Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O).

MC SYLLABUS 42

COLLOQUIUM CAPITA IMPLEMENTATIE VAN PROGRAMMEERTALEN

J.C. VAN VLIET (red.)

MATHEMATISCH CENTRUM AMSTERDAM 1980

1980 Mathematics Subject Classification: 68-06,68B05,68F05,68F20,68F25

ACM-Computing Review-categories: 4.12,5.23,5.24
ISBN 90 6196 191 2

INHOUD

Inhoud i

Voorwoord v

I NAAMLIJST-ALGORITMEN door D. Grune
1. Inleiding 1
2. Vraagstelling 2
3. Een abstracte algoritme 4
4. Algoritmen met string-vergelijking 7
4.1. Ongesorteerde array's 7
4.2. Gesorteerde array's 10
4.2.1. Het zoeken 10
4.2.2. Het invoegen 11
4.3. Gelinkte structuren 12
4.3.1. Gelinkte lijsten 12
4.3.2. Binaire bomen 13
5. Algoritmen zonder string-vergelijking 14
5.1. Tekens uit de naam 15
5.2. Andere vragen 16
6. Gemengde algoritmen 16
6.1. Vergelijkende algoritmen 17
6.2. Niet-vergelijkende algoritmen 18
6.2.1. Tekens uit de naam 18
6.2.2. Hash-functies 19
7. Een geheel andere algoritme: open-hash 20
8. Samenvatting 23
9. Enkele voorbeelden 24
Literatuur 25

II CODE-GENERATIE door L.G.L.T. Meertens
1. Inleiding 27
2. Code-generatie als partiële berekening 29
3. Optimalisatie-transformaties 32
4. Tussenfasen 34
5. Van tussencode naar doelcode 35
6. Accessbepaling 43
7. Samenvatting 44
Literatuur 45

III OVER ONTLEDEN EN GRAMMATICALE EQUIVALENTIERELATIES door A. Nijholt
1. Inleiding 47
2. Verduidelijking 50
3. Ontleden 53
3.1. Neerwaarts 54
3.2. Opwaarts 55
3.3. Neerwaarts/Opwaarts 57
3.4. Contextvrije grammatica's met reguliere expressies 57
4. Vertaler-genererende systemen 57
4.1. Neerwaarts 59
4.2. Opwaarts 60

5. Transformaties naar neerwaarts te ontleden grammatica's 61
5.1. Uitgaan van transformaties 61
5.2. Uitgaan van definities 63
6. Transformaties naar normaalvormen 64
7. Conclusies 67
Literatuur 67

IV GARBAGE COLLECTION door H.B.M. Jonkers
1. Inleiding 73
2. Geheugenbeheer 73
3. (Compactificerende) sanering 84
3.1. Sanerings-algoritmen 85
3.1.1. Merk-algoritmen 86
3.2. Compactificatie-algoritmen 95
3.2.1. Compactificatie met een representatie-afbeelding 97
3.2.2. Compactificatie met vertakkings-verzamelingen 100
4. Slot 104
Literatuur 105

V RELATIES TUSSEN TAALDEFINITIE EN TAALIMPLEMENTATIE door C. Hemerik
1. Inleiding 109
2. Over taaldefinities 111
3. Beschrijving van B, D en C 113
3.1. De brontaal B 113
3.2. De doeltaal D 113
3.3. De vertaler C 114
4. Implementatiecorrectheid in termen van denotationele semantiek 116
4.1. Inleiding 116
4.2. Denotationele definitie van B 116
4.3. Denotationele definitie van D 118
4.4. Correctheidscriteria voor C 121
4.5. Bewijs van de correctheid van C 122
5. Implementatiecorrectheid in termen van axiomatische semantiek 128
5.1. Inleiding 128
5.2. Axiomatische definitie van B 129
5.3. Axiomatische definitie van D 129
5.4. Correctheidscriteria voor C 131
5.5. Bewijs van de partiële correctheid van C 132
5.6. Terminatie 137
6. Conclusies 138
7. Bibliografie 139
Literatuur 141

VI DE CYBER ALGOL 68 VERTALER door J.J.F.M. Schlichting
1. Inleiding 143
2. Overzicht van de vertaler 144
2.1. Structuur van de vertaler 144
2.2. Naamlijst 144
2.3. Implementatietaken 145
2.4. Geheugenindeling van de vertaler 146

3.	Pass1	146
4.	Pass2	148
4.1.	Opstelling van de grammatica	148
4.2.	Het ontleedprogramma	149
4.3.	Invoer-routine	150
4.4.	Declaraties en ranges	150
4.5.	Modes	151
5.	Mode-equivalencing	152
6.	Mode checking en coercies	153
Literatuur		157

VOORWOORD

In deze syllabus zijn de voordrachten gebundeld welke gehouden zijn in het Colloquium Capita Implementatie van Programmeertalen. Dit colloquium werd in het najaar van 1979 door de Afdeling Informatica van het Mathematisch Centrum georganiseerd.

In de bijdragen van Grune (Naamlijst-algoritmen) en Jonkers (Garbage collection) staat een bepaalde benaderingswijze van het gegeven probleem centraal. Deze benaderingswijze kan worden gekarakteriseerd met de term "abstractie". Grune toont aan hoe verschillende naamlijst-algoritmen afgeleid kunnen worden op grond van bepaalde beslissingen met betrekking tot te kiezen gegevensstructuren en methoden voor het vergelijken van representaties van namen. Via een soortgelijke benadering geeft Jonkers op een taal- en machine-onafhankelijke manier een overzicht van een belangrijk deel van het onderwerp garbage collection.

In de bijdrage van Meertens wordt getracht enige structuur aan te brengen in het complexe probleem van code-generatie. Een algemene optimalisatie-techniek, welke bekend staat als het partiële-berekeningsprincipe, speelt hierbij een centrale rol.

De bijdragen van Nijholt en Hemerik dragen een meer theoretisch karakter. Nijholt geeft een aantal resultaten op het gebied van structuurbehoudende transformaties op grammatica's, en gaat nader in op de wijze waarop deze resultaten in de praktijk gebruikt kunnen worden. Tevens behandelt hij een aantal recente resultaten op het gebied van de ontledingstheorie. Hemerik gaat nader in op de vraag hoe bij een gegeven definitie van een programmeertaal een correcte implementatie te construeren. Hierbij worden de geschiktheid van de denotationele en die van de axiomatische definitiemethode met elkaar vergeleken aan de hand van een eenvoudig voorbeeld.

In de bijdrage van Schlichting wordt een globale beschrijving van het machine-onafhankelijke gedeelte van de Cyber ALGOL 68 vertaler gegeven. Deze bijdrage draagt een wat meer specialistisch karakter, en voor een goed begrip is enige voorkennis van de programmeertaal ALGOL 68, en van de problemen die optreden bij de implementatie van deze taal, vereist.

J.C. van Vliet

NAAMLIJST-ALGORITMEN

D. GRUNE
Mathematisch Centrum

1. INLEIDING

Eenenvijftig procent van alle tekstuele eenheden in een ALGOL 68 programma bestaat uit namen.

Deze zin behoeft enige verduidelijking. Een programma bestaat in eerste instantie uit tekens: letters, cijfers, haakjes, spaties, enz. Wanneer zo'n programma door een vertaler ingelezen wordt, neemt deze telkens groepjes van één of meer tekens bijeen tot tekstuele eenheden: denotaties (zoals 123.45E2), namen (zoals x, print, ABS, WRITE), al dan niet samengestelde symbolen (zoals ?, :=, **), enz. Deze tekstuele eenheden worden weer samengenomen tot syntaktische eenheden, zoals toekenningsopdrachten, conditionele opdrachten, enz.; bij elke syntaktische eenheid hoort dan een bepaalde bedoeling, en het is de taak van de code-generator iets te produceren dat deze bedoeling zo goed mogelijk weergeeft.

Met elk van de soorten tekstuele eenheden moet de vertaler iets specifieks doen: denotaties hebben hun eigen betekenis in zich en moeten gewoon onthouden worden; namen krijgen door de rest van het programma hun betekenis en moeten door de vertaler in lijsten verzameld en opgezocht worden; symbolen dragen de syntaktische structuur van het programma en moeten doorgegeven worden. Van deze drie is de behandeling van namen de ingewikkeldste en interessantste. Bovendien vormen namen vaak de grootste groep van tekstuele eenheden.

Voor ALGOL 68 worden hiervoor in [1] enige getallen gegeven: 53 geanalyseerde programma's bleken te bestaan uit:

	54 651	tekstuele eenheden,
waarvan	17 209	identifiers,
	10 627	vette woorden,
	5 916	denotaties en

20 899 symbolen.

Nu kunnen in ALGOL 68 zowel identifiers (namen van variabelen, constanten, enz., geschreven in kleine letters) als vette woorden (namen van modes en operatoren, geschreven in vette letters) door de programmeur gedeclareerd en van informatie voorzien worden. Beide groepen moeten dus als namen beschouwd worden. Samen vormen ze 27836 van de 54651 tekstuele eenheden, en we zien dus dat de 51% uit de eerste zin niet helemaal spreekwoordelijk was.

Voor andere programmeertalen is de situatie vergelijkbaar. Vrijwel alle talen hebben de structuur: scheider, naam, scheider, naam, enz. Soms komt een dubbele scheider voor, waardoor het aantal namen iets minder dan de helft is. In ALGOL 68 zijn sommige vette woorden ook scheiders, waardoor de weegschaal iets meer naar de andere kant doorslaat.

2. VRAAGSTELLING

Het bovenstaande laat wel zien dat de verwerking van namen in een vertaler een, zeker vanuit efficiëntie-standpunt, zeer belangrijke aangelegenheid is. Er is een veelheid van algoritmen voor bekend, van zeer simpel tot zeer vernuftig. In dit artikel zullen we de belangrijkste de revue laten passeren en er vanuit een bepaald gezichtspunt een oordeel over geven.

Wat moet zo'n algoritme nu precies doen? De ingelezen naam wordt als string beschouwd (b.v. toegekend aan een string-variabele, opgeslagen in een array-tje, o.i.d.). Deze string wordt aan de algoritme 'naamlijst' aangeboden, welke dan een stuk informatie oplevert. De mode van 'naamlijst' is dan proc(string)info. Aan deze info willen we nu informatie gaan toevoegen b.v. dat de naam aangetroffen is op regel 13 (voor de foutmeldingen) of gedeclareerd is als integer (voor het nagaan van juist gebruik en voor de code-generator). En we verwachten dat wanneer we dezelfde naam opnieuw aanbieden, deze informatie beschikbaar komt. Dit houdt in dat we van de naamlijst-algoritme niet alleen verwachten dat hij namen voor ons opzoekt, maar ook dat hij informatie onthoudt, een functie die eigenlijk meer aan een array doet denken dan aan een procedure en die een interne data-structuur nodig maakt. Hij kan dan in plaats van de info

beter een wijzer ernaartoe afleveren, en is dan van mode proc(string)ref info. We stellen nu als eis:

naamlijst(s1) :=: naamlijst(s2) <=> s1 = s2,

d.w.z., er is een unieke afbeelding van string's op ref info's. Dit doet nog veel sterker aan array's denken, die immers een unieke afbeelding vormen van int's op geheugenplaatsen. Een betere mode voor de naamlijst-algoritme is dus row[string]ref info (wat geen ALGOL 68 meer is, maar hopelijk wel begrijpelijk). De eis:

naamlijst[s1] :=: naamlijst[s2] <=> s1 = s2

is dan triviaal.

Bij deze beschouwingswijze doen zich echter onmiddellijk twee vragen voor.

Gewone array's worden geïndiceerd met integers, en er is een uiterst eenvoudige algoritme, om van zo'n integer en de array descriptor tot het adres van het geïndiceerde element te komen. Hoe moet dat met string's? M.a.w. wat is de afbeeldings-algoritme?

Bij gewone array's moet het aantal elementen van te voren bekend zijn, en zeer veel kleiner zijn dan het aantal integers. Er zijn dus integers die niet als index mogen worden gebruikt. We willen nu echter dat bij elke string en info hoort. Hoe kan dat?

Het antwoord op de tweede vraag is relatief eenvoudig. We doen net alsof we een oneindig groot array van info's hebben; zo goed als alle elementen daarvan zijn lege info's, die geen geheugen behoeven in te nemen, als de naamlijst-algoritme maar weet hoe een lege info eruit ziet. Hij kan dan naar behoefte lege elementen creëren (hoe en waar?).

Op de eerste vraag zijn vele antwoorden te geven. In alle gevallen zal elke string die als index gebruikt wordt, bewaard moeten worden, met op een of andere manier daaraan gehecht een wijzer naar de bijbehorende info. Deze string's moeten ergens blijven, zodat we met hetzelfde hoe-en-waar probleem als hierboven zitten. Als de naamlijst-algoritme geprogrammeerd wordt in een taal waarin vlot met string's kan worden gemanipuleerd en waarin

gemakkelijk geheugen kan worden aangevraagd, is het probleem al door iemand anders opgelost (hoe efficiënt blijft de vraag).

Vaak is dat echter niet zo. We zitten dan met een stuk geheugen van vaste lengte en moeten daarin zelf het opslaan van de namen en het plaatsen van de info-blokken programmeren. Sommige afbeeldings-algoritmen lenen zich daartoe beter dan andere. En zo'n vast stuk geheugen raakt natuurlijk een keer vol, waarna we meestal niets anders kunnen doen dan het opgeven. Het argument dat het heelal eindig is, de wereld eindig, het budget eindig, de computer eindig, dus waarom het aantal namen niet, bevredigt alleen mensen die toch nooit grote programma's schrijven.

3. EEN ABSTRACTE ALGORITME

We zullen nu een zeer algemene algoritme gaan bekijken, gebaseerd op het hierboven genoemde oneindige array van info's. Het woord 'array' echter impliceert een soort nabijheid in plaats, die voorlopig nergens voor nodig is, en we zullen er hier dan ook een verzameling van maken.

```
mode elem = struct(string name, ref info info);

set elem N = c een oneindige verzameling elem's zodanig dat:
    1. voor elke mogelijke string er precies 1 elem met die
       string is,
    2. alle info's wijzen naar verschillende, initiëel lege
       info's c;

row naamlijst = [string T] ref info:
begin set elem S:= N, int M:= 1;

    while size S > 1
    do  proc(string)answer QMS =
        c een vraag afhankelijk van M en/of S c;
        S:= c de verzameling elem's E uit S, waarvoor geldt dat
              QMS(name of E) = QMS(T) c;
        M +:= 1
    od;
    info of elem ∈ S
end
```

(deze algoritme is in pseudo-ALGOL 68 geschreven, en is dus niet zo bruikbaar. Dat is niet erg want de looptijd is toch oneindig.)

De hier gebruikte techniek voor het uitdunnen van de oneindige verzameling, net zolang tot we T gevonden hebben, is erg eenvoudig. We stellen een reeks vragen, en we houden telkens alleen die elementen over die op een vraag net zo reageren als T. Na verloop van (oneindig veel) tijd is er dan nog maar één element over, en dat moet dan T bevatten.

Hoewel terminatie van een programma dat misschien oneindig veel rekentijd kost, een onduidelijk begrip is, is het wel duidelijk dat men de QMS-vragen zo moet kiezen, dat elk een redelijke kans heeft S te verkleinen, daar anders geen voortgang wordt gemaakt.

Redelijke vragen (QMS) aangaande de string zouden kunnen zijn:

- wat is het M-de teken in de string?
- is de string groter dan, gelijk aan of kleiner dan een van te voren bepaalde string uit S?

In beide gevallen is goed te zien hoe de oneindige verzameling langzamerhand wordt uitgedund.

Vrijwel alle bekende algoritmen zijn variaties op deze algoritme. Ze verschillen in twee aspecten:

- hoe wordt de oneindige verzameling geïmplementeerd?
- welke reeks vragen (QMS) wordt er gebruikt?

Het eindig maken kan, zoals gezegd, eenvoudig geschieden door alles met lege info weg te laten. Verder willen we in de praktijk als antwoord van 'naamlijst' niet alleen de info maar ook de string zelf weer terug, en liefst een wijzer naar dit alles. Deze wijzer kunnen we dan verder in de vertaler als vertegenwoordiger van de naam hanteren.

Dit levert ons de volgende, al heel wat meer praktische, algoritme.

```
mode elem = struct(string name, info info);
info new = c de info van een nieuwe naam c;
set ref elem N:= empty;
```

```
row naamlijst = [string T] ref elem:
begin
    set ref elem S:= N, int M:= 1;

    while size S > 1
    do  proc(string)answer QMS =
        c een vraag afhankelijk van M en/of S c;
        S:= c de verzameling elem's E uit S, waarvoor geldt dat
              QMS(name of E) = QMS(T) c;
        M +:= 1
    od;

    if size S = 1
    then
        ref elem E ∈ S;
        if name of E = T
        then E
        else goto insert new
        fi
    else goto insert new
    fi

exit insert new:
    heap elem E := (T, new);
    N +:= E;
    E
end
```

De toegevoegde if-goto-exit-constructie onderscheidt de drie gevallen:

- er is geen elem overgebleven,
- er is een elem overgebleven, maar het was niet de goede,
- de goede elem is overgebleven.

De hiervoor benodigde test-structuur is ingewikkeld, geeft vaak aanleiding tot programmeerfouten en past niet goed bij de tegenwoordige structureringsmiddelen. De gegeven vorm vind ik het duidelijkst; indien gewenst kan de goto door een procedure-aanroep vervangen worden.

Veel van wat nu volgt is ontleend aan [2], waarin criteria worden ontwikkeld waaraan naamlijst-algoritmen moeten voldoen, en waarin de verschillende mogelijkheden voor data-structuren voor N (de "database") en voor vragenlijsten (QMS) systematisch onderzocht worden.

We zullen, waar mogelijk, een oordeel geven over de efficiëntie, zowel in termen van zoektijd als in termen van geheugenbeslag. Vooral deze zoektijd-efficiëntie is onderwerp van een spraakverwarring, die daarin ligt, dat soms het gedrag van het opzoeken van één naam wordt beschreven (lineair, logaritmisch, constant), en soms het gedrag van de algoritme bij het opbouwen van een namen-bestand met N namen (kwadratisch, N ln(N), lineair). Als de zoektijd lineair is, is de algoritme kwadratisch, enz.

4. ALGORITMEN MET STRING-VERGELIJKING

Een voor de hand liggende manier om aan een string een antwoord te ontlokken is om hem te vergelijken met een andere string. Deze andere string moet dan ergens uit de verzameling S komen. De precieze algoritme wordt bepaald door de manier waarop deze string uit S wordt gekozen, en door de implementatie van S.

4.1. Ongesorteerde array's

De eenvoudigste data-structuur om de elem's (wijzers!) in op te slaan is een ongesorteerd array; en de enige manier om hierin te zoeken is lineair, b.v. van voren naar achteren. S komt dan overeen met het deel-array N[M:], en de vraag QMS met "S[1] = T". Dit levert de eenvoudigste algoritme: "Lineair zoeken in een ongesorteerd array".

Het is een interessante oefening de abstracte algoritme voor dit eenvoudige geval te concretiseren. Een (reeds werkende) tussenstap zou kunnen zijn:

```
[1:10000] ref elem N1, int N2:= 0;

proc naamlijst = (string T) ref elem:
begin int S1:= 1, S2:= N2, int M:= 1;

    while S2 - S1 + 1 > 1
    do  proc QMS = (string s)bool: name of N1[S1] = s;

        if QMS(T)
        then S2:= S1
            # het ene element N1[S1] waarvoor
              QMS(name of N1[S1]) = true #
        else S1 +:= 1
            # alle elementen E waarvoor
              QMS(name of E) = false #
        fi;
        M +:= 1
    od;

    if S2 - S1 + 1 = 1
    then
        ref elem E = N1[S1];
        if name of E = T
        then E
        else goto insert new
        fi
    else goto insert new
    fi
exit insert new:
   (heap elem E := (T, new);
    if (N2+:=1) <= upb N1
    then N1[N2]:= E
    else # naamlijst vol # skip
    fi;
    E
   )
end
```

Hieraan valt uiteraard nog een heleboel te verbeteren. Overigens worden we in de eerste regel al met de noodzaak geconfronteerd een uitspraak te doen over de eindigheid van de naamlijst, wat weer zijn weerslag heeft op het invoegen onder 'insert new'.

Er is een hiermee samenhangende complicatie die uit het bovenstaande programma niet blijkt maar pas aan het licht komt als we proberen deze algoritme in een wat onvriendelijker taal, b.v. de plaatselijke assembler, te schrijven. In ALGOL 68 bedraagt het geheugenbeslag

10000 * reference + N2 * elem .

