Dataflow Computers

Arthur H. Veen
Afdeling Informatica
Mathematisch Centrum

28 maart 1980

COLLOQUIUM HOGERE PROGRAMMEERTALEN EN COMPUTERARCHITECTUUR
Mathematisch Centrum

1. IN HET BEGIN

In den beginne was er een machine, die bestond uit een geheugen en een Centrale Verwerkings-Eenheid (CVE). De werking was simpel. Het geheugen was passief; daar lagen instrukties en gegevens opgeslagen. De CVE was aktief: hij doorliep een oneindige cyclus. Iedere stap van de cyclus bestond uit het ophalen van een instruktie uit het geheugen en het uitvoeren van de bewerkingen, die door de instruktie werden gespecificeerd. De programma-wijzer bepaalde welke instruktie werd opgehaald. Deze werd na iedere instruktie met één opgehoogd. Er waren ook instrukties, die de programma-wijzer een nieuwe waarde konden geven. Een programma bevatte twee soorten instrukties: instrukties die de programma-wijzer wijzigden (control flow) en instrukties die gegevens in het geheugen wijzigden (data flow).

Dit koncept, dat later naar von Neumann werd genoemd, was eenvoudig en overzichtelijk en geheel toegesneden op de stand van de technologie van die dagen: een eenvoudige CVE kon nog maar net gemaakt worden. In de drie decennia die op de eerste computers volgden, veranderde de situatie drastisch. Complexe hardware werd steeds goedkoper, waardoor, bij het ontwerpen van een CVE, snelheid en kracht belangrjker werden dan eenvoud. Nieuwe toepassingsgebieden dienden zich aan en flexibiliteit werd belangrijk. Programma's werden steeds groter en onoverzichtelijker. Om de koppeling van randapparatuur (en later multi-programmering) te vergemakkelijken deed de interrupt zijn intrede, waardoor de control flow geheel ondoorzichtig werd. Op den duur bleef van de eenvoud van de oorspronkelijke machine weinig meer over.

Zo'n tien jaar geleden deed een nieuw, simpel koncept zijn intrede: dataflow. Een dataflow-programma bestaat alleen uit dataflow-instrukties. Zo'n instruktie

specificeert een bewerking op een aantal operanden en bevat de adressen van de instrukties, die de geproduceerde resultaten weer als operand gebruiken. Control flow instrukties zijn er niet meer. De volgorde van uitvoering wordt geheel bepaald door de beschikbaarheid van de operanden. Een instruktie wordt uitgevoerd als alle benodigde operanden beschikbaar zijn. Het enige effekt van het uitvoeren van een instruktie is de consumptie van de operanden en de produktie van de resultaat-gegevens. De instrukties zijn funktioneel, dat wil zeggen dat de waarden van de resultaat-gegevens alleen afhangen van de waarden van de operanden. Er zijn geen "onzichtbare" neveneffecten. Een dataflow-computer is een machine, die speciaal gebouwd is voor het efficiënt uitvoeren van dataflow-programma's. Hij heeft geen expliciete programma-wijzer, maar laat zich leiden door de onderlinge afhankelijkheid van gegevens in het programma.

Dit koncept heeft zijn wortels in de zogenaamde dataflow-analyse, een methode voor het analyseren van programma's voor optimaliserings-doeleinden en semantische verifikatie. Zie [6,16,13] voor vroege referenties.

In het volgende hoofdstuk zal ik nader ingaan op de ontwikkelingen, die tot dataflow-machines en -talen hebben geleid. In hoofdstuk 3 worden de principes van een dataflow-machinetaal behandeld aan de hand van een eenvoudig voorbeeld. In het daaropvolgende hoofdstuk wordt nader ingegaan op een aspekt van de machinetaal, dat bepalend is voor de graad van parallellisme. Hoofdstuk 5 presenteert het ontwerp van een realistische dataflow-machine. De laatste hoofdstukken zijn gewijd aan hogere programmeertalen voor dataflow-computers.

2. MOTIVATIE

Zoals al eerder in dit colloquium [15] is opgemerkt, zijn nieuwe ontwikkelingen op het grensgebied tussen machine-architektuur en hogere programmeertalen veelal het gevolg van impulsen uit zowel de hardware- als de software-hoek. Dataflow vormt hierop geen uitzondering.

2.1. Hardware

De drijvende kracht vanuit de hardware is in dit geval het verlangen om een computer te bouwen, die zijn snelheid haalt uit de parallelle executie van onafhankelijke delen van een programma. Het verband tussen prijs en snelheid van een processor is verre van lineair. De koper van een microprocessor krijgt relatief gezien veel meer rekenkracht voor zijn geld dan de koper van een supersnelle maxiprocessor. Met andere woorden: hoe duurder de processor, hoe ongunstiger de prijs/prestatie-verhouding. Het verschil loopt op tot één of twee ordes van grootte. Het is voor de hand liggend om te zoeken naar een architektuur, waarbij een groot aantal microprocessoren, of speciaal ontworpen LSI-komponenten van een soortgelijke complexiteit, met elkaar verbonden zijn door een eenvoudige kommunikatie-struktuur. Als we een organisatie kunnen vinden, die garandeert dat op ieder moment een redelijk deel van de processoren bezig is met relevante berekeningen, dan hebben we een machine, die veel sneller is dan een konventionele machine van gelijke prijs.

Voor sommige toepassingen is zoiets vrij gemakkelijk te realiseren. Er zijn relatief goedkope machines op de markt, die een enorme snelheid behalen door gebruik te maken van specifieke eigenschappen van algorithmen in een klein toepassingsgebied. Voorbeelden hiervan zijn Fast Fourier Transform machines of array-processoren, die speciaal zijn ontworpen voor een aantal standaard bewerkingen op vectoren. Deze voorbeelden vallen in de zogenaamde Single Instruction Multiple Data stream (SIMD) klasse van parallelle processoren: dezelfde bewerking wordt uitgevoerd op een aantal verschillende gegevens-stromen. Doordat er maar één instruktie-stroom is hebben we maar één programma-wijzer nodig. Het aantal toepassingen, dat zich voor implementatie op een SIMD-computer leent, is helaas beperkt. Een algemeen bruikbare parallelle processor zal meerdere onafhankelijke instruktiestromen en dus geen centrale programma-wijzer hebben.

Om een parallelle processor met een ruim toepassingsgebied te kunnen realiseren, is het wachten echter op een algemene methode om een tijdrovende berekening in delen te splitsen, die aan onafhankelijke processoren uitgedeeld kunnen worden. Er bestaat al lang behoefte aan zo'n methode, maar aan die behoefte bleek veel moeilijker te voldoen dan aanvankelijk werd verondersteld. Het probleem is dat

parallellisme en het von Neumann koncept fundamenteel met elkaar in strijd zijn. Het dataflow koncept is een veelbelovend alternatief.

2.2. Software

In de software-hoek zijn er ontwikkelingen, die in dezelfde richting wijzen. Met de "software crisis" hebben we al zo lang moeten leven, dat we het nauwelijks nog een crisis kunnen noemen. De problemen zijn er sindsdien nauwelijks kleiner op geworden. Vooral grote software-projekten met een lange levensduur zitten in de problemen. De programma's zijn onoverzichtelijk en slecht gestruktureerd, het gedrag is soms onvoorspelbaar, fouten zijn moeilijk op te sporen. Eén van de oplossingen (gestruktureerd programmeren) is een ontwerp- en programmeer-discipline waarmee de meest voorkomende fouten voorkomen kunnen worden. Een programma moet bestaan uit modulen, die zo veel mogelijk onafhankelijk zijn en die elkaar expliciet informatie doorgeven. Het effekt van het uitvoeren van een moduul moet zoveel mogelijk zichtbaar zijn. Onzichtbare neveneffekten zijn gevaarlijk. Om deze redenen zijn onder andere goto's en globale variabelen naar de strafbank verwezen. Gezocht wordt naar talen die het toepassen van deze principes gemakkelijker maken en het zondigen ertegen bemoeilijken. In een dataflow-taal zijn een aantal van de gesignaleerde problemen uitgebannen. Omdat er geen programma-wijzer is, bestaan er ook geen goto's. Globale variabelen zijn verdwenen, omdat procedure's geen neveneffekten mogen hebben. Of in zo'n taal ook nog geprogrammeerd kan worden, valt natuurlijk nog te bezien.

3. MACHINETAAL

We zagen in het eerste hoofdstuk al dat een dataflow-instruktie de adressen bevat van zijn opvolgers, dat wil zeggen van de instrukties die het geproduceerde resultaat als operand gebruiken. We kunnen een dataflow-programma daarom zien als een gerichte graaf: de knopen zijn de instrukties en de kanten zijn éénrichtings-datapaden. Iedere instruktie heeft één of meer invoerpaden en één of meer uitvoerpaden. De gegevens, die over de paden lopen, worden tokens genoemd. De uitvoering van een programma wordt geheel door de Laad-regel en de Executie-regel bepaald. Een instruktie heet geladen als, op alle benodigde invoerpaden een token aanwezig is

(Laad-regel). Als een instruktie geladen is dan gaat hij, op een niet gespecificeerd moment, over tot executie, waarbij alle tokens op de invoerpaden verdwijnen en na verloop van tijd een token op alle uitvoerpaden verschijnt (Executie-regel). De waarde van de uitvoertokens is een funktie van die van de invoertokens. De funktie wordt bepaald door het soort instruktie (aangegeven door de zogenaamde opkode).

In dit hoofdstuk worden de principes van een dataflow-machine behandeld. Voor dat doel wordt, bij wijze van voorbeeld, een vereenvoudigde machine met een bijbehorende machinetaal geïntroduceerd. Echte dataflow-machines verschillen aanzienlijk hiervan en van elkaar, zoals we in de volgende hoofdstukken nog zullen zien.

