstation bij transport van informatie (zie vullen van G + 2 en W!)
- bij de beslissing in het lusonderzoek
- bij de eigenlijke bewerking.

Nog sterker komt dit tot uiting in de volgende opgave: bereken $y := \sum_{1}^{n} a_i$ als n (≥ 0) gegeven is in woord G, y moet komen in woord G + 1, terwijl a_1 t/m a_n op verder niet te bespreken wijze staan in woord A t/m A + n - 1 (men kan zich indenken dat dit voor elkaar komt met behulp van een hier niet opgeschreven stuk programma dat voorafgaat aan de volgende initialisatie).

```
        OR      ...
G:      DC      n                                          } pseudo-instructie
        DA      1                                          }
A:      DA      n                                          }
W:      DA      1                                          }
ZERO:   DC      0                                          }
ONE:    DC      1                                          }
        L       ZERO        }                              }
        C       G + 1       } {G + 1} := 0                 }
        L       SI          }                              } initialisatie
        C       SI + 1      } startinstructie plaatsen     }
        L       G           }                              }
        C       W           } {W} := n                     }
LON:    L       W                                          }
        S       ONE                                        }
        C       W                                          } lusonderzoek
        P       * + 3                                      }
        J       END                                        }
SI:     L       A                                          "constante"
        .       ...         }                              }
        A       G + 1       } {G + 1} := {G + 1} + a_i     } bewerking
        C       G + 1       }                              }
        L       SI + 1      }                              }
        A       ONE         } {SI + 1} := {SI + 1} + 1     } instructiemodificatie
        C       SI + 1      }                              }
        J       LON
END:    .       ...
```

Opmerkingen.

1. Verifieer de juistheid van het programma voor n = 0 en willekeurige n > 0.
2. Bewerking en instructiemodificatie kunnen vaak verwisseld worden. Het initialiseren van de instructie op SI + 1 wordt wel eens vervangen door een herstellen van de oorspronkelijke instructie op SI + 1 nadat de lus doorlopen is. Voor de denkdiscipline is het initialiseren echter raadzamer.

3. Merk op dat de instructie op SI nooit "bereikt" wordt; deze instructie fungeert als "constante" wat niets bijzonders is omdat een instructie ook een binair bitpatroon is. Het plaatsen van zo'n constante vlak bij het programmastuk waar hij nodig is en direct na (!) een J instructie is gebruikelijk (maar niet noodzakelijk).
4. Op SI + 1 kan iedere willekeurige instructie geschreven worden, maar het neerschrijven van een H instructie is gebruikelijk (het programma zou dan bij wijze van waarschuwing stoppen als SI + 1 niet ingevuld werd).

In deze opgave zien we het von Neumann concept in werking: het programma wordt tijdens de uitvoering veranderd. (Het is geen pure procedure.) Voor een herhaalde toepassing van het programma is dat geen bezwaar, omdat het goed geïnitialiseerd wordt. Toch zijn er bezwaren tegen dit soort programma's:

- als de uitvoering van dit programma onderbroken wordt (bijvoorbeeld voor de behandeling van een interrupt) èn het geheugendeel waar dit programma zich bevindt voor iets anders nodig is, moet het veranderde programma weggeschreven worden om later een correcte voortzetting te garanderen; immers het oorspronkelijke programma kan niet gebruikt worden;
- wordt van dit programmadeel door twee (of meer) andere programma's "gelijktijdig" gebruik gemaakt, dan gaat het mis omdat de instructie op SI + 1 in het algemeen niet goed zal zijn voor beide programma's tegelijk, evenmin trouwens {W} en {G + 1};
- om dezelfde reden is deze programmeringstechniek niet geschikt om recursie te implementeren.

2.8. Indexregisters

Maken we nu gebruik van het aanwezig zijn van meer accumulatoren dan kunnen we aan de laatst genoemde bezwaren gedeeltelijk tegemoet komen, zoals het volgende programma aantoont:

```
        OR      .....
G:      DC      n
        DA      1
A:      DA      n
ZERO:   DC      0
ONE:    DC      1
        L       ZERO        A := 0
        L1      G           A1 := n
        L2      SI          A2 := A   A
LON:    S1      ONE         A1 := A1 - 1
        P1      * + 3
        J       END
SI:     A       A
        C2      SI + 2
        ..      .....
        A2      ONE
        J       LON
END:    C       G + 1
```

pseudo-instructies

initialiseren van 3 accumulatoren

lusonderzoek

"constante"

instructie modificatie en bewerking

In dit voorbeeld hebben we namelijk voldoende accumulatoren voor de taken van lusonderzoek (A1), sommatie (A) en instructie modificatie (A2). In ingewikkelder gevallen zullen we toch accumulatoren tekort kunnen komen, zodat we dan weer geheugenwoorden moeten gebruiken om accumulatoren te simuleren (met de consequentie van vele L en C instructies). Bovendien blijkt bij een nauwkeurige beschouwing dat aan bovengenoemde bezwaren nog niet altijd tegemoet wordt gekomen. Namelijk niet als het programma na afwerking van de instructie op SI + 1 onderbroken wordt door een ander programma dat van deze zelfde programmatekst gebruik maakt (zoals altijd nemen we overigens aan dat de inhoud van accumulatoren niet verloren gaat bij onderbreking van een programma, anders gaat het helemaal mis!).

Om dit soort problemen op te lossen, was de volgende stap in de ontwikkeling van rekenautomaten de invoering van het zgn. indexeren. Het besturingsorgaan werd zo uitgebreid dat het een instructie van de volgende gedaante
- operatiedeel (incl. vermelding van betrokken accumulator), gevolgd door
- adres van een andere accumulator, gevolgd door
- geheugenadres

zal uitvoeren als ware het een instructie met

- hetzelfde operatiedeel (dus betrekking hebbende op dezelfde accumulator)
- een nieuw geheugenadres, dat de som is van het oude geheugenadres en de inhoud van de "andere" accumulator.

Ter wille van de eenvoud hebben we aangenomen dat het optellen - in het besturingsorgaan - van oude geheugenadres en inhoud van de andere accumulator alleen plaats vindt voor het geheugenadresgedeelte van de instructie, dat we geen problemen krijgen in verband met de eindige grootte van een geheugen en dat deze optelling niet plaats vindt als de andere accumulator toevallig accumulator nul is (ga zelf na wat er dan allemaal mis zou kunnen gaan).

Voorbeeld: accumulator 0 bevat de integer 20, accumulator 1 de integer 15. Dan wordt de instructie A 0 100 uitgevoerd als A - 100 maar daarentegen instructie A 1 100 uitgevoerd als A - 115.

Deze werking van het besturingsorgaan draagt de naam indexeren en in plaats van accumulatoren 1, 2 en 3 spreekt men ook wel van indexregisters (B-lines, modifiers) 1, 2 en 3.

Gebruik makend van indexering wordt het laatste programma nu:

```
        OR      .....                                   }
G:      DC      n                                       }
        DA      1                                       }
A:      DA      n                                       }  pseudo-instructies
ZERO:   DC      0                                       }
ONE:    DC      1                                       }
        L   0   ZERO        A := 0                      }
        L1  0   G           A1 := n                     }  initialiseren
LON:    S1  0   ONE         A1 := A1 - 1                }
        P1  0   * + 2                                   }  lusonderzoek
        J   0   END                                     }
        A   1   A                                       bewerking
        J   0   LON
END:    C   0   G + 1
```

Opmerkingen.

1. Niet alleen is het programma weer korter geworden, maar wat veel belangrijker is, de bezwaren genoemd aan het eind van de vorige paragraaf blijken nu niet meer te gelden (tenminste zolang er genoeg indexregisters zijn!).

2. Merk op dat de getallen a_i nu in "omgekeerde" volgorde bij elkaar opgeteld worden. De gewone volgorde is echter te realiseren door accumulator 2 als teller te gebruiken en accumulator 1, beginnend met de inhoud 0, als modifier. Schrijf dit zelf uit.

Het programma voor de berekening van n * a wordt eenvoudig:

```
        OR      ...
G:      DC      n
        DC      a
        DA      1
ZERO:   DC      0
ONE:    DC      1
        L   0   ZERO
        L1  0   G
        S1  0   ONE
        P1  0   * + 2
        J   0   END
        A   0   G + 1
        J   0   * - 4
END:    C   0   G + 2
```

pseudo-instructies

initialiseren

lusonderzoek

bewerking

De switch faciliteit uit Algol is direct met een modificeerbare sprong instructie: J n 0 (n = 1,2 of 3) te realiseren. De inhoud van het genoemde indexregister bepaalt immers waarheen gesprongen wordt.

Ondanks de meerkosten voor deze ingewikkelder besturingsorganen werden vrijwel alle machines na ongeveer 1955 met indexregisters uitgerust. Aanvankelijk waren dit er 3 (zoals in ons voorbeeld) of 7, maar later ook 15 of 31 (en zelfs 99). Ter bezuiniging werden ook wel eens de eerste geheugenplaatsen als indexregisters gebruikt (ondanks het nadeel dat een extra geheugentoegang nodig is bij de uitvoering van iedere instructie). De overweging dat zich situaties kunnen voordoen waarin zelfs een flink aantal indexregisters wel eens onvoldoende kan zijn, doet de belangstelling echter verschuiven naar de oplossing die bij 0-adres machines gevonden is (zie later).

2.9. Directe, indirecte en onmiddellijke adressering

De tot dusver behandelde methodiek, waarbij in de instructie het adres van een operand genoemd wordt, heet directe adressering. In plaats van indexregisters gebruikt men in sommige machines echter een techniek, die indirecte adressering genoemd wordt (of "second level adressing"). Bij deze indirecte adressering staat in de instructie het adres van de geheugenplaats waar het adres van de operand is opgeborgen (vergelijk dit met de bekende zin uit advertenties: "brieven onder letter X aan de administratie van dit blad"). Het bezwaar van deze techniek is dat een extra geheugentoegang nodig is om een operand op te halen, maar net als met indexregisters is het dan niet nodig om een instructie te veranderen. Het equivalent van indexeren bewerkstelligt men immers nu door de inhoud van de in de instructie genoemde geheugenplaats te manipuleren. Nog weer een ander gebruik van indirecte adressering treffen we aan bij machines met een zeer groot aantal woorden in het kernengeheugen, maar met een kleine woordlengte voor instructies. In een instructiewoord staan dan niet voldoende bits ter beschikking om het gehele kernengeheugen direct te adresseren, maar in het in de instructie genoemde woord staan alle bits ter beschikking. (Deze gedachte van indirecte adressering vinden we overigens in andere uitvoeringsvormen ook terug bij de alsnog te bespreken 0-adres machines, bij machines waarbij in de instructie 2 indexregisters genoemd worden en bij machines van het PDP-8 type.)

Op de techniek van indirecte adressering zijn nog vele varianten mogelijk:

- op het in de instructie genoemde adres staat niet het adres van de operand, maar het adres van de geheugenplaats met het adres van de operand, enz. (dus "third, fourth, ... level adressing"). Met een bit in de zo benoemde woorden moet dan de voortzetting of het einde van de indirecte adressering aangeduid worden;
- indirecte adressering wordt gebruikt in combinatie met indexregisters, waarbij onderscheiden moet worden tussen indexeren voor of na de indirecte adressering.

We vermelden tenslotte nog dat deze indirecte adressering vaak gebruikt wordt bij eenvoudige procesbesturingsmachines.

In tegenstelling tot de indirecte adressering staat de zgn. onmiddellijke ("immediate" of "zero-level") adressering, waarbij in een instructie niet het adres van een operand genoemd wordt, maar onmiddellijk de operand zelf. Gezien het beperkte aantal bits dat voor het adresdeel van een instructie ter beschikking staat, kunnen slechts betrekkelijk kleine integers op die manier als operand optreden. Waar echter vaak integers als 1 of 2 nodig zijn in een programma, is onmiddellijke adressering echter een plezierige techniek, met het voordeel van het besparen van een geheugentoegang om een operand op te halen.

Voorbeeld. Duiden we indirecte, resp. onmiddellijke adressering in symbolische assembleertaal aan door achter de corresponderende directe adresseringsinstructies een *, resp. + te plaatsen, dan betekent:

A 50 dat de inhoud van woord 50 opgeteld wordt bij de accumulator,
A *50 terwijl in woord 50 bijv. 686 staat dat de inhoud van woord 686 opgeteld wordt bij de accumulator,
A +50 dat de integer 50 opgeteld wordt bij de accumulator.

2.10. Subroutines

Voor de meeste programmeringselementen van Algol hebben we nu de equivalenten in een assembleertaal leren kennen. Rest nu nog om te onderzoeken hoe procedures in een lagere taal (of subroutines zoals ze dan meestal heten) geconstrueerd worden.

Allereerst moeten de volgende overwegingen vermeld worden

a) subroutines (procedures) vormen als het ware een uitbreiding van de normale vocabulaire van een taal, daar ze afgesloten eenheden zijn (opgebouwd met de normale vocabulaire), waarvan de gebruiker de constructie en werking niet hoeft te kennen maar wel het bestaan (de naam) en het juiste gebruik (efficiënte subroutines kunnen door specialisten eens en voor altijd geconstrueerd worden en besparen dan veel programmerings- en testtijd bij het ontwikkelen van grote programma's);
b) er moet een mechanisme zijn, waarmee de werking van een subroutine bewerkstelligd wordt (in Algol gebeurt dit door de naam van de procedure te noemen) op de gewenste plaats in een "aanroepend" hoofdprogramma, gevolgd door continuering van het aanroepend programma;

c) er moet een mechanisme zijn om parameters uit het hoofdprogramma over te dragen naar de subroutines;
d) het is aan te bevelen dat een subroutine eventueel "tegelijkertijd" door meerdere hoofdprogramma's te gebruiken is, waarbij op onvoorspelbare tijdstippen door ingreepbewerkingen één van de hoofdprogramma's enige tijd beslag legt op het besturingsorgaan;
e) het is aan te bevelen dat subroutines zichzelf kunnen aanroepen en dat naast de call by value ook een call by name faciliteit aanwezig is.

In de volgende pagina's zullen we onder ogen zien hoe aan de minimumeisen a t/m c en aan de additionele verlangens d en e tegemoet kan worden gekomen. Ten aanzien van de realisering kunnen we ons twee oplossingen voorstellen, nl. ten eerste dat de subroutine gekopieerd wordt op de gewenste plaatsen in het aanroepend programma (de zgn. "open" subroutine) en ten tweede dat de subroutine slechts op één plaats in het machinegeheugen staat waarheen dan sprongen worden uitgevoerd vanuit de gewenste plaatsen in het hoofdprogramma. Het voordeel van de laatste oplossing (de "gesloten" subroutine) is uiteraard ruimtebesparing; een klein bezwaar is het tijdverlies voor de organisatie om de aanroep. We zullen eerst gesloten subroutines bespreken.

2.11. Gesloten subroutines

2.11.1. Sprong en terugkeer

Het aanroepen van een subroutine is geen probleem: men kan bijv. de naam van de subroutine gebruiken als label van de eerste (echte) instructie van de subroutine en deze dus in werking stellen door een spronginstructie in het hoofdprogramma met deze label als operand. Ook de terugsprong is geen probleem wanneer aan de subroutine meegedeeld is op welk punt het hoofdprogramma verlaten werd. Het vastleggen van het terugkeeradres kan op verschillende manieren gebeuren:
- voor machines zonder indexregisters met bijv. de volgende instructies in hoofdprogramma, resp. subroutine.

```
L      *                          SUB:  A      ....
J      SUB  →                           C      EIND
---------                               ----------
---------                               ----------
---------  ←                            ----------
                                  EIND: ----------
```

Verklaring: de instructie L * plaatst zichzelf in de accumulator, zodat in het adresgedeelte van de instructie in de accumulator de actuele plaats ("tag") van deze instructie is vastgelegd. Door de eerste instructie van de subroutine moet hierbij een dusdanige bitcombinatie bijgeteld worden dat een sprong-instructie ontstaat die, op het eind van de subroutine geplaatst, straks de terugsprong veroorzaakt naar "zoveel plaatsen voorbij" de L * instructie (hoe groot "zoveel" is, is tamelijk willekeurig maar moet gestandaardiseerd zijn voor alle bibliotheek subroutines van een rekencentrum om het gebruik van zo'n bibliotheek voor de gebruiker te vereenvoudigen).
Bij eerste kennismaking lijkt dit misschien een zeer gekunstelde methode en zou men denken dat het hoofdprogramma wel de terugkeersprong in de subroutine zou kunnen plaatsen. Dit eist echter dat de schrijver van het hoofdprogramma precies weet waar hij deze informatie in de subroutine moet plaatsen. Verandert de subroutine dan moet ook het hoofdprogramma veranderd worden!

- voor machines met indexregisters zal men liever een vast indexregister aanwijzen voor hetzelfde doel als hierboven geschetst, met het voordeel dat de accumulator dan beschikbaar komt voor overdracht van andere informatie tussen hoofdprogramma en subroutine. Dus:

```
L1    0     *                    SUB: ---------
J     0     SUB →                     ---------
----------------- ←                   J   1   2
```

De laatste instructie van de geschetste subroutine bewerkstelligt door indexering een sprong naar het woord volgende op J 0 SUB in het hoofdprogramma. Door de indexering is het bijtellen van een rare bit combinatie overbodig; de subroutine kan nu veel directer (en als pure procedure!) geschreven worden.

Een variant op deze methoden berust op het plaatsen van het terugkeeradres door het hoofdprogramma op een vaste plaats in de subroutine, bijv. de eerste

```
L     *                        SUB: ----------
C     SUB                         → ----------
J     SUB + 1 →                     ----------
----------                          ----------
----------                          ----------
```

(Met indirecte adressering kan men het opbergen van de "tag" ook op een andere plaats in de subroutine bewerkstelligen. Ga dit na!)

Omdat in deze aanroepmethoden van subroutines de L- en J-instructies altijd in vaste combinaties gebruikt worden, worden ze in de hardware vaak verenigd tot één speciale instructie, de zgn. subroutinesprong. In de meeste uitvoeringen plaatst deze het adres van de subroutinesprong (of van het daarop volgende woord) in een indexregister en springt dan naar de opgegeven plaats.

Twee zaken zullen daarom voor zichzelf spreken:
- er is een zeer directe wisselwerking tussen de hardware structuur (het instructiepakket) en de programmering van subroutine aanroepen,
- er moet een zeer consequente standaardisatie zijn van subroutine structuren, wil men subroutines indien vereist kunnen uitwisselen en met elkaar laten samenwerken.

2.11.2. Mechanismen voor overdracht van parameters

Het zal slechts zelden gebeuren dat een hoofdprogramma een subroutine aanroept zonder één of meer parameters "mee te geven" (een voorbeeld van een parameterloze subroutine is een subroutine die een "random-number" moet leveren of de procedure read). Verschillende typen van parameters zijn mogelijk, bijv.
- argumenten en resultaat vóór een subroutine, bijv. voor $y = f(x_1 x_2 \ldots x_n)$,
- adressen van argumenten, dan wel het beginadres van een geheugengebied waarin argumenten staan (bijv. het adres van het eerste element van een matrix, die getransponeerd moet worden).
- adressen voor de door de subroutine te produceren resultaten,
- parameters, die voor een subroutine met vele faciliteiten, de gewenste selecteert.

Van de vele ontworpen mechanismen om parameters door te geven noemen we:
- het direct verplaatsen van parameters van hoofdprogramma naar subroutine met behulp van een reeks van L (met adressen die in het hoofdprogramma bekend zijn) en C (met adressen die uit de subroutinebeschrijving bekend zijn) bewerkingen die óf in hoofdprogramma óf in subroutine staan. Een variant hierop is het aanwijzen van een vast "common" gebied, waarvan het beginadres zowel aan hoofdprogramma als aan subroutine bekend zijn en waarin parameters (de "globale") in een vaste volgorde geplaatst moeten worden. Bij zeer grote aantallen parameters, die weinig veranderen, is deze variant meestal minder omslachtig dan direct verplaatsen. Nog weer een andere variant is het gebruik van de geheugenlocaties volgende op de subroutinesprong,

die dank zij de "tag" immers aan hoofdprogramma en subroutine beiden bekend zijn. Het grote voordeel van de laatste variant is dat schrijvers van hoofdprogramma en subroutine alleen een afspraak moeten maken over de volgorde van parameters, niet over hun adressen.