In een assembler hebben we (meestal) geen heap-generator om geheugen voor nieuwe elem's te creëren, en we staan dan voor de keus, óf een eigen geheugenbeheersysteem te schrijven, óf botweg een array te gebruiken. Tien tegen een kiezen we voor het laatste; dit array moet dan natuurlijk evenveel plaatsen hebben als N1, zodat het geheugenbeslag bedraagt:

10000 * reference + 10000 * elem .

Doen we dit echter zo, dan merken we dat we hier weer op kunnen bezuinigen: de elem's hebben vaste plaatsen, en we kunnen dus het array met references uitsparen. De kosten bedragen dan:

10000 * elem .

Deze reductie is alleen mogelijk bij algoritmen waarbij de elementen nooit verplaatst worden.

Samenvattend kunnen we zeggen dat lineair zoeken in een ongesorteerd array een eenvoudig te programmeren algoritme is dat door zijn kwadratische zoektijd-component ongeschikt is voor grotere naamlijsten. Aanbevolen voor lijsten met minder dan 20 namen.

4.2. Gesorteerde array's

Op het eerste gezicht lijkt het een goed idee het array gesorteerd te houden, want dan kunnen we veel gemakkelijker zoeken. Maar meteen doet zich dan de vraag voor, hoe het moet als we een element moeten tussenvoegen! Voor de voortgang van het betoog wil ik hier even aannemen dat we daarvoor een goede oplossing hebben, en houd me nu eerst bezig met de verschillende zoek-methoden in een gesorteerd array.

4.2.1. Het zoeken

De meest naïeve gedachte is, de ordening te gebruiken om als we een naam tegenkomen die groter is dan die welke we zochten, te concluderen dat hij er niet is, en hem toe te voegen. Deze gedachte is te naïef: de resulterende algoritme heeft nog steeds een kwadratische tijd-component en is nauwelijks beter dan "lineair zoeken in een ongesorteerd array".

De beste manier om de ordening te benutten is binair zoeken: we bekijken het middelste element, onderzoeken of diens naam groter dan, gelijk aan, of kleiner dan T is, en reduceren de verzameling tot alles onder het middelste element, het middelste element zelf of alles boven het middelste element. De bijbehorende concrete algoritme is weer eenvoudig af te leiden. Afgezien van de bijkomende kosten, kost het in een array ter lengte N gemiddeld log N slagen om T te vinden.

Zo nu en dan wordt het idee naar voren gebracht dat dit beter kan: als we Zijlstra in de telefoongids opzoeken, beginnen we ook niet in het midden, maar ergens achteraan. De naam wordt als een scalaire waarde opgevat op de schaal van A tot ZZZ..., en op de overeenkomstige plaats in het array wordt gezocht. Implementaties van dit idee leiden vrijwel altijd tot teleurstelling, door een samenstel van oorzaken:

- de methode werkt bij gratie van een uniforme verdeling van de namen, welke voor kleinere segmenten uit het array steeds twijfelachtiger wordt,
- de methode sorteert pas bij grote aantallen namen (> 1000) merkbaar effect,
- de kosten van de interpolatie-berekeningen zijn vaak hoger dan die van een extra vergelijking.

Voor de geïnteresseerde lezer staat er in [2] een uitgewerkt voorbeeld; YAO en YAO [3] hebben het probleem volledig geanalyseerd.

4.2.2. Het invoegen

De meest voor de hand liggende manier om een element tussen te voegen is het opzij duwen van de rest van de elementen. Dit kost per nieuwe naam gemiddeld 1/2 N handelingen, als N het aantal namen tot nu toe is. Hierdoor wordt een kwadratische term geïntroduceerd, die voor middelgrote aantallen (zo'n 1000 namen) ongeveer de helft van de zoektijd gaat uitmaken.

Een praktische manier om hier wat tegen te doen is het "verdunnen" van het array met lege elementen, met nil's. We kunnen dan met opzij duwen stoppen zo gauw we een nil zouden willen opduwen. Deze nil sneuvelt dan wel in het proces, waardoor het array steeds "dichter" wordt. Op een gegeven ogenblik wordt het zicht te slecht en gaan we herverdunnen. Bij dit herverdunnen worden alle elementen verschoven, zodat het lijkt alsof hiermee een kwadratische component toegevoegd wordt. Bij zorgvuldiger analyse blijkt dat echter niet zo te zijn: hoewel de herverdunningen steeds grotere aantallen elementen moeten verschuiven, treden ze steeds minder frequent op, zodanig dat de kosten per naam constant zijn. Zie [2].

Het blijkt dat een geringe verdunning al wonderen doet: wanneer we in het begin met 10% lege elementen verdunnen, en herverdunnen tot 10% wanneer het aantal lege elementen tot 4% gedaald is, blijkt er voor elke string-vergelijking nog maar 1 element verplaatst te hoeven worden; dit alles voor een array ter lengte 1000. Zie wederom [2].

Samenvattend kunnen we zeggen dat binair zoeken in een gesorteerd array met eenvoudig invoegen een efficiënte, eenvoudig te programmeren algoritme is, geschikt voor middelgrote aantallen namen (100 tot 500).

Binair zoeken in een gesorteerd verdund array is zéér efficiënt, ook voor veel grotere aantallen namen, maar is tamelijk omslachtig te implementeren.

Een bijkomend voordeel is dat de namen in alfabetische volgorde opgeslagen zijn wat goed van pas komt voor het leveren van een cross-reference listing.

4.3. Gelinkte structuren

Willen we de verschillende elementen kunnen doorzoeken, dan moeten ze op de een of andere manier samenhangen; in een array hangen ze samen door hun onderlinge nabijheid; in een gelinkte structuur hangen ze samen via wijzers. De eenvoudigste vorm is de gelinkte lijst waarin we van elk element zijn opvolger kunnen vinden door een bijbehorende wijzer te volgen.

4.3.1. Gelinkte lijsten

Eenvoudige toepassingen hiervan leveren niets nieuws ten opzichte van een array: het enige wat we met een gelinkte lijst kunnen, is hem van voren naar achteren doorlopen, en dat konden we met een array ook. Het enige is, dat tussenvoegen nu gemakkelijker geworden is. We raken daardoor een kwadratische term kwijt, maar de kwadratische term van het lineair zoeken blijft, dus dat helpt ook niets.

Het is dan ook hoogst verbazingwekkend dat we die kwadratische term in een gesorteerde lijst toch kwijt kunnen! De hiervoor door MATTHIJSSEN en UZGALIS [4] ontwikkelde methode komt er op neer, dat na de gebruikelijke vraag of dit element de gezochte naam bevat, er nog aan P-1 willekeurige andere elementen gevraagd wordt of hun naam soms dichter bij de gezochte ligt. Met de dichtstbijliggende gaan we dan verder.

Deze techniek is bedoeld voor een gesorteerde naamlijst die zich op schijf bevindt, in de vorm van pagina's met een lengte van P namen die als eenheid ingelezen en weggeschreven worden. De link geeft dan alleen aan op welke pagina de volgende naam zich bevindt. We halen die pagina op en kijken dan of deze de naam bevat; zo niet, dan gaan we verder met de grootste naam op deze pagina die kleiner is dan de gezochte.

De theoretische eigenschappen van deze techniek zijn moeilijk vast te stellen, maar het is bekend dat voor P=2 de zoektijd evenredig is met de wortel uit het aantal namen.

Echter, om deze methode te kunnen gebruiken, moeten we in staat zijn om "P-1 andere willekeurige elementen" te onderzoeken. En het enige wat we in een gelinkte lijst hebben, is een wijzer naar het volgende element!

Om dit onoplosbare probleem op te lossen moeten we buiten het referentie-kader treden: de gelinkte lijst wordt naar alle waarschijnlijkheid in een array opgeslagen, en de naastliggende elementen zijn dan "willekeurige" elementen.

Het gepagineerd zoeken in een gesorteerde gelinkte lijst is eigenlijk alleen interessant; de praktische waarde beperkt zich tot die situaties waarbij als geheugen alleen een kleine schijf ter beschikking staat.

4.3.2. Binaire bomen

Het vergelijken van twee strings levert niet persé een ja/nee antwoord; we kunnen ook groter/gelijk/kleiner krijgen. Wanneer we dit als answer in onze abstracte algoritme gebruiken, moeten we de volgende slag verder met die verzameling namen die groter, resp. gelijk, resp. kleiner zijn, en het ligt voor de hand deze verzamelingen dan voor te stellen door wijzers ernaartoe. Dit leidt direct tot de klassieke binaire boom, zoals b.v. beschreven door KNUTH [5]. Het grote voordeel is de logaritmische zoektijd: bij verdubbeling van het namen-bestand is er maar één vergelijking méér nodig. Een nadeel is dat de extra wijzers extra ruimte kosten. Maar het grootste nadeel is niet direct duidelijk en komt pas aan het licht als we een programma gaan verwerken dat namen in alfabetische volgorde bevat. De boom ontaardt dan tot een lineaire gelinkte lijst en de zoektijd wordt lineair.

Nu kan men beweren dat er geen reden is waarom namen in een programma in volgorde zouden staan, maar de praktijk wijst anders uit. Teksten als:

```
real x0, x1, x2, x3, x4, x5, x6, x7, x8, x9;
```

of

```
char letter a = "a", letter b = "b", ...
```

zijn helemaal niet zeldzaam en veroorzaken nare sliertен in de namen-boom. Er zijn dan ook vele technieken ontwikkeld om hier iets tegen te doen. Ze hebben alle hetzelfde doel: het "kroes" houden van de boom, en doen dat door beperkingen aan de vorm van de boom op te leggen. Zo mag b.v. in een AVL-tree in alle knopen het verschil tussen de maximale diepte van de linker tak en de maximale diepte van de rechter tak hoogstens 1 bedragen. Het zal duidelijk zijn dat zo'n beperking door het invoegen van een nieuwe naam gemakkelijk verstoord wordt en dat we opzoek-efficiëntie moeten betalen met invoeg-complicatie. Voor een aanvaardbare invoeg-algoritme is

in elke knoop extra informatie nodig die plaats inneemt en bijgehouden moet worden. BAER en SCHWAB [6] geven een overzicht van de verschillende technieken en bespreken het voor en tegen van elk.

Er bestaat echter een methode om het lineair worden van de boom tegen te gaan, die deze nadelen niet of in minder mate heeft. Deze methode, die in de literatuur veel te weinig aandacht heeft gekregen, bestaat eenvoudigweg daarin dat we op willekeurige ogenblikken de knoop waar we bezig zijn, "schudden". Dit betekent dat we de structuur (A b C) d E omzetten in A b (C d E) of omgekeerd (b en d zijn namen, A, C en E zijn deelbomen). Mochten er door het invoeren van gesorteerde namen lange slierten ontstaan, dan brokkelen deze door het "schudden" vanzelf weer af. Deze methode is oorspronkelijk door KOK [7] ontwerpen voor het verwerken van reeds (bijna) gesorteerde bestanden. Voor een optimale werking moet gemiddeld éénmaal per zoek- of invoeg-actie geschud worden. Er is geen extra informatie in de knopen nodig; wel moet de algoritme het laatste gedeelte van de afgelegde weg onthouden.

Al met al kan men zeggen dat "zoeken in een klassieke binaire boom" een eenvoudig te programmeren en vaak efficiënte algoritme is die echter om zijn kwadratisch gedrag op gesorteerde lijsten niet aanbevolen kan worden. Het gebruik van "afgedwongen symmetrische" bomen brengt hierin verbetering, maar ook deze algoritmen moeten, om hun ingewikkeldheid en extra kosten aan tijd en ruimte, met de nodige reserve bekeken worden. De "geschudde" boom zou hier een aantrekkelijk alternatief kunnen zijn.

Een voordeel is dat de namen gemakkelijk in alfabetische volgorde beschikbaar zijn; ook kunnen de elementen in de knopen worden opgenomen, net als bij het ongesorteerde array (4.1).

5. ALGORITMEN ZONDER STRING-VERGELIJKING

Tot nu toe bestonden de vragen QMS, die we gebruikten om het zoekgebied te beperken, altijd uit de vergelijking van twee strings, en zat de variatie meer in de data-structuur. We zullen nu enige andere vragen trachten te verzinnen. Deze zullen dan voornamelijk van M afhangen, de teller die bijhoudt hoe vaak we al iets gevraagd hebben.

5.1. Tekens uit de naam

De meest voor de hand liggende vraag is die naar het M-de teken:

```
proc QMS = (string s) char: s[M];
```

en het antwoord is nu niet meer een 3-waardig groter/gelijk/kleiner, maar iets dat enkele tientallen waarden kan hebben. En daarmee zitten we dan meteen met de vraag wat we daar nou voor data-structuur voor moeten gebruiken.

Laten we beginnen met een gesorteerd, eventueel verdund, array. Stel, het eerste teken van de aangeboden naam is "m". We moeten dan uit het array dat interval selecteren dat alle namen beginnend met een "m" bevat; aangezien het array gesorteerd is, staan deze namen gelukkig bij elkaar. We doen dit door binair te zoeken, en als we slim zijn kunnen we het zoeken naar boven- en onder-grens combineren. Stel, het volgende teken is een "c"; we bepalen dan het sub-interval van alle woorden die als tweede teken een "c" hebben, enz.

Deze merkwaardige algoritme is geheel uit de abstracte beschouwing voortgekomen. Ik heb iets dergelijks in de literatuur nooit gezien en heb weinig idee hoe het met de efficiëntie staat. Het is misschien de moeite van het onderzoeken waard.

Op bekender terrein komen we met het gebruik van een N-voudig vertakte boom met N groter dan 2 en wel gelijk aan het aantal mogelijke antwoorden. Aan de wortel vinden we dan een knoop met 26 wijzers en verder overal knopen met 36 wijzers. Dit is waarschijnlijk de snelste naamlijst-algoritme die er bestaat: een naam ter lengte n kan met n indiceringen gevonden worden. Het geheugenbeslag is echter ontstellend.

De meeste wijzers in de knopen zijn echter "nil", en we zouden een knoop dus kunnen implementeren als een gelinkte lijst (of zelfs boom) van die wijzers die niet "nil" zijn, ieder met zijn teken. De snelheid loopt terug maar het geheugenbeslag wordt minder belachelijk.

Het is moeilijk bij deze en soortgelijke algoritmen een redelijk inzicht in het geheugenbeslag te krijgen. Er is veel ruimte voor wijzers nodig, maar de strings zelf hoeven niet meer opgeslagen te worden: de

tekens die een naam van zijn buren onderscheiden staan nu verspreid over de boom. In machinetaal kunnen ze meestal wel ergens tussen of in de wijzers weggemoffeld worden, maar in een hogere programmeertaal is er vaak een apart woord voor nodig.

Ervaring met het engelse woordenboek op UNIX (PDP11) heeft mij geleerd dat in een gesorteerde woordenlijst ongeveer de helft van de letters een onderscheidende functie als hierboven heeft. Met een gemiddelde naamlengte van 5,5 [1] geeft dat een kleine 3 knopen per naam. Hoe zich dit verhoudt tot 1 knoop per naam + het onthouden van de naam, ligt geheel aan de architectuur van de machine die gebruikt wordt.

Deze bomen vervallen niet tot lineaire lijsten als de namen in volgorde aangeboden worden, en bewaren de namen (min of meer) in alfabetische volgorde.

5.2. Andere vragen

Het is een interessante bezigheid te proberen nog andere vragen te verzinnen. COFFMAN en EVE [8] stellen voor de string om te zetten in een eenduidige stroom bits. De vraag QMS levert dan telkens de volgende 2 bits uit deze stroom af. Dit zijn 4 mogelijke antwoorden, en we kunnen de deelverzameling van alle namen met dezelfde twee bits in die positie vinden door de onderhavige wijzer te volgen.

Hoe sneller de bitstromen voor twee verschillende namen uiteenlopen, hoe kleiner het geheugenbeslag is. Het kwalitatieve gedrag is door VAN DE LUNE [9] geanalyseerd; om een kwantitatief aanvaardbaar gedrag te krijgen, zijn verfijningen noodzakelijk.

6. GEMENGDE ALGORITMEN

In het voorgaande hoofdstuk hebben we kennis gemaakt met enige op zichzelf wel interessante maar in de praktijk toch tamelijk onbruikbare algoritmen, en we vragen ons af of deze niet toch nuttig kunnen zijn in combinatie met andere algoritmen.

Nu hebben gemengde algoritmen, d.w.z. algoritmen die een tijdje de ene techniek gebruiken en dan overgaan op een andere, drie duidelijke nadelen:

- we moeten twee verschillende algoritmen programmeren,
- er zijn meestal twee verschillende data-structuur vereist,
- we moeten een of andere test uitvoeren om te beslissen wanneer we van de ene op de andere moeten overstappen.

Hier moet dan wel wat tegenover staan, willen we aan een gemengde algoritme de voorkeur geven.

In principe kunnen we elke techniek op elk ogenblik met elke andere techniek combineren, maar sommige combinaties zijn geslaagder dan andere.

Een redelijke aanpak is om te kijken op welk niveau een gegeven techniek niet meer goed werkt, en dan te kijken met welke andere techniek we verder kunnen.

6.1. Vergelijkende algoritmen

Als eenvoudig voorbeeld nemen we het binair zoeken in een gesorteerd array. Weliswaar is binair zoeken theoretisch altijd sneller dan lineair zoeken, maar in de praktijk is lineair zoeken sneller voor lengtes kleiner dan ongeveer 20 (door lagere overhead). We zouden dus, zo gauw het interval korter dan 20 is geworden, lineair kunnen gaan zoeken. De drie bovengenoemde nadelen treden nauwelijks op: de tweede algoritme is zeer eenvoudig, de data-structuur is dezelfde en de test is heel goedkoop. Aan de andere kant is de winst ook klein, dus waarschijnlijk is het niet de moeite waard.

Veel interessanter is in dezen de binaire boom. Een van de nadelen van de binaire boom is dat de "bladeren" allemaal lege wijzers bevatten, die ruimte verspillen. We kunnen dit ondervangen door als bladeren array's ter lengte T te nemen. Als zo'n array te vol raakt, wordt het in twee array's en een knoop gesplitst. Hierdoor winnen we flink wat geheugen; aan de andere kant zijn we nu weer gedwongen zo nu en dan met elementen te gaan slepen waardoor we ze niet meer in de knopen kunnen opslaan (zoals in 4.1), en we weer extra ruimte voor wijzers nodig hebben.

In [2] worden preciese berekeningen omtrent het geheugenbeslag van "breedblad-bomen" uitgevoerd. De conclusie is dat het wel wat helpt (voor een array-lengte van 32); maar het kwadratisch gedrag op gesorteerde invoer is er nog steeds, zij het verminderd met een factor T/2. In zijn algemeenheid weegt dit waarschijnlijk niet tegen de complicaties op.

6.2. Niet-vergelijkende algoritmen

Met niet-vergelijkende vragen hadden we het probleem dat ze weliswaar zeer efficiënt splitsten, maar onaanvaardbaar veel geheugen kostten. Mengen met andere technieken opent dan perspectieven.

6.2.1. Tekens uit de naam

Nemen we als eerste vraag weer "wat is het eerste teken van de string", en implementeren we dit met een array ter lengte 26, dan hebben we a.h.w. een woordenboek met duimgrepen. Door één indicering vinden we dan de gewenste letter. Binnen deze letter kunnen we dan b.v. een gelinkte lijst gebruiken of een binaire boom. In [2] wordt echter aangetoond dat "duimgrepen + gelinkte lijst" niet aanbevelenswaardig is (de lijsten worden snel te lang) en dat de voordelen van "duimgrepen + binaire boom" marginaal zijn.

Veel beter wordt de toestand als we splitsen naar de eerste twee letters: we beginnen dan met een array van 702 plaatsen (= 26*27, 26 letters + het geval niets/cijfers), waarna de lijsten of bomen redelijk klein zijn (voor naamlijsten met zo'n 1000 namen). Uit [2] blijkt dat "duimgrepen van twee letters + gelinkte lijst" marginaal slechter is dan de gewone binaire boom (en zeker veel beter voor gesorteerde invoer!), maar dat "duimgrepen van twee letters + binaire boom" aanzienlijk veel beter is. We kunnen dus elke boom-algoritme verbeteren door duimgrepen van twee letters toe te voegen.

Ook hier hebben we weer het voordeel dat de namen in alfabetische volgorde bewaard worden.

6.2.2. Hash-functies

Overigens zal het duidelijk zijn dat de namen zo niet erg gelijkmatig over de 702 bomen verdeeld zullen worden, en dat sommige bomen daardoor inefficiënt groot worden.