3.1. Een Eenvoudige Graaf

We nemen als voorbeeld de berekening van de wortels van een vierkantsvergelijking

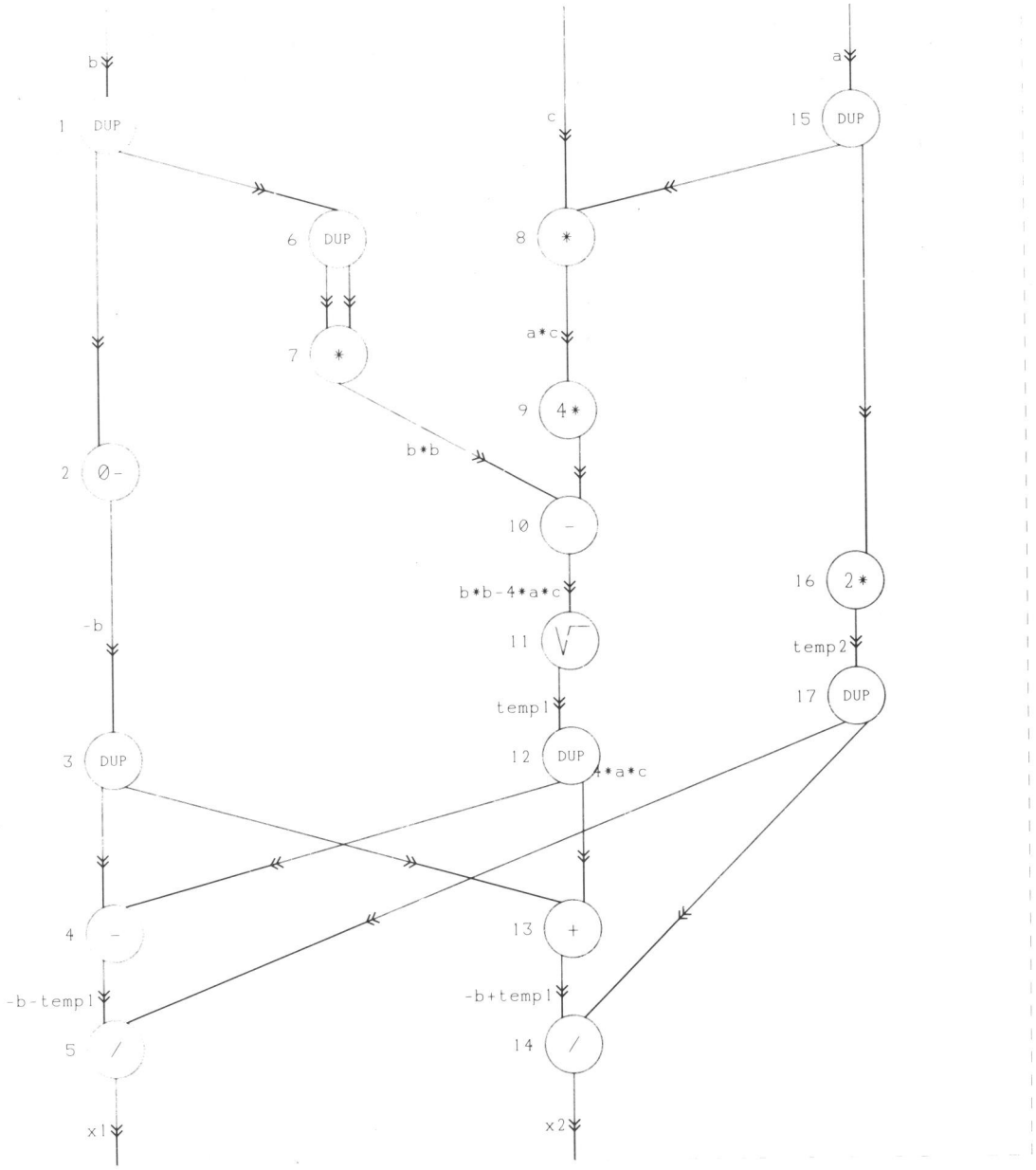
$$ax^2 + bx + c = 0$$

$$\langle x1, x2 \rangle = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

In konventionele vorm:

```
templ := wortel( b*b - 4*a*c);
temp2 := 2*a;
x1    := (-b + templ) / temp2;
x2    := (-b - templ) / temp2;
```

De dataflow-graaf ziet er als volgt uit:



Ik heb hier aangenomen dat alle rekenkundige operaties slechts één uitvoerkanaal hebben. Er zijn daarom speciale dupliceer operaties (DUP) nodig, die een duplikaat van het invoertoken op de beide uitvoerkanalen plaatsen. De benodigde konstanten zijn in de instruktiekode opgenomen. De "4*" instruktie heeft dus maar één invoertoken nodig. Bij de paden zijn namen vermeld om vergelijking met het voorafgaande en het volgende te vergemakkelijken. Deze namen horen niet bij de dataflow-graaf.

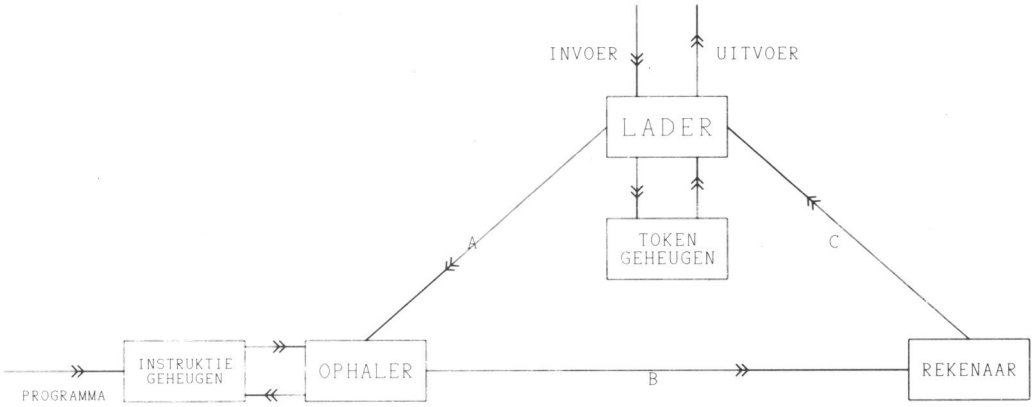
De machinekode voor deze graaf zou er als volgt uit kunnen zien:

adres	instruktie			
	opkode	konstante	opvolgers	
1	DUP		2/1	6/Ø
2	MIN	Ø/Ø	3/Ø	
3	DUP		4/Ø	13/Ø
4	MIN		5/Ø	
5	DEEL		UITVOER	
6	DUP		7/Ø	7/1
7	MAAL		10/Ø	
8	MAAL		9/1	
9	MAAL	4/Ø	10/1	
10	MIN		11/Ø	
11	WORTEL		12/Ø	
12	DUP		4/1	13/1
13	PLUS		14/Ø	
14	DEEL		UITVOER	
15	DUP		8/1	16/1
16	MAAL	2/Ø	17/Ø	
17	DUP		5/1	14/1

De kodering van de opvolgers is aangegeven als opvolgeradres/invoerkanaal. De invoerkanalen hebben nummers Ø en 1. Als de instruktie een konstante bevat, komt deze in de plaats van een invoerkanaal. Dit is aangegeven in de kolom "konstante" met waarde/invoerkanaal. Het is interessant om te zien dat de volgorde van de instrukties niet van belang is. We kunnen de instrukties willekeurig permuteren (mits we natuurlijk de opvolger-adressen mee muteren) zonder de betekenis van het programma te wijzigen. De uitvoering van het programma wordt geheel door de Laad- en de Executie-regel bepaald.

3.2. Een Vereenvoudigde Machine

In de volgende figuur is schematisch een hypothetische dataflow-machine weergegeven.



Het programma in machinekode wordt in het instruktiegeheugen geladen. Via het invoerkanaal worden gegevens ingebracht in de vorm van token-pakketten. Zo´n pakket bestaat uit een waarde en zijn bestemming. De Lader accepteert zo´n pakket en kijkt of het invoertoken de geädresseerde instruktie in de Laad-toestand brengt. Zo nee, dan wordt het token opgeborgen, zo ja, dan worden het instruktie-adres en de bijbehorende token(s) in de vorm van een invoerpakket naar de Ophaler gestuurd. Deze voegt de invoerwaarden samen met een kopie van de instruktie tot een kommando-pakket. De Rekenaar voert de instruktie uit, die door de opkode wordt gespecificeerd, gebruik makend van de invoerwaarden. Met ieder opvolger-adres wordt een kopie van de uitgerekende waarde gekombineerd tot een tokenpakket en doorgestuurd naar de Lader. Als de Lader de bestemming UITVOER tegenkomt dan wordt het token naar het uitvoerkanaal gestuurd.

Over de verbindingskanalen lopen dus drie soorten pakketten:

A tokenpakket <adres, waarde>
B invoerpakket <adres, waarde(n)>
C kommandopakket <opkode, waarde(n), adres(sen)>

Stel dat we in ons voorbeeld de wortels willen uitrekenen voor de waarden 2.5, 6 en 2 voor respektievelijk a, b en c. De invoer zou bestaan uit de volgende tokenpakketten:

```
<15/0, 2.5 >
< 1/0, 6.0 >
< 8/0, 2.0 >
```

Omdat instrukties 1 en 15 maar één operand nodig hebben, zijn zij nu geladen. De Lader stuurt de volgende invoerpakketten naar de Ophaler:

```
< 15, 2.5 >
<  1, 6.0 >
```

Instruktie 8 heeft twee operanden nodig en is dus nog niet geladen. Het desbetreffende token wordt in het token-geheugen opgeborgen. De Ophaler stuurt op zijn beurt de volgende kommando-pakketten de baan op:

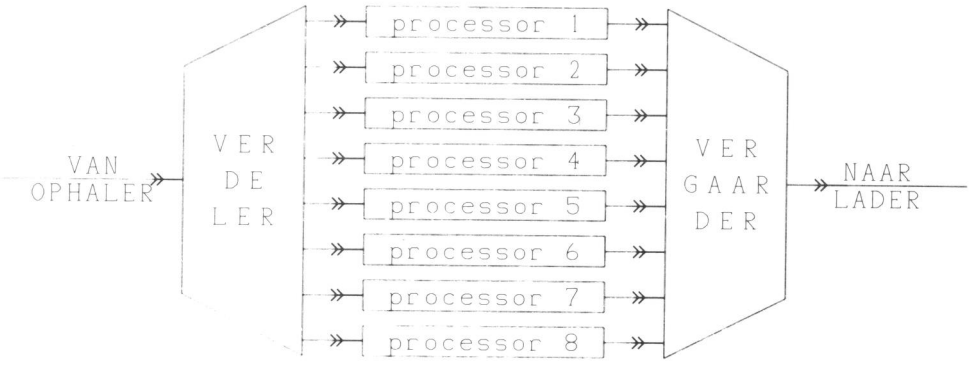
```
< DUP, 2.5, 8/1, 16/1 >
< DUP, 6.0, 2/1,  6/0 >
```

De Lader ontvangt vervolgens van de Rekenaar:

```
<  8/1, 2.5 >
< 16/1, 2.5 >
<  2/1, 6.0 >
<  6/0, 6.0 >
```