- het meegeven van parameters in indexregisters is ook gebruikelijk, maar dit kan uiteraard alleen maar toegepast worden als het aantal parameters klein is en de te benutten registers niet voor andere taken in het hoofdprogramma nodig zijn. Wanneer de te benutten registers in de subroutine nodig zijn, moet hun inhoud door de eerste instructies van de subroutine "opgeborgen" worden op geheugenplaatsen die aan de subroutine bekend zijn. Er is dan wat aantallen bewerkingen betreft nog maar nauwelijks verschil met de vorige methode, maar de indexregistermethode is toch wat beter omdat schrijvers van hoofdprogramma en subroutine van elkaar alleen hoeven te weten in wèlke registers bepaalde parameters zitten en pure procedure mogelijk zijn;
- uit de genoemde twee mechanismen volgen direct twee andere als men niet de parameters zelf meegeeft, maar de adressen van de parameters of parametergebieden (de eenvoudigste vorm van een call by name). Dit kan dan weer gebeuren in indexregisters of in de geheugenlocaties volgend op de subroutinesprong;
- tenslotte noemen we hier alvast het mechanisme van het plaatsen van parameters in een (later te behandelen) stapel.

Opgave. Ga zelf na hoe men indirecte adressering in plaats van indexregisters kan gebruiken voor deze parametermechanismen.

2.11.3. Redden van registers

Een belangrijk facet van het subroutinemechanisme hebben we tot dusver over het hoofd gezien, nl. het feit dat zowel hoofdprogramma als subroutine van accumulator(en) en indexregisters gebruik zullen maken. Wanneer daarom geen voorzieningen worden getroffen om de inhoud van deze registers te "redden" door ze tijdelijk in het geheugen op te bergen, zou het hoofdprogramma niet goed voortgezet worden nadat een subroutine zijn werk heeft gedaan.

Hoewel het denkbaar is om het hoofdprogramma de gewenste registers te laten redden, is dit niet gebruikelijk en wordt deze taak overgelaten aan de subroutine. Dit is niet alleen consequenter gezien het concept dat subroutines a.h.w. een uitbreiding vormen van het instructiepakket van een machine

(zonder neventaken op te leggen als het redden van registers), maar ook beter omdat pas in de subroutine bekend is van wèlke registers gebruik zal worden gemaakt. De structuur van een subroutine zal dus in het algemeen de volgende zijn:

- het redden van registers,
- evt. het verzorgen van terugkeeradres(sen),
- het ophalen en bewerken van parameters,
- de eigenlijke subroutine bewerking,
- het doorgeven van de resultaten,
- herstellen van registerinhouden,
- terugsprong.

Opmerking. Soms is één enkele instructie beschikbaar om een hele reeks registers te redden of te herstellen.

Waar moeten registers echter opgeborgen worden en waar moeten werkgebieden voor de subroutine gekozen worden? De eenvoudigste oplossing waar men op komt, is om hiervoor red/werkgebieden in de subroutine zelf te gebruiken. Aan deze oplossing kleven echter de volgende bezwaren:

- wanneer een programma in het kernengeheugen om de een of andere reden onderbroken wordt, kan na "geheugenverlies" voor het herstarten niet gebruik worden gemaakt van de oorspronkelijke kopie van een subroutine in het secundaire geheugen omdat de red/werkgebieden daarin blanco zijn,
- een subroutine zou zonder een additionele regel, nl. dat een eenmaal begonnen subroutine niet onderbroken mag worden, niet te gebruiken zijn door twee of meer hoofdprogramma's "tegelijkertijd", omdat dan wederom red/werkgebieden vernietigd kunnen worden,
- een subroutine zou zichzelf niet direct of indirect (d.i. via een andere subroutine) kunnen aanroepen en daarmee zou recursief gebruik onmogelijk zijn.

Een programmadeel dat zichzelf niet verandert en dat geen red/werkgebied in zichzelf bevat (al eerder pure procedure genoemd) kan dan door verschillende gebruikers benut worden op "vrijwel hetzelfde" moment en heet daarom ook wel "re-entrant". Re-entrant programma's die zichzelf kunnen aanroepen, heten recursief.

Voor geavanceerd computergebruik mogen de red/werkgebieden dus niet lokaal zijn t.o.v. een subroutine, maar moeten door het aanroepende programma bepaald worden. Het aanroepende programma zou dan via bijv. een indexregister aan de subroutine kunnen mededelen waar het redgebied ligt, nl. door in een indexregister het adres van de eerste locatie te plaatsen. Met symbolische adresarithmetiek zijn dan ook de volgende locaties te bereiken. Tijdens gebruik van deze gebieden moet onderzocht worden of de ter beschikking gestelde ruimte nog wel voldoende is.

Om bovendien red/werkgebieden helemaal veilig te stellen moet de eerste taak van de subroutine zijn (ga dit na!) om

- hetzij voor iedere afzonderlijke redoperatie het gebruikte indexregister met 1 op te hogen (niet er na omdat eerst onderzocht moet worden of er nog ruimte is) en na iedere hersteloperatie het indexregister met 1 te verlagen (bij deze methode is geen symbolische adresarithmetiek nodig, wel gaat iedere red- of hersteloperatie gepaard met een indexregister operatie, hetgeen extra rekentijd vergt),
- hetzij voor de eerste redoperatie het gebruikte indexregister meteen verhogen met het totale aantal geheugenlocaties dat voor red- en werkgebieden nodig is in de betreffende subroutine (voor het benoemen van locaties in deze gebieden is dan symbolische adresarithmetiek nodig).

De volledige structuur van een subroutine zal dus moeten zijn:

- veilig stellen van red- en werkgebieden,
- redden van registers,
- evt. verzorgen van terugkeeradres(sen),
- de eigenlijke subroutine instructies,
- herstellen van registerinhouden,
- vrijgeven van red- en werkgebieden,
- terugsprong.

2.11.4. Een gedetailleerd voorbeeld

Laten we aannemen dat niet alleen in één programma op verschillende plaatsen de absolute waarde van de som van n (≥ 0) getallen (n mogelijk zelfs verschillend op die plaatsen) bepaald moet worden, doch dat deze bewerking plaats moet vinden in meer programma's, die elkaar op onvoorspelbare ogenblikken kunnen onderbreken om zelf voort te gaan. Die onder-

brekingen kunnen ook tijdens de uitvoering van de te maken gemeenschappelijke subroutine, MAS, plaatsvinden; de inhoud van de accumulatoren wordt bij iedere onderbreking "gered" evenals het punt van onderbreking, zodat ieder onderbroken programma t.z.t. weer herstart kan worden. De taak van de subroutineschrijver is er voor te zorgen dat dit herstarten goed kan verlopen.

Met opzet is een eenvoudig voorbeeld gekozen om zoveel mogelijk aandacht te kunnen geven aan de koppeling van hoofdprogramma's en subroutine MAS. Ter wille van de eenvoud èn om pure procedures te construeren nemen we ook aan dat we maximaal drie indexregisters kunnen gebruiken, waarvan de eerste als "kladpapiertje" gebruikt mag worden. Voorts nemen we aan dat we reeds beslist hebben dat parameters van de subroutine zijn:

- het getal n
- het adres van het eerste van de op te tellen getallen
- het adres waar de som geplaatst moet worden.

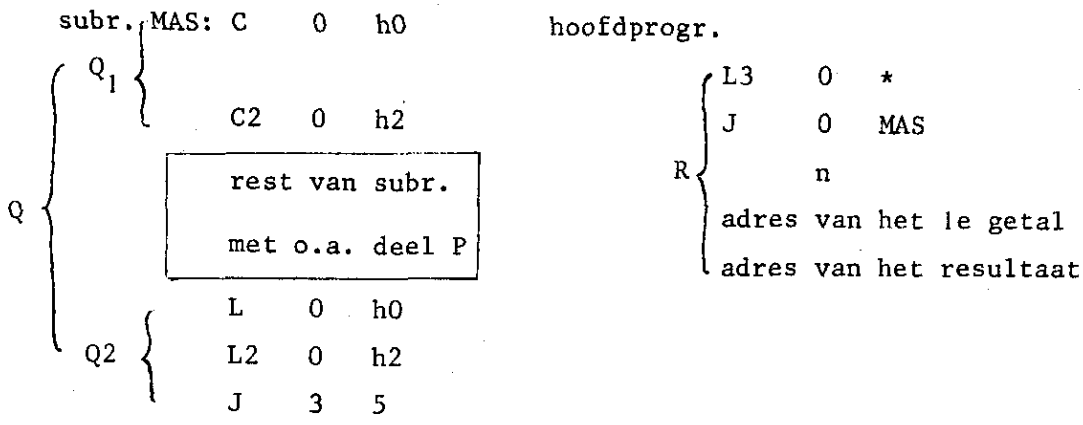
(deze beslissing had ook kunnen zijn dat vóór de op te tellen getallen het getal n geplaatst wordt, dat het resultaat achter het laatste van die getallen komt en dat het adres van n dan de enige parameter is. Werk zelf voor dit geval de subroutine uit!)

Met al deze veronderstellingen zou het bewerkingsdeel van de subroutine dan bijvoorbeeld de volgende gedaante kunnen krijgen (of het korter kan, laten we in het midden)

```
P   L1   0   g1          A := n
    L2   0   g2          A2 := "adres van het eerste getal"
    L    0   ZERO        A := 0 (begin met een lege accumulator)
    J    0   * + 3
    A    2   0           optelbewerking
    A2   0   ONE         A2 := "adres van het volgende getal"
    S1   0   ONE  }      lustest
    P1   0   * - 3 }
    L2   0   g3   }
    C    2   0    }
    P    0   * + 4  }    resultaat wegschrijven met het goede teken
    S    2   0      }
    S    2   0      }
    C    2   0    }
```

Het zal duidelijk zijn dat geheugenplaats ZERO het getal 0 bevat en ONE het getal 1, hetgeen met DC pseudoinstructies bewerkstelligd kan worden. Over de geheugenplaatsen g1 t/m g3, die met de drie parameters corresponderen, zullen we ons later uitspreken.

Omdat indexregisters beschikbaar zijn, is de koppeling van hoofdprogramma en subroutine eenvoudig als volgt te bewerkstelligen:



Op te merken is hierbij dat:

- **alleen die registers A0 en A2 gered worden die de subroutine gebruikt, A1 was kladpapier;**
- aangenomen is dat het hoofdprogramma na de werking van de subroutine weer hervat moet worden vijf plaatsen na de L3 0 * tag-instructie (waar de tussengelegen geheugenplaatsen voor gebruikt kunnen worden, zullen we nog zien);
- we nu nog moeten bekijken hoe de parameters overgedragen worden en wat te zeggen is over de geheugenplaatsen g1 t/m g3, h0 en h2.

Een reeds genoemde, doch veelal te verwerpen methode komt hierop neer dat men de geheugenplaatsen g1 t/m g3, h0 en h2 "onderbrengt in de subroutine". We zullen eerst de uitwerking geven en dan de bezwaren bespreken.

Voor het parametertransport bestaan zoals gezegd verschillende methoden: een oude, slechte methode is om in het hoofdprogramma vóór de sprong naar MAS en de "tag" instructie met behulp van L en C bewerkingen de drie parameters over te brengen naar de subroutine geheugenplaatsen g1 t/m g3. De parameters in g1 t/m g3 zijn dan meteen goed voor gebruik (verifieer dit).

Deze methode is daarom slecht omdat de schrijver van het hoofdprogramma precies moet weten welke absolute of symbolische adressen voor g1 t/m g3 gebruikt zijn in de subroutine en omdat, wanneer symbolische adressen gebruikt zijn, hoofdprogramma en subroutine samen ter vertaling moeten worden aangeboden opdat de vertaler in beide programma's aan g1 t/m g3 dezelfde absolute adressen toewijst. (Zijn er meer hoofdprogramma's dan zijn er nog meer complicaties mogelijk, waar we echter niet op in zullen gaan). Om vrijwel dezelfde reden is ook het gebruik van een "common" gebied te verwerpen.

Een goede methode voor parametertransport is de parameters bij elkaar te plaatsen in een gebied dat zonder verdere voorzieningen aan hoofdprogramma en subroutine bekend is. De eerste gedachte is om daarvoor de geheugenplaatsen te gebruiken, die direct volgen op de sprong naar de subroutine (om die reden hebben we die plaatsen dan ook vrij gehouden!) Dank zij de "tag" zijn deze plaatsen immers aan de subroutine bekend, zodat alleen nog maar de volgorde van de parameters van belang is (hier: aantal getallen, adres van het eerste getal, adres van het resultaat). Een volledige subroutine is dan bijvoorbeeld de volgende (register 3 voor de terugkeer; in dit voorbeeld zijn ook de geheugenplaatsen g1 t/m g3 niet nodig omdat buiten de accumulatoren geen tussenresultaten bewaard hoeven te worden):

```
        OR      ...
ZERO  : DC      0
ONE   : DC      1
     h0: DA     1
     h2: DA     1
MAS   : C   0   h0      } redden van A en A2
        C2  0   h2
        L1  3   2       A1 := n
        L2  3   3       A2 := "adres van het eerste getal"
        L   0   ZERO    A := 0  (begin met een lege accumulator)
        J   0   * + 3
        A   2   0       optelbewerking
        A2  0   ONE     A2 := "adres van het volgende getal"
        S1  0   ONE     }
        P1  0   * - 3   } lustest
        L2  3   4       A2 := "adres van het resultaat"
        C   2   0       }
        P   0   * + 4   }
        S   2   0       } resultaat wegschrijven met het goede teken
        S   2   0       }
        C   2   0       }
        L   0   h0      }
        L2  0   h2      } herstellen van A en A2
        J   3   5       terugsprong
```

Een variant op deze methode van parametertransport is de parameters in een bepaald gebied bij het hoofdprogramma bij elkaar te zetten en het adres (van de eerste locatie) van dit gebied via een indexregister (anders dan het derde) aan de subroutine door te geven. Een (klein) voordeel van deze methode is dat de terugsprong voor alle subroutines dan altijd kan gebeuren naar een vast aantal plaatsen na de sprong naar de subroutine (de relatieve plaats is immers dan niet meer afhankelijk van het aantal parameters). Men kan deze gedachte nog verder vervolgen in die zin dat men het adres (of de adressen; soms zijn meer terugsprongadressen nodig als de terugsprong bijvoorbeeld afhankelijk is van het resultaat) waarheen teruggesprongen moet worden ook in het parametergebied zet. Wel moeten uiteraard adressen en andere parameters in een vaste volgorde in zo'n gebied staan.

Wat is dan nu nog het bezwaar van deze subroutinevorm? Wel, als meer hoofdprogramma's "tegelijkertijd" gebruik maken van de gemeenschappelijke subroutine is er bij wederzijdse onderbreking een kans (ga na wanneer) dat redgebieden (h0 en h2) worden overgeschreven, zodat hoofdprogramma's niet goed voortgezet worden. Hetzelfde zal zeker gebeuren als subroutines zichzelf direct of indirect aanroepen. Men zal voor deze gevallen bijzondere voorzieningen moeten treffen.

Een goede, altijd te gebruiken methode blijkt meteen te volgen uit het alternatief de werk- en redgebieden dan ook maar onder te brengen in het hoofdprogramma dat de subroutine aanroept. Dit kan gebeuren door een eenvoudige voortzetting van de gedachte over het parametergebied, nl. zet er red- en werkgebieden direct achter (het geheel wordt verder "speelruimte" genoemd). Via het indexregister dat naar het adres van dit gebied wijst kan men zowel in hoofdprogramma en subroutine alle locaties bereiken. Als ieder hoofdprogramma zo zijn eigen speelruimte heeft, kunnen red- en werkgebieden ook niet ten onrechte overschreven worden. De uitwerking van deze gedachtengang is in het volgende programma gegeven. Het hoofdprogramma bevat de instructies:

L3	0 ...	adres van de speelruimte
----------		eventueel andere instructies die echter niet A3 mogen veranderen (ga dit na)
C	3 2	n wordt in de speelruimte geplaatst vanuit de accumulator
----------		eventueel andere instructies
C	3 3	het adres van de getallenreeks wordt geplaatst
----------		eventueel andere instructies
C	3 4	het adres van het resultaat wordt geplaatst
----------		eventueel andere instructies (die A3 niet veranderen!)
C	3 0	het terugsprongadres wordt geplaatst
----------		eventueel andere instructies
J	0 MAS	sprong naar de subroutine

De subroutine heeft dan de volgende gedaante:

```
      OR      ...
ZERO: DC      0
ONE : DC      1
MAS : C   3   6        }
      C2  3   7        }   redden van A en A2
      L1  3   2            A1 := n
      L2  3   3            A2 := "adres eerste getal"
      L   0   ZERO         A := 0  (begin met een lege accumulator)
      J   0   * + 3
      A   2   0            optelbewerking
      A2  0   ONE          A2 := "adres van het volgende getal"
      S1  0   ONE    }     lustest
      P1  0   * - 3  }
      L2  3   4            A2 := "adres van het resultaat"
      C   2   0      }
      P   0   * + 4  }
      S   2   0      }     wegschrijven resultaat
      S   2   0      }
      C   2   0      }
      L   3   6      }     herstellen van A en A2
      L2  3   7      }
      L1  3   0      }     terugsprong
      J   1   0      }
```

Omdat bij wederzijdse onderbreking alle accumulatoren gered en hersteld worden, zal nu alles goed verlopen. Gezien de terugsprong zal nu duidelijk zijn waarom een klad indexregister nodig was.

Voor het hier behandelde voorbeeld is de structuur van deze subroutine nu wel in orde, maar in het algemeen toch nog niet. Wat gebeurt er namelijk wanneer een subroutine , **onderweg** op zijn beurt een andere subroutine of zichzelf (dus recursie) aanroept? Als bovenstaande subroutine bijvoorbeeld na de L 0 ZERO instructie zonder meer zichzelf zou aanroepen dan wordt door het uitvoeren van de redbewerkingen informatie overschreven die later nog nodig is. Bovendien is voor die nieuwe aanroep speelruimte nodig, die

niet bij de subroutine zelf ondergebracht mag worden als die geschikt moet zijn voor gemeenschappelijk gebruik door meer hoofdprogramma's. De oplossing moet weer gevonden worden door:

- de nieuwe speelruimte eveneens onder te brengen bij het hoofdprogramma,
- er voor te zorgen dat de speelruimte van een nog niet afgewerkte subroutine niet overschreven wordt door de speelruimte van de volgende aanroep.

Het laatste is eenvoudig te bereiken door voor de volgende aanroep de inhoud van het naar de speelruimte wijzende indexregister te vermeerderen met een dusdanig bedrag dat de nog van belang zijnde speelruimte niet overschreven kan worden. Omgekeerd moet bij terugkeer uit een subroutine dat indexregister onmiddellijk met hetzelfde bedrag verminderd worden om weer toegang te krijgen tot de eigen speelruimte. Uiteraard moet dit ook gebeuren in een hoofdprogramma.

Ter toelichting van het laatste zullen we in het volgende een recursief werkende subroutine uitwerken waarmee n! berekend kan worden. We nemen aan dat parameters van deze subroutine zijn: het terugsprongadres, n (≥ 0) en dat het resultaat direct na n geplaatst moet worden; dat het hoofdprogramma op het moment van aanroepen van de subroutine al vele tussenresultaten in zijn speelruimte heeft staan. In het hoofdprogramma staan dan de volgende instructies:

```
L3   0   .....     adres van de speelruimte wordt in A3 geplaatst
A3   0   .....     wat al in de speelruimte is wordt "afgeschermd" door bij A3 de goede integer bij te tellen
C    3   0         terugsprong adres → speelruimte
C    3   1         n → speelruimte
J    0   FAC
```

De subroutine FAC zou dan de volgende gedaante kunnen hebben:

```
      OR      ...
ONE : DC      1
FOUR: DC      4
STEP: DC      4
FAC : C   3   3          redden accumulator
      L   3   1
      S   0   ONE  }
      S   0   ONE  }     test op de gevallen n = 0 of 1
      P   0   * + 3}
      L   0   ONE
      J   0   * + 12
    → L   3   1
      S   0   ONE
      A3  0   STEP       veilig stellen oude speelruimte
      C   3   1
      L1  0   *    }
      A1  0   FOUR }     klaarmaken van recursieve terugsprong
      C1  3   0    }
      J   0   FAC        recursieve subroutine aanroep
      L   3   2
      S3  0   STEP       afbreken van speelruimte
      M   3   1          n! = n * (n - 1)!
    → C   3   2
      L   3   3          herstel accumulator inhoud
      L1  3   0    }
      J   1   0    }     terugsprong naar FAC of hoofdprogramma
```

Opmerkingen.

1. De constanten op de plaatsen FOUR en STEP zijn, hoewel in grootte gelijk, apart gedefinieerd om tot uitdrukking te brengen dat ze ieder een eigen functie hebben. Ga dit na!
2. Gezien de afspraken over het eerste en derde indexregister hoeft alleen maar de accumulator inhoud gered en hersteld te worden (de instructies C 3 3 en L 3 3).