Om dat te voorkomen willen we als QMS een functie die de ingevoerde namen zo gelijkmatig mogelijk over N bomen verdeelt. Dit is dus een proc(string)int QMS die een integer tussen 1 en N aflevert, zodanig dat:

- wanneer hij op alle mogelijke strings wordt toegepast, alle antwoorden (ongeveer) even vaak voorkomen,
- het antwoord van zoveel mogelijk eigenschappen van de string afhangt.

Zo'n functie noemen we een hash-functie, een naarmate hij beter voldoen aan bovenstaand signalement is hij een betere hash-functie. We zien dat de "duimgreep met twee letters" een slechte hash-functie is omdat het antwoord uitsluitend van de eerste twee tekens afhangt (hoewel, wanneer alle strings worden aangeboden, bijna alle antwoorden even vaak voorkomen).

De gedachtengang is, dat wanneer de namen in de invoer bepaalde systematische eigenschappen hebben (b.v. vaak met Sch beginnen, of allemaal 5 letters lang zijn), ze in andere eigenschappen zullen verschillen, zodat ze door de hash-functie, die immers op alle eigenschappen let, toch verschillend geclassificeerd worden.

Bekende hash-functies zijn gebaseerd op het optellen van de tekens gevolgd door een of andere reductie tot het interval [1:N], op allerlei exclusieve-of-bewerkingen op de bitpatronen van de tekens (voor een interval $[0 : 2^N-1]$), enz.

Een dergelijke algoritme wordt een "gesloten-hash-algoritme" genoemd (voor de "open-hash" zie hoofdstuk 7). Er zijn goede redenen om als tweede data-structuur eerder gelinkte lijsten dan binaire bomen te kiezen. Ten eerste kosten binaire bomen meer ruimte voor de links, en die ruimte kunnen we beter besteden om het interval [1:N] te vergroten; ten tweede verworden bomen toch tot lijsten als de invoer gesorteerd is (en als we dat niet willen, moeten we iets ingewikkelds doen); en ten derde willen we N graag zo groot hebben dat er in elk vak zo weinig namen terecht komen dat het

verschil tussen binaire bomen en lineaire lijsten er niet meer toe doet.

Uit [2] blijkt dat "gesloten hash + gelinkte lijst" wat tijd betreft verreweg de meest efficiënte algoritme is die we tot nog toe gezien hebben. Ook wat geheugen betreft zijn er geen problemen. Deze techniek is zeer aanbevelenswaardig als:

- we verwachten dat het gemiddeld aantal namen per programma niet meer dan enkele malen N bedraagt (anders gaat de kwadratische component doorzetten),
- we geen alfabetische afdruk nodig hebben (sorteren is duurder dan het hele opbouwen!).

Wel krijgen we hier te maken met het gewone nadeel dat kleeft aan het gebruik van hash-functies: het wordt voor de programmeur en de gebruiker volslagen onduidelijk wat het programma precies doet. Dit uit zich voornamelijk op drie manieren:

- Het vinden van fouten in de algoritme wordt bemoeilijkt. Remedie: probeer het eerst met een hash-functie die alles op 1 afbeeldt.

- Fouten in de hash-functie zelf uiten zich alleen in een sub-optimaal gedrag, en blijven vaak onopgemerkt.

- Het is moeilijk tot ondoenlijk de naamlijst met de hand voor te vullen. Moet dat toch, dan is een speciaal programmaatje de aangewezen weg.

De combinatie hash-functie/voorvulling betekent meestal ook het einde van de overdraagbaarheid van het programma.

7. EEN GEHEEL ANDERE ALGORITME: OPEN-HASH

De open-hash algoritme, die in naam en uiterlijk toch veel op te gesloten-hash lijkt, is de enige mij bekende naamlijst-algoritme die niet gebaseerd is op het idee de verzameling namen door tactisch vragen telkens zo te verkleinen dat uiteindelijk de gezochte overblijft.

In plaats daarvan voeren we een eindige verzameling "pseudonymen" in. Elke naam krijgt (via een hash-functie) een oneindige reeks pseudonymen. En alle namen die we al zijn tegengekomen, kennen we onder een pseudonym, en wel onder het eerste pseudonym in zijn pseudonymen-reeks waaronder we niet al een andere naam kennen.

Het opzoeken gaat nu als volgt. We vragen de binnenkomende naam naar zijn eerste pseudonym. Als we nog niemand onder dat pseudonym kennen, was hij nieuw en wordt hij ingeschreven onder dat pseudonym. Kennen we al wel iemand onder dat pseudonym, dan kijken we of het de binnengekomene is. Zo ja, dan is hij geïdentificeerd; zo nee, dan vragen we de binengekomene naar zijn tweede pseudonym en proberen het opnieuw.

De kracht van dit op het eerste gezicht verbijsterende plan ligt in zijn snelheid: als er maar genoeg pseudonymen zijn is de kans erg groot dat een naam bekend raakt onder zijn eerste pseudonym, en dus met één vergelijking geïdentificeerd kan worden. Zijn zwakte ligt in zijn onvermogen meer namen te identificeren dan er pseudonymen zijn.

Als pseudonymen zijn de integers het gemakkelijkst: het "kennen onder een pseudonym" is dan een eenvoudige indicering. Voor een verzameling pseudonymen ter grootte P wordt de algoritme dan als volgt.

```
mode elem = struct(string name, info info);
[] ref elem N = c een array ter lengte P, geheel gevuld met
                ref elem(nil) c;

proc hash = (string s) []int:
    c een oneindig array van int's uit het interval [1 : P],
      afhankelijk van 's' c;

row open hash = [string T] ref elem:
begin
    [] int H = hash(T);
    ref elem E, bool found := false;

    for M from 1 while not found
    do  E := N[H[M]];
```

```
            if E :=: ref elem(nil)
            then E := heap elem := (T, new); found := true
            elif name of E = T
            then found := true
            fi
        od;
        E
    end
```

(Slechts conformiteit met de voorgaande programma's heeft mij er van weerhouden, dit recursief (of met goto's) op te schrijven. Het machine-code vlaggetje in een abstracte algoritme schijnt onvermijdelijk te zijn.)

Eerste vereiste voor de terminatie van "open-hash" is, dat als M maar hoog genoeg wordt, elk pseudonym een keer aan de beurt komt. Zolang er nog een lege plaats is, wordt deze dan een keer aangewezen. Is deze er echter niet meer, dan termineert de afbeeldings-algoritme niet, wanneer er een onbekende naam binnenkomt.

De gemakkelijkste manier om te zorgen dat alle pseudonymen een keer aan de beurt komen is, om voor M > 1 te nemen:

$$H[M] = H[M-1] \bmod P + 1 .$$

Het nadeel hiervan is, dat voor twee namen die dezelfde H[1] hebben, de hele reeks H[1 :] gelijk is. Dit leidt tot opeenhopingen in de naamlijst, die de efficiëntie ongunstig beïnvloeden. KNUTH [5] beschrijft op blz. 512 verschillende aanpakken om hier iets tegen te doen.

Het probleem dat ontstaat als de pseudonymen uitgeput zijn, is vervelend. HOPGOOD [10] beveelt aan de tabel op een gegeven ogenblik twee keer zo groot te maken en geeft ook grenzen aan waarbij dat zou moeten gebeuren. Meer geheugen geven alleen helpt echter niet. We moeten ook de hash-functie uitbreiden tot het interval [1 : 2*P] en dat betekent het herstructureren van de tabel. Daarvoor heb ik in de literatuur geen algoritmen kunnen vinden.

De eenvoudigste aanpak is, een nieuw array ter lengte 2P te nemen en de namen één voor één over te hevelen. Dan hebben we wel (tijdelijk) 3P aan geheugen nodig. We kunnen ook proberen de tabel in situ op te rekken, maar algoritmen daarvoor worden zeer ingewikkeld.

Zelfs als we al een oplossing vinden voor het herstructurerings-probleem, dan nog gedraagt de algoritme zich schokkerig: voor het toevoegen van één naam is soms plotseling een grote hoeveelheid geheugen en tijd nodig. Willen we deze algoritme kunnen aanbevelen, dan moeten de andere eigenschappen wel zéér goed zijn. Dat zijn ze dan ook.

De klassieke (niet-groeiende) "open-hash algoritme" is de snelste zoek-algoritme die bekend is: gemiddeld 1.5 string-vergelijkingen per naam; begint het array echter vol te raken, dan neemt de efficiëntie snel af en bij een geheel vol array werkt de algoritme niet meer.

De "groeiende open-hash algoritme" is ook zeer snel: gemiddeld nog steeds 1.5 string-vergelijkingen + een constant bedrag voor herstructurering per naam. En het belangrijkste hierbij is dat deze hoge efficiëntie gehandhaafd blijft, ongeacht het aantal namen. Dit is de enige algoritme die lineair is in het aantal namen!

Net als bij de "gesloten-hash" treden hierbij de bekende nadelen op:
- moeilijke debugging,
- slecht met de hand voor te vullen,
- weinig overdraagbaar,
- geen alfabetische volgorde.

8. SAMENVATTING

In het voorgaande hebben we een groot aantal algoritmen de revue laten passeren; vier ervan steken met kop en schouders boven de rest uit:

- binair zoeken in een verdund array:
  - is tamelijk eenvoudig te programmeren,
  - is van orde N ln(N) en heeft zeer weinig extra geheugen nodig,
  - houdt de namen in alfabetische volgorde;

- duimgrepen van twee letters + binaire bomen:
  - is tamelijk eenvoudig te programmeren,
  - is ook van orde N ln(N) maar meestal veel sneller dan de voorgaande, en heeft relatief veel geheugen nodig.
  - houdt de namen in alfabetische volgorde;

- gesloten hash + gelinkte lijsten:
  - is zeer eenvoudig te programmeren,
  - is in principe van orde N^2 maar in de praktijk van orde N en ligt wat snelheid en geheugenbeslag betreft tussen de beide voorgaande,
  - houdt de namen niet in alfabetische volgorde;

- groeiende open-hash:
  - is tamelijk ingewikkeld te programmeren (herstructureringsproblemen).
  - is van orde N, en zeer zuinig in het geheugengebruik,
  - houdt de namen niet in alfabetische volgorde.

Voor verdere details verwijs ik weer naar [2].

9. ENKELE VOORBEELDEN

Het is misschien interessant enige voorbeelden uit de praktijk te bezien.

De ALGOL 60-vertaler op de EL-X8 [11] gebruikte een gesorteerd array (en produceerde geen cross-reference listing).

De nieuwe ALEPH-vertaler (waarvoor dit hele onderzoek uiteindelijk begonnen was) gebruikt "binair zoeken in een verdund array", vanwege de overdraagbaarheid, de cross-reference listing en de bestendigheid tegen gesorteerde invoer. (Voor ALEPH zie [12]).

De A68H-vertaler van Hendrik Boom gebruikt "gesloten hash met P = 256 + binaire bomen", vanwege de relatieve eenvoud en goede bruikbaarheid voor zowel zeer kleine als zeer grote aantallen namen.

De CDC ALGOL 68-vertaler [13] gebruikt "gesloten hash met P = 128 + ongesorteerde lijsten", omdat de vertaler voornamelijk op kleinere programma's rekent, en de opzoektijd toch in het niet zinkt vergeleken bij de (zeer uitgebreide) code-optimalisatie.

De CDC ALGOL 60-vertaler ("ALGOL 5") gebruikt dezelfde algoritme, maar met gesorteerde lijsten, om gemakkelijker een cross-reference listing te kunnen geven.

De UNIX C-compiler gebruikt een niet-groeiende open-hash, met verschillende array's voor de verschillende soorten namen.

LITERATUUR

[1] GRUNE, D., *Some Statistics on ALGOL 68 Programs*, IW 109/79, Mathematisch Centrum, Amsterdam, 1979.

[2] GRUNE, D., *Choosing a Tag-list Algorithm for a Compiler with Special Application to the ALEPH Compiler*, Software-Practice and Experience, 9,7 (Juli 1979), pp.575-593.

[3] YAO, A.C. & F.F. YAO, *The Complexity of Searching an Ordered Random Table*, 17th Annual Symp. on Foundations of Comp. Science, IEEE Comp. Soc., 1976, pp.173-177.

[4] MATTHIJSSEN, R.L. & R.C. UZGALIS, *Some Simple Data Structures in a Paged Environment*, The UCLA Comp. Soc. Dept. Quarterly, 4, 1 (Januari 1976), p.67.

[5] KNUTH, D.E., *The Art of Computer Programming*, Vol. 3: *Sorting and Searching*, Addison Wesley, New York, 1973.

[6] BAER, J.-L. & B. SCHWAB, *A Comparison of Tree-Balancing Algorithms*, Comm. ACM, 20, 5 (Mei 1977), pp.322-330.

[7] KOK, G., *Alfabetiseren en andere sorteerwerkzaamheden*, MR 120/70, Mathematisch Centrum, Amsterdam, 1970, p.35.

[8] COFFMAN, E.G. & J. EVE, *File Structures Using Hash Functions*, Comm. ACM, 13, 7, (Juli 1970), pp.427-436.

[9] LUNE, J. VAN DE, *On the Asymptotic Behaviour of a Sequence Arising in Computer Science*, ZW 31/74, Mathematisch Centrum, Amsterdam, 1974.

[10] HOPGOOD, F.R.A., *A Solution to the Table Overflow Problem for Hash Tables*, Computer Bulletin, 11, 4 (Maart 1967), pp.297-300.

[11] KRUSEMAN ARETZ, F.E.J. & B.J. MAILLOUX, *The ELAN source text of the MC ALGOL 60 system for the EL X8, MR 84*, Mathematisch Centrum, 1966.

[12] GRUNE, D., R. BOSCH & L.G.L.T. MEERTENS, *ALEPH Manual*, IW 17/75, Mathematisch Centrum, Amsterdam, 1975.

[13] ALGOL 68 Version I Reference Manual, Control Data Services B.V., Rijswijk, 1976.

CODE-GENERATIE

L.G.L.T. MEERTENS
Mathematisch Centrum

1. INLEIDING

Een programma, geschreven in een programmeertaal, beschrijft een algoritmisch proces. Als we dat proces uitgevoerd willen hebben, kunnen we dat in principe zelf met potlood en papier doen. Voor sommige toepassingen gaat dat niet snel genoeg en schakelen we een electronisch apparaat in. Dat apparaat moet zich houden aan de semantiek van de taal. Als we ervan uitgaan dat die semantiek operationeel beschreven is in termen van een abstracte machine, zouden we die machine in hardware kunnen nabouwen. Om diverse, onder meer ekonomische, redenen, is dit geen algemeen gebruik. Het komt vaker voor dat er al een konkrete "doelmachine" is, en dat we die zover moeten krijgen dat hij als model kan dienen voor de abstracte machine. Dat kan via verschillende technieken, zoals interpretatie en compilatie. In beide gevallen zal, om redenen van efficiëntie, het programma in het algemeen aan een analysefase onderworpen worden voordat het proces op de doelmachine wordt gestart dat het oorspronkelijke algoritmisch proces modelleert. Het verschil zit hem in de staart. Bij compilatie wordt het geanalyseerde programma "vertaald" tot een programma voor de doelmachine. Deze laatste fase van een compilatie is de "code-generatie". Daarin wordt het te verwerken programma, dat al in een aantal eerdere fasen is voorgekauwd, uiteindelijk omgezet in een stroom instructies voor de doelmachine: de "code".

In het grijze verleden woedde een hevig debat tussen voor- en tegenstanders van hogere programmeertalen. De tegenstanders argumenteerden dat een compiler nooit de kwaliteit code kon afleveren die een getrainde pro-

grammeur (of "codeur") met de hand zou produceren. Waarschijnlijk om die reden is er van het begin af aan veel nadruk gelegd op het ontwikkelen van code-generatoren die code van "hoge kwaliteit" genereerden (d.w.z., efficiënte code). De literatuur over code-generatie beperkt zich bijna uitsluitend tot dat aspect. Er is alle reden enigszins gereserveerd te staan tegenover deze nadruk op de efficiëntie: de aldus verkregen besparingen moeten in een redelijke verhouding staan tot de extra kosten bij de toepassing (en ontwikkeling) van de code-generator. Hier geldt, als bij zoveel optimalisatie-problemen, dat in het begin drastische verbeteringen kunnen worden verkregen met simpele middelen, maar dat al spoedig een gebied bereikt wordt waar de zwaarste machinerie nog slechts een marginaal resultaat boekt.

Die eenzijdige nadruk houdt ook in dat de theorie van code-generatie voor het overige onderontwikkeld is. Dit staat in schrille tegenstelling tot de veelheid aan bruikbare resultaten op het gebied van formele talen en ontledingstechnieken. Standaardwerken op het gebied van vertalerbouw (LEWIS, ROSENKRANTZ & STEARNS[5], AHO & ULLMAN[1]) wijzen op de afhankelijkheid van de doelmachine, en beperken zich dan tot de behandeling van een voorbeeld.

In het volgende zal juist een wat afstandelijker behandeling voorop staan. Dat wil niet zeggen dat hier een algemene theorie van code-generatie wordt ontvouwd; hooguit een wat theoretischer benadering, met de nadruk op het construeren van een code-generator.

Aan de code-generatie gaan de fasen van syntactische en semantische analyse van het bronprogramma vooraf. Dit hoeft niet in te houden dat het gehele programma geanalyseerd is voordat de code-generatie een aanvang neemt. Of deze fasen geïntegreerd kunnen worden zal van de programmeertaal afhangen. In het volgende zal echter gedaan worden alsof deze fasen volledig in de tijd gescheiden zijn. Het vertrekpunt voor de code-generator is dan de geanalyseerde tekst, bijv. in de vorm van een ontledingsboom (mogelijk gelineariseerd door de prefix-representatie) aangevuld met tabellen.

De semantische analyse zal hier verder buiten beschouwing blijven. Wel zij opgemerkt dat deze in het algemeen mede zal dienen om de code-generatie gemakkelijker te laten lopen. In een taal waarin identifiers gedeclareerd

worden en daarbij een type uitgedeeld krijgen, is het bijv. onhandig bij iedere toepassing van een identifier op jacht te moeten gaan in het bronprogramma naar de bijbehorende declaratie teneinde dat type te achterhalen. Vragen die de code-generator waarschijnlijk meermalen zal stellen, kunnen dan eenmalig uitgezocht worden, waarbij het antwoord via tabellen kan worden overgedragen.

2. CODE-GENERATIE ALS PARTIËLE BEREKENING

Code-generatie kan beschouwd worden als een toepassing van een algemene optimalisatie-techniek, namelijk het partiële-berekeningsprincipe (ERSHOV[3]). Ruw gezegd komt dit principe neer op de mogelijkheid een programma gedeeltelijk uit te voeren als bepaalde invoergegevens bekend zijn. Er blijft dan een "rest-programma" over. Een voorbeeldje moge de bedoeling verduidelijken. Zij het volgende programmaatje voor machtsverheffing gegeven:

```
y:= 1;
while n > 0
do if odd n
   then y:= y * x; n:= n - 1
   fi;
   n:= n ÷ 2; x:= x * x
od.
```

Als gegeven is dat n gelijk aan 5 is, is dit ekwivalent met

```
y:= 1; y:= y * x; x:= x * x; x:= x * x; y:= y * x; x:= x * x.
```

Dit is dus het rest-programma voor n = 5.

Een programma om zo'n rest-programma te genereren kan uit het oorspronkelijke programma worden afgeleid door de niet bij voorraad uitvoerbare stukjes "uit te stoten". Hiertoe voeren we een constructie

```
>>S
```

in, waarbij S een statement of iets dergelijks is, met als betekenis: voeg "S" toe aan het rest-programma in opbouw. Hierbij kan S op de positie van uitdrukkingen constructies bevatten van de vorm

```
!E
```

waarin E een uitdrukking voorstelt die tijdens de partiële berekening geëvalueerd moet worden, waarna "!E" in S vervangen wordt door een representatie van de uitkomst. Voorbeeld:

```
>>x:= m[i + !(offset - 1)].
```

Als, op het ogenblik van uitvoering (tijdens de partiële berekening) de waarde van offset 100 is, leidt dit tot het toevoegen van

```
"x:= m[i + 99]"
```

aan het rest-programma. Voor het bovenstaande voorbeeld krijgen we:

```
>>y:= 1;
while n > 0
do if odd n
   then >>y:= y * x; n:= n - 1
   fi;
   n:= n ÷ 2; >>x:= x * x
od.
```

(De constructie "!E" komt hierin niet voor.)