Nu zijn de instrukties 2, 6, 8 en 16 geladen. Deze worden uitgevoerd en brengen op hun beurt weer nieuwe instrukties in de Laad-toestand. Uiteindelijk zal de executie van instrukties 5 en 14 tokenpakketten met bestemming UITVOER opleveren. De hier geschetste executie-volgorde van de instrukties is slechts één van de mogelijkheden. Omdat de instrukties echter funktioneel zijn hebben veranderingen in de volgorde geen invloed op het resultaat. Daarom bevat ieder pakket dat over één van de kanalen A, B of C loopt alle informatie, die nodig is voor een bepaalde aktie. De pakketten hebben geen onderlinge verwijzingen of andere afhankelijkheden. In een korrekt dataflow-programma is de volgorde, waarin de pakketten worden afgehandeld

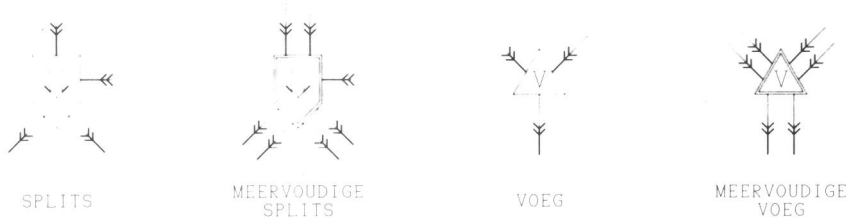
onbelangrijk. De kanalen kunnen daarom van buffers worden voorzien en de drie modulen kunnen asynchroon opereren. Dit heeft de belangrijkste konsekwentie voor de Rekenaar. Omdat alle dataflow-instrukties funktioneel zijn, zijn de geproduceerde tokenpakketten volledig bepaald door het bijbehorende kommandopakket. De Rekenaar kan daarom zonder problemen gerealiseerd worden door middel van een aantal parallelle processoren, die ieder een kommandopakket accepteren, de berekening uitvoeren en een tokenpakket produceren (zie onderstaande figuur). De Verdeler en de Vergaarder zijn simpele schakelkomponenten die pakketten naar de eerste vrije processor sturen, respektievelijk pakketten van alle processoren accepteren. Omdat de Ophaler en de Lader over een geheugen beschikken, waarvan bij de behandeling van een pakket gebruik wordt gemaakt, is het veel moeilijker om deze modulen ook parallel uit te voeren. Deze vormen dan ook potentiële bottlenecks.



3.3. Overige Instrukties

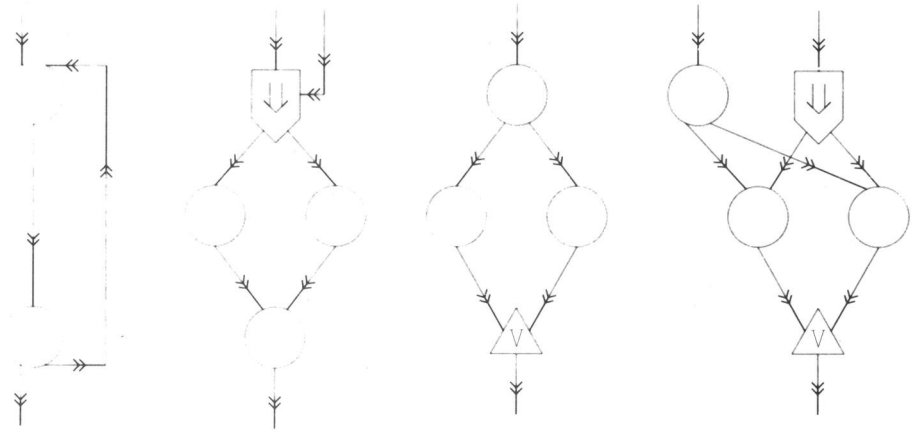
Met de soort instrukties die we tot nu toe hebben gezien kunnen we alleen zeer primitieve grafen maken, overeenkomend met programma's in een konventionele taal, waarin alleen maar simpele variabelen, toekenning en arithmetische operaties voorkomen. Ieder datapad in de graaf komt overeen met een simpele variabele. Het plaatsen van een token op een uitvoerpad komt overeen met de konventionele toekenningsoperatie. Voor interessante programma's, met konditionele expressies en iteraties, hebben we allereerst het ekwivalent van de konditionele sprong nodig. In de volgende figuur zien we de SPLITS operator. Deze stuurt een kopie van zijn invoertoken naar slechts één van zijn uitvoerkanalen. De

splitsrichting wordt bepaald door de waarde van het logische token op het rechter invoerkanaal. De executie-regel van SPLITS wijkt af van de gangbare, omdat slechts op één van de twee uitvoerpaden een token verschijnt. De meervoudige SPLITS wordt gebruikt als meerdere datapaden door dezelfde test worden gesplitst. Om de SPLITS te sturen hebben we nu ook operatoren nodig, die logische waarden produceren (relationele operaties) en manipuleren (logische operaties).

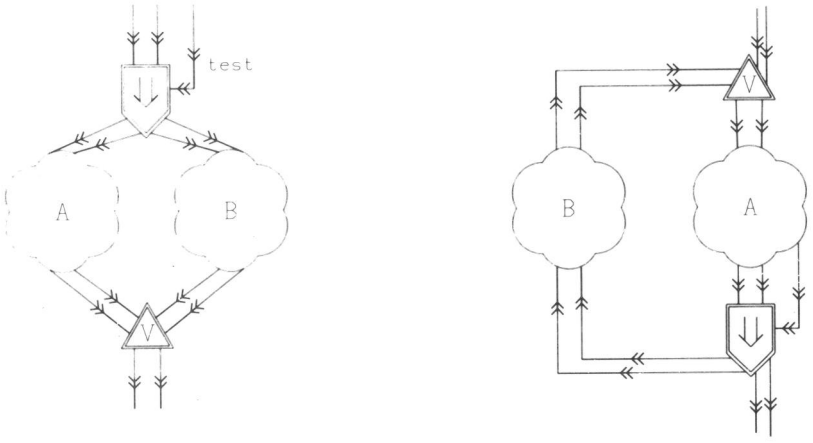


Het komplement van de SPLITS is de VOEG operatie. Deze accepteert een token van zijn linker of rechter invoerpad en plaatst een duplikaat op het uitvoerkanaal. VOEG heeft een afwijkende Laad-regel: de instruktie is geladen zodra alle invoerpaden aan de linkerkant òf alle invoerpaden aan de rechterkant een token bevatten. Als zowel tokens op linker als rechter paden binnenkomen, dan is de volgorde van de uitvoertokens niet gedetermineerd.

Een procedure P is een graaf die wordt gemarkeerd door de twee instrukties INGANG P en UITGANG. Deze procedure wordt aangeroepen door de AANROEP P instruktie. Als deze instruktie tot executie overgaat, wordt een kopie gemaakt van de met P korresponderende graaf, de invoertokens worden op de invoerpaden van INGANG P geplaatst en de uitvoerpaden van de UITGANG instruktie worden verbonden met de uitvoerpaden van de AANROEP P instruktie. Omdat bij iedere aanroep een nieuwe kopie wordt gemaakt, kunnen de verschillende aanroepen van procedures geen invloed op elkaar hebben. Recursie krijgen we er zodoende gratis bij.



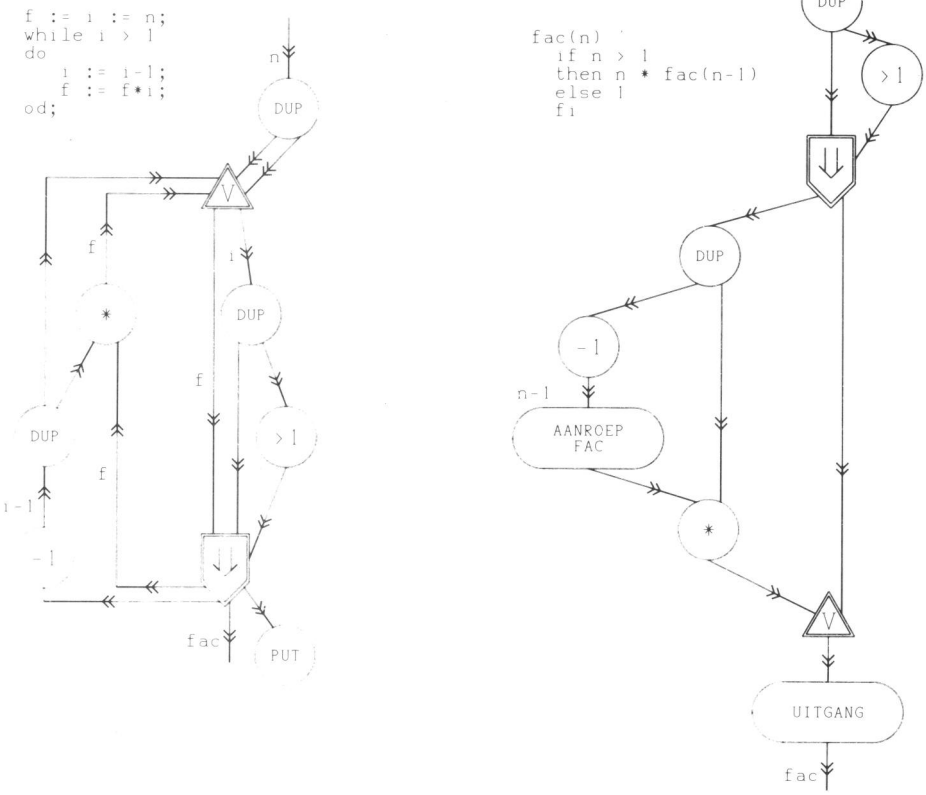
Omdat SPLITS en VOEG een afwijkende Laad-, respektievelijk Executie-regel hebben, moeten we oppassen hoe we deze operatoren in een programma verwerken. Een willekeurige cyclus in een gerichte graaf geeft ook problemen. Bovenstaande figuur laat een aantal narigheden zien. Een veilige manier om SPLITS, VOEG of een cyclus te gebruiken, zien we in de volgende figuur, waar de effekten van het afwijkende gedrag van SPLITS en VOEG elkaar opheffen:



if test then A else B fi

while A do B od

Hieronder zien we als voorbeeld van iteratie de graaf om n! uit te rekenen met daar naast dezelfde berekening in recursieve vorm:



4. KRINGLOOP EN KLEUREN

In het laatste hoofdstuk zagen we twee manieren om een stuk programma meer dan eens te gebruiken: iteratie en recursie. Iteratie en recursie zijn voorbeelden van een kringloop. Een programma zonder kringloop is moeilijk denkbaar: de programmeur zou zijn tijd verdoen met het schrijven van instrukties die slechts één keer worden uitgevoerd. In een dataflow-machine vereist kringloop speciale voorzieningen. Worden er namelijk tokens geïnjecteerd in een subgraaf waar nog tokens in achter gebleven zijn van een vorige ronde van de kringloop, dan zouden de tokens behorende bij verschillende rondes met elkaar verward kunnen worden, met alle gevolgen van dien. De machine uit het vorige hoofdstuk gebruikt twee verschillende voorzieningen. Bij

iteratie vormen de meervoudige SPLITS- en de meervoudige VOEG-instruktie een sluis, die ervoor zorgt dat tokens van een nieuwe ronde pas doorgelaten worden als de vorige geheel beëindigd is. Bij recursie wordt een verse kopie van de gehele graaf gemaakt. In het algemeen bestaan er drie methoden om dergelijke interferenties van tokens te vermijden:

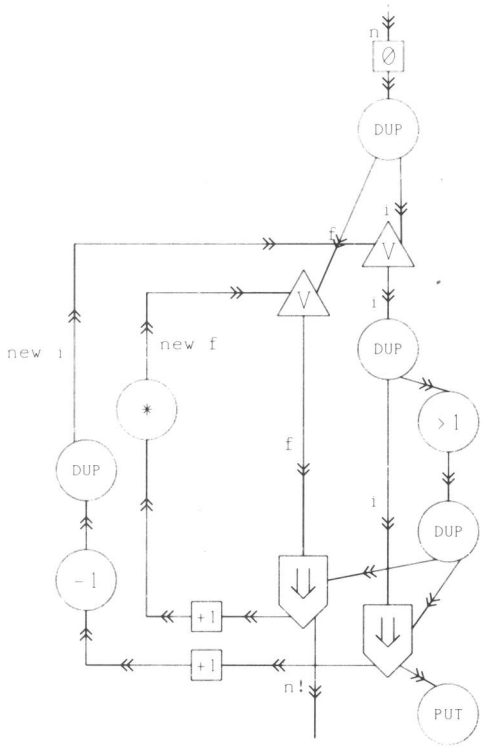
SLUIS De subgraaf wordt omsloten door twee poorten en tokens voor een nieuwe ronde worden slechts toegelaten als alle tokens van een vorige ronde zijn verdwenen[23].

KOPIE Aan het begin van een nieuwe ronde wordt de subgraaf gekopiëerd[20].

KLEUR Iedere ronde krijgt een unieke identifikatie (kleur) en alle tokens die bij één ronde horen worden gemarkeerd met deze kleur. De Laad-regel wordt gewijzigd. Een instruktie is nu geladen als op alle invoerpaden een token met dezelfde kleur aanwezig is[2,7].

De SLUIS oplossing heeft een drastische beperking van het parallellisme tot gevolg: iedere ronde moet wachten op de voltooiing van de vorige. Bij KOPIE en KLEUR kunnen wel verschillende ronden tegelijkertijd aktief zijn. Deze twee laatste oplossingen bieden daarom de beste mogelijkheden om een dataflow-machine optimaal te benutten. KOPIE echter vereist een enorm instruktiegeheugen, omdat voor iedere ronde de volledige subgraaf moet worden gekopiëerd, inclusief de instrukties die niet worden gebruikt. Denk eens aan een grote recursieve procedure met veel conditionele instrukties. Een instruktie zou eigenlijk pas gekopiëerd moeten worden op het moment dat er invoertokens voor arriveren. De administratie die hiervoor nodig is, komt overeen met die voor het KLEUR mechanisme. We kunnen het KLEUR-mechanisme ook zien als een efficiënte implementatie van KOPIE. Bij KOPIE loopt maar één token over een datapad. Nadat het token gepasseerd is verdwijnt het datapad. Bij KLEUR kunnen over een datapad meerdere tokens lopen, maar nooit meerdere van dezelfde kleur.

Ter illustratie een voorbeeld van de iteratie voor n! met kleuren geïmplementeerd:



De vierkante hokjes zijn operatoren die alleen de kleur van een token wijzigen. De twee operatoren in dit voorbeeld zetten de kleur op Ø of hogen hem op. De meervoudige SPLITS en VOEG zijn door hun enkelvoudige broertjes vervangen, want we hebben geen SLUIS meer nodig. Als nu de vermenigvuldigoperatie veel langer duurt dan alle andere operaties, dan kan de rij tokens op datapad i (met waarden n-1,n-2,...,1) al geproduceerd zijn voordat de eerste vermenigvuldiging is voltooid. Al deze tokens i hebben als bestemming de vermenigvuldig instruktie, maar ze hebben allemaal een verschillende kleur. Pas als weer een f beschikbaar komt, wordt een token i met de juiste kleur opgezocht en naar de vermenigvuldiginstruktie gestuurd. In dit voorbeeld wordt het dure gedeelte van de berekening dus weer sequentieel uitgevoerd, maar dit is te wijten aan het inherent sequentiële karakter van dit algorithme. In het algorithme:

```
fac(n)
   f(1,n)

f(min,max)
   if (min = max)
      min
   else
      f(min, [(min+max)/2]) * f([(min+max)/2]+1, max)
```

verdeelt f het aantal vermenigvuldigingen steeds in tweeën. Hierdoor kunnen maximaal n ronden tegelijkertijd aktief zijn. Dit illustreert het betreurenswaardige feit, dat het maximaal benutten van de capaciteit van een parallelle machine niet alleen een geëigende taal vereist, maar in vele gevallen ook geëigende algorithmen. In het algemeen liggen in het parallel uitvoeren van kringloopstappen de grootste mogelijkheden voor het benutten van parallellisme.

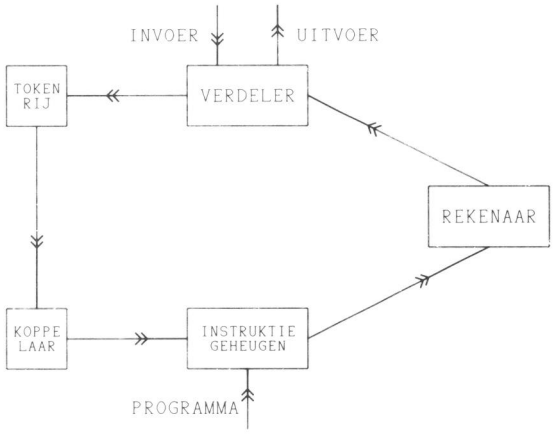
Het kleurmechanisme, zoals in dit voorbeeld is geschetst, is een vergaande vereenvoudiging. Als we rekening willen houden met geneste iteraties, meervoudige recursie en alle kombinaties hiervan dan hebben we een veel complexer schema nodig. In het volgende hoofdstuk zien we daar een voorbeeld van. Door het kleurmechanisme niet geheel onzichtbaar te maken, maar in de machinetaal primitieven op te nemen, die kleuren van tokens kunnen manipuleren, opent zich plotseling een scala van interessante mogelijkheden, waar ik in hoofdstuk 6 nog op terug zal komen.

5. DE MANCHESTER DATAFLOW MACHINE

In de afgelopen vijf jaar zijn er een aantal ontwerpen van dataflow-machines gepubliceerd en zeker twee daarvan zijn inmiddels gerealiseerd. Een daarvan zal ik hier uitvoerig beschrijven om een indruk te geven van de complexiteit van een realistische machine. Voor de overige machines zie[21,23,5,2].

Zeker vier onderzoeksgroepen in Engeland houden zich op dit moment met dataflow bezig. Op Manchester University wordt een prototype geconstrueerd, dat in 1981 operationeel moet zijn. Het is de enige machine waarvan een gedetailleerd ontwerp is gepubliceerd[24,10]. De machine werkt met kleuren om een zo groot mogelijk parallellisme uit een kringloop te

halen. Iedere nieuwe ronde van een kringloop krijgt een unieke kleur toegewezen, die hem onderscheidt niet alleen van de andere ronden van de kringloop maar ook van andere kringlopen. Arvind & Gostelow [2] hebben een schema voorgesteld, waarbij een instruktie uit de kleur van een invoertoken een nieuwe unieke kleur kan berekenen zonder van globale informatie gebruik te maken. Helaas groeit volgens dit schema het aantal bits dat nodig is om de kleur te koderen lineair met de recursiediepte. In Manchester is een variatie bedacht, die wel gebruik maakt van globale informatie maar veel minder geheugen gebruikt[11]. De kleur bestaat uit drie velden van elk twaalf bits, één voor een index in een datastruktuur (array van vaste lengte), één voor de teller in een iteratie en één voor een unieke procedure-kleur. De eerste twee kunnen door de instruktie lokaal worden berekend, voor de laatste is een instantie in de machine nodig, die unieke procedure-kleuren uitgeeft. Een iteratie krijgt ook een eigen procedure-kleur om geneste iteraties mogelijk te maken.



Bovenstaande figuur geeft de architektuur van de machine weer. De invoergegevens komen binnen in de Verdeler als een tokenpakket (95 bits), bestaande uit kleur (36), bestemming (22) en waarde (37). De Tokenrij is een FIFO buffer met plaats voor 16K van deze pakketten. De Koppelaar is het interessantste gedeelte van deze machine. Hij houdt de Laadregel bij en moet dus voor ieder binnenkomend token bepalen of de bestemmings-instruktie nu geladen is en in dat geval de eventueel bijbehorende invoertokens bijvoegen. Een vereenvoudiging is, dat instrukties met meer dan twee invoerpaden al op hoger niveau zijn gesplitst in eenvoudiger instrukties,

zodat in de machinetaal alleen operaties met één of twee invoerpaden bestaan. Het aantal invoerpaden van een instruktie wordt aangegeven door een bit in het bestemmingsveld. "Alleenstaande" tokens worden direkt doorgestuurd. Voor de overige tokens wordt gekeken of de partner al aanwezig is. Zo ja, dan wordt deze opgehaald en samen met de andere doorgestuurd naar het instruktie-geheugen. Als de partner nog niet aanwezig is, dan wordt het token opgeborgen. Voor dit opzoeken simuleert de Koppelaar een associatief geheugen door middel van hardware hash-tabellen. De zoek-sleutel wordt gevormd door 54 bits van de bestemming en de kleur. De Koppelaar kan 16K tokens herbergen. Het instruktie-geheugen is gesegmenteerd en biedt plaats aan 8-16K instrukties (68 of 34 bits). De Rekenaar bestaat uit twee trappen. De eerste trap voert de instrukties uit die globale informatie gebruiken ("Genereer Procedure Kleur"). De tweede trap is opgebouwd uit 15 elementen. Ieder element is een gemicroprogrammeerde bit-slice processor.