Bij het lezen van het voorgaande komt de vraag op of er niet een eenvoudiger oplossing is voor deze programmeringsproblemen van recursief werkende subroutines. Het komt er eigenlijk op neer dat we behoefte hebben aan een hardware mechanisme dat ons in staat stelt om ergens iets in het geheugen te schrijven en het meteen daarna onmogelijk te maken deze informatie door overschrijving te vernietigen. We zouden hiermee een "stapel" informatie kunnen bewaren met de eigenschap dat nieuwe informatie slechts boven op de stapel geplaatst kan worden, terwijl omgekeerd informatie slechts uitgelezen kan worden door achter elkaar steeds van de momentane bovenste stapelplaats informatie op te halen. Een voor de programmeur anoniem register zou dan als "stapelwijzer" kunnen fungeren in die zin dat de stand van de stapelwijzer de eerstvolgende "vrije" geheugenplaats aangeeft. Zoals we later zullen zien wordt zo'n stapel (cellar, stack, last-in-first-out queue) mechanisme inderdaad gebruikt in 0-adres machines. Bij n-adres ($n \geq 1$) machines moet het stapelmechanisme met software gesimuleerd worden.

Een probleem dat we hier niet verder in detail zullen uitwerken is de call by name voor actuele parameters die geïndexeerde variabelen of expressies zijn. Het zal echter duidelijk zijn dat we in dit geval als parameter naar de subroutine een adres moeten doorgeven en wel een adres van een te construeren subroutine die het adres van de geïndexeerde variabele berekent, hetzij de expressie evalueert.

2.12. Open subroutines (macro's)

Omdat ze wel programmeerwerk maar geen geheugenruimte sparen, zijn open subroutines minder gebruikelijk dan gesloten subroutines. Eerst een voorbeeld van een open subroutine: wanneer in het hardware repertoire van een machine niet een instructie voorkomt die de absolute waarde van een getal uit het geheugen in de accumulator plaatst, is het plezierig om aan het software repertoire een open subroutine of zoals ook genoemd een "macro-instructie" toe te voegen die ons in staat stelt om dit te doen met bv.

```
LA          ADRES
```

Het assembleer programma, dat een in een symbolische assembleertaal geschreven programma omzet in een machinetaalprogramma, moet dan herkennen dat met LA geen hardware instructie correspondeert en moet in zijn lijst van geldige macro-instructies constateren dat deze LA instructie vervangen moet worden door een aantal "gewone" instructies. Welke gewone instructies dit zijn, moet gedeclareerd zijn, bijv. voor een machine zonder indexregisters als volgt

```
LA          MACRO(ADRES)
            L     ADRES
            P     * + 3      } "macrobody"
            S     ADRES
            S     ADRES
            END
```

Op de plaats van de macro-instructie LA wordt dus de macrobody met het in de macro-instructie opgegeven adres gesubstitueerd, hetgeen zoals eenvoudig geverifieerd kan worden het gewenste effect heeft. We zien aan dit eenvoudige voorbeeld verder dat een macro vooral dan te prefereren is boven een gesloten subroutine wanneer het aantal subroutine instructies betrekkelijk klein is. Het inschakelen van een gesloten subroutine vormt dan eigenlijk een onevenredig zware geheugen- en verwerkingstijd belasting.

In een macrodefinitie kunnen zoveel "dummy variabelen" (hierboven ADRES) optreden als nuttig is, bv. twee in de macro WISSEL, die de inhoud van twee geadresseerde woorden verwisselt:

```
WISSEL      MACRO(ADRES1,ADRES2)
            L     ADRES1
            C     HULP
            L     ADRES2
            C     ADRES1     } "macrobody"
            L     HULP
            C     ADRES2
            END
```

Als in het vorige voorbeeld zijn de "symbolische (pseudo) instructies" MACRO en END, die de macrobody omsluiten, slechts van belang voor de assembler; de gebruiker die deze macro aldus gedeclareerd heeft kan in de statements van zijn programma verder volstaan met instructies als

```
WISSEL    AAA,    BBB
```

waarin AAA en BBB willekeurige symbolische adressen zijn. De assembler moet er verder voor zorgen dat het werkgebied HULP "lokaal" blijft t.o.v. deze macro-instructie en dat er geen verwarring kan ontstaan met werkgebieden die

in het hoofdprogramma ook HULP genoemd zijn.

Het gebruik van macro's maakt het mogelijk om tijdens de programmeringsphase bepaalde beslissingen uit te stellen. Weet men nog niet of men een gesloten of open subroutine zal programmeren, dan kan men bijv. in zijn programma alvast zetten:

```
        CALL        SUB, PAR 1, PAR 2
```

en achteraf besluiten om één van de volgende macro's te schrijven:

```
CALL   MACRO   (SUBR, PARM1, PARM2)        CALL   MACRO   (SUBR, PARM1, PARM2)
       L       *      ┐                             ----------      ┐
       J       SUBR   │                             ----------      │ macrobody
       PARM1          │                             ----------      │
       PARM2          ┘                             ----------      ┘
       END                                          END
```

(waarbij voor het aanroepen van de gesloten subroutine terwille van de eenvoud een zeer simpel mechanisme als voorbeeld is genomen. Deze macro vernietigt de inhoud van de accumulator!). Op het probleem hoe deze "expansie van een macro" tot stand komt, zullen we hier niet verder ingaan. Ook niet op macrofaciliteiten die de laatste tijd in zwang komen als

- voorwaardelijke expansie (welke reeks gewone instructies gegenereerd wordt hangt af van bijv. bepaalde parameters; het voordeel is dat afhankelijk van omstandigheden tijdens expansie slechts een deel van de hele macrobody overgeschreven wordt, terwijl een gesloten subroutine in zijn geheel moet worden gegeven),
- de mogelijkheid dat een macro weer andere macro's oproept ("nesting").

De reden dat we hier niet op ingaan, is dat deze bijzondere faciliteiten nog te zeer gebonden zijn aan bepaalde assembleertalen en deze weer aan bepaalde machinetypen. Dit neemt niet weg dat macro's en hun faciliteiten vrijwel de enige mogelijkheid geven om op een laag taalniveau (dus onder dat van proceduretalen als Algol) een standaardisatie en uniformiteit te bereiken op het gebied van de harde programmatuur. Als het zou lukken om over deze standaardisatie overeenstemming te bereiken, zou men inderdaad machineonafhankelijke programmatuur kunnen ontwikkelen zonder het bezwaar van efficiëntieverlies te ontmoeten. Met de macro expansie kan men immers naar hartelust bijzonderheden van een bepaalde machine zo goed mogelijk tot zijn recht laten komen.

(Een alternatieve gedachte, nl. om Algol voor het schrijven van harde programmatuur uit te breiden met een aantal faciliteiten, is overigens wel zo aantrekkelijk.)

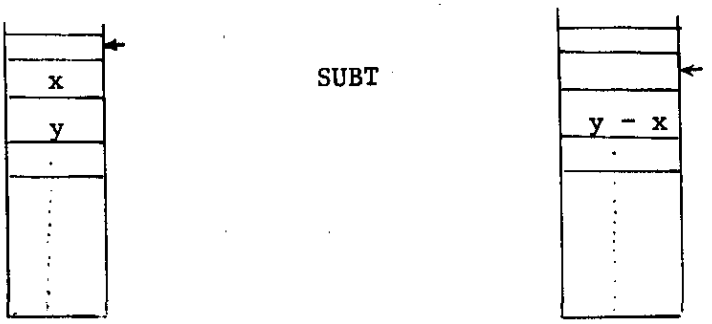
2.13. Stapel of 0-adres machines

In het voorgaande is reeds enkele malen gezinspeeld op de zgn. stapel of 0-adres machines, waarbij bijna alle instructies "adresloos" zijn. Uiteraard brengt dat met zich mee dat de operand(en) of operandadres(sen), waarop de bedoelde instructies betrekking hebben, vooraf op voor de machinebesturing bekende en opeenvolgende maar voor de programmeur onbekende geheugenplaatsen gebracht moet(en) zijn.

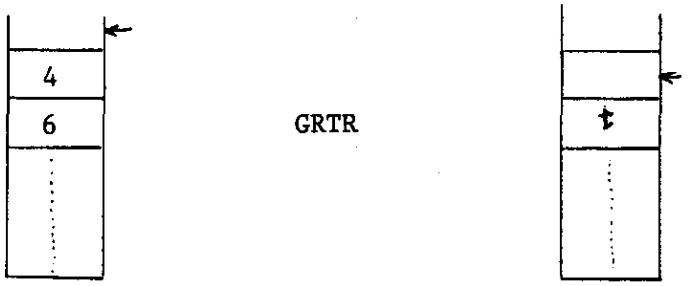
Bij geavanceerde machines is het gebruikelijk dat

- de bovenste twee stapelplaatsen ondergebracht zijn in het rekenkundige orgaan, opdat operaties op hier geplaatste operand(en) snel uitgevoerd kunnen worden (is de hele stapel in het kernengeheugen ondergebracht, dan zijn extra geheugentoegangen nodig),
- voor de "dieper gelegen" stapelplaatsen in het kernengeheugen ruimte gereserveerd wordt, waarbij een "anonieme" (niet toegankelijk voor de programmeur) stapelwijzer, S, de geheugenplaats volgende op de laatst gevulde geheugenplaats aangeeft,
- de machinebewerkingen beheerst worden door drie typen instructies, nl.
  . adresloze "operatoren" werkend op de bovenste of twee bovenste stapelplaats(en),
  . instructies (met een adresdeel) waarmee één operand boven op de stapel geplaatst wordt,
  . andere instructies, o.a. om een adres van een operand boven op de stapel te plaatsen.
- twee anonieme registers plaats en maximum grootte van de stapelruimte in het kernengeheugen bepalen, zodat niet per ongeluk buiten deze stapelruimte geschreven kan worden.

Om getallen op de top van de stapel te krijgen gebruiken we de instructie VALC(x), waarbij x de naam van een getal is met een bekend adres in het geheugen (VALC = VALUE CALL).
Een adres van een getal met naam x krijgen we op de top van de stapel door: NAMC(x); nu verschijnt het adres van x op de stapel (NAMC = NAME CALL).
Zowel VALC als NAMC bewerken een verhoging van de stapelwijzer met 1.
De vier bekende rekenkundige bewerkingen noemen we: ADD, SUBT, MULT en DIVD.
Alle werken op de bovenste twee elementen van de stapel volgens het volgende voorbeeld:



Het vergelijken van twee getallen geschiedt met de adresloze instructies GRTR, LESS en EQUL; voor bv. GRTR als volgt: als het een na bovenste element van de stapel groter is dan het bovenste, dan krijgt dat een na bovenste element de waarde t (= true) (indien niet, dan de waarde f) en de stapelwijzer zakt één positie. Dus



LESS en EQUL werken analoog.

De sprongopdrachten (absoluut en voorwaardelijk) worden aangeduid door:

BRUN(L) Branch Unconditional naar label L.

BRFL(L) Branch if False naar label L, d.w.z. als de top van de stapel f bevat (d.i. een false uitkomst van een vergelijkopdracht) dan wordt naar L gesprongen, de stapelwijzer zakt altijd een plaats.

BRTR(L) Branch if true naar label L, de stapelwijzer zakt altijd een plaats.

Om informatie van de top van de stapel weer naar het geheugen te brengen maken we gebruik van:

STOD Store Destructive: de inhoud van de opeen na hoogste stapelplaats wordt gebracht naar het adres genoemd in de top van de stapel; de stapelwijzer zakt twee plaatsen!

STON Store Non-destructive: als STOD, de stapelwijzer zakt slechts één plaats en de informatie die zojuist weggebracht is, blijft echter achter op de top van de stapel.

Contact met "buiten" vindt plaats via:

READ Op de top van de stapel verschijnt het eerstvolgend getal van een band; de stapelwijzer wordt één hoger.

PRINT De top van de stapel wordt uitgeprint, de wijzer zakt een plaats.

De oplossing van de laatste opgave van §2.5 wordt:

instructie	stapel
VALC(a)	a
VALC(b)	a,b
VALC(c)	a,b,c
MULT	a,b * c
SUBT	a - b * c
VALC(d)	a - b * c,d
VALC(e)	a - b * c,d,e
VALC(f)	a - b * c,d,e,f
MULT	a - b * c,d,e * f
SUBT	a - b * c,d - e * f
DIVD	(a - b * c)/(d - e * f)
NAMC(r)	
STOD	

Vergelijk deze oplossing met die van §2.5 voor een 1-adres machine.

De som van drie getallen, die op de band staan, berekenen en printen we met:

instructie	stapel
READ	g_1
READ	g_1, g_2
READ	g_1, g_2, g_3
ADD	$g_1, g_2 + g_3$
ADD	$g_1 + g_2 + g_3$
PRINT	

Wanneer we te maken hebben met een machine waarvan de bovenste twee woorden van de stapel snelle registers zijn, is een sneller programma voor beide voorbeelden denkbaar. Het tweede wordt dan bv.:

instructie	stapel
READ	g_1
READ	g_1, g_2
ADD	$g_1 + g_2$
READ	$g_1 + g_2, g_3$
ADD	$g_1 + g_2 + g_3$
PRINT	

In de voorbeelden zien we dat noch de vulling van de stapel, noch de stand van de stapelwijzer voor de programmaschrijver van belang is, wat het gebruik van het woord anoniem rechtvaardigt. Door vergelijken van de twee voorbeelden zien we verder dat we ons ook geen zorgen hoeven te maken over het opbergen van tussenresultaten (bijv. $g_1 + g_2$), hetgeen bij 1-adres machines wel het geval is (er moet natuurlijk wel een boodschap of automatische speciale voorziening zijn wanneer de oorspronkelijk toegewezen stapelruimte te klein blijkt te worden).

De gedachte om voor tussenresultaten de stapel te gebruiken kan verder uitgebreid worden tot procedures. Als in het eerste voorbeeld a niet een variabele is maar een procedure die nog uitgewerkt moet worden dan is hiervoor ook de stapel te gebruiken voor o.a.

- het bewaren van terugkeerinformatie nodig bij het verlaten van de procedure,
- het bewaren van de actuele parameters,
- het bewaren van de locale variabelen van de procedure,
- het bewaren van de tussenresultaten tijdens de werking van de procedure.

In dit college zal niet verder ingegaan worden op de practische realisering van het verwerken van procedures bij stapelmachines, doch zij volstaan met het vermelden van de display vector die ingevoerd moet worden om problemen op te vangen als o.a. het toegelaten zijn van globale parameters en van het verlaten van procedures met goto statements. Ook indexering van variabelen zullen we in deze inleiding niet behandelen.

Wanneer we terugkijken naar de bij gesloten subroutines besproken problemen t.a.v. het redden van registers en werkgebieden bij het schrijven van recursieve subroutines en de daarvoor benodigde geprogrammeerde stapelbewerkingen bij 1-adres machines, dan zal aan te voelen zijn dat bij 0-adres machines voor deze recursieproblematiek een veel eleganter oplossing mogelijk is. Ook dit zullen we echter hier niet uitwerken. Een ander voordeel is dat de adresloze instructies met een klein aantal bits te representeren zijn, wat tot een dichte pakking van programma's aanleiding geeft.

2.14. Instructie repertoires

Ten behoeve van de voorgaande beschouwingen zijn slechts ongeveer 10 instructies gebruikt, terwijl in de meeste moderne computers meestal 150-200 instructies beschikbaar zijn. Dit op het eerste gezicht grote aantal instructies is te verklaren als we in aanmerking nemen dat bijv. voor de vier voornaamste arithmetische bewerkingen (+, -, × en /) de volgende varianten mogelijk zijn:

- vele data typen: integers, reals, double length reals, multi-length integers;
- werken met de operand of de absolute waarde van de operand;
- directe en onmiddellijke adressering;
- het resultaat van een bewerking kan in een (accumulator) register of het primaire geheugen komen.

Zouden alle combinaties mogelijk (en zinvol) zijn, dan zou dit al aanleiding geven tot 128 verschillende instructies.

Nog niet besproken zijn een aantal "logische" accumulator opdrachten voor bijv.

- and, not, inclusive or en exclusive or bewerkingen,

- de opschuif of rondschuif bewerkingen, waarmee de bitpatronen in een (accumulator-) register naar "links of naar rechts" worden opgeschoven (met bijv. verlies van wat "rechts naar buiten" verdwijnt) of rondgeschoven (wat er "rechts uitgaat komt er links weer in").

Aangezien we niet gesproken hebben over de interpretatie van deze bitpatronen zullen we niet verder ingaan op de zin van deze "logische" opdrachten, maar hier volstaan met de opmerking dat met deze opdrachten het "in- en uitpakken" (vijf 8-bits karakters worden bijv. achter elkaar geplaatst in een 40-bit woord) van karakters eenvoudig te realiseren is, evenals het vermenigvuldigen met machten van 2.

Eveneens direct aan te voelen, zijn de verschillende conditonele sprong-instructies, omdat als criteria gebruikt kunnen worden:

- het positief, negatief of nul zijn van (accumulator-)registers of geheugen inhouden,
- het (niet) groter, kleiner of gelijk zijn van de inhoud van twee registers of van een register en een geheugen inhoud.

Naast de net genoemde groepen van instructies, waarmee operanden bewerkt kunnen worden en waarmee het afwerken van een programma "gestuurd" wordt, is er een groep van instructies waarmee in de eerste plaats informatietransporten bewerkstelligd worden. Afhankelijk van het type invoer/uitvoerorgaan kunnen dit transporten zijn van telkens één enkel karakter (bij ponsband) tot ruim 100 karakters (bij een regeldrukker) met transportsnelheden tot enkele duizenden karakters per seconde. Bij informatietransporten naar secundaire geheugensystemen gaat het om meer massale transporten van honderden tot tienduizenden karakters per blok met snelheden - afhankelijk van de soort van het secundaire geheugen - van circa 100.000 karakters per sec. Behalve de instructies waarmee zo'n informatietransport geïnitieerd wordt, bestaan instructies om te onderzoeken of het transport is uitgevoerd, of het goed verlopen is, of een transportkanaal beschikbaar is, instructies om te transporteren informatie om te coderen (d.i. een ander bitpatroon te geven) of te verfraaien (d.i. "editing" met spaties, punten, komma's, geldsymbolen, sterretjes, enz.)

Nog weer een andere groep instructies hangt samen met het interrupt mechanisme van een rekenautomaat. Onderzocht kan o.a. worden:

- of een storing is opgetreden tijdens informatietransport (o.a. "pariteitscontrôle"),
- of een niet-bestaande instructie werd aangediend,
- of een bepaalde tijd verstreken is,
- of getracht werd in een "afgeschermd" stuk geheugen te schrijven (storage protection),
- of "externe" signalen (en welke) tot de rekenautomaat zijn doorgedrongen,
- of bij bepaalde arithmetische bewerkingen zinloze resultaten zijn geproduceerd ("divide check", "underflow" en "overflow" bij het produceren van te kleine of te grote reals).

Voor details t.a.v. instructies raadplege men een handleiding van een computerfirma.

2.15. Enkele vormen van computergebruik

In de begintijd van computergebruik was het gebruikelijk dat een programmeur voor een afgesproken tijd de beschikking kreeg over de rekenautomaat. Voorzien van zijn programma, gegevens en de documentatie (meestal alleen aantekeningen) van zijn probleem zorgde hij er voor dat zijn programma in het machinegeheugen terecht kwam (een standaard "monitor" programma hielp hem daarbij), dat onder besturing van zijn programma de gegevens wanneer nodig werden ingelezen en verwerkt en dat de resultaten uitgetypt werden. Met handschakelaars op het bedieningspaneel was hij verder in staat om eventuele fouten in zijn reeds ingelezen programma's te verbeteren of de afloop van een berekening te beïnvloeden. Deze vorm van "single job processing" bevredigde hem omdat hij het gevoel had direct zijn resultaten te krijgen.

Met het sneller worden van machines daalden de verwerkingstijden van een gemiddeld karwei van kwartieren naar één minuut of minder; de machines werden ook ingewikkelder wat de bediening betrof. Om zo min mogelijk tijd te verliezen voor manuale handelingen werd het net geschetste "open-shop" systeem vervangen door een "closed shop" systeem, waarbij een ploeg van professionele operateurs de verschillende machines bediende. De programmeur of machinegebruiker kwam niet meer verder dan de balie en moest afwachten wanneer zijn karwei tussen de vele andere verwerkt zou worden. Tussen het inleveren van zijn karwei en het terugkrijgen van resultaten kon een etmaal of meer liggen, al kostte zijn karwei niet meer dan een enkele minuut.