Als het oorspronkelijke programma een keuze-constructie bevat (bijv. if ... then ... else ... fi) waarvan de test niet tijdens de partiële berekening al geëvalueerd kan worden tot een waarde, zal ook in het rest-programma een keuze-constructie moeten voorkomen, en moet ieder der niet uitgesloten alternatieven onderworpen worden aan een partiële berekening. Verder is het voor veel toepassingen van het principe wenselijk dat de par-

tiële berekening gegarandeerd convergeert. Bij herhalings-constructies (zoals while ... do ... od) is daarom behoedzaamheid geboden.

Toepassing van het partiële-berekeningsprincipe op een interpretator levert een code-generator op. De semantiek van een programmeertaal kan operationeel beschreven worden als een interpretator S (een programma) met parameters P (het bronprogramma) en X (de invoer voor P). Door uit S(P, X) het rest-programma S_P af te leiden voor gegeven P, is een vorm van doelcode bereikt. $S_P(X)$ beschrijft dan een berekening die ekwivalent is met S(P, X), maar (waarschijnlijk aanmerkelijk) efficiënter.

Om het beoogde effect te bereiken, is het wel noodzakelijk dat S aan heel bepaalde voorwaarden voldoet. Het gaat er immers niet om een willekeurige vorm van doelcode te bereiken, maar code voor een bepaalde doelmachine. De interpretator S moet dus zelf min of meer een programma in die code zijn. Min of meer, want ten aanzien van delen waarvan vaststaat dat ze in het proces van partiële berekening zullen worden geëlimineerd, kunnen willekeurige uitbreidingen worden toegelaten. Een tweede afzwakking is het onderwerp van de volgende sectie.

Het beschrevene is niet een geheel nieuwe techniek van code-generatie. Integendeel, het is een abstracte beschrijving van datgene wat ontwerpers van code-generatoren in het algemeen al doen.

Een voorbeeld: Een interpretator voor een taal waarin statements voorkomen als

```
a:= b + c * d
```

kan beschreven worden door een systeem van routines die ieder een bepaald "sjabloon" behandelen. Een greep hieruit zou kunnen gevormd worden door:

```
I(V:= E):
    l:= Iloc(V);
    v:= Ival(E);
    STORE(v, l).
```

```
Ival(E1 + E2):
    v1:= Ival(E1);
    v2:= Ival(E2);
    ADD(v1, v2, r);
    return(r).
```

Om hieruit een generator af te leiden, worden de locale variabelen als l en v van de routines vervangen door een "model" van de variabelen in de doelmachine. De routines kunnen dan een vorm krijgen als:

```
G(V:= E):
    ml:= Gloc(V);
    mv:= Gval(E);
    >>STORE(!mv, !ml).

Gval(E1 + E2):
    mv1:= Gval(E1);
    mv2:= Gval(E2);
    mr:= temploc;
    >> ADD(!mv1, !mv2, !mr);
    return(mr).
```

Zoals al vermeld, zullen we in de praktijk niet werken met het ruwe bronprogramma, maar zal P zoiets zijn als een ontledingsboom aangevuld met tabellen en dergelijke. Dit is een geschikte representatie van het programma om sjablonen als "V:= E" te kunnen herkennen.

3. OPTIMALISATIE-TRANSFORMATIES

In het algemeen zal een code-generator, afgeleid door rechtlijnige toepassing van het partiële-berekeningsprincipe, wanneer geen bijzondere maatregelen worden genomen, vaak "domme" code genereren. Maatregelen om daarvoor een stokje te steken door de code-generator aan te passen leiden weer tot een toename in de complexiteit. Maar als er nog een optimalisatie-fase op de code-generatie volgt, kunnen we ons gerust permitteren in een aantal

gevallen zulke domme code te genereren, in de wetenschap dat deze vervolgens tot iets aanvaardbaars wordt opgepoetst.

Uit de talloze mogelijkheden volgen er hier enkele. Het gaat hierbij steeds om transformaties van de code die de semantiek onveranderd laten.

a) Kijkgat-optimalisatie (McKEEMAN[4]). De gegenereerde code wordt door een "kijkgat" beschouwd waarbij steeds enkele opeenvolgende instructies tegelijk zichtbaar zijn. Als hierin een rijtje van pakweg twee of drie instructies voorkomt dat vervangen kan worden door iets efficiënters (volgens een tevoren bepaald, beperkt repertoire van vervangingsschema's) is een kijkgat-optimalisatie mogelijk.

b) Vooruitschuiven van constanten. Als in een lineaire rij instructies (zonder mogelijkheid tot binnenspringen) van variabelen bekend is dat ze een constante waarde hebben, kan deze bij een volgend gebruik van de variabele worden ingevuld. Hierdoor wordt bijv.

```
a:= 1; b:= 2 * a; c:= b + 5
```

vervangen door

```
a:= 1; b:= 2; c:= 7.
```

(Merk op dat bovendien expressies waar mogelijk vast geëvalueerd zijn.)

c) Vooruitschuiven van variabelen. Als in een instructierij bekend is dat twee variabelen dezelfde waarde hebben, kan bij gebruik van de ene variabele de andere worden genomen. Voorbeeld:

```
b:= a; c:= b; d:= c + 1
```

kan vervangen worden door

```
b:= a; c:= a; d:= a + 1.
```

d) Eliminatie van dode toekenningen. Als een variabele niet meer gebruikt wordt (of een nieuwe waarde krijgt toegekend voor gebruikt te worden), kan de toekenning aan die variabele uit de code geschrapt worden. Zo wordt het effect van

```
a:= b; b:= c; a:= d
```

ook bereikt door

```
b:= c; a:= d.
```

Dit is, zoals gezegd, maar een greep uit de mogelijke transformaties. Sommige hebben voornamelijk zin in combinatie met andere, zoals het vooruitschuiven van variabelen om dode toekenningen te elimineren. Welke locale optimalisaties zinvol zijn, hangt af van de "domheid" van de code-generator. Het eerder gegeven rest-programma voor verheffing tot de vijfde macht kan worden geoptimaliseerd tot

```
y:= 1 * x; x:= x * x; x:= x * x; y:= y * x
```

door het vooruitschuiven van de constante 1 en het elimineren van enkele dode toekenningen.

4. TUSSENFASEN

De (denkbeeldige) interpretator S zal een abstracte machine modelleren waarop P kan worden uitgevoerd. Nu zijn de konkrete machines die ons ter beschikking staan in het algemeen ver verwijderd van het abstractieniveau van de operationele semantiek van programmeertalen. Om de partiële-berekeningstechniek toe te passen, zou de interpretator eerst omgewerkt moeten worden tot een interpretator voor de feitelijke doelmachine. Dit is echter vaak ongewenst; het zou de complexiteit van het probleem tot onhanteerbare proporties kunnen opblazen. Daarom zal het vaak zinvol zijn een tussenmachine met een eigen tussencode te verzinnen en het probleem in twee stappen aan te pakken. Die tussenmachine kan bijv. "lage" grootheden als wijzers

kennen, maar nog abstraheren van sommige konkrete problemen van het geheugenbeheer en de registers. De vrijheid bij het ontwerpen van de tussenmachine moet uiteraard gebruikt worden om deze nauw te laten aansluiten bij de abstracte machine van de operationele semantiek. We krijgen dan een proces in twee stappen,

B -> T -> D,

omzetting van de brontaal B naar de tussentaal T, en omzetting van T naar de doeltaal D. (Meer tussenfasen zijn natuurlijk ook mogelijk.)

Deze gang van zaken biedt extra voordelen. Het belangrijkste voordeel is wellicht dat bij het overdragen van de verlater naar een andere doelmachine niet de gehele code-generator hoeft te worden herschreven, maar alleen de fase T -> D. Een ander voordeel is aangeduid door CARTER[2]. Bij het toepassen van locale optimaliserende code-transformaties, zoals in de vorige sectie aangeduid, moet uiteraard voorop staan dat de betekenis van het programma onveranderd blijft. De complexe semantiek van konkrete machines maakt echter het toepassen van zulke transformaties tot een hachelijke aangelegenheid. We hebben wel de semantiek van de tussenmachine in de hand, en kunnen deze zo helder houden als ons zelf belieft. Ook in het licht van de mogelijkheid dat de fase T -> D voor verschillende doelmachines moet worden herschreven, is het gewenst zoveel mogelijk de optimalisatie aan deze fase te laten voorafgaan.

5. VAN TUSSENCODE NAAR DOELCODE

Bij een fase als T -> D is het niet meer mogelijk D volledig naar onze hand te zetten. Tot op zekere hoogte kan dit wel. Als de programmeertaal bijv. reële aritmetiek kent maar de doelmachine daarvoor geen ingebouwde voorzieningen heeft, kunnen subroutines voor de reële aritmetische operaties ontwikkeld worden; de instructies die deze subroutines aanroepen kunnen dan beschouwd worden als een uitbreiding van het instructierepertoire van de doelmachine. Deze uitbreidingen vormen tezamen het "run-time-systeem". We zullen aannemen dat dergelijke uitbreidingen al geschied zijn,

zodat instructies uit de tussencode kunnen worden omgezet in een redelijk bescheiden aantal doelinstructies.

Hoewel vanuit een abstract standpunt het omzetten van code naar code ook beschouwd kan worden als een toepassing van het partiële-berekeningsprincipe, biedt dit toch betrekkelijk weinig houvast als het erom gaat deze fase te construeren. Een betere ingang wordt verkregen door de volgende beschouwing:

Als eerste benadering zouden we ons kunnen voorstellen dat iedere tusseninstructie vervangen wordt door een of meer doelinstructies. Het uitvoeren van tusseninstructies bewerkstelligt een verandering van de toestand S_t van de tussenmachine in een nieuwe toestand S_t'. Deze verandering moet afgebeeld worden op de doelmachine. Hiertoe worden de toestanden S_d van de doelmachine geïnterpreteerd als toestanden van de tussenmachine. Die interpretatie geschiedt via een interpretatiefunctie I, en wel volgens $S_t = I(S_d)$. De tusseninstructie moet nu vervangen worden door zodanige doelinstructies dat S_d overgaat in een nieuwe toestand S_d', waarbij de geldigheid van de interpretatie overeind blijft. Dat wil zeggen: uit $S_t = I(S_d)$ moet $S_t' = I(S_d')$ volgen. Nu zijn er talloze manieren om I te kiezen. Je zou verwachten dat I voor een bepaalde code-omzettingsfase eens en voor al vast gekozen moet worden. Maar daarvoor is geen enkele noodzaak. Het enige dat nodig is, is dat I bij ieder punt vastligt, zodat niet bijv. vanuit twee verschillende interpretaties een sprong kan plaatsvinden naar dezelfde label in het doelprogramma.

Laten we besluiten bij het punt waar we met de code-omzetting zijn aangeland (bij de overgang van S_t naar S_t') over te gaan op een nieuwe interpretatie J, die zo gekozen wordt dat $J(S_d) = I(S_d')$. De doelinstructies moeten nu S_d omzetten in een S_d'', zodanig dat uit $S_t = I(S_d)$ de geldigheid van $S_t' = J(S_d'')$ volgt. Hieraan is al voldaan door de keuze $S_d'' = S_d$, zodat er geen doelinstructies meer nodig zijn. De tusseninstructie is dus "vertaald" in nul doelinstructies, dankzij de overgang van I naar J.

Natuurlijk kan deze gedachtengang niet onbeperkt worden gevolgd, anders zou ieder programma tot een leeg doelprogramma kunnen worden teruggebracht. Een eerste beperking is gelegen in het feit dat bij een punt dat langs meer wegen bereikt kan worden, een vaste interpretatie moet worden gekozen. Als

een van de wegen een andere interpretatie heeft, zeg J i.p.v. I, moeten doelinstructies ingelast worden om J tot I te herleiden, d.w.z., instructies die de toestand S_d omzetten in een toestand S'_d zodat $J(S_d) = I(S'_d)$.

Maar er is meer. Om deze methode toe te passen zal de code-omzetter moeten manipuleren met representaties van de interpretatiefunctie. Deze representaties moeten binnen zekere grenzen van eenvoud blijven om de methode hanteerbaar te houden. Er moet een vaste functie II zijn die vertelt welke interpretatie bij een gegeven representatie mI hoort. Omdat het er uiteindelijk om gaat de uitspraak $S_t = II(mI)(S_d)$ invariant te laten, kan II eventueel ook impliciet gegeven zijn door middel van deze uitspraak. Voor een tusseninstructie hoeft geen code gegenereerd te worden als we vanuit een interpretatiefunctie I = II(mI) naar een nieuwe functie J toe willen, en het lukt een mJ te vinden met J = II(mJ). Maar in het algemeen zal zo'n mJ niet bestaan. Het streven blijft er echter op gericht zoveel mogelijk "symbolisch" te rekenen op de interpretatiefunctie en zo weinig mogelijk doelcode te genereren.

Als illustratie zal de omzetting beschreven worden van code voor een stapel-machine naar een machine met één register. Een greep uit het instructierepertoire van de stapel-machine:

```
PUSH i : p:= p+1; S[p]:= i
POP    : p:= p-1;
ADD    : p:= p-1; S[p]:= S[p] + S[p+1]
```

en van de doelmachine:

```
Li     : R:= M[i]
Si     : M[i]:= R
Ai     : R:= R + M[i]
Ci     : R:= i.
```

Het is in principe heel goed mogelijk iedere instructie om te zetten in een min of meer vast rijtje doelinstructies. Voor een programmaatje als

```
PUSH 100
PUSH 200
ADD
PUSH 300
PUSH 400
ADD
ADD
```

krijgen we dan typisch zoiets als

```
C100          " PUSH 100
S0
------------------------
C200          " PUSH 200
S1    *
------------------------
L1    *       " ADD
A0
S0
------------------------
C300          " PUSH 300
S1
------------------------
C400          " PUSH 400
S2    *
------------------------
L2    *       " ADD
A1
S1    *
------------------------
L1    *       " ADD
A0
S0.
```

De met "*" aangegeven opvolgingen "Si; Li" kunnen hieruit door kijkgat-optimalisatie worden geschrapt. Dan nog blijft er een formidabele hoe-

veelheid instructies over. In dit voorbeeld is gewerkt met een vaste interpretatie:

$$S[i] = M[i],\ i = 0, \ldots, p.$$

(Omdat p tijdens code-omzetting kan worden bijgehouden, is het niet noodzakelijk p in de toestand van de doelmachine op te nemen.)

Met een variabele interpretatie kunnen we veel betere resultaten boeken. Hiertoe voeren we het representatie-type

```
access::= constant(v: int) | mem | reg
```

in. Dit wil zeggen: een waarde van het type access is of een "constant" en bevat dan een geheel getal, of "mem", of "reg". De laatste twee waarden zijn atomair; er valt niets meer over te melden dan dat ze van het type access zijn en verschillend.

Met behulp hiervan kunnen we de wijze beschrijven waarop de stapel in de doelmachine gemodelleerd is. De representatie van de interpretatie wordt dan gegeven door een array mS, die voor iedere stapel-cel een "access" bevat. In de toestand van de doelmachine kunnen deze afgebeeld worden op een geheugencel ("mem"), het register ("reg") of niet afgebeeld hoeven te zijn omdat de waarde van de inhoud tijdens code-omzetting bekend is ("constant"). Door het simpele voorbeeld is het bij een access "mem" niet nodig bij te houden welke cel het betreft, als we maar bijhouden tot waar we gebleven zijn met het uitdelen van geheugen (in de variabele m).

De interpretatie is nu gegeven door:

```
S[i] = case mS[i] in
          constant(v): v,
          mem        : M[f(i)],
          reg        : R
       esac,
```

voor i = 0, ..., p. Hierbij is f een hulpfunctie die gegeven is door:

```
f(0) = 0,
f(k+1) = f(k) + case mS[k] in
                    constant(): 0,
                    mem       : 1,
                    reg       : 0
                esac.
```

Deze functie telt het aantal bezette geheugencellen. Voor de eerder genoemde variabele m hebben we dan: m = f(p+1) - 1, en dus in het bijzonder, als mS[p] = mem, m = f(p).

Bovendien zullen we eisen dat voor ten hoogste één i tegelijk mS[i] = reg geldt, en dat deze i dan voldoet aan f(i) = m+1 (d.w.z., "boven" i komt geen access mem voor). Deze restrictie is niet wezenlijk, maar vereenvoudigt de behandeling.

De omzetting van PUSH, POP en ADD kan nu als volgt beschreven worden:

```
G(PUSH i):
    p:= p+1; mS[p]:= constant(i).

G(POP):
    p:= p-1;
    if mS[p+1] = mem then m:= m-1 fi.

G(ADD):
    p:= p-1;
    mS[p]:= Gadd(mS[p], mS[p+1]).
```

Hierna zal Gadd ontwikkeld worden als een verzameling kleine routines om de verschillende access-combinaties te behandelen. Merk op dat vanwege de eerder genoemde restrictie (reg, mem) en (reg, reg) niet kunnen voorkomen.

Een aardige optimalisatie ligt voor de hand als een van beide parameters constant(0) is; immers, zoals bekend, is 0+x = x+0 = x. Ook op het symbolische niveau waarop we werken, kunnen we deze identiteit toepassen.

De bedoeling is dat deze optimalisatie, indien toepasbaar, voorrang heeft. (Anders blijft de code correct, maar is hij minder efficiënt.) We krijgen dan:

```
Gadd(constant(0), a2):
    return(a2).

Gadd(a1, constant(0)):
    return(a1).
```

Ook de behandeling van twee constanten verloopt zeer soepel:

```
Gadd(constant(v1), constant(v2)):
    return(constant(v1+v2)).
```

Dit is hoever we kunnen komen door alleen de interpretatie aan te passen. Als we iets anders hebben, zullen er een of meer doelinstructies gegenereerd moeten worden. Er is eigenlijk maar één situatie waarin de optelling rechtstreeks gegenereerd kan worden. De A-instructie werkt immers op het register en een geheugencel. Om deze toe te passen, zijn een access mem en een access reg vereist:

```
Gadd(mem, reg):
    >>A!m; m:= m-1;
    return(reg).
```

De overblijvende combinaties zullen op een of andere manier naar (mem, reg) "toegepraat" moeten worden. Aangezien een access reg het makkelijkst te verkrijgen is, beginnen we met de gevallen waarin al één access mem is. Een tweede access constant kan met de C-instructie naar reg worden toegebracht:

```
Gadd(mem, constant(v2)):
    >>C!v2;
    return(Gadd(mem, reg)).
```

Van een mem-access kunnen we ook gemakkelijk een reg krijgen door toepas-

sing van de L-instructie:

```
Gadd(mem, mem):
    >>L!m; m:= m-1;
    return(Gadd(mem, reg)).
```

In geval de tweede parameter van Gadd een access mem is, kunnen we eenvoudig de parameters omwisselen om een eerder behandelde combinatie te krijgen. In feite blijkt het nog slechts om één mogelijkheid te gaan:

```
Gadd(constant(v1), mem):
    return(Gadd(mem, constant(v1)).
```

In de resterende combinaties heeft al een van de parameters een access van de vorm reg. Dit helpt echter weinig, omdat het andere access een constant is; hiervan kan geen mem-access gemaakt worden zonder het register te gebruiken. Wel is het zo dat een S-instructie van reg naar mem gaat, zodat we daarmee een reeds behandelde combinatie kunnen bereiken:

```
Gadd(reg, constant(v2)):
    m:= m+1; >>S!m;
    return(Gadd(mem, constant(v2))).
```

Er blijft nog één combinatie over, waarvoor eveneens de parameters kunnen worden omgewisseld om een bekende combinatie te verkrijgen:

```
Gadd(constant(v1), reg):
    return(Gadd(reg, constant(v1))).
```

Als we deze code-omzetter toepassen op het eerdere voorbeeld, dan blijkt er nog in het geheel geen code gegenereerd te worden, aangezien bij deze flexibele representatie alle programmaatjes die alleen uit PUSH-, POP- en ADD-instructies bestaan al tijdens code-omzetting kunnen worden geëxecuteerd. Andere instructies kunnen dit echter onmogelijk maken (bijv. een READ-instructie). De gegeven stukken code-omzetter blijven dan echter overeind staan; hun correctheid is gebaseerd op het invariant houden van de

interpretatie S_t = II(mI)(S_d) en kan niet teloor gaan door verdere toevoegingen aan de code-omzetter.

6. ACCESSBEPALING

In het algemeen zullen de accessen veel ingewikkelder zijn dan in dit kunstmatige voorbeeld. In een taal met een blok-structuur zal het access bijv. het display-niveau en de offset ten opzichte van een geheugen-segment bevatten. Bij een machine met meer registers moet een access als reg worden gecompliceerd tot iets als reg(i: 0..7).