De machine is niet vrij van deadlock. De Koppelaar kan namelijk vol raken. Iedere instruktie konsumeert één of twee tokens en produceert één of twee tokens. Instrukties kunnen daarom in drie klassen worden verdeeld: expanderend, inkrimpend of neutraal, al naar gelang ze het aantal in omloop zijnde tokens vermeerderen, verminderen of gelijk laten. Wanneer er nu tijdens een bepaalde fase van de uitvoering van een programma veel expanderende instrukties worden uitgevoerd (bijvoorbeeld door het simultaan starten van vele ronden van een uitgebreide kringloop), dan groeit het aantal tokens in het systeem en de Koppelaar en de Tokenrij raken steeds voller. Als deze beiden tot hun limiet zijn gevuld, zal de Rekenaar zijn tokens niet meer kwijt kunnen en de machine komt abrupt tot stilstand. Dit zal niet snel gebeuren, omdat naast de 32K tokens die in de Tokenrij en de Koppelaar terecht kunnen, ook nog tokens naar een overstroom-bak kunnen afvloeien. Hoe groot deze bak is, wordt niet vermeld, maar het is wel duidelijk, dat de snelheid van de Koppelaar evenredig daalt met het aantal overgestroomde tokens. De volgende modifikatie op het ontwerp zou een aanzienlijke verlichting kunnen brengen. In ieder token-pakket wordt een extra bit opgenomen om aan te kunnen geven of de opvolger-instruktie expanderend, inkrimpend of neutraal is. De Tokenrij wordt gewijzigd, zodat hij de tokens nu niet meer in de oorspronkelijke volgorde aan de Koppelaar doorgeeft, maar

alle expanderende of inkrimpende of neutrale tokens vóór laat gaan. De keuze wordt bepaald aan de hand van signalen, die de aktiviteit van de andere delen van de machine aangeven. Bij een onderbezette Rekenaar krijgen expanderende tokens voorrang, terwijl als de overstroombak tokens bevat bij voorkeur inkrimpende tokens worden doorgestuurd.

Als de machine optimaal werkt dan wordt iedere 200-300 nsec een instruktie uitgevoerd. Het geheugen, dat in de diverse onderdelen wordt gebruikt (zo'n 460 Kbyte) heeft een snelheid van 100 nsec. De cyclustijd van het microprogramma is 200 nsec. Het zijn dus snelle komponenten die worden gebruikt (bipolair). De maximum snelheid komt toch niet boven de vijf miljoen instrukties per seconde (5 MIPS) uit. Een DEC VAX-11 heeft bijvoorbeeld een snelheid van ongeveer 1 MIPS. Hierbij moeten we nog bedenken dat, onder andere vanwege de kleuren-administratie, het aantal machine-instrukties in een dataflow-programma groter is dan in een ekwivalent konventioneel programma. Het probleem is, dat belangrijke onderdelen en vooral de Koppelaar niet eenvoudig parallel zijn uit te voeren.

Het hier beschreven prototype is echter slechts een deel van de uiteindelijk te bouwen machine. Deze gelaagde machine zal bestaan uit een aantal ringen, die ieder vrijwel hetzelfde zijn als het prototype zoals dat nu wordt gebouwd. Alleen de Verdeler is anders. De Verdelers van alle ringen tesamen vormen een schakelnetwerk, waardoor tokens tussen de verschillende ringen kunnen worden uitgewisseld. De route van de tokens door het netwerk wordt bepaald door het bestemmingsveld. Het verkeer door het netwerk, en dus de belasting van dit centrale onderdeel van de machine, wordt dan bepaald door de verdeling van de instrukties over de afzonderlijke ringen. Nog veel onderzoek is nodig om tot een goede strategie voor deze verdeling te komen. Er is overigens weinig bekend over deze gelaagde machine.

6. FUNKTIONALITEIT EN STROMEN

Voor ik de mogelijkheden voor hogere programmeertalen voor dataflow computers behandel, wil ik eerst wat dieper ingaan op een centraal begrip in deze context: funktionaliteit. Alle operatoren in de machinetaal in hoofdstuk 3 vertonen een funktioneel gedrag, dat wil zeggen dat de waarden van de uitvoertokens geheel bepaald worden door de waarden

van de invoertokens. Dit geldt ook voor procedure-aanroepen: de uitvoertokens van een AANROEP instruktie zijn geheel bepaald door de waarden van de invoertokens. Alle procedures zijn dus funkties en alle informatie die een procedure nodig heeft moet in de invoertokens besloten liggen. De procedure heeft geen toegang tot globale informatie en hij heeft ook geen intern geheugen waardoor informatie van één aanroep aan een andere aanroep kan worden doorgegeven. Dit is tegelijkertijd de kracht en de zwakte van dataflow-talen. Omdat aanroepen elkaar niet ondergronds kunnen beïnvloeden is de onderlinge afhankelijkheid van procedure-aanroepen direkt uit de dataflow-graaf af te lezen: De Laad-regel en de Executie-regel zijn voldoende om een deterministisch gedrag te verzekeren. De strikte eis van funktionaliteit vormt echter ook een grote handicap, want niet-funktionele procedures zijn bij het programmeren vaak erg handig en soms zelfs onmisbaar.

Een voorbeeld van het eerste is een procedure NAMEN in een vertaler die de namentabel beheert. Verspreid door de vertaler komen twee soorten aanroepen voor: NAMEN(SCHRIJF, NAAM) waardoor met naam een adres wordt geassocieerd, dat in de tabel wordt opgeslagen en NAMEN(LEES, NAAM) die als waarde het met NAAM geassocieerde adres oplevert. De SCHRIJF aanroep heeft een neveneffekt waardoor de LEES aanroep niet meer funktioneel is, maar van voorgaande aanroepen van NAMEN afhangt. De in dataflow-talen meest gebruikelijke oplossing is om de waarde van het interne geheugen (de tabel) als extra in- en uitvoerwaarden mee te geven. De aanroepen zijn dan van de vorm:

```
<dummy,TABEL52> := NAMEN(SCHRIJF, NAAM, TABEL51);
<adres,dummy  > := NAMEN(LEES   , NAAM, TABEL52);
```

Door iedere SCHRIJF opdracht wordt dus een nieuwe tabel gecreëerd. Voor een complex programma als een vertaler wordt dit al gauw ondoenlijk, omdat de tabel bij iedere wijziging gekopieerd moet worden. De programma's worden onoverzichtelijk, doordat de programmeur voor iedere wijziging een nieuwe naam voor de tabel moet bedenken. Het druist ook in tegen modulariteits-principes: we willen juist het beheer over de namentabel aan een aparte procedure overlaten en we willen dus zeker geen interne informatie exporteren.

Hetzelfde probleem komen we op grotere schaal tegen bij

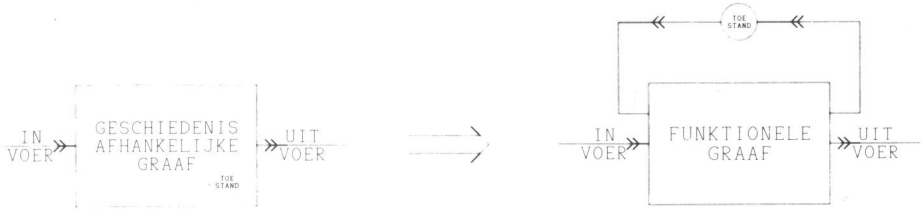
het kommuniceren met een centraal gegevensbestand. De methode van het doorgeven van de oude en nieuwe waarde van het intern geheugen bij iedere aanroep wordt hier geheel onbruikbaar. Het is immers juist één van de voordelen van een centrale bestandsbeheerder dat de programma's, die van het bestand gebruik maken, niet van elkaar op de hoogte hoeven te zijn.

We kunnen deze kommunikatie met een gegevensbestand zien als een vorm van in- en uitvoer, waarbij in het algemeen het probleem zich voordoet, dat de wereld buiten het dataflow-programma niet funktioneel is en de kommunikatie ermee zorgvuldig moet worden gesynchroniseerd. Een goede oplossing voor het in- en uitvoerprobleem kan daarom model staan voor een algemene oplossing voor komponenten in een dataflow-programma met een niet-funktioneel gedrag.

Laten we eerst twee oorzaken van niet-funktioneel gedrag onderscheiden: niet-determinisme en geschiedenis-afhankelijkheid. Een tekst-editor, die gebruikt wordt om een nieuwe tekst te creëren (het modificeren van een bestaande tekst laten we hier even buiten beschouwing), vertoont een niet-funktioneel gedrag: De invoer opdracht "TYP LAATSTE REGEL" leidt niet altijd tot dezelfde uitvoer, maar is afhankelijk van voorgaande invoer. Een goede editor is echter wel consistent: als we later dezelfde reeks opdrachten invoeren, verwachten we hetzelfde resultaat. Op dit niveau is de editor dus wel funktioneel. Meerdere gebruikers kunnen er dan ook simultaan mee werken als de verschillende reeksen opdrachten maar uit elkaar worden gehouden. Zo'n programma noemen we deterministisch en geschiedenis-afhankelijk. Een echte random-generator is ook niet funktioneel: "TYP RANDOM-GETAL" geeft meestal niet dezelfde uitvoer. De uitvoer wordt echter niet door voorafgaande invoer bepaald, maar door andere, buiten ons model vallende, faktoren. Zo'n programma heet niet-deterministisch.

Terug naar de dataflow-graaf. Van geschiedenis-afhankelijke grafen willen we kunnen zeggen op welk niveau zo'n graaf funktioneel is. Dit bepaalt namelijk welke tokens onafhankelijk van elkaar en dus parallel kunnen worden verwerkt (vergelijk de opdrachten van verschillende gebruikers van een editor) en welke op elkaar zullen moeten wachten. Deze afhankelijkheid kunnen we beschrijven met behulp van zogenaamde stromen. Een stroom is een geordende

reeks tokens, die over hetzelfde datapad loopt. Een deterministische geschiedenis-afhankelijke graaf heeft een intern geheugen, d.w.z. de graaf bevat een cyclus met een token dat niet bij iedere aanroep dezelfde initiële waarde heeft. Het gedrag is ekwivalent met dat van een funktionele graaf waaraan we de waarde van het geheugen (de toestand) als extra in- en uitvoer meegeven (externe cyclus).



De toestand (het rondcirkelende token) wordt geïnitialiseerd door een speciaal invoer-token, dat het Stroom-Scheidings-Token (SST) wordt genoemd. Een stroom wordt omsloten door SST's. Een deterministische graaf is funktioneel op het niveau van een stroom: bij dezelfde invoerstroom hoort altijd dezelfde uitvoer. Tokens die tot verschillende stromen behoren, kunnen onafhankelijk van elkaar (en dus parallel) behandeld worden. We moeten dan wel zorgen, dat we met iedere stroom een apart intern geheugen associeren. Om het parallellisme niet onnodig te beperken willen we ook nog graag, dat de tokens, die tot één stroom behoren, alleen dan sequentieel worden verwerkt als het strikt noodzakelijk is (dat wil zeggen binnen een geschiedenis-afhankelijke graaf). Onafhankelijke bewerkingen op de individuele tokens moeten parallel worden uitgevoerd.