Omdat de electromechanische invoer/uitvoer-bewerkingen bij steeds sneller wordende inwendige operaties de totale verwerkingstijd van een job gingen bepalen, werd single job processing vervangen door batch-processing. Met simpele computers werd de geponste informatie van een reeks karweien getransporteerd naar een magneetband, deze werd dan verbonden met een snelle computer die alle zo vergaarde ("batched") karweien zonder stoppen achter elkaar verwerkte en de resultaten weer op een andere magneetband schreef. Met simpele computers (vaak waren 2 of 3 van deze "off-line" apparaten nodig om de snelle machine bij te houden) werden de resultaten dan afgedrukt. Het werk van operateurs bestond uit niet veel anders dan laden van kaarten, afscheuren van papier en verwisselen van magneetbanden. De snelle machine hield zelf wel de administratie bij van wat hij gedaan had, bij de verwerking gebruik makend van de besturingsinformatie die iedere gebruiker voor zijn programma geponst had. Bezwaren van deze batched processing waren dat het coördineren van het werk voor de verschillende machines vaak problemen gaf, dat in de praktijk niemand meer wist waar een karwei gebleven was totdat de resultaten verschenen en dat bij het manipuleren van magneetbanden fouten werden gemaakt. Bovendien bepaalde de volgorde van opladen van een magneetband ook de volgorde van verwerken en uitprinten zodat een paar grote karweien aan het begin van de band tientallen kleine karweien uren konden ophouden.

Een volgende verbetering was pas mogelijk na de invoering van multiprogramming. Massale informatie transporten (zoals het inlezen van een programma) nemen milliseconden in beslag, terwijl instructietijden in microseconden uitgedrukt moeten worden. Tijdens een karwei zou de centrale verwerkingseenheid (CVE: besturing en arithmetisch orgaan) daardoor in feite vaak weinig gebruikt kunnen worden. De eigenlijk simpele taak van besturing van informatietransporten werd daarom overgebracht naar de kanalen (tussen kernengeheugen en randapparatuur) die, nadat ze door de CVE aan de gang waren gezet, zelfstandig de gewenste informatietransporten konden verzorgen, terwijl de CVE zich verder bezig hield met het eigenlijke rekenwerk. Met andere woorden, men kreeg "overlap" van rekenen en informatietransport. Ook dit was nog niet afdoende want het rekenwerk kon wel eens opgehouden worden omdat het benodigde gegevenstransport nog niet beëindigd was.

In het streven naar zo volledig mogelijke benutting van de CVE kwam men toen tot het idee van

- "spooling" of "symbiont" operaties (multi-programmering in beperkte zin): de CVE gebruikt zijn wachttijd om periphere bewerkingen zo nodig aan de gang te houden. Voor iedere off-line bewerking vereist dit wel een eigen magneetband en invoer/uitvoer apparaat en meestal ook eigen kanaal, maar het voordeel is toch dat "tegelijkertijd" met een rekenkarwei, off-line laden van een magneetband (met volgende karweien) of uitprinten van een magneetband (met de resultaten van vorige karweien) plaats kan vinden,
- volledige multi-programmering, waarbij in de plaats van off-line bewerkingen andere rekenkarweien voortgezet worden. Hiervoor is nodig dat voor die andere rekenkarweien tenminste het "actieve" deel van het programma in het kernengeheugen aanwezig is, wat de behoefte aan grotere kernengeheugens deed toenemen.

Uiteraard gebeurt dit omschakelen van het ene karwei naar het andere weer onder besturing van een (althans gedeeltelijk) altijd in het kernengeheugen aanwezig standaard programma dat tegenwoordig een deel is van het bedrijfssysteem (operating system of control program). Dit bedrijfssysteem, waarvan ook het interrupt systeem een deel is, geeft de gebruiker veel faciliteiten. Het vereist ook vele hardware voorzieningen, waar we niet op in zullen gaan, en software voorzieningen, o.a. dat een machinetaalprogramma op iedere plaats in het geheugen kan werken omdat niet van tevoren vaststaat op welke plaats het eens moet werken. Multi-programmering is dus een gebruiksfaciliteit, waarbij de CVE zo volledig mogelijk benut wordt, doordat deze verscheidene programma's, die (althans gedeeltelijk) gelijktijdig in het kernengeheugen aanwezig zijn, afwisselend aan de gang zet. Naast het reeds genoemde bezwaar van grote kernengeheugens, waarvan een gebruiker slechts een zeer beperkt deel mag gebruiken, betekent dit grote aantallen magneetband eenheden en/of invoer/uitvoer apparaten, omdat ieder programma zijn eigen peripherie-apparaten moet hebben. Omdat de prijs van een CVE sterk daalt t.o.v. peripherieapparatuur is niet goed te overzien hoe ver men met multi-programmering moet gaan.

De gebruiker geeft met een besturingstaal (job control language) aan welke faciliteiten van het bedrijfssysteem hij wil gebruiken (periphere eenheden, benodigde ruimte, gewenste compilers, prioriteiten, enz.).

Zolang magneetbanden als buffer tussen invoer/uitvoer en berekeningsphase gebruikt worden, is de werkwijze van een rekencentrum nog steeds die van batch processing, al is het werk van de operateur wat eenvoudiger geworden. Daarentegen wordt nu een werkvoorbereider nodig, die eigenschappen, verwerkingstijden en afleveringstijden van programma's in aanmerking nemend, een optimale oplaadvolgorde van programma's moet bepalen.

Een volgende verbetering was mogelijk door voor de bufferfunctie de magneetbanden door "direct access" eenheden te vervangen (trommels en schijven). Door aan programma's prioriteitsgetallen toe te kennen, kan de verwerkingsvolgorde een andere zijn dan de oplaadvolgorde en door manuaal ingrijpen kan men zelfs na het opladen nog de verwerkingsvolgorde veranderen. Een volgende faciliteit, die aan deze queued jobprocessing toegevoegd kon worden, was mogelijk geworden door de ontwikkeling van apparatuur en programmatuur waardoor invoer/uitvoer apparatuur, door telefoonlijnen gekoppeld met de rekenautomaat, op willekeurig lange afstand opgesteld kon worden.

Bestaat deze apparatuur uit band- of kaart-lezers of -ponsers en regeldrukkers, dan geeft remote-batch-processing de mogelijk om ook van ver weg een karwei in de queue te plaatsen en later op de eigen regeldrukker de resultaten te krijgen.

Ondanks de verbeteringen, die o.a. door multi-programmering mogelijk werden, kleven aan alle batch methoden de bezwaren van het verlies van direct contact tussen gebruiker (programmeur) en machine. Is het karwei eenmaal ingezonden dan is er geen verdere actie t.a.v. de afloop van verwerking meer mogelijk. Om dit te ondervangen ging men zich bezinnen op de mogelijkheden van conversational processing. Dit is een manier van computergebruik waarbij de gebruiker met behulp van een via een telefoonlijn gekoppelde schrijfmachine tot op zekere hoogte de afloop van verwerking van zijn programma zou kunnen beïnvloeden, dan wel bij het schrijven van een programma direct een reactie van de rekenautomaat zou kunnen krijgen op evidente fouten.
Gekoppeld met de wens om op door de gebruiker te bepalen tijdstippen programma's en gegevens in de rekenautomaat te kunnen brengen of ze uit te lezen, leidde dit tot de ontwikkeling van time-sharing. De bedoeling is hierbij om een gebruiksfaciliteit te scheppen waarbij iedere gebruiker achter zijn schrijfmachine gezeten het gevoel heeft dat hij de volledige beschikking heeft over de machinefaciliteiten (met uitzondering van "sequential" werkende randapparatuur als magneetbandeenheden, kaartlezers, regeldrukkers, enz.). De hoofd-

gedachte bij de realisering is dat voor iedere gebruiker een actief deel van zijn programma in het kernengeheugen staat, welk deel dan met een bepaalde frequentie gedurende enkele tientallen milliseconden geactiveerd wordt. Daar de menselijke waarnemingstijd en reactietijd van de orde van seconden is, zou de gebruiker niet merken dat de rekenautomaat meestal voor andere gebruikers (hetzij ook gekoppeld, hetzij voor batched jobs) bezig was. Hoewel vele systemen zelfs al commercieel beschikbaar zijn, is de economie nog een twijfelpunt. Niet alleen voor de eigenaar van het systeem, die met een moeilijk voorspelbare hoeveelheid ongebruikte tijd kan blijven zitten, maar ook voor de leider van een programmeursgroep. Time sharing kan gemakkelijk leiden tot een slordige manier van werken (minder goed doordenken en vaker testen), zodat het helemaal niet zeker is dat time sharing effectiever is dan een snelle batch verwerking. Menselijke factoren zouden er echter wel de oorzaak van kunnen zijn dat de toekomst aan time sharing is (men heeft liever zijn eigen vervoermiddel dan het openbaar vervoer!).

Soms wordt in plaats van multi-programming ook wel van multi-processing gesproken, maar deze term wordt de laatste tijd meer gereserveerd voor die situatie, waarin meer dan één CVE problemen uit een gemeenschappelijk kernengeheugen verwerkt. Naast dit kernengeheugen zijn ook andere eenheden als regel voor gemeenschappelijk gebruik beschikbaar. Voordelen van deze configuratie zijn grotere bedrijfszekerheid en een grotere doorzet bij een relatief geringe extra investering. Als regel verwerkt iedere CVE zijn "eigen" karweien van begin tot eind.

Wanneer de CVE's gemeenschappelijk aan eenzelfde karwei werken, dan spreekt men meestal van parallel processing, een gebruikswijze die nog in de kinderschoenen staat.

Bij sommige rekenautomaat toepassingen zijn automaat en menselijk of physisch of chemisch proces direct met elkaar verbonden. Wanneer de automaat het initiatief neemt tot het uitwisselen van informatie dan spreekt men meestal van on-line processing. Is de kringloop gesloten zodat proces en automaat een geheel vormen, dan wordt de term in-line processing gebruikt. Gaat het initiatief voor informatie uitwisseling uit van het proces en wordt het hele informatie verwerkingstempo ook door het proces bepaald dan wordt gesproken van real-time processing. Ter waarschuwing zij opgemerkt dat deze termen ook wel door elkaar gebruikt worden.

Enkele karaktersets

Schrijfmachine		X8-ponsband	7-bits ISO	BCD	EBCDIC
A	U.C.	12-1	4-1	12-1	12-1
B		12-2	4-2	12-2	12-2
C		14-3	4-3	12-3	12-3
D		12-4	4-4	12-4	12-4
E		14-5	4-5	12-5	12-5
F		14-6	4-6	12-6	12-6
G		12-7	4-7	12-7	12-7
H		13-0	4-8	12-8	12-8
I		15-1	4-9	12-9	12-9
J		10-1	4-10	11-1	11-1
K		10-2	4-11	11-2	11-2
L		8-3	4-12	11-3	11-3
M		10-4	4-13	11-4	11-4
N		8-5	4-14	11-5	11-5
O		8-6	4-15	11-6	11-6
P		10-7	5-0	11-7	11-7
Q		11-0	5-1	11-8	11-8
R		9-1	5-2	11-9	11-9
S		6-2	5-3	0-2	0-2
T		4-3	5-4	0-3	0-3
U		6-4	5-5	0-4	0-4
V		4-5	5-6	0-5	0-5
W		4-6	5-7	0-6	0-6
X		6-7	5-8	0-7	0-7
Y		7-0	5-9	0-8	0-8
Z		5-1	5-10	0-9	0-9
a	L.C.	12-1	6-1		12-0-1
b		12-2	6-2		12-0-2
c		14-3	6-3		12-0-3
d		12-4	6-4		12-0-4
e		14-5	6-5		12-0-5
f		14-6	6-6		12-0-6
g		12-7	6-7		12-0-7
h		13-0	6-8		12-0-8
i		15-1	6-9		12-0-9
j		10-1	6-10		12-11-1
k		10-2	6-11		12-11-2
l		8-3	6-12		12-11-3
m		10-4	6-13		12-11-4
n		8-5	6-14		12-11-5
o		8-6	6-15		12-11-6
p		10-7	7-0		12-11-7
q		11-0	7-1		12-11-8
r		9-1	7-2		12-11-9
s		6-2	7-3		11-0-2
t		4-3	7-4		11-0-3
u		6-4	7-5		11-0-4
v		4-5	7-6		11-0-5
w		4-6	7-7		11-0-6
x		6-7	7-8		11-0-7
y		7-0	7-9		11-0-8
z		5-1	7-10		11-0-9

Enkele karaktersets (vervolg)

Schrijfmachine		X8-ponsband	7 bits ISO	BCD	EBCDIC
0	1.c.	4-0	3-0	0	0
1		0-1	3-1	1	1
2		0-2	3-2	2	2
3		2-3	3-3	3	3
4		0-4	3-4	4	4
5		2-5	3-5	5	5
6		2-6	3-6	6	6
7		0-7	3-7	7	7
8		1-0	3-8	8	8
9		3-1	3-9	9	9
+		14-0	2-11	12	12-6-8
-		8-0	2-13	11	11
.		13-3	2-14	12-3-8	12-3-8
,		11-3	2-12	0-3-8	0-3-8
<		6-1	3-12	12-6-8	12-4-8
10		7-3			
¯		1-6	5-15	0-5-8	0-5-8
^	u.c.	4-0	5-14		
v		0-1			
*		0-2	2-10	11-4-8	11-4-8
/		2-3	2-15	0-1	0-1
=		0-4	3-13	3-8	6-8
;		2-5	3-11	11-6-8	11-6-8
[		2-6	5-11	12-5-8	12-2-8
]		0-7	5-13	11-5-8	11-2-8
(		1-0	2-8	0-4-8	12-5-8
)		3-1	2-9	12-4-8	11-5-8
"		14-0	2-2		7-8
:		13-3	3-10	5-8	2-8
¬		8-0	7-12	11-7-8	11-7-8
?		11-3	3-15	12-0	0-7-8
>		6-1	3-14	6-8	0-6-8
'		7-3	2-7	4-8	5-8
|		1-6	7-14	12-7-8	12-7-8
$			2-4	11-3-8	11-3-8
&			2-6	12	12
1.c.		15-2			
u.c.		15-4			
erase		15-7			
car. return		3-2			
space		2-0			
stopcode		1-3			
blank					
#					3-8
@					4-8

Opmerking. X8 en ISO codes: vervang de decimale karakters door binaire; 0 = geen gat, 1 = gat, BCD en EBCDIC kaartcodes: in de genoemde rijen (12, 11, 0 t/m 9) een gat. Voor sommige karakters is geen standaardisatie tot stand gekomen.

Informatiedragers

In een informatiedrager wordt informatie in de vorm van bits vastgelegd, vrijwel altijd in magnetische vorm, zoals

- in kernengeheugens door magnetisering van ferriet-ringetjes. De inschrijf- en uitleestijd is niet afhankelijk van het adres van het geselecteerde woord en meestal van de orde van een mms (= 10^{-6} sec) per woord (van ongeveer 32 bits). De kosten zijn ongeveer een dubbeltje per bit, de capaciteit gaat tot ongeveer 10^{7} bits.
- in trommelgeheugens door magnetisering van plekjes op een aantal paralelle "sporen" op het cylinder oppervlak van een met constante snelheid draaiende trommel. Ieder spoor heeft zijn eigen vaste "lees/schrijfkop", zodat de leestijd van een bepaalde hoeveelheid informatie op een spoor in de eerste plaats bepaald wordt door de rotatietijd van de trommel (van de orde van ms (= 10^{-3} sec)) plus de electronische leestijd (van de orde van mms). De kosten zijn ongeveer 1 dubbeltje per 25 bits, de capaciteit gaat tot ca. 10^{9} bits.
- in schijfgeheugens door magnetisering van plekjes op een aantal concentrische sporen op iets dat op een aantal boven elkaar geplaatste grammofoonplaten lijkt. Omdat er voor alle sporen op één plaat samen meestal maar één, beweegbare lees/schrijfkop is, bestaat de leestijd behalve uit de tijden genoemd bij trommelgeheugens ook nog uit de "opzoektijd" van het spoor, die als regel van de orde van tientallen ms is. Schijfgeheugens zijn veelal verwisselbaar; de kosten zijn ongeveer 1 dubbeltje per 500 bits, de capaciteiten per schijfpakket is van de orde van 10^{9} bits. (Opgemerkt moet worden dat sommige trommelgeheugens het karakter van schijfgeheugens hebben en omgekeerd sommige schijfgeheugens het karakter van trommelgeheugens; dit wordt door het vast of beweegbaar zijn van de lees/schrijfkop bepaald!)
- in magneetstripgeheugens eveneens door magnetisatie van plekjes op stukjes magneetband vastgeplakt op een soepele drager die op het cylinderoppervlak van een trommeltje gespannen kan worden; al is het mechanisme dus anders dan bij schijfgeheugens toch is de karakteristiek van het lezen van een hoeveelheid informatie dezelfde, zij het ongeveer 10 maal langzamer;
- op magneetbanden eveneens door plaatselijke magnetisatie; voor het lezen of schrijven van informatie moet echter de magneetband in beweging worden gebracht (orde enkele ms) en onder de lees/schrijfkop worden getrokken

voor het eigenlijke informatietransport (van de orde van tientallen mms) waarna de band weer tot stilstand komt. Blokken informatie zijn door "gaps" van elkaar gescheiden; de grootte van de gaps is niet constant! Om een bepaalde hoeveelheid informatie te bereiken moeten alle daarvoor staande stukken informatie gelezen worden, zodat het gebruik van magneetbanden alleen efficiënt is bij een "sequentiële" informatie organisatie (bij de drie voorgaande media is "random"access mogelijk).

Bij de moderne machines is alleen informatie in een kernengeheugen direct toegankelijk voor de centrale verwerkingseenheid; informatie opgeslagen in andere media moet eerst naar het kernengeheugen getransporteerd worden, zodat deze andere geheugens achtergrond of secundaire geheugens genoemd worden.

Strikt genomen zijn ook ponskaarten, ponsbanden, printers en t.v. schermen informatiedragers; zij onderscheiden zich van de vorige media doordat zij slechts éénmaal met informatie "gevuld" kunnen worden resp. het informatietransport slechts in één richting kan gaan. Bij de andere informatiedragers "verdwijnt" oude informatie door het overschrijven met nieuwe informatie (zoals bij een recorderband) en in vele kernengeheugens zelfs door het afschakelen van de machine ("volatile memory").

De koppeling van secundaire informatiedragers verloopt als regel via zgn. besturingseenheden ("control units", waarin o.a. codeconversies, tijdelijke informatie opslag, controle's op juiste werking, synchronisatie en hulpbewerkingen als starten van een band worden geregeld) en kanalen ("channels", waardoor informatie uit de besturingseenheden komend naar plaatsen in het kernengeheugen wordt getransporteerd). De individuele informatiedragers hebben om kosten te sparen meestal niet hun eigen kanaal naar het kernengeheugen. Kanalen zijn duur en, indien individueel gebruikt, slecht bezet omdat de zoektijden 10 tot 500 ms zijn tegenover informatie transporttijden van de orde van mms. Om informatie uit een secundair geheugen te krijgen wordt daarom als regel eerst een "voorbericht" uit het kernengeheugen over een beschikbaar kanaal naar de betreffende besturingseenheid gestuurd, waarna het kanaal weer vrijgegeven wordt. Is de gewenste geheugeneenheid dan klaar om informatie te ontvangen of te zenden, dan wordt opnieuw de hulp van het kanaal ingeroepen voor het eigenlijke informatietransport. De "kanaaluiteinden" in het kernengeheugen kunnen bij snelle machines vrijwel tegelijkertijd toegang tot het kernengeheugen krijgen, zodat informatietransporten

zelfs simultaan met informatiemanipulaties in het kernengeheugen kunnen plaatsvinden. We spreken dan van parallel verlopende processen (die uiteraard gecoördineerd moeten worden om chaos te voorkomen).

Het samenstel van kernengeheugen, secundaire geheugens, kanalen en invoer/uitvoer-eenheden wordt computer "configuratie" genoemd. Deze roept associaties op met telefooncentrale-systemen en de daarmee samenhangende wachttijdproblemen, die inderdaad een belangrijke rol kunnen spelen!

Hulpapparatuur

Hulpapparaten worden gebruikt om buiten de rekenautomaat ("off-line") informatie vast te leggen in informatiedragers of van de ene informatiedrager over te brengen naar een andere of om bewerkingen met sommige informatiedragers uit te voeren. Zonder verder op details in te gaan kunnen de volgende hulpapparaten genoemd worden:

ponsapparaten : op een toetsenbord ingeslagen informatie wordt gecodeerd vastgelegd in ponskaart of ponsband; soms tegelijk met een leesbare vastlegging op papier of kaart (interpreter);

controle ponsapparaten: op een toetsenbord ingeslagen informatie wordt vergeleken met de informatie in een tegelijkertijd doorgevoerde reeds geponste kaart; bij discrepantie wordt bijv. als waarschuwing het toetsenbord geblokkeerd;

magneetbandschrijvers : als de twee vorige apparaten, maar informatie wordt/is vastgelegd op een magneetband (sneller en geruisloos);

reproducers : voor het maken van copieën van ponskaarten of ponsbanden;

converters : van ponsband naar ponskaart of magneetband en viceversa;

verifiers : voor het vergelijken van ponsbanden;

collators : voor het vergelijken of samenvoegen of scheiden van ponskaarten;

sorteerapparaten : voor het sorteren van ponskaarten op grond van daarin geponste informatie;

tabulators : voor het aflijsten op papier van informatie in ponskaarten.