Wanneer de accessen in diversiteit en complexiteit toenemen, wordt het niet realistisch om, zoals hierboven bij Gadd is gedaan, "met de hand" alle combinaties af te lopen. Het wordt dan zaak de daar geschetste redenering zoveel mogelijk te systematiseren en te mechaniseren. Hiertoe is het wenselijk een arsenaal van manipulaties te hebben die (onder gegeven condities) de geldigheid van de interpretatie invariant laten bij onveranderlijke toestand van de tussenmachine. (Voor het boven gegeven voorbeeld: indien mS[p] = mem, dan is

```
>>L!m; m:= m-1; mS[p]:= reg
```

er een.) Bij een machine met meer registers en instructies in de trant van ADD, Ri, Rj, Rk zal er een routine in de code-omzetter zijn die een willekeurig access omtovert in een reg-access.

Hierbij komen we aan het probleem van de register-toewijzing. Het loont in het algemeen de moeite resultaten zo mogelijk in een register te houden. Dat scheelt transporten van het geheugen naar het register, en bij tussenresultaten mogelijk ook het omgekeerde transport. Ook bij machines waar de instructies rechtstreeks uit het geheugen kunnen putten, is het meestal sneller als de operanden in een register staan. Nu is het aantal registers eindig en meestal vrij beperkt. Af en toe zal bij de code-omzetting een access reg(i) moeten worden "losgelaten" om een register vrij te maken. Dit is in feite hetzelfde probleem als dat van de "working set" in een paginerings-systeem. Een aanvaardbare oplossing is het register vrij te geven dat

het langst niet is geraadpleegd. Voor het working-set-probleem is een theoretische optimale oplossing bekend: geef de pagina vrij die het langst niet gebruikt zal worden. Deze oplossing is in de praktijk onbruikbaar omdat de toekomst onbekend is. Voor een lineaire rij instructies is de toekomst echter wel bekend! Door de rij van achter naar voren door te lopen, kan een optimale oplossing eenvoudig berekend worden. Zo'n achterwaartse wandeling kan tevens eenvoudig constateren waar variabelen dood worden, waardoor de ruimte die zij in beslag nemen kan worden vrijgegeven.

Helaas gooien keuze- en herhalings-constructies hier roet in het eten. Daardoor zijn we toch gedwongen tot een heuristische aanpak.

Desalniettemin kan het zinvol zijn de register-toewijzing, en in het algemeen de toewijzing van accessen, te laten voorafgaan aan de code-omzetting. In dat geval is het wel zo verstandig deze niet dwingend op te leggen aan de code-omzetter, maar als advies aan te bieden. De code-omzetter dient zelf op principes gestoeld te zijn waarbij de correctheid voorop staat. Als het advies binnen deze principes toepasbaar is (en niet herkenbaar dwaas), wordt het toegepast; anders wordt het in de wind geslagen. Hiermee wordt bereikt dat de binding tussen de access-adviseur en de code-generator zeer los is. Beide kunnen onafhankelijk gewijzigd worden zonder dat de correctheid van de doelcode in gevaar komt. Dit laat meer ruimte toe om wat te experimenteren en te sleutelen met de access-adviseur. Bij de keuze- en herhalings-constructies is het mogelijk een simpele heuristiek te kiezen die mogelijk een ontoepasbaar advies geeft (bijv. onverenigbare adviezen voor verschillende alternatieven).

7. SAMENVATTING

Een code-generator, geconstrueerd volgens de boven aangegeven lijnen, bestaat idealiter uit de volgende fasen:

i) de fase B -> T;
ii) locale optimalisatie T -> T;
iii) access-adviesbepaling;
iv) code-omzetting T -> D.

Zoals al eerder gesteld is voor de vertaler als geheel, hoeven deze fasen niet per se alle aanwezig te zijn en hoeven zij ook niet per se volledig in de tijd gescheiden te zijn. Verder is het zeer wel mogelijk dat er meer tussenfasen zijn.

In ieder geval is het geboden deze fasen met een zo scherp mogelijke interface te ontwikkelen. Een expliciete tekstuele voorstelling van de tussencode zal daarbij behulpzaam zijn. Deze zal bovendien bij de ontwikkeling grote diensten bewijzen. Verder zij opgemerkt dat het ook verstandig is de optimalisatie en de adviesbepaling zo te maken dat deze ontkoppeld of door iets veel eenvoudigers vervangen kunnen worden als blijkt dat ze meer kosten dan opleveren.

De feitelijke verschijningsvorm van de fasen hoeft evenmin op het oog te lijken op de hier gegeven abstracte schetsen. Een voor de hand liggende implementatie van de sjabloon-herkenning is bijvoorbeeld een LL(1)-achtige ontledingstechniek toe te passen op een prefix-representatie van de ontledingsboom (bijv. via recursieve afdaling of met een stapel) en daaraan "semantische acties" te verbinden. Voor de code-omzetting kunnen tabellen waarin de beschikbare manipulaties gerepresenteerd zijn lange lappen programma vervangen.

Welke aanpak ook gekozen wordt, code-generatie is een complex probleem, complex ook vergeleken bij het ontledingsprobleem; alle mogelijkheden om daarin structuur aan te brengen die de complexiteit tot hanteerbare proporties terugbrengt, moeten dankbaar worden aangegrepen.

LITERATUUR

[1] AHO, A.V. & J.D. ULLMAN, *Principles of Compiler Design*, Addison-Wesley, Reading, Mass., 1977.

[2] CARTER, J.L., *A case study of a new code generation technique for compilers*, Comm. ACM 20 (1977) 914-920.

[3] ERSHOV, A.P., *On the partial computation principle*, Information Processing Letters 6 (1977) 38-41.

[4] McKEEMAN, W., *Peephole optimization,* Comm. ACM 8 (1965) 443-444.

[5] LEWIS, P.M., D.J. ROSENKRANTZ & R.E. STEARNS, *Compiler Design Theory,* Addison-Wesley, Reading, Mass., 1976.

OVER ONTLEDEN EN GRAMMATICALE EQUIVALENTIERELATIES

A. NIJHOLT
Vrije Universiteit, Amsterdam

1. INLEIDING

In dit artikel zullen we ons bezighouden met ontleden en met grammaticale equivalentierelaties. Ontleden (Engelse term: parsing) of beter geformuleerd (vergelijk BRANDT CORSTIUS [9]) herkennend ontleden is het beslissen of een symboolrij element is van de taal van de beschouwde grammatica en, indien dit het geval is, het geven van 'structuur' aan deze symboolrij. Voor onze doeleinden zal 'structuur' vrijwel altijd zijn, of worden weergegeven door, een rijtje productieregels van de grammatica die in de afleiding van de symboolrij (de zin) gebruikt zijn. Zij G een (contextvrije) grammatica. Noteer de taal van G als L(G).

Veronderstel we hebben een tweetal grammatica's G_1 en G_2. We noemen G_1 en G_2 zwak equivalent als ze dezelfde taal genereren, dwz. als $L(G_1) = L(G_2)$. Als we nu een ontledingsboom voor een zin w in de grammatica G_1 en een ontledingsboom voor dezelfde zin w in de grammatica G_2 zouden gaan bekijken dan zal niet noodzakelijkerwijs van enige gelijkheid sprake zijn. Evenmin zal dit het geval hoeven te zijn voor representaties van de bomen dmv. rijtjes van de gebruikte productieregels. Er zijn andere vormen van equivalentie. Hierbij wordt niet altijd zwakke equivalentie vooronderstelt. Wij zullen dat wel doen.

Bij deze andere vormen van equivalentie wordt tevens aandacht geschonken aan de bomen en aan de voortbrengingen van de zinnen.

Equivalentierelaties tussen grammatica's kunnen bijvoorbeeld zinvol gebruikt worden als we een grammatica transformeren naar een andere grammatica die 'prettiger' eigenschappen heeft wat betreft de wijze waarop de bijbehorende taal ontleed kan worden.

Soms zal dit gebeuren door een transformatie naar een zogenaamde 'normaalvorm'. In dat geval voldoen de productieregels (of de voortbrengingen

van de zinnen) aan een aantal voorwaarden. We zullen hier later op terugkomen.

HOTZ [26] merkt hierover op:

Resultate über die strukturelle Verwandschaft verschiedener Sprachen existieren kaum. Selbst bei der Herleitung von Normalformentheoremen für Grammatiken hat man sich mit der Feststellung der schwachen Äquivalenz begnügt.

In deze notities over ontleden en grammaticale equivalentierelaties zullen we ons laten leiden door mogelijk practische toepassingen van de theorie van grammaticale equivalentierelaties. Daartoe zullen we o.a. een aantal bestaande vertaler-genererende systemen [1] bekijken.

In hun overzichtsartikel over 'Translator Writing Systems' beschrijven FELDMAN en GRIES [14] o.a. vertaler-genererende systemen waarbij de syntactische analyse gebaseerd is op precedentie-technieken. Niet elke contextvrije grammatica kan omgezet worden tot een ontleder die op basis van dergelijke technieken zinnen kan ontleden. Grammatica's waarvoor dit wel mogelijk is worden in dat artikel 'precedence grammars' genoemd. Feldman en Gries merken op:

"*Moreover, one must manipulate a grammar for an average programming language considerably before it is a precedence grammar..... The final grammar could not be presented to a programmer as a reference to the language*".

Ook in andere situaties waarbij de gekozen ontledingsmethode gebaseerd is op andere dan precedentie-technieken zullen veelal transformaties nodig zijn om de grammatica geschikt te maken voor de ontledingsmethode. Het niet meer 'natuurlijk' zijn van de nieuw verkregen grammatica is niet belangrijk als de benodigde transformaties om de invoergrammatica geschikt te maken voor het systeem door het vertaler-genererende systeem zelf verzorgd worden.

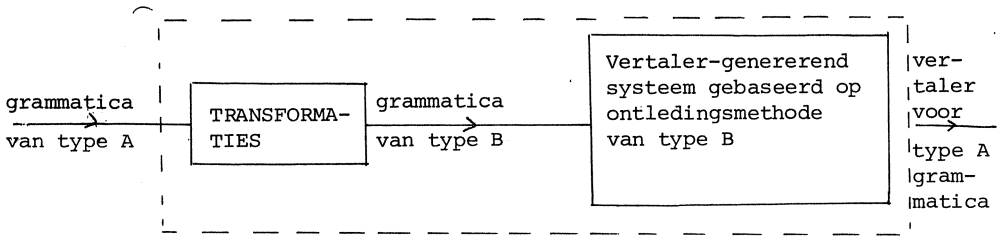
Daarbij moet dan wel opgemerkt worden dat door de transformaties de structuur van de oorspronkelijke grammatica verloren zal gaan. Dit kan betekenen dat de nieuwe grammatica niet meer dezelfde vertaling naar semantische acties (die leiden tot code-generatie) zal geven als de oorspronkelijke grammatica. Informeel: de grammatica's zijn niet noodzakelijkerwijs 'semantisch equivalent'.

---

1) Wij gebruiken de naam vertaler-genererend systeem. In het Engels worden o.a. de namen compiler writing system, translator writing system, compiler-compiler en parser generating system gebruikt.

Ook dit is geen probleem als het vertaler-genererende systeem in staat is om uit de 'structuur' van de nieuwe grammatica de 'structuur' van de oorspronkelijke grammatica te herleiden. In dat geval kan de gebruiker van het systeem onkundig blijven van de transformaties op zijn, als invoer gegeven, syntactische en semantische regels. We zullen aandacht besteden aan dit soort transformaties.

Als elke grammatica G van type A getransformeerd kan worden tot een grammatica G' van type B en we kunnen immer voor elke zin w de structuur van w met betrekking tot G' omzetten tot de structuur van w met betrekking tot G (in dat geval zeggen we G' *overdekt* G), dan kan een vertaler-genererend systeem als weergegeven in Figuur 1 gebruikt worden.



Figuur 1. Vertaler-genererend systeem voor type A grammatica's.

We sluiten deze Inleiding af met een aantal notities over de verdere inhoud van dit artikel. We zullen ons niet bezighouden met de meer geavanceerde vormen van het toekennen van semantiek (attributen) aan de nonterminale symbolen en aan de productieregels van een grammatica. In BOCHMANN [7] wordt bericht over transformaties op attribuut-grammatica's.

Andere grammaticale equivalentierelaties dan de *overdekking* zullen in dit artikel niet aan de orde komen. Vrij bekende relaties zijn die van structurele equivalentie (PAULL and UNGER [53,54]) en die van grammar functor (zie bijvoorbeeld HOTZ [26] en de daar gegeven literatuurverwijzingen, BENSON [6] en NIJHOLT [49]). In WOOD [69] is een uitgebreide bibliografie van 'grammatical similarity relations' te vinden.

Voor zover wij kunnen nagaan zijn er geen practische toepassingen in de vertaler-bouw van dit soort relaties. Evenals het begrip *overdekking* dat we hierna formeel zullen introduceren kunnen deze relaties wel zinvol zijn bij theoretische beschouwingen over het ontleden.

De bedoelingen van dit artikel zijn tweeërlei. In de eerste plaats willen we de lezer kennis laten nemen van een aantal resultaten op het gebied van '*overdekkingen*' en van de wijze waarop deze resultaten eventueel gebruikt kunnen worden of al gebruikt worden. Deze resultaten zijn veelal verkregen binnen een theoretisch kader. Opmerkelijk is echter dat in de oudere en niet zozeer theoretisch gerichte literatuur dit idee van overdekking al veelvuldig aan de orde kwam en gebruikt werd. Het begrip overdekking werd formeel ingevoerd (zie de definities van *cover* in GRAY en HARRISON [20] en AHO en ULLMAN [2]) zonder dat dit werd opgemerkt. Integendeel, een aantal problemen werd geïntroduceerd (en soms ook nog onjuist opgelost) die al eerder, in een niet-theoretisch raamwerk, waren opgelost door bouwers van vertalers en vertaler-genererende systemen. In de tweede plaats zullen we van de gelegenheid gebruik maken om een aantal recente artikelen op het gebied van het ontleden de revue te laten passeren.

In de volgende sectie zullen we een aantal definities geven. Sectie 3 gaat over ontleden. In sectie 4 zullen we een aantal vertaler-genererende systemen bekijken en daarbij vooral letten op de kenmerken die in het kader van dit artikel interessant zijn. In sectie 5 worden een aantal voorbeelden gegeven van transformaties naar gemakkelijk neerwaarts (top-down) of opwaarts (bottom-up) te ontleden grammatica's. Het artikel wordt afgesloten met een aantal overdekkingsresultaten wat betreft grammatica's in een normaalvorm.

2. VERDUIDELIJKING

We zullen ons zoveel mogelijk houden aan de definities en notaties van AHO en ULLMAN [2].

Een *alfabet* is een eindige verzameling van symbolen. Zij V een alfabet. V^+ staat voor de verzameling van rijtjes symbolen (woorden) uit V. Het *lege woord* wordt aangeduid met ε. $V^* = V^+ \cup \{\varepsilon\}$.

Een *contextvrije grammatica* (of kortweg grammatica) is een viertal $G = (N,\Sigma,P,S)$ waarbij:

- N is het alfabet van *nonterminale* symbolen
- Σ is het alfabet van *terminale* symbolen, $V = N \cup \Sigma$
- $P \subseteq N \times V^*$ is de eindige verzameling van *productieregels*
- S is het *startsymbool*

We schrijven $A \rightarrow \alpha$ in P in plaats van (A,α) in P.

Nonterminale symbolen zullen in het algemeen aangeduid worden met hoofdletters uit het begin van het alfabet; terminale symbolen met kleine letters uit het begin van het alfabet. In sommige voorbeelden zullen ook de symbolen + en × en, en id als terminaal symbool fungeren. De letters X,Y en Z staan voor symbolen in V waarvan (nog) niet bekend is of ze tot N of tot Σ behoren. Kleine letters uit het Griekse alfabet zullen gebruikt worden om rijtjes symbolen uit V^* te representeren.

Op de gebruikelijke wijze definiëren we *afleidingen* (voortbrengingen). Veronderstel we hebben rijtjes α_1 en α_2 en een productieregel $A \to \beta$, dan kunnen we schrijven $\alpha_1 A \alpha_2 \Rightarrow \alpha_1 \beta \alpha_2$ (een afleiding van één stap). De relatie $\Rightarrow \subseteq V^* \times V^*$ kan gegeneraliseerd worden tot $\overset{+}{\Rightarrow}$ (een afleiding van één of meer stappen) en $\overset{*}{\Rightarrow}$ (een afleiding van nul of meer stappen).
Als $\alpha_1 \in \Sigma^*$, dan schrijven we

$$\alpha_1 A \alpha_2 \underset{L}{\Rightarrow} \alpha_1 \beta \alpha_2 \qquad \textit{(linkerafleiding)}$$

en als $\alpha_2 \in \Sigma^*$, dan schrijven we

$$\alpha_1 A \alpha_2 \underset{R}{\Rightarrow} \alpha_1 \beta \alpha_2 \qquad \textit{(rechterafleiding)}.$$

Evenals voor $\Rightarrow$ kunnen we $\overset{+}{\underset{L}{\Rightarrow}}$, $\overset{*}{\underset{L}{\Rightarrow}}$, $\overset{+}{\underset{R}{\Rightarrow}}$ en $\overset{*}{\underset{R}{\Rightarrow}}$ invoeren.

We veronderstellen dat de productieregels van een grammatica genummerd zijn met nummers uit het alfabet

$$\Delta_G = \{i \mid 1 \leq i \leq |P|\}$$

waarbij $|P|$ het aantal elementen van P aanduidt. Als $i.A \to \alpha$ in P (dwz. $A \to \alpha$ is de i-de regel) dan schrijven we soms $\alpha_1 A \alpha_2 \overset{i}{\Rightarrow} \alpha_1 \beta \alpha_2$ (of evt. $\alpha_1 A \alpha_2 \overset{i}{\underset{L}{\Rightarrow}} \alpha_1 \beta \alpha_2$ en $\alpha_1 A \alpha_2 \overset{i}{\underset{R}{\Rightarrow}} \alpha_1 \beta \alpha_2$). Dit kan weer gegeneraliseerd worden, bijvoorbeeld $\gamma \overset{\pi}{\underset{L}{\Rightarrow}} \delta$ als π het rijtje is van productieregels die gebruikt zijn in de linkerafleiding van δ uit γ.

De *taal* gegenereerd door een grammatica G is de verzameling

$$L(G) = \{w \mid S \overset{*}{\Rightarrow} w\}.$$

Zij $w \in L(G)$. Als $S \overset{\pi}{\underset{L}{\Rightarrow}} w$, dan wordt π een *linkerontleding* ('left parse') van w genoemd. Evenzo, als $S \overset{\pi}{\underset{R}{\Rightarrow}} w$, dan wordt π^R (π van rechts naar links gelezen) een *rechterontleding* ('right parse') van w genoemd.

Voorbeeld 1. (FOSTER [16])
Beschouw de grammatica G met productieregels

1. $S \rightarrow T$
2. $S \rightarrow S + T$
3. $T \rightarrow \underline{id}$
4. $T \rightarrow T \times \underline{id}$

De symboolrij $\underline{id} + \underline{id} \times \underline{id}$ is element van de taal van de grammatica. Een rechterontleding voor deze zin is 31342. Deze rechterontleding kan opgevat worden als een rijtje semantische acties (bijvoorbeeld aanroepen voor procedures die code-generatie verzorgen).

Ten behoeve van later gebruik is het nodig enige begrippen in te voeren. Wij zeggen dat een grammatica

- *niet linksrecursief* (NLR) is, als voor elk nonterminaal symbool A een afleiding $A \overset{+}{\Rightarrow} A\alpha$, voor zekere $\alpha \in V^*$, niet mogelijk is.
- in *Greibach normaalvorm* (GNV) is, als elke productieregel van de vorm $A \rightarrow a\alpha$ is ($a \in \Sigma$ en $\alpha \in N^*$).
- ε-vrij is, als $P \subseteq N \times V^+$
- *links gefactoriseerd* is, als er geen twee productieregels van de vorm $A \rightarrow \alpha\beta$ en $A \rightarrow \alpha\gamma$ met $\alpha \neq \varepsilon$ en $\beta \neq \gamma$ in P zijn.

Zij $\alpha \in V^*$. Het aantal symbolen van α noteren we met $|\alpha|$. Het rijtje bestaande uit de eerste k symbolen van α noteren we met $k{:}\alpha$. Als $|\alpha| < k$, dan $k{:}\alpha = \alpha$. Definiëer

$$\text{EERST}_k(\alpha) = \{k{:}w \mid \alpha \overset{*}{\Rightarrow} w\}.$$

We zijn nu voldoende voorbereid om het begrip *overdekking* formeel te introduceren.