Een uitgebreid KLEUR-mechanisme kan aan deze tegenstrijdige eisen voldoen. Zoals we in hoofdstuk 4 zagen, zorgen de kleuren voor maximaal parallellisme bij kringlopen. Door nu in de machinetaal operaties op kleuren toe te voegen kan de scheidslijn tussen parallel en sequentieel worden beïnvloed. Introduktie van een token met een nieuwe kleur in een graaf is ekwivalent met het kopiëren van die graaf. Twee tokens met verschillende kleur kunnen immers niet met elkaar gekoppeld worden, dus kunnen ze elkaar ook niet beïnvloeden. Dit geldt natuurlijk alleen als de graaf geen veranderingen in de kleur aanbrengt. Als de tokens van een stroom van de geschiedenis-afhankelijkheid van een graaf gebruik moeten

maken, dan zullen ze allemaal dezelfde instantie van die graaf moeten doorlopen. Met andere woorden: ze moeten allemaal dezelfde kleur krijgen, maar ze moeten toch onderscheiden worden van de tokens van een andere stroom. Dit wordt op de volgende manier bewerkstelligd. Een deel van de kleur heeft de funktie van rangnummer-veld. Opeenvolgende tokens van een stroom hebben opeenvolgende rangnummers en overigens dezelfde kleur. De geschiedenis-afhankelijke graaf wordt omsloten door een stroomsluis bestaande uit een STROOM-IN en een STROOM-UIT instruktie. De STROOM-IN instruktie heeft tot taak de binnenkomende tokens in volgorde te accepteren en ze, ontdaan van het rangnummerveld, aan de graaf door te geven. De tokens van een stroom hebben dus binnen de graaf een identieke kleur. De STROOM-UIT instruktie accepteert de resultaat-tokens uit de graaf en vult het rangnummer-veld weer in. De STROOM-IN en STROOM-UIT instrukties zijn gesynchroniseerd, dat wil zeggen STROOM-IN levert het volgende token pas af als STROOM-UIT het resultaat van het voorgaande token heeft verwerkt. Hierdoor is deterministisch gedrag verzekerd.

7. HOGERE PROGRAMMEERTALEN

De machinetaal uit hoofdstuk 3 is natuurlijk niet erg geschikt om in te programmeren. De instrukties zijn van een veel te laag niveau en de mogelijkheden voor foute konstrukties zijn legio (zie 3.3). Hetzelfde geldt voor de kleuren manipulatie mechanismen, die in het vorige hoofdstuk zijn beschreven. We hebben een hogere programmeertaal nodig, waarin het "goed" programmeren is en die vrij eenvoudig naar een dataflow-machinetaal te vertalen is. Uit het oogpunt van kontinuïteit en gebruikersgemak zou het de voorkeur verdienen voor deze taal een bestaande hogere programmeertaal te kiezen en een vertaler te schrijven die programma's in die taal geschreven, accepteert en machine-kode voor een dataflow-computer aflevert. Daarvoor moet de vertaler echter (statisch) de volledige data-afhankelijkheid van de statements in een programma vaststellen. Voor de meeste talen (speciaal die met conditionele sprongen) is dit een ingewikkelde procedure en noodzakelijkerwijze zal de vertaler hier en daar een benadering moeten maken. Om determinisme te verzekeren zal dit altijd een pessimistische schatting moeten zijn en het parallellisme in het vertaalde programma zal dan ook verre van optimaal zijn.

Vandaar dat in eerste instantie een aantal nieuwe talen zijn ontstaan die speciaal voor dataflow-machines zijn gemaakt. Behalve de problemen, die we al van het ontwerpen van konventionele talen kennen, dienen zich, bij het ontwerpen van deze dataflow-talen, een aantal nieuwe problemen aan:

- Door de afwezigheid van een centraal geheugen heeft de vertaalde kode geen toegang tot globale informatie.

- Alle dataflow-instrukties zijn funktioneel, de buitenwereld (achtergrond-geheugen, de gebruiker van een interaktief programma) is dat niet. Alle operaties op deze buitenwereld (in- en uitvoer) moeten daarom zorgvuldig worden gesynchroniseerd.

- Om de graad van parallellisme zo hoog mogelijk te houden, moet de taal de juiste konstrukties bevatten. Als een (abstrakt) algorithme wordt omgezet in een programma in deze taal dan mag er geen parallellisme verloren gaan.

- We zijn gewend om bij het programmeren sequentieel te denken. De dataflow-taal moet liefst zo zijn, dat het (mentaal) sequentieel interpreteren van een programma de betekenis niet aantast.

Zeker vijf hogere programmeertalen zijn in de afgelopen jaren speciaal voor dataflow-machines ontworpen. Drie daarvan zal ik in dit hoofdstuk behandelen. Zie voor de overige talen [25,22]. Al deze talen hebben een aantal belangrijke eigenschappen gemeen:

- Het zijn zogenaamde Talen met Eenmalige Toekenning (TET's): aan een variabele kan maar één maal in een programma een waarde worden toegekend. Van variabel en toekennen is daarom nauwelijks nog sprake en men gebruikt dan ook wel de woorden waarde en definitie. De reden voor de eenmalige toekenning is eenvoudig: in een dataflow-machine beschikken we niet over geheugenplaatsen waar naar believen waarden in geschreven of uit gelezen kunnen worden. We hebben alleen maar te maken met datapaden waar, afgezien van kringloop, slechts één waarde over loopt. De paden kunnen we namen geven. Toekenning (definitie) korrespondeert dan met het plaatsen van een token op een uitvoerpad en referentie

korrespondeert met het konsumeren van een token van het invoerpad. Meerdere referenties aan dezelfde naam leiden tot dupliceer (DUP) knopen.

- Om statisch op eenvoudige wijze te kunnen bepalen of aan de regel van eenmalige toekenning is voldaan zijn konditionele statements niet toegestaan. Bij konditionele expressies moet de else tak altijd aanwezig zijn. De definities van meer dan één waarde kunnen door dezelfde konditie worden gestuurd als in:

  <min,max> := if a > b then <b,a> else <a,b> fi

  Dit heeft het nadeel dat een waarde en zijn definiërende expressie ver uit elkaar kunnen komen te staan (om van het komma's tellen nog maar niet te spreken).

- Omdat procedures funktioneel zijn hebben zij geen inwendig geheugen, dat gebruikt kan worden om informatie van een aanroep naar een andere door te geven. Evenmin zijn er globale variabelen. De invoerwaarden en de resultaten zijn strikt gescheiden. Alle parameters zijn call-by-value.

- Een programma bestaat uit een reeks definities (toekenningen), waardoor een naam aan de waarde van een expressie wordt gebonden. Expressies worden, voor zover mogelijk, gelijktijdig geëvalueerd. Alleen als in een expressie aan een naam wordt gerefereerd, wordt zonodig gewacht tot de bijbehorende expressie is geëvalueerd. Wat parallelle executie betreft, is hier dus de situatie precies omgekeerd in vergelijking met konventionele talen. Een konventioneel programma wordt geheel sequentiëel uitgevoerd, tenzij parallelle executie door speciale primitieven wordt aangegeven. Bij dataflow-talen wordt juist alles parallel uitgevoerd tenzij speciale omstandigheden dat verhinderen.

Ik zal de twee voorbeelden uit hoofdstuk 3 in de drie talen weergeven om een idee te geven hoe een programma er uitziet.

7.1. LAPSE

Voor de in hoofdstuk 5 beschreven machine is onlangs op de Manchester University een hogere programmeertaal ontworpen, die de naam LAPSE heeft gekregen[8]. De syntax is op PASCAL gebaseerd.

Het stuk programma, dat de wortels van een vierkantsvergelijking oplost, ziet er, in LAPSE, als volgt uit:

```
decl templ, temp2: real;
 ...
x1 := (-b + templ) / temp2;
x2 := (-b - templ) / temp2;
templ := wortel( b*b - 4*a*c );
temp2 := 2*a;
```

Dit ziet er dus vrijwel hetzelfde uit als in normaal PASCAL, alleen is de volgorde van de statements niet van belang: x1 en x2 worden pas uitgerekend als templ en temp2 zijn berekend.

De elementaire datatypen zijn boolean, integer en real. De toegestane datastrukturen zijn (net als in PASCAL) record en array (met vaste grenzen).

Alle funkties zijn op hetzelfde niveau gedeklareerd (geen geneste procedure-deklaraties). Funkties mogen elkaar willekeurig aanroepen en recursie is zonder beperking toegestaan. Het kleur-mechanisme zorgt ervoor dat parameters onafhankelijk van elkaar worden doorgegeven, zodat de uitvoering van een deel van een funktie al kan beginnen, voordat alle parameters beschikbaar zijn (asynchrone parameter konsumptie). Ook de resultaten worden asynchroon opgeleverd.

Het is typerend voor dataflow, dat iteratie meer problemen oplevert dan recursie. De iteratie-konstruktie heeft dan ook de vorm van een funktie gekregen en is geïmplementeerd als een efficiënte vorm van staartrecursie :

```
iteration fit(i, f: integer);
repeat
     i := i - 1;
     f := f * i;
until i < 1 end
```

```
function fac(n: integer): integer;
decl dummy: integer;
begin
     [fac, dummy] := fit(n, n);
end
```

Een iteratie heeft dus een naam en de aanroep intialiseert de iteratie-variabelen. Het nul maal doorlopen van een iteratie is onmogelijk.

Aan elementen van een array kan op de normale wijze worden gerefereerd (als in B := AR[i]). Omdat een dataflow-machine geen adresseerbaar gegevens-geheugen heeft, wordt de array-referentie gesimuleerd door een kopie van het gehele array AR te maken en vervolgens alle elementen behalve i weg te gooien. Aan een individueel element kan niet worden toegekend. Als dit was toegestaan, zou statisch niet meer bepaald kunnen worden of aan de regel van Eenmalige Toekenning is voldaan. De waarden van een array worden altijd gelijktijdig toegekend in een forall konstruktie. Bijvoorbeeld

```
forall i in [1..10] do AB[i] := A[i] + B[i] od
```

creëert een nieuw array AB met grenzen 1 en 10. Alle doorgangen van een forall worden parallel geëvalueerd. Om redenen van implementatie kan binnen een forall geen funktie-aanroep voorkomen.

Voor de in- en uitvoer bestaan zeer eenvoudige lees- en schrijf-operaties. Deze kunnen niet binnen een iteratie of een funktie worden gebruikt. De programmeur moet zelf voor de synchronisatie zorgen, door middel van extra parameters. In

```
[s2,B] := read(file, s1);
[s1,A] := read(file, true );
```

wordt eerst A en dan B ingelezen. De dummy-variabele s1 zorgt voor de synchronisatie. We zien hier overigens hetzelfde probleem en dezelfde oplossing als bij de procedure NAMEN uit hoofdstuk 6.