Omdat deze apparaten betrekkelijk langzaam werken en bedienend personeel vragen is de tendens om de functies van reproducers t/m tabulators niet meer op de afzonderlijke (relatief goedkope) apparaten uit te voeren maar in multi-programmering met een rekenautomaat.

Grafiekenschrijvers ("plotters") kunnen zowel "off-line" als "on-line" (direct aan een automaat gekoppeld) gebruikt worden.

Getalrepresentaties

Positionele notatie

Het abstracte begrip getal is niet afhankelijk van de representatie of voorstelling op papier, in een ponskaart of in een computergeheugen, daarbij gebruik makend van een tientallig, tweetallig of r-tallig stelsel. Voor positieve gehele getallen in een r-tallig talstelsel spreken we de positionele notatie af, d.w.z. dat

$$a_{n-1}\ a_{n-2} \cdots a_1\ a_0 \quad \text{met } 0 \leq a_k \leq r-1 \text{ en } a_k \text{ geheel}$$

voorstelt het getal

$$N_r = \sum_{k=0}^{n-1} a_k \cdot r^k$$

waarin de a_k de cijfersymbolen in het betreffende talstelsel zijn (hiervoor schrijft men als regel de vertrouwde decimale cijfersymbolen, zo nodig gevolgd door de letters van het alfabet).

Voor de representatie in een computergeheugen komt vooral het binaire talstelsel in aanmerking, doch voor menselijk gebruik is dit minder plezierig door de lengte van de getallen. Als tussenweg gebruikt men daarom vaak een 8-tallig stelsel omdat een getal in het 8-tallig stelsel direct om te schrijven is tot een getal in het 2-tallig stelsel door ieder van de cijfersymbolen in het 8-tallig stelsel (dus 0 t/m 7) te vervangen door hun binaire equivalenten (dus 000, 001, 010, 011, 100, 101, 110, 111).

Omrekening van een integer in het r-tallig stelsel naar het s-tallig stelsel maakt gebruik van de gelijkheid

$$\sum_{k=0}^{n-1} a_k r^k = N_r \equiv N_s = \sum_{j=0}^{m-1} b_j s^j$$

waaruit door deling door r volgt

$$N_{r/r} = \sum_{k=1}^{n-1} a_k r^{k-1} + a_{0/r} = \sum_{j=0}^{m-1} b_j s^j/r$$

zodat a_0 de rest is van het getal N na deling door r, in welk talstelsel wij deze deling ook uitvoeren. Door deze deling in het s-tallig stelsel te herhalen vinden we van achter naar voren gaande de cijfers van N in het r-tallig stelsel als het in het s-tallig stelsel gegeven is.

Voorbeeld 1: Wat is $(149)_{10}$ in het 8-tallig stelsel?

149/8 = 18 rest 5
18/8 = 2 rest 2
2/8 = 0 rest 2

dus

$$(149)_{10} = (2\ 2\ 5)_8 = (010\ 010\ 101)_2.$$

Voorbeeld 2: Wat is $(123)_5$ in het 10-tallig stelsel ($10 = (20)_5$) ?

123/20 = 3 rest $(13)_5$ $\quad (13)_5 = (5+3)_{10} = 8$
3/20 = 0 rest 3

dus

$$(123)_5 = 38\ .$$

Opgave: Hoe verloopt de omrekening van fracties $\sum_{k=-n}^{-1} a_k r^k$?

Effect van eindig aantal posities; positieve en negatieve integers

Het grootste getal dat we met n posities voor een r-tallig stelsel kunnen voorstellen is

$$\sum_{k=0}^{n-1} (r-1)r^k = r^n - 1\ .$$

Immers tellen we hier 1 bij op dan is het resultaat met n posities nul omdat de carry verloren gaat. M.a.w. met n posities kunnen we slechts de getallen

mod (r^n) voorstellen, hetgeen aanschouwelijk is te maken door de getallen $0,\ldots,r^n-1$ op een cirkel af te beelden.

Voor de representatie van positieve en negatieve integers zijn verschillende systemen bedacht, waarvan vier in de praktijk vaak voorkomen. De meest bekende vorm is de absolute waarde en teken representatie, zoals in het decimale systeem gebruikelijk. Bij een 2-tallig stelsel wordt de eerste bit gebruikt om het teken voor te stellen (0 voor positieve getallen, 1 voor negatieve), de overige n - 1 bits vormen de absolute waarde van het getal, zodat integers in het gebied $-(2^{n-1}-1)$ tot $(2^{n-1}-1)$ voorgesteld kunnen worden. Omdat 0 zowel als +0 als als -0 kan voorkomen is het oppassen geblazen bij het vergelijken van 2 getallen of het testen op 0. Hier geldt

$$N = (1 - 2a_{n-1}) \sum_{k=0}^{n-2} a_k \, 2^k \; .$$

Een tweede systeem is het r-complement systeem. Hierbij worden de positieve getallen in het gebied $0 \leq \text{getal} \leq \frac{1}{2}r^n - 1$ direct positioneel gerepresenteerd, maar bij de negatieve getallen in het gebied $-\frac{1}{2}r^n \leq \text{getal} < 0$ wordt eerst r^n opgeteld alvorens ze positioneel te representeren (voor een 10-tallig stelsel met n = 2 stellen dus 0,...,49 de corresponderende positieve getallen voor en 50,...,99 de negatieve getallen -50,...,-1). Het voordeel van dit systeem is dat er slechts één nul voorkomt, een klein nadeel dat de gebieden van positieve en negatieve getallen niet precies even groot zijn ($-\frac{1}{2}r^n$ kan wel worden gerepresenteerd, maar $+\frac{1}{2}r^n$ niet).

Een derde systeem is het (r - 1) complement systeem. Hierbij worden de positieve getallen in het gebied $0 \leq \text{getal} \leq \frac{1}{2}r^n - 1$ direct positioneel gerepresenteerd, maar bij de negatieve getallen in het gebied $-(\frac{1}{2}r^n - 1) \leq \text{getal} < 0$ wordt eerst $(r^n - 1)$ opgeteld alvorens ze positioneel te representeren (voor een 10-tallig stelsel met n = 2 stellen dus 0,...,49 de corresponderende positieve getallen voor en 50,...,99 de negatieve getallen -49,...,-0). De gebieden van positieve en negatieve getallen zijn nu weer even groot, doch we moeten wederom rekening houden met het voorkomen van +0 en -0.

Een vierde systeem is tenslotte het excess r^n systeem, waarbij bij getallen in het gebied $-\frac{1}{2}r^n \leq \text{getal} \leq \frac{1}{2}r^n - 1$ eerst $\frac{1}{2}r^n$ wordt opgeteld alvorens ze direct positioneel af te beelden (voor een 10-tallig stelsel met n = 2

stellen 0,...,49 de negatieve getallen -50,...,-1 voor en de getallen 50,...,99 de positieve getallen 0,...,+49). Als bij het r-complement systeem is er dus maar één nul.

Opgave: Toon aan dat voor het 2-tallig stelsel en n bits

a) in het 2-complement systeem $N = \sum_{k=0}^{n-2} (a_k - a_{n-1})2^k - a_{n-1}$;

b) in het 2-complement systeem een negatief getal uit zijn absolute waarde verkregen wordt door alle bits te inverteren en dan 1 op te tellen bij de minst significante bit;

c) in het 1-complement systeem $N = \sum_{k=0}^{n-2} (a_k - a_{n-1})2^k$;

d) in het 1-complement systeem een negatief getal uit zijn absolute waarde verkregen wordt door alle bits te inverteren (0 → 1, 1 → 0);

e) in het excess 2^n systeem het bitpatroon met uitzondering van de eerste bit identiek is met dat van het 2-complement systeem.

Positieve en negatieve reals

Tot dusver hebben we het alleen gehad over "fixed-point" integers of fracties waarbij de positie van de radix-punt wel voor vermenigvuldigingen van belang is, maar niet voor optellingen of aftrekkingen. Reals worden vrijwel altijd in het computergeheugen voorgesteld in "floating-point" (drijvende komma) vorm, dus zoals in het r-tallig stelsel:

$$\text{integer}_{,}\, r^{\text{exponent}} \quad \text{of} \quad \text{fractie}_{,}\, r^{\text{exponent}}$$

(ten onrechte wordt i.p.v. fractie ook wel het woord mantisse gebruikt). Als regel worden hierbij de volgende afspraken gehanteerd:

- r = 2 of 8 of 16 (afhankelijk van het machinetype);
- $1/r \leq |\text{fractie}| < 1$ (het floating-point getal heet dan genormaliseerd);
- voor de representatie van de exponent wordt meestal de excess r^n vorm gebruikt, voor de fractie (of integer) een van de andere drie vormen;
- fractie (of integer) worden met de exponent als regel in één computerwoord "samengepakt".

De aanleiding tot normalisatie is dat men zoveel mogelijk van de aanwezige bits ter beschikking wil stellen om de fractie voor te stellen (de relatieve fout in de fractie is dan $\frac{1}{2}\times$ de laatste bitwaarde), en om de absolute fout zo klein mogelijk te maken de exponent zo klein mogelijk houdt. Men moet echter wel bedenken dat normalisatie anderzijds in berekeningen optredende onnauwkeurigheden "verduistert", zoals het volgende voorbeeld laat zien:

$$0.1234 \times 10^5 - 0.1233 \times 10^5 = 0.0001 \times 10^5 = 0.1000 \times 10^1 .$$

De einduitkomst na normalisatie suggereert ten onrechte een nauwkeurigheid van 4 cijfers!

Dat men niet altijd 2 als basis kiest, maar bijv. ook wel eens 16, hangt samen met het gewenste bereik van de reals. Zijn in totaal bijv. 32 bits beschikbaar, dan is 24 + 8 een mogelijke verdeling voor fractie en exponent. Rekening houdend met de tekenbit is dan $2^7 - 1 = 127$ de grootst mogelijke exponent. Was 2 de basis dan is de grens van het bereik van de reals 2^{127}; was daarentegen 16 de basis dan is het bereik $2^{4\times127}$. We moeten hier echter wel meteen bij bedenken dat bij een basis van 16 door normalisatie de fractie 1/16 (binair .0001) zou kunnen zijn zodat dan de eerste drie posities van de 24 bestemd voor het fractiedeel in feite verloren zijn voor de representatie van de fractie. Vergroting van het bereik gaat hier dus ten koste van de relatieve nauwkeurigheid.

Bij een berekening kan de exponent van het resultaat wel eens te groot ("overflow") of te klein ("underflow") zijn om gerepresenteerd te kunnen worden. Dit hoort door de hardware gesignaleerd te worden. Bij fixed-point getallen kan natuurlijk ook overflow optreden als we buiten het bereik komen.

Nauwkeurigheid van berekeningen

Door afrondingen en door verlies van cijfers kunnen bij berekeningen met vaste woordlengten de resultaten onnnauwkeurig zijn, zoals in de numerieke wiskunde verder onderzocht wordt. Volg voorlopig de volgende vuistregels bij het programmeren (evt. door expressies te herarrangeren):

- stel delingen zo lang mogelijk uit, vooral als de deler van de vorm (a - b) is; stel ook vermenigvuldigingen zo lang mogelijk uit;
- moeten positieve en negatieve getallen bij elkaar opgeteld worden, tel dan eerst de positieve getallen bij elkaar op, dan de negatieve en dan de twee tussenresultaten;

- moet een reeks in grootte afnemende getallen (alle zelfde teken) bij elkaar opgeteld worden, begin dan de optellingen met het kleinste getal.

De achtergrond voor deze vuistregels is dat bij werken met een vast aantal cijfers verschillende eigenschappen, waar we anders nauwelijk bij nadenken, niet gelden bij het rekenen met floating-point getallen. Bijvoorbeeld:

associatieve optelling:	(1004. - 1003.) + 1.500 = 2.500
	1004. + (- 1003. + 1.500) = 2.000
associatieve vermenigvuldiging:	(2.001 × 5.009) × 8.008 = 80.24
	2.001 × (5.009 × 8.008) = 80.26
distributieve eigenschap:	(4000. × .9009) - (4000. × .9000) = 4.000
	4000. × (.9009 - .9000) = 3.600.

Dat sommige compilers ten behoeve van een sneller rekenproces de arithmetische expressies van een programma in feite omvormen is daarom onaanvaardbaar omdat de programmeur heel vaak een arithmetische expressie bewust op een bepaalde manier neerschrijft!

Summary

The report gives a complete defining description of the international algorithmic language ALGOL 60. This is a language suitable for expressing a large class of numerical processes in a form sufficiently concise for direct automatic translation into the language of programmed automatic computers.

The introduction contains an account of the preparatory work leading up to the final conference, where the language was defined. In addition the notions reference language, publication language, and hardware representations are explained.

In the first chapter a survey of the basic constituents and features of the language is given, and the formal notation, by which the syntactic structure is defined, is explained.

The second chapter lists all the basic symbols, and the syntactic units known as identifiers, numbers, and strings are defined. Further some important notions such as quantity and value are defined.

The third chapter explains the rules for forming expressions and the meaning of these expressions. Three different types of expressions exist: arithmetic, Boolean (logical), and designational.

The fourth chapter describes the operational units of the language, known as statements. The basic statements are: assignment statements (evaluation of a formula), go to statements (explicit break of the sequence of execution of statements), dummy statements, and procedure statements (call for execution of a closed process, defined by a procedure declaration). The formation of more complex structures, having statement character, is explained. These include: conditional statements, for statements, compound statements, and blocks.

In the fifth chapter the units known as declarations, serving for defining permanent properties of the units entering into a process described in the language, are defined.

The report ends with two detailed examples of the use of the language and an alphabetic index of definitions.

Introduction

Background

After the publication[1,2] of a preliminary report on the algorithmic language ALGOL, as prepared at a conference in Zürich in 1958, much interest in the ALGOL language developed.

As a result of an informal meeting held at Mainz in November 1958, about forty interested persons from several European countries held an ALGOL implementation conference in Copenhagen in February 1959. A "hardware group" was formed for working cooperatively right down to the level of the paper tape code. This conference also led to the publication by Regnecentralen, Copenhagen, of an *Algol Bulletin*, edited by PETER NAUR, which served as a forum for further discussion. During the June 1959 ICIP Conference in Paris several meetings, both formal and informal ones, were held. These meetings revealed some misunderstandings as to the intent of the group which was primarily responsible for the formulation of the language, but at the same time made it clear that there exists a wide appreciation of the effort involved. As a result of the discussions it was decided to hold an international meeting in January 1960 for improving the ALGOL language and preparing a final report. At a European ALGOL Conference in Paris in November 1959 which was attended by about fifty people, seven European representatives were selected to attend the January 1960 Conference, and they represent the following organizations: Association Française de Calcul, British Computer Society, Gesellschaft für Angewandte Mathematik und Mechanik, and Nederlands Rekenmachine Genootschap. The seven representatives held a final preparatory meeting at Mainz in December 1959.

Meanwhile, in the United States, anyone who wished to suggest changes or corrections to ALGOL was requested to send his comments to the *Communications of the ACM*, where they were published. These comments then became the basis of consideration for changes in the ALGOL language. Both the SHARE and USE organizations established ALGOL working groups, and both organizations were represented on the ACM Committee on Programming Languages. The ACM Committee met in Washington in November 1959 and considered all comments on ALGOL that had been sent to the ACM *Communications*. Also, seven representatives were selected to attend the January 1960 international conference. These seven representatives held a final preparatory meeting in Boston in December 1959.

January 1960 Conference

The thirteen representatives[3], from Denmark, England, France, Germany, Holland, Switzerland, and the United States, conferred in Paris from January 11 to 16, 1960.

---

[1] Preliminary report — International Algebraic Language, Comm. Assoc. Comp. Mach. 1, No. 12 (1958), 8.

[2] Report on the Algorithmic Language ALGOL by the ACM Committee on Programming Languages and the GAMM Committee on Programming, edited by A. J. PERLIS and K. SAMELSON, Numerische Mathematik Bd. 1, S. 41—60 (1959).

[3] WILLIAM TURANSKI of the American group was killed by an automobile just prior to the January 1960 Conference.

Prior to this meeting a completely new draft report was worked out from the preliminary report and the recommendations of the preparatory meetings by Peter Naur and the Conference adopted this new form as the basis for its report. The Conference then proceeded to work for agreement on each item of the report. The present report represents the union of the Committee's concepts and the intersection of its agreements.

April 1962 Conference [Edited by M. Woodger]

A meeting of some of the authors of Algol 60 was held on 2nd–3rd April 1962 in Rome, Italy, through the facilities and courtesy of the International Computation Centre. The following were present:

Authors	*Advisers*	*Observer*
F. L. Bauer	M. Paul	W. L. van der Poel
J. Green	R. Franciotti	(Chairman, IFIP TC 2.1
C. Katz	P. Z. Ingerman	Working Group Algol)
R. Kogon (representing J. W. Backus)		
P. Naur		
K. Samelson	G. Seegmüller	
J. H. Wegstein	R. E. Utman	
A. van Wijngaarden		
M. Woodger	P. Landin	

The purpose of the meeting was to correct known errors in, attempt to eliminate apparent ambiguities in, and otherwise clarify the Algol 60 Report. Extensions to the language were not considered at the meeting. Various proposals for correction and clarification that were submitted by interested parties in response to the Questionnaire in *Algol Bulletin* No. 14 were used as a guide.

This report[1] constitutes a supplement to the Algol 60 Report which should resolve a number of difficulties therein. Not all of the questions raised concerning the original report could be resolved. Rather than risk hastily drawn conclusions on a number of subtle points, which might create new ambiguities, the committee decided to report only those points which they unanimously felt could be stated in clear and unambiguous fashion.

Questions concerned with the following areas are left for further consideration by Working Group 2.1 of IFIP, in the expectation that current work on advanced programming languages will lead to better resolution:

1. Side effects of functions.
2. The call by name concept.

[1] [Editor's note: — The present edition follows the text which was approved by the Council of IFIP. Although it is not clear from the Introduction, the present version is the original report of the January 1960 conference modified according to the agreements reached during the April 1962 conference. Thus the report mentioned here is incorporated in the present version. The modifications touch the original report in the following sections: Changes of text: 1 with footnote; 2.1 footnote; 2.3; 2.7; 3.3.3; 3.3.4.2; 4.1.3; 4.2.3; 4.2.4; 4.3.4; 4.7.3; 4.7.3.1; 4.7.3.3; 4.7.5.1; 4.7.5.4; 4.7.6; 5; 5.3.3; 5.3.5; 5.4.3; 5.4.4; 5.4.5. Changes of syntax: 3.4.1; 4.1.1; 4.2.1; 4.5.1.]

3. **own**: static or dynamic.

4. For statement: static or dynamic.

5. Conflict between specification and declaration.

The authors of the ALGOL 60 Report present at the Rome Conference, being aware of the formation of a Working Group on ALGOL by IFIP, accepted that any collective responsibility which they might have with respect to the development, specification, and refinement of the ALGOL language will from now on be transferred to that body.

This report has been reviewed by IFIP TC 2 on Programming Languages in August 1962 and has been approved by the Council of the International Federation for Information Processing.

As with the preliminary ALGOL report, three different levels of language are recognized, namely a Reference Language, a Publication Language, and several Hardware Representations.

Reference Language

1. It is the working language of the committee.

2. It is the defining language.

3. The characters are determined by ease of mutual understanding and not by any computer limitations, coders notation, or pure mathematical notation.

4. It is the basic reference and guide for compiler builders.

5. It is the guide for all hardware representations.

6. It is the guide for transliterating from publication language to any locally appropriate hardware representations.

7. The main publications of the ALGOL language itself will use the reference representation.

Publication Language

1. The publication language admits variations of the reference language according to usage of printing and handwriting (e.g., subscripts, spaces, exponents, Greek letters).

2. It is used for stating and communicating processes.

3. The characters to be used may be different in different countries, but univocal correspondence with reference representation must be secured.

Hardware Representations

1. Each one of these is a condensation of the reference language enforced by the limited number of characters on standard input equipment.

2. Each one of these uses the character set of a particular computer and is the language accepted by a translator for that computer.

3. Each one of these must be accompanied by a special set of rules for transliterating from publication or reference language.

For transliteration between the reference language and a language suitable for publications, among others, the following rules are recommended.

Reference language	*Publication language*
Subscript brackets []	Lowering of the line between the brackets and removal of the brackets.
Exponentiation ↑	Raising of the exponent.
Parentheses ()	Any form of parentheses, brackets, braces.
Basis of ten $_{10}$	Raising of the ten and of the following integral number, inserting of the intended multiplication sign.