Veronderstel we hebben twee alfabets V_1 en V_2. Een homomorfisme $\psi: V_1^* \rightarrow V_2^*$ is gedefinieerd als volgt: $\psi(\varepsilon) = \varepsilon$, voor elke $X \in V_1$ is $\psi(X) \in V_2^*$ en $\psi(\alpha\beta) = \psi(\alpha)\psi(\beta)$. We zeggen dat grammatica $G' = (N',\Sigma,P',S')$ de grammatica $G = (N,\Sigma,P,S)$ *links tot rechts overdekt* als er een homomorfisme $\psi: \Delta_{G'}^* \rightarrow \Delta_G^*$ bestaat zodanig dat aan de volgende twee voorwaarden voldaan is:

(i) als $S' \overset{\pi'}{\underset{L}{\Rightarrow}} w$ dan $S \overset{\pi}{\underset{R}{\Rightarrow}} w$, waarbij $\pi^R = \psi(\pi')$, en

(ii) als $S \overset{\pi}{\underset{R}{\Rightarrow}} w$, dan bestaat er een rijtje π' zodanig dat $S' \overset{\pi'}{\underset{L}{\Rightarrow}} w$ met $\psi(\pi') = \pi^R$.

Het zal duidelijk zijn dat we zonder problemen ook andere vormen van overdekking kunnen invoeren. We noteren bovenstaande links tot rechts overdekking als G'[L/R]G. Als we zowel voor G' als G linkerontledingen beschouwen dan zeggen we G' *links overdekt* G, notatie G'[L/L]G. Evenzo G'[R/L]G (G' *rechts tot links overdekt* G) en G'[R/R]G (G' *rechts overdekt* G).
Als we in het algemeen over overdekkingen praten dan bedoelen we een van bovengenoemde vier vormen.

Voorbeeld 1 (vervolgd).
We kunnen de gegeven grammatica G transformeren tot een niet-linksrecursieve grammatica G' zodanig dat G'[L/R]G en G'[R/R]G. Grammatica G' heeft productieregels [2]

$S \to TH_1 \mid TH_1X$	$H_1 \to \varepsilon$ {1}
$T \to \underline{id}\ H_3 \mid \underline{id}\ H_3Y$	$H_2 \to \varepsilon$ {2}
$X \to +\ TH_2 \mid +\ TH_2X$	$H_3 \to \varepsilon$ {3}
$Y \to \times \underline{id}\ H_4 \mid \times\ \underline{id}\ H_4Y$	$H_4 \to \varepsilon$ {4}

Als we nu homomorfisme ψ toepassen op het rijtje productieregels van G' dat gebruikt is in een linker- of in een rechterontleding van de zin $\underline{id} + \underline{id} \times \underline{id}$ dan krijgen we precies het rijtje 31342 weer terug. Einde van Voorbeeld 1.

3. ONTLEDEN

> *"Ik heb het gevoel", zei de keizer met samengeknepen oogleden, "dat het je weleens je hoofd zou kunnen kosten, wanneer ik je betoog ging ontleden".*
> *"Lieve help!" riep de droomuitlegger. "Niet ontleden. Niet ontleden, wat ik u bidden mag!"*
>
> (*Harry Mulisch, De diamant*)

---

2) Conventies.
We laten elke productieregel volgen door {π}, waar $\pi \in \Delta_G^*$ het beeld is van de productieregel onder de (overdekkings) homomorfisme ψ: $\Delta_{G'}^* \to \Delta_G^*$. Dus i.A → α{π} of A → α{π} betekent dat de productieregel i wordt afgebeeld op het rijtje regels π. Als π = ε dan nemen we niet de moeite {ε} te noteren.

Er zijn tientallen ontledingsmethoden ontwikkeld voor contextvrije talen en voor een aantal deelklassen van de klasse van contextvrije talen. Voor practische toepassingen beperkt men zich meestal tot methoden die ontwikkeld zijn en werken voor de zogeheten deterministische talen. Vaak definieert de syntactische structuur van grote delen van een programmeertaal een deterministische taal. Daarom is die beperking mogelijk en aangezien de ontledingsmethoden voor deterministische talen vaak eenvoudig geïmplementeerd kunnen worden en efficiënt werken is het ook wenselijk.

We zullen in het kort een aantal ontledingstechnieken behandelen. Het is gebruikelijk onderscheid te maken tussen neerwaarts en opwaarts werkende methoden. Eerst beschouwen we de neerwaartse methode.

3.1. Neerwaarts

Beschouw een linkerafleiding

$$\omega_0 \underset{L}{\overset{P_0}{\Longrightarrow}} \omega_1 \underset{L}{\overset{P_1}{\Longrightarrow}} \cdots\cdots \omega_{n-1} \underset{L}{\overset{P_{n-1}}{\Longrightarrow}} \omega_n$$

met $\omega_0 = S$ en $\omega_n \in \Sigma^*$. Bij het neerwaarts ontleden is het doel, gegeven ω_0 en ω_n, het vinden van de rij $P_0P_1\ldots P_{n-1}$ van productieregels die gebruikt zijn in de afleiding van ω_n van S.

Te beginnen bij ω_0, de taak is, gegeven ω_{j-1} $(1 \leq j < n)$ te bepalen hoe ω_{j-1} tot ω_j moet worden herschreven opdat tenslotte ω_n bereikt wordt. Aangezien we linkerafleidingen beschouwen kunnen we schrijven $\omega_{j-1} = wA\alpha$; w is dus een prefix van ω_n. Bij het deterministisch neerwaarts ontleden kunnen we, nadat we $\omega_0,\omega_1,\ldots,\omega_{j-2}$ en $\omega_{j-1} = wA\alpha$ verwerkt hebben, productieregel P_{j-1} en daarmee ω_j bepalen door te kijken naar de symbolen van ω_n rechts van w.

De klasse van grammatica's waarvoor deze methode werkt is de klasse van LL(k)-grammatica's, waarbij k slaat op het aantal symbolen wat ten hoogste 'vooruitgekeken' mag worden.

Formeel, een grammatica is LL(k) als voor elke linkerafleiding $S \underset{L}{\overset{*}{\Rightarrow}} wA\alpha$ en productieregels $A \to \beta$ en $A \to \gamma$ geldt dat als

$$EERST_k(\beta\alpha) \cap EERST_k(\gamma\alpha) \neq \emptyset$$

dan $\beta = \gamma$.

Zowel in AHO en ULLMAN [2] als in AHO en ULLMAN [3] worden LL(k)-grammatica's en ontleders op een duidelijke manier beschreven. Vaak kan men zich beperken tot het vooruitkijken van één symbool.

Dit type grammatica's werd geïntroduceerd door LEWIS en STEARNS [39] en de theorie werd ontwikkeld in ROSENKRANTZ en STEARNS [59]. Vergelijkbaar onderzoek werd gedaan door WOOD [68] en V̌CULIK [10]. Recent onderzoek op het gebied van LL(k)-grammatica's en talen is beschreven in BEATTY [5] en in SIPPU en SOISALON-SOININEN [61]. LL-ontleders zijn eenvoudig, gemakkelijk te implementeren en efficiënt. Daarom zijn veel pogingen ondernomen om niet-LL-grammatica's te transformeren tot LL-grammatica's. In sectie 5 komen we hier op terug.

3.2. Opwaarts

Veel methoden, speciaal die gebruikt worden voor vertaler-genererende systemen, zijn gebaseerd op opwaartse ontledingstechnieken.
Beschouw nu een rechterafleiding

$$\omega_n \overset{P_n}{\underset{R}{\Longrightarrow}} \omega_{n-1} \underset{R}{\Longrightarrow} \cdots \omega_2 \overset{P_2}{\underset{R}{\Longrightarrow}} \omega_1 \overset{P_1}{\underset{R}{\Longrightarrow}} \omega_0$$

met $\omega_n = S$ en $\omega_0 \in \Sigma^*$.

Bij het opwaarts ontleden is het doel het rijtje productieregels $P_1P_2\ldots P_n$ te vinden die gebruikt zijn (in de omgekeerde volgorde) om ω_0 af te leiden van ω_n.

We schrijven $\omega_j = \alpha Aw$ en $\omega_{j-1} = \alpha\beta w$. Het probleem bestaat nu uit het vinden van β, het vinden van de plaats waar β voorkomt in ω_{j-1} en het bepalen door welk nonterminaal symbool β vervangen moet worden opdat ω_{j-1} overgaat in ω_j. De positie van β in ω_{j-1} kunnen we aangeven met $|\alpha\beta|$, d.w.z. met de lengte van de symboolrij $\alpha\beta$.

Het paar $(A \to \beta, |\alpha\beta|)$ heet het *handvat* van ω_{j-1}. Bij het opwaarts ontleden beginnen we met ω_0. Het bepalen van het handvat $(A \to \beta, |\alpha\beta|)$ en het voortdurend vervangen van β in ω_{j-1} $(1 \leq j \leq n)$ door A (het '*reduceren*') moet voortgezet worden, indien mogelijk, totdat tenslotte het startsymbool $\omega_n = S$ is bereikt.

Bij de deterministische opwaartse ontledingsmethoden kan ω_j bepaald worden nadat de symboolrijen $\omega_0, \omega_1, \ldots, \omega_{j-1}$ bepaald zijn. Het handvat $(A \to \beta, |\alpha\beta|)$ van ω_{j-1} kan worden bepaald doordat de context van β dit handvat eenduidig bepaalt. De context rechts van β (het aantal symbolen van w

dat vooruitgekeken wordt) wordt beperkt door een bovengrens k, $k \geq 0$. In het meest algemene geval krijgen we de volgende definitie.
Als we rechterafleidingen hebben

$$S \underset{R}{\overset{*}{\Longrightarrow}} \alpha Aw \underset{R}{\Longrightarrow} \alpha\beta w = \gamma w$$

en

$$S \underset{R}{\overset{*}{\Longrightarrow}} \alpha'A'x \underset{R}{\Longrightarrow} \alpha'\beta'x = \gamma w'$$

en de eerste k symbolen van w en w' zijn gelijk, dan eisen we dat de handvaten van γw en $\gamma w'$ gelijk zijn.

De grammatica's die rechterafleidingen hebben die voldoen aan deze definitie zijn de LR(k)-grammatica's.

In GELLER en HARRISON [17] is een uitgebreide studie van LR(k)-grammatica's gegeven. Ontledingsmethoden voor LR-grammatica's zijn beschreven in KNUTH [29], DEREMER [12], AHO en JOHNSON [1] en GELLER en HARRISON [18]. Twee eenvoudig te implementeren technieken zijn gebaseerd op SLR(1)-grammatica's (simple LR(1)) en op LALR(1)-grammatica's (look-ahead LR(1)).

Er gelden de volgende inclusierelaties voor deze klassen van grammatica's:

$$LR(0) \subsetneq SLR(1) \subsetneq LALR(1) \subsetneq LR(1)$$

De context van β in ω_{j-1} kan op een minder algemene manier gebruikt worden. Op die manier kunnen deelklassen van de LR(k)-grammatica's gedefinieerd worden. Voorbeelden zijn de 'bounded (right) context' grammatica's en de verschillende klassen van precedentiegrammatica's (zie AHO en ULLMAN [2]). De opwaartse ontledingsmethoden voor deze klassen van grammatica's worden ook wel genoemd '*shift-reduce*' methoden. De reden is dat bij de implementatie van een dergelijke methode m.b.v. een stapel er steeds een besluit moet worden genomen of er een volgend symbool van de zin op de stapel gezet moet worden ('*shift*') of dat, in het geval een handvat is gevonden, de bovenste symbolen van de stapel (corresponderend met het rechterlid van de gevonden productieregel) verwijderd moeten worden ('*reduce*'). Bij niet-LR-grammatica's zullen er actieconflicten aanwezig zijn, d.w.z. er zijn situaties waarbij de ontleder niet kan beslissen of er een shift-actie of een reduce-actie gedaan moet worden (*shift/reduce conflict*) of de ontleder

kan niet beslissen wat het linkerlid van een gevonden rechterlid van een productieregel is (*reduce/reduce conflict*).

3.3. Neerwaarts/Opwaarts

Een derde klasse van ontledingsmethoden wordt wel aangeduid met de verzamelnaam 'semi top-down' methoden. Vaak zijn niet zo zeer deze methoden interessant als wel de door de methoden gedefinieerde klassen van grammatica's. In sectie 5 zullen we aandacht besteden aan een aantal van deze klassen.

Bij sommige van deze typen grammatica's stapt men af van de gebruikelijke linker- en rechterontledingen als resultaat van het (zuivere) ontleedproces. Voorbeelden zijn de 'left corner' ontledingen (ROSENKRANTZ en LEWIS [58]), de 'left part' ontledingen (NIJHOLT [48]) en de gegeneraliseerde 'left corner' ontledingen (DEMERS [11]).

Een bijzondere klasse van grammatica's is die van de *strict deterministische grammatica's* (HARRISON en HAVEL [23,24]). Het meest karakteristieke kenmerk is dat de definitie niet gebaseerd is op de afleidingen van de grammatica maar op de producties, terwijl toch de klasse van (prefixvrije) deterministische talen wordt gedefinieerd. Een gevolg is dat het erg simpel is om te beslissen of een grammatica strict deterministisch is. Beschouwingen gebaseerd op deze klasse van grammatica's hebben geleid tot nuttige ideeën op het gebied van ontleden en vertalen (zie o.a. GELLER en HARRISON [18], LEWIS, ROSENKRANTZ en STEARNS [38] en NIJHOLT [47]).

3.4. Contextvrije grammatica's met reguliere expressies

Langzamerhand worden ook de grammatica's met productieregels waarvan het rechterlid uit een reguliere expressie bestaat (of evt. een eindige automaat of transitiediagram) theoretisch behandeld. Zowel neerwaartse methoden ('extended LL(k), zie bijv. HEILBRUNNER [25]) als opwaartse methoden ('extended LR(k)', zie bijv. MADSEN en KRISTENSEN [40] en LALONDE [33,34]) kunnen worden gebruikt.

4. VERTALER-GENERERENDE SYSTEMEN

Veronderstel we hebben een taal die beschreven kan worden met een contextvrije grammatica. We willen een ontleder of een vertaler voor deze

grammatica bouwen. Duidelijk is dat men bij het definiëren van de grammatica al rekening kan houden met het ontwerpen van de ontleder (vergelijk WIRTH [67]).

Het bouwen van de vertaler kan met de hand gebeuren of met behulp van een vertaler-genererend systeem. Als het met de hand gedaan wordt en een zekere ontledingsmethode is gekozen dan zal men vaak nog gedwongen zijn om de grammatica te manipuleren, transformaties toe te passen etc. om de grammatica geschikt te maken voor deze ontledingsmethode. Hetzelfde kan het geval zijn bij het gebruik van een vertaler-genererend systeem.

Een dergelijk systeem accepteert als invoer de syntactische regels (tezamen met semantische informatie) van een grammatica en produceert als uitvoer een ontleder of een vertaler voor deze grammatica. In een vertaler-genererend systeem is een keus gemaakt voor een bepaald type ontledingsmethode. Als het systeem voorzien wordt van syntactische regels dan zal geprobeerd worden een ontleder van dit type te genereren.

Als de syntactische regels een grammatica specificeren die niet op deze wijze ontleed kan worden dan kan het volgende gebeuren:

- Het systeem meldt zijn falen om een ontleder te bouwen aan de gebruiker; het verstrekt informatie waarom het misliep. Deze informatie kan door de gebruiker benut worden om de grammatica (en de wijze waarop semantiek is toegevoegd aan de productieregels) te veranderen opdat ze geschikt wordt voor dit systeem.
- Het systeem past transformaties toe op de invoergrammatica en de toegevoegde semantiek om ze geschikt te maken voor het systeem. Duidelijk is dat dit zo dient te gebeuren dat de door de gebruiker gewenste semantiek bewaard blijft.
- Hoewel de syntactische regels niet een grammatica specificeren die geschikt is om door de aan het systeem ten grondslag liggende ontledingsmethode verwerkt te worden, genereert het systeem toch een ontleder. Dit kan gebeuren doordat het systeem gebruikt maakt van door de gebruiker verstrekte extra informatie en/of doordat het systeem, volgens welomschreven regels, zelfstandig besluiten neemt omtrent de bouw van de ontleder. In dat geval vervalt het voordeel van het automatisch correct zijn van de gebouwde ontleder.

We willen een aantal voorbeelden van vertaler-genererende systemen bekijken. Het is geenszins de bedoeling een overzicht te geven van recent

gebouwde vertaler-genererende systemen. In FELDMAN en GRIES [14] is een uitbegreid overzicht te vinden van vertaler-genererende systemen van voor 1968. In RÄIHÄ en SAARINEN [57] en in RÄIHÄ [55] zijn meer recente overzichten te vinden.

4.1. Neerwaarts

FOSTER [15,16] beschrijft een vertaler-genererend systeem dat gebaseerd is op een neerwaartse ontledingsmethode. In dit systeem, passende in de Britse traditie van neerwaarts ontleden (vergelijk een vertaler-genererend systeem van BROOKER en MORRIS uit 1962 (beschreven in GRIES [21]) en later werk van WOOD [68]), wordt de invoergrammatica eerst door een speciaal programma SID (Syntax Improving Device) bewerkt opdat de grammatica geschikt wordt voor een neerwaartse ontledingsmethode.

Een van de transformaties die gebruikt wordt is de eliminatie van linker-recursie (zie Voorbeeld 1). Tevens wordt geprobeerd een soort linker-factorisatie toe te passen, dwz. er wordt geprobeerd een links-gefactoriseerde grammatica te construeren. Evenals de eliminatie van linker-recursie dient dit zodanig te gebeuren dat de oorspronkelijke semantiek gehandhaafd blijft.

Vergelijkbare transformaties als gebruikt in SID en met hetzelfde doel ingevoerd zijn de transformaties beschreven in STEARNS [64] (zie ook Appendix A van LEWIS II, ROSENKRANTZ en STEARNS [37]). Andere voorbeelden van vertaler-genererende systemen die gebaseerd zijn op een neerwaartse ontledingsmethode zijn te vinden in BOCHMAN en WARD [8], een systeem dat gebaseerd is op de LL(1)-ontledingsmethode, en in MILTON, KIRCHHOFF en ROWLAND [45], een systeem dat eveneens gebaseerd is op de LL(1)-ontledingsmethode. Dit laatste systeem, Aparse [3] geheten, is een systeem dat de door de gebruiker gegeven semantiek (in dit geval de 'inherited attributes') aanwendt om de ontleder te bouwen. Het gevolg is dan ook dat een grotere klasse dan de LL(1)-grammatica's met dit systeem verwerkt kunnen worden.

---

3) Dit systeem, operationeel sinds Juni 1978, is nog niet door Bell Laboratories (waar het gemaakt is) op de markt gebracht.

4.2. Opwaarts

Wat betreft de vertaler-genererende systemen die gebaseerd zijn op opwaartse ontledingstechnieken willen we de volgende noemen.

LECARME en BOCHMANN [36] beschrijven een vertaler-genererend systeem dat gebaseerd is op precedentietechnieken. Transformaties worden genoemd om de invoergrammatica geschikt te maken voor dit soort technieken.

In MICKUNAS en SCHNEIDER [42] wordt een vertaler-genererend systeem, ontwikkeld op de Purdue University, beschreven. De eerste fase van dit systeem bestaat uit het omzetten van de invoergrammatica tot een grammatica in een eenvoudige normaalvorm. Zoals verwacht mag worden blijft bij deze transformatie de oorspronkelijke semantiek behouden. De auteurs kondigen aan dat hun systeem uitgebreid zal worden op een zodanige wijze dat, wanneer als invoer een LR(k)-grammatica gegeven wordt, deze eerst getransformeerd wordt tot een grammatica waarbij ten hoogste een symbool 'vooruitgekeken' wordt. Bijvoorbeeld, SLR(1) en LALR(1) zijn methoden die gebruikt kunnen worden. Deze transformaties zijn beschreven in MICKUNAS, LANCASTER en SCHNEIDER [44] en in MICKUNAS [43]. De transformaties leveren rechteroverdekkingen op.