7.2. Value-Oriented Algorithmic Language

Op het MIT Laboratory for Computer Science is de dataflow-taal VAL ontwikkeld[1]. De syntax is gebaseerd op CLU[18,17].

De wortels van de vierkantsvergelijking worden opgeleverd door de volgende expressie:

```
let  templ, temp2: real ;
     templ := wortel( b*b - 4*a*c );
     temp2 := 2*a;
in
     (-b + templ) / temp2,
     (-b - templ) / temp2
endlet
```

Deze let konstruktie introduceert een nieuw blok, waarbinnen namen kunnen worden ge(her)definiëerd. De let konstruktie is zelf ook een expressie die dezelfde waarde(n) heeft als de expressie(s) tussen in en endlet. Deze konstruktie kan dus genest worden.

De elementaire datatypen zijn boolean, integer, real en character. Deze kunnen gekombineerd worden tot records, arrays (met vaste grenzen) en unions.

Funkties kunnen genest gedeklareerd worden. Anders dan bij een let konstruktie zijn de aktuele parameters de enige waarden waar een funktie toegang toe heeft. Om onduidelijke redenen is recursie niet toegestaan.

De iteratie konstruktie ziet er als volgt uit:

```
for fac, f: integer := n, n;
do
      if f > 1
      then iter f := f-1; fac := fac * n; enditer
      else fac
      endif
endfor
```

In de eerste regel worden de iteratie variabelen gedeklareerd en tegelijkertijd geïnitialiseerd. Als de iter-enditer tak is uitgevoerd wordt het do-endfor gedeelte weer gestart.

Aan elementen van arrays kan op normale wijze worden gerefereerd. Toekenning aan een arrayelement kan alleen plaatsvinden door een nieuw array te creëren. Bijvoorbeeld

```
B := A[5 : k]
```

maakt een nieuw array B, dat gelijk is aan A met het vijfde element vervangen door k. Arrays kunnen ook worden gemaakt met de forall konstruktie als in

```
AB := forall i in [1,10] do
      construct A[i] * B[i]
      endall
```

waar een array AB met grenzen 1 en 10 wordt gecreëerd. In

```
sum := forall i in [1,10] do
       eval plus A[i] * B[i]
       endall
```

wordt het inprodukt van arrays A en B als waarde opgeleverd. In plaats van plus kunnen ook andere operaties gebruikt, mits zij zowel associatief als kommutatief zijn. De bedoeling van deze konstruktie is om aan te geven dat iteratie-ronden onafhankelijk zijn en dus parallel kunnen worden uitgevoerd. De eval konstruktie kan de berekende resultaten asynchroon verwerken (dankzij de speciale eigenschappen van de operator).

Als tijdens de executie van een dataflow-programma een fout wordt gekonstateerd, kan niet eenvoudigweg de executie worden onderbroken en een foutboodschap worden verstuurd. Andere berekeningen kunnen immers nog in volle gang zijn. In VAL is een volledige kollektie foutwaarden gedefiniëerd, alsook het gedrag van alle operatoren die zo'n waarde als invoer krijgen. De foutwaarde zal uiteindelijk in de uitvoer verschijnen. Er bestaat een voorstel om aan een foutwaarde een spoor van alle bezochte knopen te hangen, zodat de oorsprong van de fout gemakkelijker is op te sporen[19].

Er zijn geen in- en uitvoer-primitieven gedefiniëerd. Dit wordt wel als een noodzakelijke maar gecompliceerde uitbreiding aangekondigd.

7.3. Irvine Dataflow

De dataflow-groep op de University of California in Irvine heeft voor de machine die hen voor ogen staat de taal ID ontworpen[3]. Er bestaat een vertaler voor, die machinetaal aflevert die vooralsnog wordt gebruikt als invoer voor een simulator. Er zijn ook uitgebreide simulatie statistieken gepubliceerd [9].

De wortels van de vierkantsvergelijking worden opgeleverd door:

```
( templ <- wortel(b↑2 - 4*a*c);
  temp2 <- 2*a;
  return (-b + templ) / temp2, (-b - templ) / temp2

)
```

Deze konstruktie (omsloten door haakjes) heet een blok expressie. Blokken mogen worden genest. Binnen een blok betekent een toekenning een (her)definitie van de naam in de linkerkant. ID kent geen deklaraties.

Tot de elementaire datatypen behoren integer, real, boolean, string en error. Procedure is ook een datatype met als belangrijkste operatie AANROEP. De tekst van een procedure wordt ingevoerd als konstante van dit type. Hierdoor kan overal op de plaats van een expressie ook een procedure definitie staan en kunnen procedures als parameters worden doorgegeven. Een proceduretekst kan een naam hebben om recursie mogelijk te maken. Alle invoerwaarden van een procedure moeten beschikbaar zijn voordat de funktie tot executie over kan gaan (synchrone parameter konsumptie). In ID zijn nieuwe operator-definities toegestaan en de taal heeft ook een abstrakt data type mechanisme, vergelijkbaar met de klassen uit SIMULA[4].

De iteratie voor n! wordt in ID een blokexpressie

```
( initial fac <- n; f<- n;
  while f > 1
  do new fac <- fac*n;
     new f <- f-1;
  return fac )
```

Alle iteratie-variabelen (variabelen waarmee informatie van één ronde naar de volgende kan worden doorgegeven) moeten in het initial gedeelte worden geïnitialiseerd. Binnen de iteratie is de waarde van de vorige ronde beschikbaar (fac) alsook de nieuwe waarde (new fac).

Ook bestaat er de

for i from 1 to n do

konstruktie.

ID kent een datastruktuur stroom. De elementaire operaties die hierop zijn gedefinieerd zijn dusdanig, dat de elementen van een stroom alleen opeenvolgend kunnen worden verwerkt. Voor stromen is een uitzondering op de regel van synchrone parameter konsumptie gemaakt: de elementen van een stroom worden tijdens executie van een funktie gekonsumeerd. Hierdoor kunnen op eenvoudige wijze modules worden gecreëerd die met elkaar kommuniceren door het sturen van boodschappen (vergelijk de Communicating Sequential Processes van Hoar [12]).

Een interessante konstruktie in ID is verder de beheerder (resource manager). Dit is een speciaal soort procedure, die apart wordt gecreëerd en gebruikt. Anders dus dan bij de normale procedure, waar immers iedere aanroep tot een aparte instantie van de procedure leidt. Ieder gebruik van een beheerder richt zich op dezelfde instantie. De ingang van een beheerder is een niet-deterministische voeg-operator, die de tokens, die van verschillende paden binnenkomen, tot één stroom samenvoegt. De volgorde van de tokens in deze stroom is niet gedetermineerd. Het in- en uitvoer probleem is hiermee elegant op te lossen. Een gegevensbestand-beheerder en een beheerder voor randapparatuur is gemakkelijk te realiseren.

8. EEN HYBRIDE TAAL

De TET's (talen voor eenmalige toekenning) uit het vorige hoofdstuk zijn alle drie speciaal voor een dataflow-machine ontworpen. De afstand tot konventionele talen is aanzienlijk, hoewel de gelijkenis in syntax soms anders doet vermoeden. Een programmeur, die gewend is aan het programmeren in een konventionele taal, zal zich een heel andere

programmeer-stijl eigen moeten maken. Dit is een voordeel voor zover het gaat om het afleren van slechte gewoonten. Maar niet alle mogelijkheden die konventionele talen bieden, maar TET´s missen, zijn verderfelijk. In dit hoofdstuk beschrijf ik de principe´s van een dataflow-taal, die zo dicht mogelijk aansluit bij een konventionele taal (zie ook [26]). De bedoeling hiervan is, onder andere, om te onderscheiden welke van de eigenschappen van dataflow-talen essentiëel zijn en welke niet. Het ideaal is, dat programma´s normaal in een (nette) konventionele taal geschreven kunnen worden, die dan door een vertaler dermate slim naar dataflow-machinekode vertaald worden, dat de graad van parallellisme even hoog is als wanneer een TET was gebruikt. In de rest van dit hoofdstuk zal ik laten zien, dat een taal die sterk op een konventionele taal lijkt, ook geschikt is als brontaal voor een dataflow-machine. Dat zal ik doen, door bij iedere konstruktie in de taal, die tot problemen kan leiden aan te geven hoe een vertaler een programma in zo´n taal om kan zetten naar een ekwivalent programma in een TET.

Ik ga, bij wijze van voorbeeld, uit van een op het Mathematisch Centrum ontworpen en geïmplementeerde taal SUMMER[14]. De hybride taal zou, evenals SUMMER, in het bijzonder geschikt moeten zijn voor het schrijven van string-verwerkende programma´s, zoals vertalers, tekstverwerkers en editor´s. Anders dan bij numerieke toepassingen, waar dataflow-machines en -talen grotendeels op zijn gericht, lijkt op dit toepassingsgebied geschiedenisafhankelijkheid van procedures een grote rol te spelen en tegelijkertijd stroom een voor de hand liggend data-type te zijn. SUMMER kent geen goto´s en geen pointers, maar wel globale variabelen. De taal heeft ook abstrakte data-typen, waardoor klasse-objekten kunnen worden gecreëerd.

We stellen allereerst vast, dat het verbod op konditionele statements overbodig is. Ieder konditioneel statement kan via een eenvoudig algorithme worden omgezet in een toekenning met een konditionele expressie aan de rechterkant. Aan de linkerkant van de nieuwe toekenning komt dan een reeks variabelen te staan, die de vereniging vormt van de variabelen, waaraan in de twee takken van het oorspronkelijke statement wordt toegekend. In de takken van de konditionele expressie komen de expressies van de oorspronkelijke

toekenningen te staan of de variabelen zelf. Bijvoorbeeld:

```
if k > Ø
then
     a := k*3;
     b := Ø;
else
     b := 1;
     c := k*3;
fi
```

wordt

```
<a,b,c> :=
if k > Ø
then <k*3,Ø,c>
else <a,1,k*3>
fi
```

De regel van Eenmalige Toekenning is ook overbodig. Bij het vertalen wordt voor iedere variabele bij een toekenning een nieuwe unieke naam gegenereerd. Referenties aan deze variabele in latere statements worden ook door deze nieuwe naam vervangen. Iedere keer dat aan een variabele opnieuw wordt toegekend, wordt weer een nieuwe naam gegenereerd. Als aan een variabele wordt gerefereerd waar nog geen unieke naam voor is gegenereerd dan is het programma inkorrekt.

```
open := 16;
brug := 83.5 + open;
open := brug * open;
verlies := open / 2;
```

wordt

```
var1 := 16;
var2 := 83.5 + var1;
var3 := var2 * var1;
var4 := var3 / 2;
```

Iteraties geven niet meer problemen dan in de gebruikelijke dataflow-talen. Alleen iteratie-variabelen vereisen een speciale behandeling. Dit zijn variabelen waaraan, binnen een iteratie-blok, eerst wordt gerefereerd (oude waarde) en later wordt toegekend (nieuwe waarde). Voor iedere variabele wordt de laatst gegenereerde nieuwe waarde

op de gebruikelijke wijze (via KLEUR of SLUIS-mechanisme) als oude waarde de volgende ronde ingestuurd.