Description of the reference language

Was sich überhaupt sagen läßt, läßt sich klar sagen; und wovon man nicht reden kann, darüber muß man schweigen.
Ludwig Wittgenstein

1. Structure of the language

As stated in the introduction, the algorithmic language has three different kinds of representations—reference, hardware, and publication—and the development described in the sequel is in terms of the reference representation. This means that all objects defined within the language are represented by a given set of symbols—and it is only in the choice of symbols that the other two representations may differ. Structure and content must be the same for all representations.

The purpose of the algorithmic language is to describe computational processes. The basic concept used for the description of calculating rules is the well known arithmetic expression containing as constituents numbers, variables, and functions. From such expressions are compounded, by applying rules of arithmetic composition, self-contained units of the language—explicit formulae—called assignment statements.

To show the flow of computational processes, certain non-arithmetic statements and statement clauses are added which may describe e.g., alternatives, or iterative repetitions of computing statements. Since it is necessary for the function of these statements that one statement refers to another, statements may be provided with labels. A sequence of statements may be enclosed between the statement brackets **begin** and **end** to form a compound statement.

Statements are supported by declarations which are not themselves computing instructions, but inform the translator of the existence and certain properties of objects appearing in statements, such as the class of numbers taken on as values by a variable, the dimension of an array of numbers, or even the set of rules defining a function. A sequence of declarations followed by a sequence of statements and enclosed between **begin** and **end** constitutes a block. Every declaration appears in a block in this way and is valid only for that block.

A program is a block or compound statement which is not contained within another statement and which makes no use of other statements not contained within it.

In the sequel the syntax and semantics of the language will be given[1].

1.1. Formalism for syntactic description

The syntax will be described with the aid of metalinguistic formulae[2]. Their interpretation is best explained by an example:

⟨ab⟩ ::= (| [| ⟨ab⟩(| ⟨ab⟩ ⟨d⟩

Sequences of characters enclosed in the bracket ⟨⟩ represent metalinguistic variables whose values are sequences of symbols. The marks ::= and | (the latter with the meaning of **or**) are metalinguistic connectives. Any mark in a formula, which is not a variable or a connective, denotes itself (or the class of marks which are similar to it). Juxtaposition of marks and/or variables in a formula signifies juxtaposition of the sequences denoted. Thus the formula above gives a recursive rule for the formation of values of the variable ⟨ab⟩. It indicates that ⟨ab⟩ may have the value (or [or that given some legitimate value of ⟨ab⟩, another may be formed by following it with the character (or by following it with some value of the variable ⟨d⟩. If the values of ⟨d⟩ are the decimal digits, some values of ⟨ab⟩ are:

[((*1*(*37*(
(*12345*(
(((
[*86*

In order to facilitate the study, the symbols used for distinguishing the metalinguistic variables (i.e. the sequences of characters appearing within the brackets ⟨⟩ as ab in the above example) have been chosen to be words describing approximately the nature of the corresponding variable. Where words which have appeared in this manner are used elsewhere in the text they will refer to the corresponding syntactic definition. In addition some formulae have been given in more than one place.

Definition:

⟨empty⟩::=
(i.e. the null string of symbols).

2. Basic symbols, identifiers, numbers, and strings. Basic concepts

The reference language is built up from the following basic symbols:

⟨basic symbol⟩::=⟨letter⟩|⟨digit⟩|⟨logical value⟩|⟨delimiter⟩

2.1. Letters

⟨letter⟩::= *a*|*b*|*c*|*d*|*e*|*f*|*g*|*h*|*i*|*j*|*k*|*l*|*m*|*n*|*o*|*p*|*q*|*r*|*s*|*t*|*u*|*v*|*w*|*x*|*y*|*z*|
A|*B*|*C*|*D*|*E*|*F*|*G*|*H*|*I*|*J*|*K*|*L*|*M*|*N*|*O*|*P*|*Q*|*R*|*S*|*T*|*U*|*V*|*W*|*X*|*Y*|*Z*

[1] Whenever the precision of arithmetic is stated as being in general not specified, or the outcome of a certain process is left undefined or said to be undefined, this is to be interpreted in the sense that a program only fully defines a computational process if the accompanying information specifies the precision assumed, the kind of arithmetic assumed, and the course of action to be taken in all such cases as may occur during the execution of the computation.

[2] Cf. J. W. BACKUS, The syntax and semantics of the proposed international algebraic language of the Zürich ACM-GAMM conference. ICIP Paris, June 1959.

This alphabet may arbitrarily be restricted, or extended with any other distinctive character (i.e. character not coinciding with any digit, logical value or delimiter).

Letters do not have individual meaning. They are used for forming identifiers and strings[1] (cf. sections 2.4. Identifiers, 2.6. Strings).

2.2.1. Digits

⟨digit⟩::= *0*|*1*|*2*|*3*|*4*|*5*|*6*|*7*|*8*|*9*

Digits are used for forming numbers, identifiers, and strings.

2.2.2. Logical values

⟨logical value⟩::= **true**|**false**

The logical values have a fixed obvious meaning.

2.3. Delimiters

⟨delimiter⟩::= ⟨operator⟩|⟨separator⟩|⟨bracket⟩|⟨declarator⟩|⟨specificator⟩
⟨operator⟩::= ⟨arithmetic operator⟩|⟨relational operator⟩|⟨logical operator⟩|
⟨sequential operator⟩
⟨arithmetic operator⟩::= +|−|×|/|÷|↑
⟨relational operator⟩::= <|≦|=|≧|>|≠
⟨logical operator⟩::= ≡|⊃|∨|∧|¬
⟨sequential operator⟩::= **go to**|**if**|**then**|**else**|**for**|**do**[2]
⟨separator⟩::= ,|.|$_{10}$|:|;|:=|␣|**step**|**until**|**while**|**comment**
⟨bracket⟩::= (|)|[|]|'|'|**begin**|**end**
⟨declarator⟩::= **own**|**Boolean**|**integer**|**real**|**array**|**switch**|**procedure**
⟨specificator⟩::= **string**|**label**|**value**

Delimiters have a fixed meaning which for the most part is obvious or else will be given at the appropriate place in the sequel.

Typographical features such as blank space or change to a new line have no significance in the reference language. They may, however, be used freely for facilitating reading.

For the purpose of including text among the symbols of a program the following "comment" conventions hold:

The sequence of basic symbols:	is equivalent to
; **comment** ⟨any sequence not containing ;⟩;	;
begin comment ⟨any sequence not containing ;⟩;	**begin**
end ⟨any sequence not containing **end** or ; or **else**⟩	**end**

By equivalence is here meant that any of the three structures shown in the left hand column may be replaced, in any occurrence outside of strings, by the symbol shown on the same line in the right hand column without any effect on

[1] It should be particularly noted that throughout the reference language underlining [in typewritten copy; boldface type in printed copy — Ed.] is used for defining independent basic symbols (see sections 2.2.2 and 2.3). These are understood to have no relation to the individual letters of which they are composed. Within the present report [not including headings — Ed.] underlining [boldface — Ed.] will be used for no other purposes.

[2] **do** is used in for statements. It has no relation whatsoever to the *do* of the preliminary report, which is not included in Algol 60.

e action of the program. It is further understood that the comment structure ncountered first in the text when reading from left to right has precedence in ing replaced over later structures contained in the sequence.

2.4. Identifiers

2.4.1. Syntax.

⟨identifier⟩ :: = ⟨letter⟩ | ⟨identifier⟩⟨letter⟩ | ⟨identifier⟩⟨digit⟩

2.4.2. Examples

q
Soup
V17a
a34kTMNs
MARILYN

2.4.3. Semantics. Identifiers have no inherent meaning, but serve for the identification of simple variables, arrays, labels, switches, and procedures. They may be chosen freely (cf. however section 3.2.4. Standard functions).

The same identifier cannot be used to denote two different quantities except when these quantities have disjoint scopes as defined by the declarations of the program (cf. section 2.7. Quantities, kinds and scopes and section 5. Declarations).

2.5. Numbers

2.5.1. Syntax.

⟨unsigned integer⟩ :: = ⟨digit⟩ | ⟨unsigned integer⟩⟨digit⟩
⟨integer⟩ :: = ⟨unsigned integer⟩ | + ⟨unsigned integer⟩ | − ⟨unsigned integer⟩
⟨decimal fraction⟩ :: = . ⟨unsigned integer⟩
⟨exponent part⟩ :: = $_{10}$⟨integer⟩
⟨decimal number⟩ :: = ⟨unsigned integer⟩ | ⟨decimal fraction⟩ |
⟨unsigned integer⟩⟨decimal fraction⟩
⟨unsigned number⟩ :: = ⟨decimal number⟩ | ⟨exponent part⟩ |
⟨decimal number⟩⟨exponent part⟩
⟨number⟩ :: = ⟨unsigned number⟩ | + ⟨unsigned number⟩ | − ⟨unsigned number⟩

2.5.2. Examples

0	-200.084	$-.083_{10}-02$
177	$+\ 07.43_{10}8$	$-_{10}7$
.5384	$9.34_{10}+10$	$_{10}-4$
$+0.7300$	$2_{10}-4$	$+_{10}+5$

2.5.3. Semantics. Decimal numbers have their conventional meaning. The exponent part is a scale factor expressed as an integral power of 10.

2.5.4. Types. Integers are of type **integer**. All other numbers are of type **real** (cf. section 5.1 Type declarations).

2.6. Strings

2.6.1. Syntax.

⟨proper string⟩ :: = ⟨any sequence of basic symbols not containing ' or '⟩ |
⟨empty⟩
⟨open string⟩ :: = ⟨proper string⟩ | '⟨open string⟩' | ⟨open string⟩⟨open string⟩
⟨string⟩ :: = '⟨open string⟩'

2.6.2. Examples. *'δk„ — '[[['Λ =/:' Tt''*
'..This␣is␣a␣'string''

2.6.3. Semantics. In order to enable the language to handle arbitrary sequences of basic symbols the string quotes ' and ' are introduced. The symbol ␣ denotes a space. It has no significance outside strings.

Strings are used as actual parameters of procedures (cf. sections 3.2. Function designators and 4.7. Procedure statements).

2.7. Quantities, kinds and scopes

The following kinds of quantities are distinguished: simple variables, arrays, labels, switches, and procedures.

The scope of a quantity is the set of statements and expressions in which the declaration of the identifier associated with that quantity is valid. For labels see section 4.1.3.

2.8. Values and types

A value is an ordered set of numbers (special case: a single number), an ordered set of logical values (special case: a single logical value), or a label.

Certain of the syntactic units are said to possess values. These values will in general change during the execution of the program. The values of expressions and their constituents are defined in section 3. The value of an array identifier is the ordered set of values of the corresponding array of subscripted variables (cf. section 3.1.4.1).

The various "types" (**integer, real, Boolean**) basically denote properties of values. The types associated with syntactic units refer to the values of these units.

3. Expressions

In the language the primary constituents of the programs describing algorithmic processes are arithmetic, Boolean, and designational expressions. Constituents of these expressions, except for certain delimiters, are logical values, numbers, variables, function designators, and elementary arithmetic, relational, logical, and sequential operators. Since the syntactic definition of both variables and function designators contains expressions, the definition of expressions, and their constituents, is necessarily recursive.

⟨expression⟩::=⟨arithmetic expression⟩|⟨Boolean expression⟩|
⟨designational expression⟩

3.1. Variables

3.1.1. Syntax.

⟨variable identifier⟩::=⟨identifier⟩
⟨simple variable⟩::=⟨variable identifier⟩
⟨subscript expression⟩::=⟨arithmetic expression⟩
⟨subscript list⟩::=⟨subscript expression⟩|⟨subscript list⟩, ⟨subscript expression⟩
⟨array identifier⟩::=⟨identifier⟩
⟨subscripted variable⟩::=⟨array identifier⟩ [⟨subscript list⟩]
⟨variable⟩::=⟨simple variable⟩|⟨subscripted variable⟩

3.1.2. Examples. *epsilon*
det A
a 17
Q[7, 2]
x[sin(n × pi/2), Q[3, n, 4]]

3.1.3. Semantics. A variable is a designation given to a single value. This value may be used in expressions for forming other values and may be changed at will by means of assignment statements (section 4.2). The type of the value of a particular variable is defined in the declaration for the variable itself (cf section 5.1. Type declarations) or for the corresponding array identifier (cf. section 5.2. Array declarations).

3.1.4. Subscripts. 3.1.4.1. Subscripted variables designate values which are components of multidimensional arrays (cf. section 5.2. Array declarations). Each arithmetic expression of the subscript list occupies one subscript position of the subscripted variable and is called a subscript. The complete list of subscripts is enclosed in the subscript brackets [ ]. The array component referred to by a subscripted variable is specified by the actual numerical value of its subscripts (cf. section 3.3. Arithmetic expressions).

3.1.4.2. Each subscript position acts like a variable of type **integer** and the evaluation of the subscript is understood to be equivalent to an assignment to this fictitious variable (cf. section 4.2.4). The value of the subscripted variable is defined only if the value of the subscript expression is within the subscript bounds of the array (cf. section 5.2. Array declarations).

3.2. Function designators

3.2.1. Syntax.

⟨procedure identifier⟩ :: = ⟨identifier⟩
⟨actual parameter⟩ :: = ⟨string⟩ | ⟨expression⟩ | ⟨array identifier⟩ |
⟨switch identifier⟩ | ⟨procedure identifier⟩
⟨letter string⟩ :: = ⟨letter⟩ | ⟨letter string⟩ ⟨letter⟩
⟨parameter delimiter⟩ :: = , |) ⟨letter string⟩ : (
⟨actual parameter list⟩ :: = ⟨actual parameter⟩ |
⟨actual parameter list⟩ ⟨parameter delimiter⟩ ⟨actual parameter⟩
⟨actual parameter part⟩ :: = ⟨empty⟩ | (⟨actual parameter list⟩)
⟨function designator⟩ :: = ⟨procedure identifier⟩ ⟨actual parameter part⟩

3.2.2. Examples. *sin(a − b)*
J(v + s, n)
R
S(s − 5) Temperature: (T) Pressure: (P)
Compile (': =') Stack: (Q)

3.2.3. Semantics. Function designators define single numerical or logical values whieh result through the application of given sets of rules defined by a procedure declaration (cf. section 5.4. Procedure declarations) to fixed sets of actual parameters. The rules governing specification of actual parameters are given in

section 4.7. Procedure statements. Not every procedure declaration defines the value of a function designator.

3.2.4. Standard functions. Certain identifiers should be reserved for the standard functions of analysis, which will be expressed as procedures. It is recommended that this reserved list should contain:

abs (E) for the modulus (absolute value) of the value of the expression E
sign (E) for the sign of the value of E (*+1* for E>*0*, *0* for E=*0*, *−1* for E<*0*)
sqrt (E) for the square root of the value of E
sin (E) for the sine of the value of E
cos (E) for the cosine of the value of E
arctan (E) for the principal value of the arctangent of the value of E
ln (E) for the natural logarithm of the value of E
exp (E) for the exponential function of the value of E (e^E)

These functions are all understood to operate indifferently on arguments both of type **real** and **integer**. They will all yield values of type **real**, except for *sign* (E) which will have values of type **integer**. In a particular representation these functions may be available without explicit declarations (cf. section 5. Declarations).

3.2.5. Transfer functions. It is understood that transfer functions between any pair of quantities and expressions may be defined. Among the standard functions it is recommended that there be one, namely

entier (E),

which "transfers" an expression of real type to one of integer type, and assigns to it the value which is the largest integer not greater than the value of E.

3.3. Arithmetic expressions

3.3.1. Syntax.

⟨adding operator⟩::=+|−
⟨multiplying operator⟩::=×|/|÷
⟨primary⟩::=⟨unsigned number⟩|⟨variable⟩|⟨function designator⟩|
(⟨arithmetic expression⟩)
⟨factor⟩::=⟨primary⟩|⟨factor⟩↑⟨primary⟩
⟨term⟩::=⟨factor⟩|⟨term⟩⟨multiplying operator⟩⟨factor⟩
⟨simple arithmetic expression⟩::=⟨term⟩|⟨adding operator⟩⟨term⟩|
⟨simple arithmetic expression⟩⟨adding operator⟩⟨term⟩
⟨if clause⟩::= **if** ⟨Boolean expression⟩ **then**
⟨arithmetic expression⟩::=⟨simple arithmetic expression⟩|
⟨if clause⟩⟨simple arithmetic expression⟩ **else** ⟨arithmetic expression⟩

3.3.2. Examples.

Primaries:

$7.394_{10}-8$
sum
$w[i+2, 8]$
$cos(y+z\times 3)$
$(a-3/y+vu\uparrow 8)$

Factors:

omega

$sum \uparrow cos(y + z \times 3)$

$7.394_{10} - 8 \uparrow w[i + 2, 8] \uparrow (a - 3/y + vu \uparrow 8)$

Terms:

U

$omega \times sum \uparrow cos(y + z \times 3)/7.394_{10} - 8 \uparrow w[i + 2, 8] \uparrow (a - 3/y + vu \uparrow 8)$

Simple arithmetic expression:

$U - Yu + omega \times sum \uparrow cos(y + z \times 3)/7.394_{10} - 8 \uparrow w[i + 2, 8] \uparrow (a - 3/y + vu \uparrow 8)$

Arithmetic expressions:

$w \times u - Q(S + Cu) \uparrow 2$

if $q > 0$ **then** $S + 3 \times Q/A$ **else** $2 \times S + 3 \times q$

if $a < 0$ **then** $U + V$ **else if** $a \times b > 17$ **then** U/V **else if** $k \neq y$ **then** V/U **else** 0

$a \times sin(omega \times t)$

$0.57_{10}12 \times a[N \times (N - 1)/2, 0]$

$(A \times arctan(y) + Z) \uparrow (7 + Q)$

if q **then** $n - 1$ **else** n

if $a < 0$ **then** A/B **else if** $b = 0$ **then** B/A **else** z

3.3.3. Semantics. An arithmetic expression is a rule for computing a numerical value. In case of simple arithmetic expressions this value is obtained by executing the indicated arithmetic operations on the actual numerical values of the primaries of the expression, as explained in detail in section 3.3.4 below. The actual numerical value of a primary is obvious in the case of numbers. For variables it is the current value (assigned last in the dynamic sense), and for function designators it is the value arising from the computing rules defining the procedure (cf. section 5.4.4. Values of function designators) when applied to the current values of the procedure parameters given in the expression. Finally, for arithmetic expressions enclosed in parentheses the value must through a recursive analysis be expressed in terms of the values of primaries of the other three kinds.

In the more general arithmetic expressions, which include if clauses, one out of several simple arithmetic expressions is selected on the basis of the actual values of the Boolean expressions (cf. section 3.4. Boolean expressions). This selection is made as follows: The Boolean expressions of the if clauses are evaluated one by one in sequence from left to right until one having the value **true** is found. The value of the arithmetic expression is then the value of the first arithmetic expression following this Boolean (the largest arithmetic expression found in this position is understood). The construction:

else ⟨simple arithmetic expression⟩

is equivalent to the construction:

else if true then ⟨simple arithmetic expression⟩

3.3.4. Operators and types. Apart from the Boolean expressions of if clauses, the constituents of simple arithmetic expressions must be of types **real** or **integer**

(cf. section 5.1. Type declarations). The meaning of the basic operators and the types of the expressions to which they lead are given by the following rules:

3.3.4.1. The operators $+$, $-$, and $\times$ have the conventional meaning (addition, subtraction, and multiplication). The type of the expression will be **integer** if both of the operands are of **integer** type, otherwise **real**.

3.3.4.2. The operations ⟨term⟩/⟨factor⟩ and ⟨term⟩ $\div$ ⟨factor⟩ both denote division, to be understood as a multiplication of the term by the reciprocal of the factor with due regard to the rules of precedence (cf. section 3.3.5). Thus for example

$$a/b \times 7/(p-q) \times v/s$$

means

$$((((a \times (b^{-1})) \times 7) \times ((p-q)^{-1})) \times v) \times (s^{-1})$$

The operator / is defined for all four combinations of types **real** and **integer** and will yield results of **real** type in any case. The operator $\div$ is defined only for two operands both of type **integer** and will yield a result of type **integer**, mathematically defined as follows:

$$a \div b = sign(a/b) \times entier(abs(a/b))$$

(cf. sections 3.2.4 and 3.2.5).

3.3.4.3. The operation ⟨factor⟩ $\uparrow$ ⟨primary⟩ denotes exponentiation, where the factor is the base and the primary is the exponent. Thus for example

$2 \uparrow n \uparrow k$ means $(2^n)^k$

while

$2 \uparrow (n \uparrow m)$ means $2^{(n^m)}$

Writing i for a number of **integer** type, r for a number of **real** type, and a for a number of either **integer** or **real** type, the result is given by the following rules:

$a \uparrow i$	If $i > 0$:	$a \times a \times \cdots \times a$ (i times), of the same type as a.	
	If $i = 0$,	if $a \neq 0$:	1, of the same type as a.
		if $a = 0$:	undefined.
	If $i < 0$,	if $a \neq 0$:	$1/(a \times a \times \cdots \times a)$ (the denominator has $-i$ factors), of type **real**.
		if $a = 0$:	undefined.
$a \uparrow r$	If $a > 0$:	$\exp(r \times ln(a))$, of type **real**.	
	If $a = 0$,	if $r > 0$:	0.0, of type **real**.
		if $r \leqq 0$:	undefined.
	If $a < 0$:	always undefined.	