Het bouwen van vertaler-genererende systemen gebaseerd op de LALR(1)-ontledingsmethode is een populaire bezigheid geworden. Het eerste op deze methode gebaseerde systeem werd gebouwd in Toronto (LALONDE, LEE en HORNING [35]). Dit systeem werd gevolgd door het Yacc systeem (Bell Laboratories, JOHNSON [28]). Dit systeem is geïmplementeerd op het Unix Time Sharing Systeem. Als de invoergrammatica niet LALR(1) is dan kan extra informatie, verstrekt door de gebruiker, er voor zorgen dat actieconflicten opgelost worden. Anders neemt het systeem zelf beslissingen voor het oplossen van deze conflicten. In KRON, HOFFMANN en WINKLER [30] wordt een soortgelijke filosofie gehanteerd.

Het vertaler-genererend systeem HLP (Helsinki Language Processor), beschreven in RÄIHÄ et al. [56], vereist als invoer een LALR(1)-grammatica. Diagnostische informatie wordt geproduceerd als de grammatica niet LALR(1) blijkt te zijn.

Tengevolge van de populariteit van de LALR(1)-methode wordt ook veel tijd besteed aan onderzoek op het gebied van 'error-recovery' technieken die gebruikt kunnen worden in vertaler-genererende systemen die gebaseerd zijn op LR- en LALR-technieken (zie bijv. SIPPU en SOISALON-SOININEN [60] en GRAHAM, HALEY en JOY [19]) en naar methoden om dit soort ontleders zo

efficiënt mogelijk te implementeren (zie bijv. THOMPSON [65] en DEREMER en PENNELLO [13]).

5. TRANSFORMATIES NAAR NEERWAARTS TE ONTLEDEN GRAMMATICA's

We onderscheiden een tweetal benaderingen bij het transformeren van grammatica's naar beter neerwaarts te ontleden grammatica's.

Bij de eerste benadering gaan we uit van de transformaties. Bijvoorbeeld, de eerste fase van een vertaler-genererend systeem dat gebaseerd is op een LL-ontledingsmethode kan bestaan uit een aantal transformaties die toegepast worden op de invoergrammatica om haar, indien mogelijk, LL(k) te maken.

De tweede benadering bestaat uit het karakteriseren van klassen van grammatica's die succesvol getransformeerd kunnen worden (bijvoorbeeld naar LL(k)-grammatica's).

Bij beide benaderingen wensen we dat de nieuw verkregen grammatica de oorspronkelijke grammatica overdekt.

5.1. Uitgaan van transformaties

Zowel het elimineren van linkerrecursie als het links factoriseren van een grammatica behoren tot de meest bekende transformaties van context-vrije grammatica's. LL(k)-grammatica's zijn niet linksrecursief. Daarom zal, indien aanwezig, de linkerrecursie in een LL(k) te maken grammatica geëlimineerd moeten worden.

In eerste instantie werd gedacht dat eliminatie van linkerrecursie niet altijd tot een rechtsoverdekkende grammatica zou kunnen leiden (AHO en ULLMAN [2], GRAY en HARRISON [20]). In NIJHOLT [46] werd aangetoond dat dit wel mogelijk was en in SOISALON-SOININEN [62] werd aangetoond dat de manier waarop KURKI-SUONIO [32] vanuit een meer practisch oogpunt tegen dit probleem aankeek in feite ook leidt tot een niet-linksrecursieve rechtsoverdekkende grammatica.

Ook FOSTER [15] had bij zijn werk aan een vertaler-genererend systeem dit probleem eigenlijk al opgelost. Bij al deze transformaties wordt de oorspronkelijke grammatica G omgezet in een grammatica G' zodanig dat G'[R/R]G.

Ook het links-factoriseren van een grammatica G levert een grammatica G' op zodanig dat G'[R/R]G.

Dit is eenvoudig in te zien. Veronderstel we hebben een tweetal

productieregels in G

$$i.A \rightarrow \alpha\phi$$

en

$$j.A \rightarrow \alpha\psi$$

waarbij $\alpha \neq \varepsilon$ en α is de langste gemeenschappelijke prefix van $\alpha\phi$ en $\alpha\psi$. We kunnen deze productieregels vervangen door

$$A \rightarrow \alpha H_{ij}$$
$$H_{ij} \rightarrow \phi \quad \{i\}$$
$$H_{ij} \rightarrow \psi \quad \{j\}$$

waarbij H_{ij} een nieuw ingevoerd nonterminaal symbool is. Dit proces kan voortgezet worden totdat we een links-gefactoriseerde grammatica G' hebben verkregen. Er geldt dan G'[R/R]G.

Zowel in STEARNS [64] als in LEWIS, ROSENKRANTZ en STEARNS [37] worden o.a. deze methoden gebruikt om een niet-LL(1)-grammatica LL(1) te maken. STEARNS [64] merkt op:

"*Although these transformations are not guaranteed to make grammars* LL(1) *they seem to work out when applied to real programming languages*."

Veronderstel nu dat we inderdaad via dergelijke methoden een LL(k)-grammatica G' gevonden hebben. Is het resultaat G'[R/R]G voldoende opdat G' de oorspronkelijke semantiek kan definiëren? Dat kan inderdaad.

Hoewel het ontleedproces neerwaarts is kunnen bij een LL-ontleder voor G' de productieregels in de volgorde van een rechterontleding als uitvoer worden gegeven. Dit is een direct gevolg van een resultaat van LEWIS en STEARNS [39] dat zegt dat dit soort simpele vertalingen (van een zin naar een rechterontleding) met een LL(k)-grammatica gedefinieerd kan worden. Daarom, als we voor de oorspronkelijke grammatica de semantische acties in de volgorde van een rechterontleding hebben gedefinieerd (dwz. een semantische actie na een reductie) dan kan door dit overdekkingsresultaat grammatica G' gebruikt worden i.p.v. grammatica G.

Ook voor andere dan de overdekkingsrelatie is deze benadering gevolgd. Bijvoorbeeld, in PAULL en UNGER [54] wordt geprobeerd een willekeurige grammatica om te zetten in een structureel equivalente LL-grammatica. In

HUNT en ROSENKRANTZ [27] wordt een algoritme gegeven dat een willekeurige grammatica omzet, indien mogelijk, in een LL(k)-grammatica op een zodanige wijze dat er sprake is van een zogeheten 'Reynolds-overdekking'.

5.2. Uitgaan van definities

Er zijn klassen grammatica's waarvan we zeker weten dat elk van die grammatica's getransformeerd kan worden tot een LL(k)-grammatica. Een eenvoudig voorbeeld is een klasse van grammatica's besproken in NIJHOLT en SOISALON-SOININEN [51].

Veronderstel we hebben een grammatica waarvan de linkerafleidingen en de productieregels aan de volgende conditie voldoen:

Zij $A \to \alpha\delta_1$ en $A \to \alpha\delta_2$ twee verschillende productieregels, waarbij $\alpha \neq \varepsilon$ en α is de langste gemeenschappelijke prefix van $\alpha\delta_1$ en $\alpha\delta_2$ en zij

$$S \overset{*}{\underset{L}{\Rightarrow}} wA\gamma$$

een linkerafleiding, dan geldt

$$EERST_k(\delta_1\gamma) \cap EERST_k(\delta_2\gamma) = \emptyset.$$

Wanneer we een dergelijke grammatica G links-factoriseren op de eerder aangegeven wijze, dan wordt een LL(k)-grammatica G' verkregen. Ook kan aangetoond worden dat enkel dit soort grammatica's LL(k) gemaakt kan worden door links-factoriseren.

In het algemeen zullen de eigenschappen G'[L/R]G, G'[L/L]G en G'[R/L]G niet gelden. Wel zal gelden G'[R/R]G. Zoals in sectie 5.1 uitgelegd kan G' nu gebruikt worden i.p.v. grammatica G.

Een andere klasse van grammatica's waarvan bekend is dat elk van haar grammatica's getransformeerd kan worden tot een LL(k)-grammatica is de klasse van LC(k)-grammatica's (ROSENKRANTZ en LEWIS [58]). Deze klasse van grammatica's is gegeneraliseerd in SOISALON-SOININEN [63] tot de klasse van PLR(k)-grammatica's. Aangetoond kan worden dat de klasse van PLR(k)-grammatica's precies die grammatica's omvat die via links-factoriseren om te zetten zijn tot LC(k)-grammatica's.

Het is mogelijk elke PLR(k)-grammatica G rechtstreeks om te zetten tot een LL(k)-grammatica G' zodanig dat G'[R/R]G en G'[L/R]G. M.a.w., ook hier

weer kunnen we de ontleder van G baseren op een LL(k)-methode.
In HAMMER [22] kunnen soortgelijke resultaten gevonden worden.

We hebben ons in deze secties beperkt tot voorbeelden van transformaties naar neerwaarts te ontleden grammatica's. Evenzo zijn er vele transformaties die ten doel hebben eenvoudig werkende opwaartse ontledingsmethoden te kunnen toepassen. O.a. in GRAY en HARRISON [20], McAFEE en PRESSER [41] en LEWIS, ROSENKRANTZ en STEARNS [37] zijn hiervan voorbeelden te vinden. In het algemeen leveren dit soort transformaties rechteroverdekkingen op.

6. TRANSFORMATIES NAAR NORMAALVORMEN

Zowel voor practische doeleinden als voor theoretische beschouwingen over het ontleden is het vaak gemakkelijk om uit te gaan van een grammatica in een zekere normaalvorm.

Dat kan bijvoorbeeld inhouden dat de grammatica geen productieregels van de vorm $A \to \varepsilon$ (ε-regels) of van de vorm $A \to B$ heeft. Ook kan het soms wenselijk zijn uit te gaan van een grammatica die niet linksrecursief, of niet rechtsrecursief, of links-gefactoriseerd of in Greibach normaalvorm is. Ook de Chomsky normaalvorm (waarbij elke productieregel van de vorm $A \to BC$, $A \to a$ of $S \to \varepsilon$ is) is een regelmatig gebruikte normaalvorm.

Het is daarom nuttig voor dit soort normaalvormen overdekkingsresultaten te hebben, niet alleen voor willekeurige contextvrije grammatica's, maar ook voor bijvoorbeeld LL(k), LR(k) of strict deterministische grammatica's.

We zullen ons beperken tot het geven van een tweetal tabellen waarin enige resultaten zijn verzameld.

De eerste tabel geldt voor willekeurige contextvrije grammatica's. We geven aan of het mogelijk is een overdekkende grammatica G' te vinden van het type ε-VRIJ (geen regels van de vorm $A \to \varepsilon$), het type ε-VRIJ NLR (ε-vrij en niet linksrecursief) en van het type GNV (in Greibach normaalvorm). De te overdekken grammatica G mag zijn van het type

WILL	willekeurig
ε-VRIJ	ε-vrij
NLR	niet linksrecursief
ε-VRIJ NLR	ε-vrij en niet linksrecursief
GNV	Greibach normaalvorm.
NRR	niet rechtsrecursief
ε-VRIJ NRR	ε-vrij en niet rechtsrecursief

G / G'		WILL	ε-VRIJ	NLR	ε-VRIJ NLR	GNV	NRR	ε-VRIJ NRR
ε-vrij	L/L	nee	ja	nee	ja	ja	ja	ja
	L/R	nee	nee	nee	nee	nee	ja	ja
	R/L	nee	nee	ja	ja	ja	nee	nee
	R/R	nee	ja	ja	ja	ja	nee	ja
ε-vrij NLR	L/L	nee	nee	nee	ja	ja	nee	nee
	L/R	nee	nee	nee	nee	nee	nee	ja
	R/L	nee	nee	ja	ja	ja	nee	nee
	R/R	nee	ja	ja	ja	ja	nee	ja
GNV	L/L	nee	nee	nee	ja	ja	nee	nee
	L/R	nee	nee	nee	nee	nee	nee	ja
	R/L	nee	nee	ja	ja	ja	nee	nee
	R/R	nee	ja	ja	ja	ja	nee	ja

Tabel I. Overdekkingsresultaten.

Een voorbeeld van een grammatica die niet links-overdekt kan worden met een ε-vrije grammatica is gedefinieerd met de volgende productieregels

$$S \rightarrow aSL \mid aRL$$
$$R \rightarrow bRL \mid b$$
$$L \rightarrow \varepsilon$$

Dit voorbeeld werd gegeven door UKKONEN [66].

We modificeren deze grammatica enigszins opdat we de volgende LL(1)-grammatica krijgen.

$S \rightarrow aSL \mid bRL$
$R \rightarrow bRL \mid c$
$L \rightarrow \varepsilon$

Ook voor deze grammatica zal dan gelden dat zij niet links-overdekt kan worden door een ε-vrije grammatica (en dus ook niet door een ε-vrije LL-grammatica).

In de tweede tabel zijn een aantal resultaten voor LL(k)-grammatica's verzameld.

G / G'		WILL LL	ε-VRIJ LL	GNV LL
WILL LL	L/L	ja	ja	ja
	L/R	ja	ja	ja
	R/L	ja	ja	ja
	R/R	ja	ja	ja
ε-VRIJ LL	L/L	nee	ja	ja
	L/R	nee	nee	nee
	R/L	?	ja	ja
	R/R	?	ja	ja
GNV LL	L/L	nee	ja	ja
	L/R	nee	nee	nee
	R/L	?	ja	ja
	R/R	?	ja	ja

Tabel II. Overdekkingsresultaten voor LL-grammatica's.

7. CONCLUSIES

De twee doeleinden van dit artikel werden in de inleiding geschetst.

De meeste van de hier genoemde resultaten zijn beschreven in NIJHOLT [50]. We hebben ons hier beperkt tot situaties waarbij we linker- en rechterontledingen onderscheidden. Bij het opwaarts ontleden zal men veelal ook semantische acties willen hebben bij de shift-acties van de ontleder. Dit kan eventueel gesimuleerd worden met reducties, maar het gevolg daarvan kan zijn dat nieuwe actieconflicten in de ontleder worden geïntroduceerd. Een eerste aanzet voor de beschrijving van dit soort situaties (met één semantische actie per productieregel) is te vinden in [52].

LITERATUUR

[1] AHO, A.V. & S.C. JOHNSON, *LR parsing,* Computing Surveys 6 (1974), 99-124.

[2] AHO, A.V. & J.D. ULLMAN, *The Theory of Parsing, Translation and Compiling,* Vol. 1 & 2, Prentice Hall, Englewood Cliffs, N.J. 1972 en 1973.

[3] AHO, A.V. & J.D. ULLMAN, *Principles of Compiler Design,* Addison-Wesley, Reading, Mass., 1977.

[4] ALBLAS, H. et al., *Vertalerbouw,* Academic Service, Den Haag, 1977.

[5] BEATTY, J.C., *Iteration theorems for the* LL(k) *languages,* Ph.D. Thesis 1977, Lawrence Livermore Laboratory, University of California, Livermore.

[6] BENSON, D.B., *Some preservation properties of normal form grammars,* SIAM J. Comput. 6 (1977), 381-402.

[7] BOCHMANN, G.V., *Semantic equivalence of covering attribute grammars,* Publ. # 218, December 1975, Université de Montreal.

[8] BOCHMANN, G.V. & P. WARD, *Compiler writing system for attribute grammars,* Publ. #199, 1975, Université de Montreal.

[9] BRANDT CORSTIUS, H., *Computer-taalkunde,* Coutinho, Muiderberg, 1978.

[10] CULIK II, K., *Contribution to deterministic top-down analysis of context-free languages,* Kybernetika 5 (1968), 422-431.

[11] DEMERS, A.J., *Generalized left corner parsing,* 4th ACM Symp. on Principles of Programming Languages 1977, 170-182.

[12] DEREMER, F.L., *Simple* LR(k) *grammars,* Comm. ACM 14 (1971), 453-460.

[13] DEREMER, F.L. & T.J. PENNELLO, *Efficient computation of* LALR(1) *look-ahead sets,* SIGPLAN Notices 14, Nr. 8, August 1979, 176-187.

[14] FELDMAN, J. & D. GRIES, *Translator writing systems,* Comm. ACM 11 (1968), 77-113.

[15] FOSTER, J.M., *A syntax improving program,* Computer Journal 11 (1968), 31-34.

[16] FOSTER, J.M., *Automatic Syntactic Analysis,* MacDonald, London, 1970.

[17] GELLER, M.M. & M.A. HARRISON, *On* LR(k) *grammars and languages,* Theoret. Comput. Sci. 4 (1977), 245-276.

[18] GELLER, M.M. & M.A. HARRISON, *Characteristic parsing: A framework for producing compact deterministic parsers,* I & II, J. Comput. System Sci. 14 (1977), 265-317 en 318-345.

[19] GRAHAM, S.L., C.B. HALEY & W.N. JOY, *Practical* LR *error recovery,* SIGPLAN Notices 14, Nr. 8, August 1979, 168-175.

[20] GRAY, J. & M.A. HARRISON, *On the covering and reduction problems for context-free grammars,* J. Assoc. Comput. Mach. 19 (1972), 385-395.

[21] GRIES, D., *Compiler Construction for Digital Computers,* John Wiley and Sons, New York, N.Y., 1971.

[22] HAMMER, M., *A new grammatical transformation into* LL(k) *form,* 6th ACM Symp. on Theory of Computing 1974, 266-275.

[23] HARRISON, M.A. & I.M. HAVEL, *Strict deterministic grammars,* J. Comput. System Sci. 7 (1973), 237-277.

[24] HARRISON, M.A. & I.M. HAVEL, *On the parsing of deterministic languages,* J. Assoc. Comput. Mach. 21 (1974), 525-548.

[25] HEILBRUNNER, S., *On the definition of* ELR(k) *and* ELL(k) *grammars,* Acta Informatica 11 (1979), 169-176.

[26] HOTZ, G., *Strukturelle Verwandschaften von Semi-Thue Systemen,* in: *Category Theory Applied to Computation and Control,* E.G. Manes (ed.), LNCS 25, Springer, Berlin, 1975, 174-179.

[27] HUNT, H.B. & D.J. ROSENKRANTZ, *Complexity of grammatical similarity relations*, Proc. of the Conf. on Theoretical Computer Science, Waterloo 1977, 139-145.

[28] JOHNSON, S.C., *Yacc-yet another compiler-compiler*, Comp. Sci. Techn. Rep. No. 32, Bell Laboratories, 1975.

[29] KNUTH, D.E., *On the translation of languages from left to right*, Information and Control 8 (1965), 607-639.

[30] KRON, H.H., H.J. HOFFMANN & G. WINKLER, *On a* SLR(k)*-based parser system which accepts non-*LR(k) *grammars*, in: G.I.-4. Jahrestagung, D. Siefkes (ed.), LNCS 26, Springer, Berlin, 1975, 214-223.

[31] KUNO, S., *The augmented predictive analyzer for context-free languages - Its relative efficiency*, Comm. ACM 9 (1966), 810-823.

[32] KURKI-SUONIO, R., *On top-to-bottom recognition and left recursion*, Comm. ACM 9 (1966), 527-528.

[33] LALONDE, W.R., *Regular right part grammars and their parsers*, Comm. ACM 20 (1977), 731-741.

[34] LALONDE, W.R., *Constructing* LR *parsers for regular right part grammars*, Acta Informatica 11 (1979), 177-193.

[35] LALONDE, W.R., E.S. LEE & J.J. HORNING, *An* LALR(k) *parser generator*, Proc. IFIP 1971 Congress, North-Holland Publ. Co., Amsterdam, 1972, 513-518.

[36] LECARME, O.L. & G.V. BOCHMANN, *A (truly) usable and portable compiler writing system*, Proc. IFIP 1974 Congress, North-Holland Publ. Co., Amsterdam, 1974, 218-221.

[37] LEWIS, P.M., D.J. ROSENKRANTZ & R.E. STEARNS, *Compiler Design Theory*, Addison-Wesley Publ. Co., Reading, Mass., 1976.

[38] LEWIS, P.M., D.J. ROSENKRANTZ & R.E. STEARNS, *Attributed translations*, J. Comput. System Sci. 9 (1974), 279-307.

[39] LEWIS, P.M. & R.E. STEARNS, *Syntax-directed transduction*, J. Assoc. Comput. Mach. 15 (1968), 465-488.

[40] MADSEN, O.L. & B.B. KRISTENSEN, LR*-parsing of extended context-free grammars*, Acta Informatica 7 (1976), 61-73.

[41] McAFEE, J. & L. PRESSER, *An algorithm for the design of simple precedence grammars,* J. Assoc. Comput. Mach. 19 (1972), 675-698.

[42] MICKUNAS, M.D. & V.B. SCHNEIDER, *A parser generating system for constructing compressed compilers,* Comm. ACM 16 (1973), 669-674.

[43] MICKUNAS, M.D., *On the complete covering problem for* LR(k) *grammars,* J. Assoc. Comput. Mach. 23 (1976), 17-30.