De grootste problemen komen we tegen bij kommunikatie met niet-funktionele elementen. In een TET blijft dat beperkt tot kommunikatie met de wereld buiten de dataflow-machine (in- en uitvoer), maar in een konventionele taal doen dit soort problemen zich bovendien voor bij bijvoorbeeld toekenning aan een array-element of aanroepen aan procedures die toegang hebben tot globale informatie. Welke vrijheid we in onze taal toelaten wordt bepaald door een afweging van drie faktoren: het gemak van de programmeur, de eenvoud van de vertaler en de graad van parallellisme in de gegenereerde machine-kode.

We nemen als voorbeeld weer de namentabel uit hoofdstuk 6. Een stukje SUMMER programma, waarin van een namentabel gebruik wordt gemaakt, ziet er als volgt uit:

```
namen[var1] := nieuw_adres;
adres2      := namen[var2];
namen[var3] := volgend_adres;
```

De waarde tussen [en] heet de sleutel. De vertaler kan voor de synchronisatie van deze toegangen tot de tabel zorgen door voor iedere toegang zogenaamde synchronisatie-signalen te genereren. Bovenstaand programma wordt dan omgezet in:

```
s5           := namen.schrijf( var1, nieuw_adres, s4);
<s6,adres2>  := namen.lees   ( var2, s5);
s7           := namen.schrijf( var3, volgend_adres, s6);
```

Hier zorgen dus de signalen s5 en s6 ervoor dat de toegang tot de tabel volledig wordt gesequentialiseerd. Dit is hetzelfde procedé dat de programmeur in LAPSE moet volgen om de in- en uitvoer te synchroniseren.

Als we het gemak van de programmeur voorop stellen dan laten we toe dat de tabel als globaal wordt gedeklareerd, zodat iedere procedure er toegang toe heeft. In elke procedure, die van de tabel gebruik maakt wordt dan een extra signaal toegevoegd aan zowel de parameterlijst als de resultatenlijst. Dit heeft tot gevolg dat de signalen alle toegangen tot de tabel aaneenrijgen tot één lineaire keten. We kunnen alle globale variabelen in één grote tabel

onderbrengen en daar hetzelfde sequentialisatie-algorithme voor gebruiken. Dit heeft als konsekwentie dat alle gebruik van globale variabelen in het gehele programma volledig wordt gesequentialiseerd. In een programma waar vrijelijk van globale variabelen gebruik wordt gemaakt blijft op die manier weinig parallellisme over.

Als behoud van parallellisme belangrijker is dan eenvoud van de vertaler, kunnen we beter voor iedere globale variabele een aparte signalenketen maken. Dan wordt aan de parameter- en aan de resulaten-lijst van iedere procedure een hele rij signalen toegevoegd; één signaal voor iedere globale variabele, die in de procedure of in een (direkt of indirekt) aangeroepen procedure wordt gebruikt. Als de parameters van procedures asynchroon worden gekonsumeerd, wordt door deze methode een maximaal parallellisme verkregen ten koste van een extra belasting van vertaler en machine.

Een andere benadering is om de vrijheid van de programmeur enigszins te beperken. In de taal wordt een speciale konstruktie opgenomen, die een objekt creëert en tegelijkertijd als globale variabele beschikbaar stelt. Alle statements die (dynamisch) binnen deze konstruktie vallen, hebben toegang tot dit objekt. Bovenstaand voorbeeld wordt dan:

```
install (tabel)
     schrijf(var1, nieuw_adres);
     adres2 := lees(var2);
     schrijf(var3, volgend_adres);
endinstall
```

Deze konstruktie kan worden genest. Belangrijk is dat bij install de tabel wordt gecreëerd en bij endinstall weer wordt vernietigd. Dit heeft als voordeel, dat statements die binnen afzonderlijke installs vallen, niet met elkaar hoeven te worden gesynchroniseerd. Dit vermindert de complexiteit van de vertaler en verhoogt de graad van parallellisme in de machinekode.

Een interessante mogelijkheid doet zich hierbij voor. We kunnen speciale "tabellen met eenmalige toekenning" definiëren. Dit zijn tabellen waaraan nooit twee schrijfopdrachten met dezelfde sleutel mogen worden gegeven. Lees- en schrijf-opdrachten aan zo'n tabel hoeven dan niet meer expliciet met elkaar te worden gesynchroniseerd. Als zo'n

tabel een lees-opdracht ontvangt met een sleutel waarvoor nog geen schrijf-opdracht is ontvangen, dan wordt het resultaat pas teruggestuurd als die schrijf-opdracht binnenkomt. Bij vernietiging van het objekt door de endinstall krijgt iedere nog wachtende lees-opdracht de waarde "ongedefinieerd". Op deze manier worden multi-pass vertalers overbodig: de tabel zorgt zelf voor de juiste volgorde van afhandeling.

9. KONKLUSIES

Dataflow-machines zijn een veelbelovend alternatief voor snelle computers die volgens het von Neumann principe werken. In de machinekode van dataflow-computers is bij iedere instruktie alle noodzakelijke informatie over de data-afhankelijkheid voor handen. In zulke machines kunnen daarom vele processoren parallel instrukties van één programma uitvoeren zonder dat hiervoor, tijdens executie, een ingewikkelde analyse nodig is. Er zijn een aantal hogere programmeertalen ontwikkeld, die speciaal als brontaal voor dergelijke machines zijn bedoeld. In het algemeen leggen deze talen de programmeur een aantal ongewenste beperkingen op. Een enigszins gewijzigde versie van een konventionele taal lijkt echter ook geschikt om als brontaal voor een dataflow-machine te dienen.

LITERATUUR

[1] Ackerman, W. and Jack B. Dennis, "VAL - A Value-Oriented Algorithmic Language Preliminary Reference Manual," TR-218, MIT/Laboratory for Computer Science (June 1979).

[2] Arvind, and Kim P. Gostelow, "A Computer Capable of Exchanging Processors for Time," Information Processing 77, pp.849-853, North Holland (1977).

[3] Arvind, Kim P. Gostelow, and Wil Plouffe , "An Asynchronous Programming Language and Computing Machine," Technical Report #114a, University of California, Irvine, Information and Computer Science Dept (Dec 1978).

[4] Birtwistle, G. M. and others, SIMULA begin, Auerbach Publications, Philadelphia (1973).

[5] Davis, A. L., "Architecture of DDM1: A Recursively Structured Data Driven Machine," Technical Report, University of Utah, Salt Lake City, Utah (1977).

[6] Dennis, J. B., J. B. Fosseen, and J. P. Linderman, "Data Flow Schemas," Symposium on Theoretical Programming, pp.217-246 (1972).

[7] Dennis, J. B., "First Version of a Data Flow Procedure Language," pp. 362-376 in Lecture Notes in Computer Science, Springer, New York (1974).

[8] Glauert, J. R. W., "A Single Assignment Language for Data Flow Computing," Dissertation, Dept. of Computer Science - University of Manchester (Jan 1978).

[9] Gostelow, Kim P. and Robert E. Thomas, "Performance of a Data Flow Computer," Technical Report #127a, University of California, Irvine, Information and Computer Science Dept (Oct 1979).

[10] Gurd, John and Ian Watson, "A Data Driven System for High Speed Parallel Computing," Report, Dept. of Computer Science - University of Manchester.

[11] Gurd, John, Ian Watson, and John Glauert, "A Multilayered Data Flow Architecture," Internal Report, Dept. of Computer Science - University of Manchester (July 1978). Second issue

[12] Hoar, C. A. R., "Communicating Sequential Processes," CACM Vol. 21(8), pp.666-677 (Aug 1978).

[13] Karp, R. M. and R. E. Miller, "Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing," SIAM Journal of Applied Mathematics Vol. 14 (Nov 1966).

[14] Klint, Paul, "An Overview of the SUMMER Programming Language," Conference Record of the Seventh Annual ACM Symposium of Programming Languages, pp.47-55, ACM (Jan 1980).

[15] Klint, Paul, "De Symbiose van Programmeertaal en Computer," in deze syllabus, Mathematisch Centrum (Feb 1980).

[16] Kosinski, P. R., "A Data Flow Language for Operating Systems Programming," Proceedings of ACM SIGPLAN-SIGOPS Interface Meeting SIGPLAN Notices Vol. 8(9), pp.89-94 (Sep 1973).

[17] Liskov, B. H. and others, "Abstraction Mechanisms in CLU," CACM Vol. 20(8), pp.564-576 (Aug 1977).

[18] Liskov, B. H. and others, "CLU Reference Manual," Computation Structures Group Memo 161, MIT/Laboratory for Computer Science, Cambridge, Mass. (Jul 1978).

[19] McGraw, James R., "Data Flow Computing - The VAL Language," Computation Structures Group Memo 188, MIT/Laboratory for Computer Science (Jan 1980).

[20] Miranker, Glen Seth, "Implementation of Procedures on a Class of Data Flow Processors," Proceedings of the 1977 International Conference on Parallel Processing, pp.77-86.

[21] Misunas, D. P., "A Computer Architecture for Data Flow Computation," Technical Memorandum 100, MIT/Laboratory for Computer Science (1978).

[22] Plas, A. and others, "LAU System Architecture: A Parallel Data-driven Processor Based on Single Assignment," Proceedings of the 1976 International Conference on Parallel Processing, pp.293-302, IEEE (Aug 1976).

[23] Rumbaugh, J., "A Data Flow Multiprocessor," IEEE Trans. Comp. Vol. C-26(2), pp.138-146 (Feb 1977).

[24] Watson, Ian and John Gurd, "A Prototype Data Flow Computer with Token Labelling," Proceedings National Computing Conference Vol. 48, pp.623-628, AFIPS (Jun 1979).

[25] Weng, K. S., "Stream-Oriented Computation in Recursive Data Flow Schemas," Technical Memorandum 68, MIT/Project Mac (1975).

[26] Whitelock, P. J., "A Conventional Language for Data Flow Computation," dissertation (Oct 1978).