3.3.5. Precedence of operators. The sequence of operations within one expression is generally from left to right, with the following additional rules:

3.3.5.1. According to the syntax given in section 3.3.1 the following rules of precedence hold:

first:	$\uparrow$
second:	$\times / \div$
third:	$+ -$

3.3.5.2. The expression between a left parenthesis and the matching right parenthesis is evaluated by itself and this value is used in subsequent calculations. Consequently the desired order of execution of operations within an expression can always be arranged by appropriate positioning of parentheses.

3.3.6. Arithmetics of **real** quantities. Numbers and variables of type **real** must be interpreted in the sense of numerical analysis, i.e. as entities defined inherently with only a finite accuracy. Similarly, the possibility of the occurrence of a finite deviation from the mathematically defined result in any arithmetic expression is explicitly understood. No exact arithmetic will be specified, however, and it is indeed understood that different hardware representations may evaluate arithmetic expressions differently. The control of the possible consequences of such differences must be carried out by the methods of numerical analysis. This control must be considered a part of the process to be described, and will therefore be expressed in terms of the language itself.

3.4. Boolean expressions

3.4.1. Syntax.

⟨relational operator⟩ :: = < | ≦ | = | ≧ | > | ≠

⟨relation⟩ :: = ⟨simple arithmetic expression⟩ ⟨relational operator⟩ ⟨simple arithmetic expression⟩

⟨Boolean primary⟩ :: = ⟨logical value⟩ | ⟨variable⟩ | ⟨function designator⟩ | ⟨relation⟩ | (⟨Boolean expression⟩)

⟨Boolean secondary⟩ :: = ⟨Boolean primary⟩ | ¬ ⟨Boolean primary⟩

⟨Boolean factor⟩ :: = ⟨Boolean secondary⟩ | ⟨Boolean factor⟩ ∧ ⟨Boolean secondary⟩

⟨Boolean term⟩ :: = ⟨Boolean factor⟩ | ⟨Boolean term⟩ ∨ ⟨Boolean factor⟩

⟨implication⟩ :: = ⟨Boolean term⟩ | ⟨implication⟩ ⊃ ⟨Boolean term⟩

⟨simple Boolean⟩ :: = ⟨implication⟩ | ⟨simple Boolean⟩ ≡ ⟨implication⟩

⟨Boolean expression⟩ :: = ⟨simple Boolean⟩ | ⟨if clause⟩ ⟨simple Boolean⟩ **else** ⟨Boolean expression⟩

3.4.2. Examples. $x = -2$

$Y > V \vee z < q$

$a + b > -5 \wedge z - d > q \uparrow 2$

$p \wedge q \vee x \neq y$

$g \equiv \neg a \wedge b \wedge \neg c \vee d \vee e \supset \neg f$

if $k < 1$ **then** $s > w$ **else** $h \leqq c$

if if if a **then** b **else** c **then** d **else** f **then** g **else** $h < k$

3.4.3. Semantics. A Boolean expression is a rule for computing a logical value. The principles of evaluation are entirely analogous to those given for arithmetic expressions in section 3.3.3.

3.4.4. Types. Variables and function designators entered as Boolean primaries must be declared **Boolean** (cf. section 5.1. Type declarations and section 5.4.4. Values of function designators).

3.4.5. The operators. Relations take on the value **true** whenever the corresponding relation is satisfied for the expressions involved, otherwise **false**.

The meaning of the logical operators $\neg$ (not), $\wedge$ (and), $\vee$ (or), $\supset$ (implies), and $\equiv$ (equivalent), is given by the following function table.

b1	**false**	**false**	**true**	**true**
b2	**false**	**true**	**false**	**true**
$\neg b1$	**true**	**true**	**false**	**false**
$b1 \wedge b2$	**false**	**false**	**false**	**true**
$b1 \vee b2$	**false**	**true**	**true**	**true**
$b1 \supset b2$	**true**	**true**	**false**	**true**
$b1 \equiv b2$	**true**	**false**	**false**	**true**

3.4.6. Precedence of operators. The sequence of operations within one expression is generally from left to right, with the following additional rules:

3.4.6.1. According to the syntax given in section 3.4.1 the following rules of precedence hold:

first: arithmetic expressions according to section 3.3.5.
second: $< \leqq = \geqq > \neq$
third: $\neg$
fourth: $\wedge$
fifth: $\vee$
sixth: $\supset$
seventh: $\equiv$

3.4.6.2. The use of parentheses will be interpreted in the sense given in section 3.3.5.2.

3.5. Designational expressions

3.5.1. Syntax.

⟨label⟩::=⟨identifier⟩|⟨unsigned integer⟩
⟨switch identifier⟩::=⟨identifier⟩
⟨switch designator⟩::=⟨switch identifier⟩[⟨subscript expression⟩]
⟨simple designational expression⟩::=⟨label⟩|⟨switch designator⟩|
(⟨designational expression⟩)
⟨designational expression⟩::=⟨simple designational expression⟩|
⟨if clause⟩ ⟨simple designational expression⟩ **else** ⟨designational expression⟩

3.5.2. Examples. *17*
p9
Choose[*n* − *1*]
Town[**if** *y*<*0* **then** *N* **else** *N*+*1*]
if *Ab*<*c* **then** *17* **else** *q*[**if** *w* ≦ *0* **then** *2* **else** *n*]

3.5.3. Semantics. A designational expression is a rule for obtaining a label of a statement (cf. section 4. Statements). Again the principle of the evaluation is entirely analogous to that of arithmetic expressions (section 3.3.3). In the general case the Boolean expressions of the if clauses will select a simple designational expression. If this is a label the desired result is already found. A switch designator refers to the corresponding switch declaration (cf. section 5.3. Switch declarations) and by the actual numerical value of its subscript expression

selects one of the designational expressions listed in the switch declaration by counting these from left to right. Since the designational expression thus selected may again be a switch designator this evaluation is obviously a recursive process.

3.5.4. The subscript expression. The evaluation of the subscript expression is analogous to that of subscripted variables (cf. section 3.1.4.2). The value of a switch designator is defined only if the subscript expression assumes one of the positive values *1, 2, 3, ..., n, where n is the number of entries in the switch list.*

3.5.5. Unsigned integers as labels. Unsigned integers used as labels have the property that leading zeroes do not affect their meaning, e.g. *00217* denotes the same label as *217*.

4. Statements

The units of operation within the language are called statements. They will normally be executed consecutively as written. However, this sequence of operations may be broken by go to statements, which define their successor explicitly, and shortened by conditional statements, which may cause certain statements to be skipped.

In order to make it possible to define a specific dynamic succession, statements may be provided with labels.

Since sequences of statements may be grouped together into compound statements and blocks the definition of statement must necessarily be recursive. *Also since declarations, described in section 5, enter fundamentally into the* syntactic structure, the syntactic definition of statements must suppose declarations to be already defined.

4.1. Compound statements and blocks

4.1.1. Syntax.

⟨unlabelled basic statement⟩::=⟨assignment statement⟩|⟨go to statement⟩|
⟨dummy statement⟩|⟨procedure statement⟩
⟨basic statement⟩::=⟨unlabelled basic statement⟩|⟨label⟩:⟨basic statement⟩
⟨unconditional statement⟩::=⟨basic statement⟩|
⟨compound statement⟩|⟨block⟩
⟨statement⟩::=⟨unconditional statement⟩|⟨conditional statement⟩|
⟨for statement⟩
⟨compound tail⟩::=⟨statement⟩ **end**|⟨statement⟩;⟨compound tail⟩
⟨block head⟩::=**begin** ⟨declaration⟩|⟨block head⟩;⟨declaration⟩
⟨*unlabelled compound*⟩::=***begin*** ⟨*compound tail*⟩
⟨unlabelled block⟩::=⟨block head⟩;⟨compound tail⟩
⟨compound statement⟩::=⟨unlabelled compound⟩|
⟨label⟩:⟨compound statement⟩
⟨block⟩::=⟨unlabelled block⟩|⟨label⟩:⟨block⟩
⟨program⟩::=⟨block⟩|⟨compound statement⟩

This syntax may be illustrated as follows: Denoting arbitrary statements, declarations, and labels, by the letters S, D, and L, respectively, the basic syntactic units take the forms:

Compound statement:

L: L: ... **begin** S; S; ... S; S **end**

Block:

L: L: ... **begin** D; D; .. D; S; S; ... S; S **end**

It should be kept in mind that each of the statements S may again be a complete compound statement or block.

4.1.2. Examples.

Basic statements:

$a := p + q$

go to *Naples*

Start: *Continue*: $W := 7.993$

Compound statement:

begin $x := 0$; **for** $y := 1$ **step** *1* **until** n **do** $x := x + A[y]$;
if $x > q$ **then go to** *STOP* **else if** $x > w - 2$ **then go to** S;
Aw: *St*: $W := x + bob$ **end**

Block:

Q: **begin integer** i, k; **real** w;
for $i := 1$ **step** *1* **until** m **do**
for $k := i + 1$ **step** *1* **until** m **do**
begin $w := A[i, k]$;
$A[i, k] := A[k, i]$;
$A[k, i] := w$ **end** *for i and k*
end *block Q*

4.1.3. Semantics. Every block automatically introduces a new level of nomenclature. This is realized as follows: Any identifier occurring within the block may through a suitable declaration (cf. section 5. Declarations) be specified to be local to the block in question. This means (a) that the entity represented by this identifier inside the block has no existence outside it and (b) that any entity represented by this identifier outside the block is completely inaccessible inside the block.

Identifiers (except those representing labels) occurring within a block and not being declared to this block will be non-local to it, i.e. will represent the same entity inside the block and in the level immediately outside it. A label separated by a colon from a statement, i.e. labelling that statement, behaves as though declared in the head of the smallest embracing block, i.e. the smallest block whose brackets **begin** and **end** enclose that statement. In this context a procedure body must be considered as if it were enclosed by **begin** and **end** and treated as a block.

Since a statement of a block may again itself be a block the concepts local and non-local to a block must be understood recursively. Thus an identifier, which is non-local to a block A, may or may not be non-local to the block B in which A is one statement.

4.2. Assignment statements

4.2.1. Syntax.

⟨left part⟩::= ⟨variable⟩: = | ⟨procedure identifier⟩: =

⟨left part list⟩::= ⟨left part⟩ | ⟨left part list⟩ ⟨left part⟩

⟨assignment statement⟩::= ⟨left part list⟩ ⟨arithmetic expression⟩ |
⟨left part list⟩ ⟨Boolean expression⟩

4.2.2. Examples. $s := p[0] := n := n + 1 + s$
$n := n + 1$
$A := B/C - v - q \times S$
$S[v, k + 2] := 3 - arctan(s \times zeta)$
$V := Q > Y \wedge Z$

4.2.3. Semantics. Assignment statements serve for assigning the value of an expression to one or several variables or procedure identifiers. Assignment to a procedure identifier may only occur within the body of a procedure defining the value of a function designator (cf. section 5.4.4). The process will in the general case be understood to take place in three steps as follows:

4.2.3.1. Any subscript expressions occurring in the left part variables are evaluated in sequence from left to right.

4.2.3.2. The expression of the statement is evaluated.

4.2.3.3. The value of the expression is assigned to all the left part variables, with any subscript expressions having values as evaluated in step 4.2.3.1.

4.2.4. Types. The type associated with all variables and procedure identifiers of a left part list must be the same. If this type is **Boolean**, the expression must likewise be **Boolean**. If the type is **real** or **integer**, the expression must be arithmetic. If the type of the arithmetic expression differs from that associated with the variables and procedure identifiers, appropriate transfer functions are understood to be automatically invoked. For transfer from **real** to **integer** type the transfer function is understood to yield a result equivalent to

entier(E + 0.5)

where E is the value of the expression. The type associated with a procedure identifier is given by the declarator which appears as the first symbol of the corresponding procedure declaration (cf. section 5.4.4).

4.3. Go to statements

4.3.1. Syntax.
⟨go to statement⟩ :: = **go to** ⟨designational expression⟩

4.3.2. Examples. **go to** *8*
go to *exit*[*n* + *1*]
go to *Town* [**if** *y* < *0* **then** *N* **else** *N*+*1*]
go to if *Ab*<*c* **then** *17* **else** *q*[**if** *w*<*0* **then** *2* **else** *n*]

4.3.3. Semantics. A go to statement interrupts the normal sequence of operations, defined by the write-up of statements, by defining its successor explicitly by the value of a designational expression. Thus the next statement to be executed will be the one having this value as its label.

4.3.4. Restriction. Since labels are inherently local, no go to statement can lead from outside into a block. A go to statement may, however, lead from outside into a compound statement.

4.3.5. Go to an undefined switch designator. A go to statement is equivalent to a dummy statement if the designational expression is a switch designator whose value is undefined.

4.4. Dummy statements

4.4.1. Syntax.
⟨dummy statement⟩::= ⟨empty⟩

4.4.2. Examples.

L:
begin ...; *John*: **end**

4.4.3. Semantics. A dummy statement executes no operation. It may serve to place a label.

4.5. Conditional statements

4.5.1. Syntax.
⟨if clause⟩::= **if** ⟨Boolean expression⟩ **then**
⟨unconditional statement⟩::= ⟨basic statement⟩ | ⟨compound statement⟩ | ⟨block⟩
⟨if statement⟩::= ⟨if clause⟩ ⟨unconditional statement⟩
⟨conditional statement⟩::= ⟨if statement⟩ | ⟨if statement⟩ **else** ⟨statement⟩ | ⟨if clause⟩ ⟨for statement⟩ | ⟨label⟩: ⟨conditional statement⟩

4.5.2. Examples. **if** $x>0$ **then** $n:=n+1$
if $v>u$ **then** $V:q:=n+m$ **else go to** R
if $s<0 \vee P \leqq Q$ **then** AA: **begin if** $q<v$ **then** $a:=v/s$
else $y:=2\times a$ **end else if** $v>s$ **then** $a:=v-q$
else if $v>s-1$ **then go to** S

4.5.3. Semantics. Conditional statements cause certain statements to be executed or skipped depending on the running values of specified Boolean expressions.

4.5.3.1. If statement. The unconditional statement of an if statement will be executed if the Boolean expression of the if clause is true. Otherwise it will be skipped and the operation will be continued with the next statement.

4.5.3.2. Conditional statement. According to the syntax two different forms of conditional statements are possible. These may be illustrated as follows:

if B1 **then** S1 **else if** B2 **then** S2 **else** S3; S4

and

if B1 **then** S1 **else if** B2 **then** S2 **else if** B3 **then** S3; S4

Here B1 to B3 are Boolean expressions, while S1 to S3 are unconditional statements. S4 is the statement following the complete conditional statement.

The execution of a conditional statement may be described as follows: The Boolean expressions of the if clauses are evaluated one after the other in sequence from left to right until one yielding the value **true** is found Then the unconditional statement following this Boolean is executed. Unless this statement defines its successor explicitly the next statement to be executed will be S4, i.e. the statement following the complete conditional statement. Thus the effect of the delimiter **else** may be described by saying that it defines the successor of the statement it follows to be the statement following the complete conditional statement.

The construction

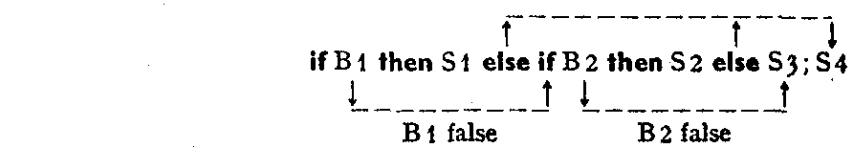
else ⟨unconditional statement⟩

is equivalent to

else if true then ⟨unconditional statement⟩

If none of the Boolean expressions of the if clauses is true, the effect of the whole conditional statement will be equivalent to that of a dummy statement.

For further explanation the following picture may be useful:



4.5.4. Go to into a conditional statement. The effect of a go to statement leading into a conditional statement follows directly from the above explanation of the effect of **else**.

4.6. For statements

4.6.1. Syntax.

⟨for list element⟩::=⟨arithmetic expression⟩|
⟨arithmetic expression⟩ **step** ⟨arithmetic expression⟩ **until** ⟨arithmetic expression⟩|
⟨arithmetic expression⟩ **while** ⟨Boolean expression⟩

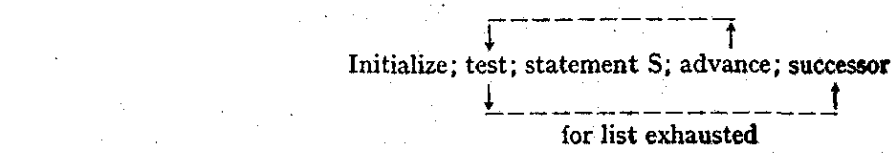
⟨for list⟩::=⟨for list element⟩|⟨for list⟩, ⟨for list element⟩

⟨for clause⟩::= **for** ⟨variable⟩:=⟨for list⟩ **do**

⟨for statement⟩::=⟨for clause⟩⟨statement⟩|
⟨label⟩:⟨for statement⟩

4.6.2. Examples. **for** $q:=1$ **step** s **until** n **do** $A[q]:=B[q]$
for $k:=1, V1\times 2$ **while** $V1<N$ **do**
for $j:=I+G, L, 1$ **step** 1 **until** $N, C+D$ **do** $A[k,j]:=B[k,j]$

4.6.3. Semantics. A for clause causes the statement S which it precedes to be repeatedly executed zero or more times. In addition it performs a sequence of assignments to its controlled variable. The process may be visualized by means of the following picture:



In this picture the word initialize means: perform the first assignment of the for clause. Advance means: perform the next assignment of the for clause. Test determines if the last assignment has been done. If so, the execution continues with the successor of the for statement. If not, the statement following the for clause is executed.

4.6.4. The for list elements. The for list gives a rule for obtaining the values which are consecutively assigned to the controlled variable. This sequence of

values is obtained from the for list elements by taking these one by one in the order in which they are written. The sequence of values generated by each of the three species of for list elements and the corresponding execution of the statement S are given by the following rules:

4.6.4.1. Arithmetic expression. This element gives rise to one value, namely the value of the given arithmetic expression as calculated immediately before the corresponding execution of the statement S.

4.6.4.2. Step-until-element. An element of the form A **step** B **until** C, where A, B, and C are arithmetic expressions, gives rise to an execution which may be described most concisely in terms of additional ALGOL statements as follows:

V: = A;
L1: **if** (V − C) × *sign* (B) > *0* **then go to** *Element exhausted*;
Statement S;
V: = V + B;
go to *L1*;

where V is the controlled variable of the for clause and *Element exhausted* points to the evaluation according to the next element in the for list, or if the step-until-element is the last of the list, to the next statement in the program.

4.6.4.3. While-element. The execution governed by a for list element of the form E **while** F, where E is an arithmetic and F a Boolean expression, is most concisely described in terms of additional ALGOL statements as follows:

L3: V: = E;
if ¬ F **then go to** *Element exhausted*;
Statement S;
go to *L3*;

where the notation is the same as in 4.6.4.2 above.

4.6.5. The value of the controlled variable upon exit. Upon exit out of the statement S (supposed to be compound) through a go to statement the value of the controlled variable will be the same as it was immediately preceding the execution of the go to statement.

If the exit is due to exhaustion of the for list, on the other hand, the value of the controlled variable is undefined after the exit.

4.6.6. Go to leading into a for statement

The effect of a go to statement, outside a for statement, which refers to a label within the for statement, is undefined.

4.7. Procedure statements

4.7.1. Syntax.

⟨actual parameter⟩ :: = ⟨string⟩ | ⟨expression⟩ | ⟨array identifier⟩ |
⟨switch identifier⟩ | ⟨procedure identifier⟩
⟨letter string⟩ :: = ⟨letter⟩ | ⟨letter string⟩ ⟨letter⟩
⟨parameter delimiter⟩ :: = , |) ⟨letter string⟩ : (
⟨actual parameter list⟩ :: = ⟨actual parameter⟩ |
⟨actual parameter list⟩ ⟨parameter delimiter⟩ ⟨actual parameter⟩
⟨actual parameter part⟩ :: = ⟨empty⟩ | (⟨actual parameter list⟩)
⟨procedure statement⟩ :: = ⟨procedure identifier⟩ ⟨actual parameter part⟩

4.7.2. Examples. *Spur (A) Order: (7) Result to: (V)*
Transpose (W, v+1)
Absmax (A, N, M, Yy, I, K)
Innerproduct (A[t, P, u], B[P], 10, P, Y)
These examples correspond to examples given in section 5.4.2.

4.7.3. Semantics. A procedure statement serves to invoke (call for) the execution of a procedure body (cf. section 5.4. procedure declarations). Where the procedure body is a statement written in ALGOL the effect of this execution will be equivalent to the effect of performing the following operations on the program at the time of execution of the procedure statement:

4.7.3.1. Value assignment (call by value). All formal parameters quoted in the value part of the procedure declaration heading are assigned the values (cf. section 2.8. Values and types) of the corresponding actual parameters, these assignments being considered as being performed explicitly before entering the procedure body. The effect is as though an additional block embracing the procedure body were created in which these assigments were made to variables local to this fictitious block with types as given in the corresponding specifications (cf. section 5.4.5). As a consequence, variables called by value are to be considered as non-local to the body of the procedure, but local to the fictitious block (cf. section 5.4.3).

4.7.3.2. Name replacement (call by name). Any formal parameter not quoted in the value list is replaced, throughout the procedure body, by the corresponding actual parameter, after enclosing this latter in parentheses wherever syntactically possible. Possible conflicts between identifiers inserted through this process and other identifiers already present within the procedure body will be avoided by suitable systematic changes of the formal or local identifiers involved.

4.7.3.3. Body replacement and execution. Finally the procedure body, modified as above, is inserted in place of the procedure statement and executed. If the procedure is called from a place outside the scope of any non-local quantity of the procedure body the conflicts between the identifiers inserted through this process of body replacement and the identifiers whose declarations are valid at the place of the procedure statement or function designator will be avoided through suitable systematic changes of the latter identifiers.

4.7.4. Actual-formal correspondence. The correspondence between the actual parameters of the procedure statement and the formal parameters of the procedure heading is established as follows: The actual parameter list of the procedure statement must have the same number of entries as the formal parameter list of the procedure declaration heading. The correspondence is obtained by taking the entries of these two lists in the same order.

4.7.5. Restrictions. For a procedure statement to be defined it is evidently necessary that the operations on the procedure body defined in sections 4.7.3.1 and 4.7.3.2 lead to a correct ALGOL statement.

This imposes the restriction on any procedure statement that the kind and type of each actual parameter be compatible with the kind and type of the corresponding formal parameter. Some important particular cases of this general rule are the following:

4.7.5.1. If a string is supplied as an actual parameter in a procedure statement or function designator, whose defining procedure body is an ALGOL 60 statement (as opposed to non-ALGOL code, cf. section 4.7.8), then this string can only be used within the procedure body as an actual parameter in further procedure calls. Ultimately it can only be used by a procedure body expressed in non-ALGOL code.

4.7.5.2. A formal parameter which occurs as a left part variable in an assignment statement within the procedure body and which is not called by value can only correspond to an actual parameter which is a variable (special case of expression).

4.7.5.3. A formal parameter which is used within the procedure body as an array identifier can only correspond to an actual parameter which is an array identifier of an array of the same dimensions. In addition if the formal parameter is called by value the local array created during the call will have the same subscript bounds as the actual array.

4.7.5.4. A formal parameter which is called by value cannot in general correspond to a switch identifier or a procedure identifier or a string, because these latter do not possess values (the exception is the procedure identifier of a procedure declaration which has an empty formal parameter part (cf. section 5.4.1) and which defines the value of a function designator (cf. section 5.4.4). This procedure identifier is in itself a complete expression).

4.7.5.5 Any formal parameter may have restrictions on the type of the corresponding actual parameter associated with it (these restrictions may, or may not, be given through specifications in the procedure heading). In the procedure statement such restrictions must evidently be observed.

4.7.6. Deleted.

4.7.7. Parameter delimiters. All parameter delimiters are understood to be equivalent. No correspondence between the parameter delimiters used in a procedure statement and those used in the procedure heading is expected beyond their number being the same. Thus the information conveyed by using the elaborate ones is entirely optional.

4.7.8. Procedure body expressed in code. The restrictions imposed on a procedure statement calling a procedure having its body expressed in non-ALGOL code evidently can only be derived from the characteristics of the code used and the intent of the user and thus fall outside the scope of the reference language.

5. Declarations

Declarations serve to define certain properties of the quantities used in the program, and to associate them with identifiers. A declaration of an identifier is valid for one block. Outside this block the particular identifier may be used for other purposes (cf. section 4.1.3).

Dynamically this implies the following: at the time of an entry into a block (through the **begin** since the labels inside are local and therefore inaccessible from outside) all identifiers declared for the block assume the significance implied by the nature of the declarations given. If these identifiers had already been defined by other declarations outside they are for the time being given a new significance.

Identifiers which are not declared for the block, on the other hand, retain their old meaning.

At the time of an exit from a block (through **end**, or by a go to statement) all identifiers which are declared for the block lose their local significance.

A declaration may be marked with the additional declarator **own**. This has the following effect: upon a reentry into the block, the values of own quantities will be unchanged from their values at the last exit, while the values of declared variables which are not marked as own are undefined. Apart from labels and formal parameters of procedure declarations and with the possible exception of those for standard functions (cf. sections 3.2.4 and 3.2.5) all identifiers of a program must be declared. No identifier may be declared more than once in any one block head.

Syntax.

⟨declaration⟩::= ⟨type declaration⟩ | ⟨array declaration⟩ |
⟨switch declaration⟩ | ⟨procedure declaration⟩

5.1. Type declarations

5.1.1. Syntax.

⟨type list⟩::= ⟨simple variable⟩ | ⟨simple variable⟩, ⟨type list⟩
⟨type⟩::= **real** | **integer** | **Boolean**
⟨local or own type⟩::= ⟨type⟩ | **own** ⟨type⟩
⟨type declaration⟩::= ⟨local or own type⟩ ⟨type list⟩

5.1.2. Examples. **integer** *p, q, s*
own Boolean *Acryl, n*

5.1.3. Semantics. Type declarations serve to declare certain identifiers to represent simple variables of a given type. Real declared variables may only assume positive or negative values including zero. Integer declared variables may only assume positive and negative integral values including zero. Boolean declared variables may only assume the values **true** and **false**.

In arithmetic expressions any position which can be occupied by a real declared variable may be occupied by an integer declared variable.

For the semantics of **own**, see the fourth paragraph of section 5 above.

5.2. Array declarations

5.2.1. Syntax.

⟨lower bound⟩::= ⟨arithmetic expression⟩
⟨upper bound⟩::= ⟨arithmetic expression⟩
⟨bound pair⟩::= ⟨lower bound⟩ : ⟨upper bound⟩
⟨bound pair list⟩::= ⟨bound pair⟩ | ⟨bound pair list⟩, ⟨bound pair⟩
⟨array segment⟩::= ⟨array identifier⟩ [⟨bound pair list⟩] |
⟨array identifier⟩, ⟨array segment⟩
⟨array list⟩::= ⟨array segment⟩ | ⟨array list⟩, ⟨array segment⟩
⟨array declaration⟩::= **array** ⟨array list⟩ |
⟨local or own type⟩ **array** ⟨array list⟩

5.2.2. Examples. **array** *a, b, c*[*7:n, 2:m*], *s*[*−2:10*]
own integer array *A*[**if** *c<0* **then** *2* **else** *1:20*]
real array *q*[*−7:−1*]

5.2.3. Semantics. An array declaration declares one or several identifiers to represent multidimensional arrays of subscripted variables and gives the dimensions of the arrays, the bounds of the subscripts, and the types of the variables.

5.2.3.1. Subscript bounds. The subscript bounds for any array are given in the first subscript bracket following the identifier of this array in the form of a bound pair list. Each item of this list gives the lower and upper bound of a subscript in the form of two arithmetic expressions separated by the delimiter : . The bound pair list gives the bounds of all subscripts taken in order from left to right.

5.2.3.2. Dimensions. The dimensions are given as the number of entries in the bound pair lists.

5.2.3.3. Types. All arrays declared in one declaration are of the same quoted type. If no type declarator is given the type **real** is understood.

5.2.4. Lower upper bound expressions.

5.2.4.1. The expressions will be evaluated in the same way as subscript expressions (cf. section 3.1.4.2).

5.2.4.2. The expressions can only depend on variables and procedures which are non-local to the block for which the array declaration is valid. Consequently in the outermost block of a program only array declarations with constant bounds may be declared.

5.2.4.3. An array is defined only when the values of all upper subscript bounds are not smaller than those of the corresponding lower bounds.

5.2.4.4. The expressions will be evaluated once at each entrance into the block.

5.2.5. The identity of subscripted variables. The identity of a subscripted variable is not related to the subscript bounds given in the array declaration. However, even if an array is declared **own** the values of the corresponding subscripted variables will, at any time, be defined only for those of these variables which have subscripts within the most recently calculated subscript bounds.

5.3. Switch declarations

5.3.1. Syntax.

⟨switch list⟩::= ⟨designational expression⟩|
⟨switch list⟩, ⟨designational expression⟩

⟨switch declaration⟩::= **switch** ⟨switch identifier⟩:= ⟨switch list⟩

5.3.2. Examples. **switch** $S:=S1, S2, Q[m]$, **if** $v>-5$ **then** $S3$ **else** $S4$
switch $Q:=p1, w$

5.3.3. Semantics. A switch declaration defines the set of values of the corresponding switch designators. These values are given one by one as the values of the designational expressions entered in the switch list. With each of these designational expressions there is associated a positive integer, 1, 2, ..., obtained by counting the items in the list from left to right. The value of the switch designator corresponding to a given value of the subscript expression (cf. section 3.5. Designational expressions) is the value of the designational expression in the switch list having this given value as its associated integer.

5.3.4. Evaluation of expressions in the switch list. An expression in the switch list will be evaluated every time the item of the list in which the expression occurs is referred to, using the current values of all variables involved.

5.3.5. Influence of scopes. If a switch designator occurs outside the scope of a quantity entering into a designational expression in the switch list, and an evaluation of this switch designator selects this designational expression, then the conflicts between the identifiers for the quantities in this expression and the identifiers whose declarations are valid at the place of the switch designator will be avoided through suitable systematic changes of the latter identifiers.

5.4. Procedure declarations

5.4.1. Syntax.

⟨formal parameter⟩::= ⟨identifier⟩
⟨formal parameter list⟩::= ⟨formal parameter⟩|
⟨formal parameter list⟩ ⟨parameter delimiter⟩ ⟨formal parameter⟩
⟨formal parameter part⟩::= ⟨empty⟩|(⟨formal parameter list⟩)
⟨identifier list⟩::= ⟨identifier⟩| ⟨identifier list⟩, ⟨identifier⟩
⟨value part⟩::= **value** ⟨identifier list⟩;| ⟨empty⟩
⟨specifier⟩::= **string**| ⟨type⟩| **array**| ⟨type⟩ **array**| **label**| **switch**|
procedure| ⟨type⟩ **procedure**
⟨specification part⟩::= ⟨empty⟩| ⟨specifier⟩ ⟨identifier list⟩;|
⟨specification part⟩ ⟨specifier⟩ ⟨identifier list⟩;
⟨procedure heading⟩::= ⟨procedure identifier⟩ ⟨formal parameter part⟩;
⟨value part⟩ ⟨specification part⟩
⟨procedure body⟩::= ⟨statement⟩| ⟨code⟩
⟨procedure declaration⟩::= **procedure** ⟨procedure heading⟩ ⟨procedure body⟩|
⟨type⟩ **procedure** ⟨procedure heading⟩ ⟨procedure body⟩

5.4.2. Examples (see also the examples at the end of the report).

```
procedure Spur(a) Order: (n) Result: (s); value n;
array a; integer n; real s;
begin integer k;
s:=0;
for k:=1 step 1 until n do s:=s+a[k,k]
end

procedure Transpose (a) Order: (n); value n;
array a; integer n;
begin real w; integer i, k;
for i:=1 step 1 until n do
    for k:=1+i step 1 until n do
    begin w:=a[i,k];
          a[i,k]:=a[k,i];
          a[k,i]:=w
    end
end Transpose

integer procedure Step (u); real u;
Step:=if 0≦u∧u≦1 then 1 else 0
```

```
procedure Absmax (a) size: (n, m) Result: (y) Subscripts: (i, k);
comment The absolute greatest element of the matrix a, of size n by m is transferred
to y, and the subscripts of this element to i and k;
array a; integer n, m, i, k; real y;
begin integer p, q;
y:=0;
for p:=1 step 1 until n do for q:=1 step 1 until m do
if abs(a[p, q])>y then begin y:=abs(a[p, q]); i:=p; k:=q end end Absmax
```

```
procedure Innerproduct (a, b) Order: (k, p) Result: (y); value k;
integer k, p; real y, a, b;
begin real s;
s:=0;
for p:=1 step 1 until k do s:=s+a×b;
y:=s
end Innerproduct
```

5.4.3. Semantics. A procedure declaration serves to define the procedure associated with a procedure identifier. The principal constituent of a procedure declaration is a statement or a piece of code, the procedure body, which through the use of procedure statements and/or function designators may be activated from other parts of the block in the head of which the procedure declaration appears. Associated with the body is a heading, which specifies certain identifiers occurring within the body to represent formal parameters. Formal parameters in the procedure body will, whenever the procedure is activated (cf. section 3.2. Function designators and section 4.7. Procedure statements) be assigned the values of or replaced by actual parameters. Identifiers in the procedure body which are not formal will be either local or non-local to the body depending on whether they are declared within the body or not. Those of them which are non-local to the body may well be local to the block in the head of which the procedure declaration appears. The procedure body always acts like a block, whether it has the form of one or not. Consequently the scope of any label labelling a statement within the body or the body itself can never extend beyond the procedure body. In addition, if the identifier of a formal parameter is declared anew within the procedure body (including the case of its use as a label as in section 4.1.3), it is thereby given a local significance and actual parameters which correspond to it are inaccessible throughout the scope of this inner local quantity.

5.4.4. Values of function designators. For a procedure declaration to define the value of a function designator there must, within the procedure body, occur one or more explicit assignment statements with the procedure identifier in a left part; at least one of these must be executed, and the type associated with the procedure identifier must be declared through the appearance of a type declarator as the very first symbol of the procedure declaration. The last value so assigned is used to continue the evaluation of the expression in which the function designator occurs. Any occurrence of the procedure identifier within the body of the procedure other than in a left part in an assignment statement denotes activation of the procedure.

5.4.5. Specifications. In the heading a specification part, giving information about the kinds and types of the formal parameters by means of an obvious notation, may be included. In this part no formal parameter may occur more than once. Specifications of formal parameters called by value (cf. section 4.7.3.1) must be supplied and specifications of formal parameters called by name (cf. section 4.7.3.2) may be omitted.

5.4.6. Code as procedure body. It is understood that the procedure body may be expressed in non-ALGOL language. Since it is intended that the use of this feature should be entirely a question of hardware representation, no further rules concerning this code language can be given within the reference language.

Examples of procedure declarations

Example 1

```
procedure euler (fct, sum, eps, tim); value eps, tim; integer tim;
real procedure fct; real sum, eps;
comment euler computes the sum of fct(i) for i from zero up to infinity by means of
a suitably refined euler transformation. The summation is stopped as soon as tim
times in succession the absolute value of the terms of the transformed series are found
to be less than eps. Hence, one should provide a function fct with one integer argument,
an upper bound eps, and an integer tim. The output is the sum sum. euler is parti-
cularly efficient in the case of a slowly convergent or divergent alternating series;
begin integer i, k, n, t; array m[0:15]; real mn, mp, ds;
i:=n:=t:=0; m[0]:=fct(0); sum:=m[0]/2;
nextterm: i:=i+1; mn:=fct(i);
          for k:=0 step 1 until n do
              begin mp:=(mn+m[k])/2; m[k]:=mn; mn:=mp end means;
          if (abs(mn)<abs(m[n]))∧(n<15) then
              begin ds:=mn/2; n:=n+1; m[n]:=mn end accept
          else ds:=mn;
          sum:=sum+ds;
          if abs(ds)<eps then t:=t+1 else t:=0;
          if t<tim then go to nextterm
end euler
```

Example 2[1]

```
procedure RK(x, y, n, FKT, eps, eta, xE, yE, fi); value x, y; integer n;
Boolean fi; real x, eps, eta, xE; array y, yE; procedure FKT;
comment RK integrates the system y'k=fk(x, y1, y2, ..., yn) (k=1, 2, ... n)
of differential equations with the method of Runge-Kutta with automatic search for
appropriate length of integration step. Parameters are: The initial values x and y[k]
```

[1] This RK-program contains some new ideas which are related to ideas of S. GILL, A process for the step by step integration of differential equations in an automatic computing machine. Proc. Camb. Phil. Soc. 47 (1951) p. 96, and E. FRÖBERG, On the solution of ordinary differential equations with digital computing machines, Fysiograf. Sällsk. Lund, Förhd. 20 Nr. 11 (1950) p. 136–152. It must be clear however that with respect to computing time and round-off errors it may not be optimal, nor has it actually been tested on a computer.

for x and the unknown functions $y_k(x)$. The order n of the system. The procedure FKT(x, y, n, z) which represents the system to be integrated, i.e. the set of functions f_k. The tolerance values eps and eta which govern the accuracy of the numerical integration. The end of the integration interval xE. The output parameter yE which represents the solution at x=xE. The Boolean variable fi, which must always be given the value **true** *for an isolated or first entry into RK. If however the functions y must be available at several meshpoints $x_0, x_1, \ldots, x_n$, then the procedure must be called repeatedly (with $x=x_k$, $xE=x_{k+1}$, for $k=0, 1, \ldots, n-1$) and then the later calls may occur with fi =* **false** *which saves computing time. The input parameters of* FKT *must be x, y, n, the output parameter z represents the set of derivatives $z[k]=f_k(x, y[1], y[2], \ldots, y[n])$ for x and the actual y's. A procedure comp enters as a non-local identifier;*

```
begin
    array z, y1, y2, y3[1:n]; real x1, x2, x3, H; Boolean out;
    integer k, j; own real s, Hs;
    procedure RK1ST(x, y, h, xe, ye); real x, h, xe; array y, ye;
        comment RK1ST integrates one single Runge-Kutta step with initial
        values x, y[k] which yields the output parameters xe = x + h and ye[k],
        the latter being the solution at xe.
        Important: the parameters n, FKT, z enter RK1ST as non-local entities;
        begin
            array w[1:n], a[1:5]; integer k, j;
            a[1]:=a[2]:=a[5]:=h/2; a[3]:=a[4]:=h; xe:=x;
            for k:=1 step 1 until n do ye[k]:=w[k]:=y[k];
            for j:=1 step 1 until 4 do
            begin
                FKT(xe, w, n, z);
                xe:=x+a[j];
                for k:=1 step 1 until n do
                begin
                    w[k]:=y[k]+a[j]×z[k];
                    ye[k]:=ye[k]+a[j+1]×z[k]/3
                end k
            end j
        end RK1ST;
Begin of program:
    if fi then begin H:=xE-x; s:=0 end else H:=Hs;
    out:= false;
AA: if (x+2.01×H-xE>0)≡(H>0) then
    begin Hs:=H; out:=true; H:=(xE-x)/2 end if;
    RK1ST(x, y, 2×H, x1, y1);
BB: RK1ST(x, y, H, x2, y2); RK1ST(x2, y2, H, x3, y3);
    for k:=1 step 1 until n do
        if comp(y1[k], y3[k], eta)>eps then go to CC;
    comment comp(a, b, c) is a function designator, the value of which is the
    absolute value of the difference of the mantissae of a and b, after the exponents
```

of these quantities have been made equal to the largest of the exponents of the originally given parameters a, b, c;
x:=*x3*; **if** *out* **then go to** *DD*;
for *k*:=*1* **step** *1* **until** *n* **do** *y*[*k*]:=*y3*[*k*];
if *s*=*5* **then begin** *s*:=*0*; *H*:=*2*×*H* **end** *if*;
s:=*s*+*1*; **go to** *AA*;

CC: *H*:=*0.5*×*H*; *out*:= **false**; *x1*=*x2*;
for *k*:=*1* **step** *1* **until** *n* **do** *y1*[*k*]:=*y2*[*k*];
go to *BB*;

DD: **for** *k*:=1 **step** *1* **until** *n* **do** *yE*[*k*]:=*y3*[*k*]
end *RK*

Alphabetic index of definitions of concepts and syntactic units

All references are given through section numbers. The references are given in three groups:

def Following the abbreviation "def", reference to the syntactic definition (if any) is given.

synt Following the abbreviation "synt", references to the occurrences in metalinguistic formulae are given. References already quoted in the def-group are not repeated.

text Following the word "text", the references to definitions given in the text are given.

The basic symbols represented by signs other than underlined *(bold faced. Publishers remark)* words have been collected at the beginning. The examples have been ignored in compiling the index.

Technical University Delft
Delft, Holland
W. L. van der Poel,
(Chairman of Working Group 2.1 on ALGOL of the
International Federation for Information Processing)