[44] MICKUNAS, M.D., R.L. LANCASTER & V.B. SCHNEIDER, *Transforming* LR(k) *grammars to* LR(1), SLR(1) *and* (1,1) *bounded right context grammars,* J. Assoc. Comput. Mach. 23 (1976), 511-533.

[45] MILTON, D.R., L.W. KIRCHHOFF & B.R. ROWLAND, *An* ALL(1) *compiler generator,* SIGPLAN Notices 14, No. 8, August 1979, 152-157.

[46] NIJHOLT, A., *On the covering of left-recursive grammars,* Conf. Record 4th ACM Symp. on Principles of Programming Languages 1977, 86-96.

[47] NIJHOLT, A., *On the covering of parsable grammars,* J. Comput. System Sci. 15 (1977), 99-110.

[48] NIJHOLT, A., *On the parsing and covering of simple chain grammars,* in: Automata, Languages and Programming, G. Ausiello & C. Böhm (eds.), LNCS 62, Springer, Berlin, 1978, 330-344.

[49] NIJHOLT, A., *Grammar functors and covers: From non-left-recursive to Greibach normal form grammars,* BIT 19 (1979), 73-78.

[50] NIJHOLT, A., IR-49, IR-52 en IR-xx, Drie interne rapporten, Vrije Universiteit, Amsterdam.

[51] NIJHOLT, A. & E. SOISALON-SOININEN, Ch(k) *grammars - A characterization of* LL(k) *languages,* in: Mathematical Foundations of Computer Science, J. Becvar (ed.), LNCS 74, Springer, Berlin, 1979, 390-397.

[52] NIJHOLT, A., *Structure preserving transformations on non-left-recursive grammars,* in: Automata, Languages and Programming, H.A. Maurer (ed.), LNCS 71, Springer, Berlin, 1979, 446-459.

[53] PAULL, M.C. & S.H. UNGER, *Structural equivalence of context-free grammars,* J. Comput. System Sci. 2 (1968), 427-463.

[54] PAULL, M.C. & S.H. UNGER, *Structural equivalence and* LL-k *grammars,* IEEE Conf. Record of 9th Ann. Symp. on Switching and Automata Theory 1968, 160-175.

[55] RÄIHÄ, K.-J., *On attribute grammars and their use in a compiler writing system,* Report A-1977-4, Dept. of Computer Science, University of Helsinki.

[56] RÄIHÄ, K.-J. et al., *The compiler writing system* HLP, Report A-1978-2, Dept. of Computer Science, University of Helsinki.

[57] RÄIHÄ, K.-J. & M. SAARINEN, *Developments in compiler writing systems,* in: G.I.-6. Jahrestagung, E.J. Neuhold (ed.), Springer, Berlin, 1976, 164-178.

[58] ROSENKRANTZ, D.J. & P.M. LEWIS, *Deterministic left-corner parsing,* IEEE Conf. Record 11-th Ann. Symp. on Switching and Automata Theory, 1970, 139-152.

[59] ROSENKRANTZ, D.J. & R.E. STEARNS, *Properties of deterministic top-down grammars,* Information and Control 17 (1970), 226-256.

[60] SIPPU, S. & E. SOISALON-SOININEN, *On defining error recovery in context-free parsing,* in: Automata, Languages and Programming, A. Salomaa & M. Steinby (eds.), LNCS 52, Springer, Berlin, 1977, 492-503.

[61] SIPPU, S. & E. SOISALON-SOININEN, *On constructing* LL(k) *parsers,* in: Automata, Languages and Programming, H.A. Maurer (ed.), LNCS 71, Springer, Berlin, 1979, 585-595.

[62] SOISALON-SOININEN, E., *On the covering problem for left-recursive grammars,* Theoret. Comput. Science 8 (1979), 1-12.

[63] SOISALON-SOININEN, E., *Characterization of* LL(k) *languages by restricted* LR(k) *grammars,* Report A-1977-3, Dept. of Computer Science, University of Helsinki.

[64] STEARNS, R.E., *Deterministic top-down parsing,* Proc. 5th Princeton Conf. on Information Sciences and Systems, 1971, 182-188.

[65] THOMPSON, D.H., *The design and implementation of an advanced* LALR *parse table construct,* UT-CSRG-79, University of Toronto, 1977.

[66] UKKONEN, E., *The nonexistence of some covering context-free grammars,* Info. Proc. Letters 8 (1978), 187-192.

[67] WIRTH, N., *The design of a PASCAL compiler,* Software - Pract. and Experience 1 (1971), 309-333.

[68] WOOD, D., *The theory of left factored languages,* Computer Journal 12/13 (1969/1970), 349-356 en 55-62.

[69] WOOD, D., *Bibliography of grammatical similarity,* EATCS-Bulletin 5, June 1978, 15-22.

GARBAGE COLLECTION

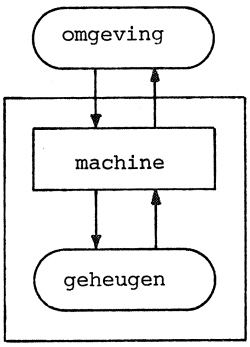
H.B.M. JONKERS
Mathematisch Centrum

1. INLEIDING

In deze lezing zal gepoogd worden om op een taal- en machine-onafhankelijke manier een overzicht van (een belangrijk deel van) het onderwerp "garbage collection" te geven. Hoewel de titel van deze lezing hier reeds mee in tegenspraak is, zal daarbij zo veel mogelijk Nederlandse terminologie gebruikt worden met, waar dat nuttig is, vermelding van het Engelse equivalent. Afgezien van enige vertrouwdheid met het begrip "implementatie van een programmeertaal" wordt geen voorkennis verondersteld en wordt alle benodigde terminologie ter plaatse geïntroduceerd. Dit houdt in dat het eerste deel van deze lezing voornamelijk bestaat uit begripsvorming en zich bezig houdt met het onderwerp "geheugenbeheer". In het tweede deel wordt dan het eigenlijke onderwerp van deze lezing behandeld.

2. GEHEUGENBEHEER

Een van de vele manieren om de semantiek van een programmeertaal te definieren is het beschrijven van een "abstracte machine", waarop programma's uit de betreffende programmeertaal direct geëxecuteerd kunnen worden. De semantiek van een programma is dan gedefinieerd door de acties van deze machine bij executie van het programma. ALGOL 68 is een voorbeeld van een programmeertaal waarvan de semantiek ook werkelijk zo gedefinieerd is. De acties van dergelijke machines kunnen onderscheiden worden in "uitwendige acties" en "inwendige acties". De uitwendige acties opereren op een "omgeving" en representeren datgene wat de gebruiker van de executie van een programma merkt. De inwendige acties opereren op een inwendige omgeving van de machine, het zgn. "geheugen", en blijven geheel verborgen voor de gebruiker (zie afbeelding 1).



Afbeelding 1

Het geheugen van een dergelijke abstracte machine kan beschouwd worden als een verzameling "objecten", en de inwendige acties van de machine kunnen beschouwd worden als "bewerkingen" op deze objecten. In principe bestaan er drie soorten bewerkingen: "creatie", "gebruik" en "vernietiging" van objecten. Dientengevolge kan men tijdens de executie van een programma op de abstracte machine "levende" en "dode" objecten onderscheiden. Afgezien van enige programmeertalen van lager niveau, zoals BASIC, kan het aantal objecten dat tijdens de executie van een programma op de bijbehorende abstracte machine gecreëerd wordt astronomisch hoog zijn. Gezien het hypothetische karakter van de abstracte machine heeft de vraag waar deze objecten bij hun creatie vandaan komen (en waar ze na hun vernietiging blijven) louter filosofische waarde. Er wordt eenvoudig aangenomen dat de abstracte machine een onuitputtelijke voorraad ongebruikte objecten heeft, waaruit er bij creatie van een object telkens één gepakt wordt. Met andere woorden, de abstracte machine wordt geacht een oneindig geheugen te hebben.

Hoe anders is de situatie waarmee de implementator van een programmeertaal geconfronteerd wordt. Voor hem staat een "concrete machine" met een in vergelijking tot de abstracte machine armzalig klein geheugen. Bovendien zijn alle objecten uit het geheugen van de concrete machine, de "cellen", volkomen gelijkvormig, terwijl de objecten uit het geheugen van de abstracte machine de meest uiteenlopende vormen kunnen hebben. De implementator ziet zich nu voor de taak gesteld ervoor te zorgen dat de concrete machine zich wat zijn uitwendig gedrag betreft gedraagt als de

abstracte machine behorend bij de te implementeren programmeertaal L. De gebruiker die een programma in L aanbiedt aan de concrete machine zal, indien de implementatie correct is, dit programma zich bij executie op de concrete machine zien gedragen zoals door de semantiek van L voorgeschreven.

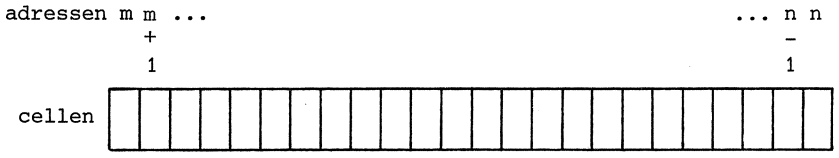
Hoewel dus het uitwendige gedrag van de concrete machine tijdens de executie van een programma in de geïmplementeerde taal volledig voorgeschreven is, is de implementator in de regel volledig vrij in de keuze van het inwendig gedrag van de concrete machine; de gebruiker merkt daar toch niets van. Wegens de hemelsbreed verschillende structuur van de geheugens van de abstracte en de concrete machine is deze vrijheid uiteraard ook wel nodig. Zij stelt de implementator in staat de objecten en bewerkingen van de abstracte machine te representeren door bewerkingen en objecten van de concrete machine, op willekeurig welke manier hem het beste uitkomt.

De meest ingewikkelde representaties zijn in een implementatie mogelijk. Enigszins vereenvoudigend komen zij echter alle op het volgende neer. Ieder object uit het geheugen van de abstracte machine wordt in de concrete machine gerepresenteerd door een verzameling cellen, een "locatie". De creatie van een object wordt gerepresenteerd door het "toewijzen" van een locatie aan het object, waarna we zeggen dat het object zich op deze locatie "bevindt". De vernietiging van een object wordt gerepresenteerd door het object van zijn locatie te "verwijderen". De verzameling van alle cellen die op een bepaald moment van de executie van een programma niet aan een object zijn toegewezen, zullen we het "vrije geheugen" noemen. De (denkbeeldige) instantie die het toewijzings- en ontruimingsbeleid regelt zullen we aanduiden met de "geheugenbeheerder".

De twee voornaamste problemen waarmee een implementator bij het ontwerpen van een geheugenbeheerder geconfronteerd wordt zijn:

(1) Eindigheid van het geheugen.
Het aantal cellen in het geheugen van de concrete machine is eindig. Ten einde aan ieder van het mogelijk zeer grote aantal te creëren objecten een locatie toe te kunnen wijzen, zal de geheugenbeheerder op zeer zuinige wijze met het geheugen moeten omspringen. Dit laatste houdt in dat cellen die niet langer gebruikt worden zoveel mogelijk opnieuw gebruikt worden.

(2) Lineariteit van het geheugen.
Het geheugen van de concrete machine vormt in feite een verzameling cellen, die ieder een uniek "adres" hebben. Deze adressen kunnen beschouwd worden als natuurlijke getallen en de verzameling van alle adressen vormt meestal een interval van de natuurlijke getallen, zodat het geheugen van de concrete machine opgevat kan worden als een aaneengesloten rij cellen (zie afbeelding 2). Om verschillende redenen is het wenselijk dat objecten een aaneengesloten deelrij, een zgn. "compacte" locatie, krijgen toegewezen. Ongecontroleerde toewijzing en weer vrijgeven van geheugen kan er echter toe leiden dat het vrije geheugen er uit komt te zien als gatenkaas, een verschijnsel dat bekend staat als "fragmentatie". Dit kan er uiteindelijk toe leiden dat het onmogelijk is geheugen toe te wijzen aan een object, alhoewel de totale omvang van het vrije geheugen meer dan voldoende is.



Afbeelding 2

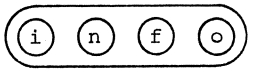
Een voor de hand liggende methode om het eerste probleem aan te pakken is om de geheugenbeheerder een administratie van het vrije geheugen te laten bijhouden. Bij creatie van een object wordt met behulp van deze administratie een stukje vrij geheugen aan het object toegewezen, waarna dit stukje geheugen uit de administratie wordt geschrapt. Bij vernietiging van het object wordt dit stukje geheugen weer vrijgegeven en in de

administratie opgenomen. Als nu bovendien nog een aantal plezierige eigenschappen betreffende de levensduur van objecten bekend is, kan vaak met een minimum aan administratie ook het probleem van de fragmentatie van het geheugen op efficiënte wijze opgelost worden. Zo kan in implementaties van ALGOL 60, waar alle objecten een geneste levensduur hebben, het werk van de geheugenbeheerder beperkt worden tot het bijhouden van een stapelwijzer [25].

In bovenstaande aanpak wordt er min of meer van uitgegaan dat de gebruiksduur van objecten gelijk is aan hun levensduur. Ongelukkigerwijze is er een groot aantal programmeertalen (o.a. LISP, PASCAL, ALGOL 68) bij de implementatie waarvan men er daarmee niet komt. De reden is dat in deze talen objecten voorkomen, die in de abstracte machine weliswaar gecreëerd, maar nooit expliciet vernietigd worden. Hoewel de levensduur van deze objecten dus "oneindig" is, zal hun gebruiksduur dit echter veelal niet zijn. In de abstracte machine levert dit natuurlijk geen problemen op. In de concrete machine kan een uitgebreid gebruik van dergelijke objecten bij bovengenoemde aanpak echter zeer snel leiden tot uitputting van het vrije geheugen. Dit kan alleen voorkomen worden door de geheugenbeheerder toe te rusten met de mogelijkheid om geheugen vrij te geven, dat bezet wordt door weliswaar levende, maar niet langer gebruikte objecten. De cruciale vraag hierbij is: hoe stelt de geheugenbeheerder vast dat een object niet langer gebruikt wordt? Om een antwoord op deze vraag te geven is het noodzakelijk iets dieper in te gaan op de structuur van objecten en de manier waarop de abstracte machine met deze objecten omgaat.

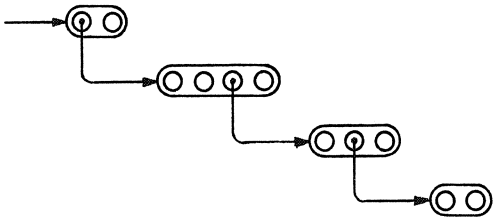
Een object uit het geheugen van de abstracte machine kan gezien worden als een "doos" waarin zich "informatie" bevindt. De doos is opgebouwd uit een aantal "vakken" waarin zich de "informatie-eenheden" bevinden (zie afbeelding 3). De vakken van deze doos zullen we de "componenten" van het object noemen. De machine kan zich "toegang" tot een doos verschaffen en de inhoud ervan raadplegen ("lezen") of veranderen ("schrijven"). De aard van de informatie die in een object ligt opgeslagen kan zeer verschillend zijn. Er is echter één informatie-eenheid die speciale aandacht verdient: de "verwijzing". Een verwijzing "verwijst" naar een object. Ieder object heeft een unieke verwijzing, die aan hem is toegekend bij zijn creatie (zoals een kind een naam krijgt bij zijn geboorte). De enige manier voor de machine om toegang te krijgen tot een object is via zijn verwijzing. D.w.z., de

machine beschikt over een mechanisme waar het een verwijzing in kan stoppen en dat vervolgens het unieke object waarnaar verwezen wordt te voorschijn haalt.



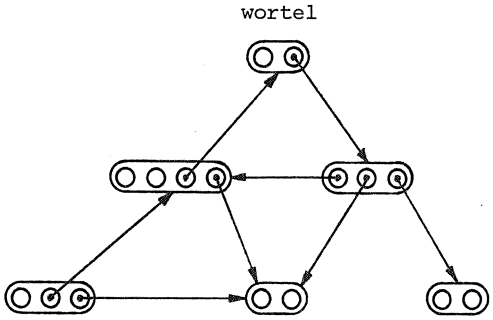
Afbeelding 3

Aangezien een verwijzing zelf informatie is, die gelezen en geschreven kan worden, kan de abstracte machine (gestuurd door het programma) "toegangspaden" volgen. Dit houdt in dat de machine zich via een aantal elkaar opvolgende toegangs- en leesoperaties toegang verschaft tot een object (zie afbeelding 4). Ten einde op deze manier toegang tot ook maar één object te kunnen krijgen zijn echter "startpunten" nodig: er moeten objecten zijn, waarvan de verwijzingen direct beschikbaar zijn voor de machine, d.w.z. zonder eerst een toegangspad te hoeven volgen. Zonder verlies van algemeenheid kunnen we aannemen dat er maar één zo'n object is, dat we de "wortel" zullen noemen.



Afbeelding 4

Het geheugen van de abstracte machine kan nu gezien worden als een graaf, waarvan de objecten de knopen en de verwijzingen de pijlen zijn. Voor een aantal programmeertalen (zoals ALGOL 68) is dit een veel ingewikkelder soort graaf dan we kennen uit de grafentheorie. Dat komt doordat in die talen objecten elkaar vaak op de meest ingewikkelde manieren kunnen overlappen. Ter wille van de beknoptheid zullen we hier echter aannemen dat de graaf een "nette" graaf is, oftewel dat objecten elkaar niet overlappen (zie afbeelding 5).



Afbeelding 5

Met betrekking tot het gebruik van verwijzingen doet zich nog een belangrijk probleem voor. Na de vernietiging van een object kunnen andere levende objecten in principe nog verwijzingen naar het object bevatten. Het is uiteraard niet de bedoeling dat deze "postume verwijzingen" (Eng. "dangling references") gebruikt worden om toegang te krijgen tot het dode object. Ten aanzien van het voorkómen van het gebruik van postume verwijzingen kan men globaal gesproken vier standpunten innemen:

(1) Maak het de gebruiker onmogelijk de verwijzing van een object met eindige levensduur in handen te krijgen.
Dit is de eenvoudigste, maar tevens de meest beperkende methode om het gebruik van postume verwijzingen te voorkomen. Deze methode wordt b.v. in PASCAL gebruikt.

(2) Zorg ervoor, dat wanneer een object X een verwijzing naar een object Y bevat, de levensduur van X bevat is in die van Y.
Deze methode wordt in ALGOL 68 gebruikt, waar dit aanleiding geeft tot een aantal regels die bekend staan als de "scope rules".

(3) Test tijdens executie op het gebruik van postume verwijzingen.
Dit is een veel minder beperkende methode dan (1) en (2), die echter wel de hoogste kosten met zich mee brengt (vooral in tijd). Vandaar waarschijnlijk dat deze methode weinig gebruikt wordt.

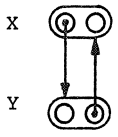
(4) Laat de verantwoordelijkheid aan de gebruiker over.
Dit is het meest liberale, maar ook het onveiligste standpunt, dat o.a. in PL/I gehuldigd wordt.

Keren we nu terug naar het probleem hoe de geheugenbeheerder kan vaststellen dat een object niet langer gebruikt wordt. Daarbij kan men zich allereerst afvragen wanneer de geheugenbeheerder dient vast te stellen dat een object niet meer gebruikt wordt. Het absolute summum zou uiteraard zijn dat dit gebeurt op het moment dat het object voor de laatste keer gebruikt wordt, zodat het aan het object toegewezen geheugen meteen vrijgegeven kan worden. In het algemeen is echter op het moment dat een object voor de laatste keer gebruikt wordt nog niet bekend dat dit inderdaad de laatste keer is, aangezien dit meestal afhangt van dingen die nog moeten gebeuren. Het is dus onvermijdelijk dat de geheugenbeheerder het in onbruik raken van een object pas enige tijd later vaststelt.

De oudste oplossing van het probleem om het in onbruik raken van objecten vast te stellen gaat uit van het principe: wie weet beter dan de gebruiker wanneer een object in onbruik raakt? Hij immers heeft het programma geschreven. De gebruiker wordt daarom de mogelijkheid gegeven in zijn programma aan te geven welke objecten volgens hem niet meer gebruikt worden. De meeste implementaties spelen de goedgelovige Thomas en interpreteren deze hints als opdrachten om het geheugen vrij te geven (en veel anders kunnen ze meestal ook niet). Hierdoor kunnen eenvoudig postume verwijzingen ontstaan, waardoor de gebruiker op geen enkele manier tegen zijn eigen fouten beschermd wordt. Desalniettemin wordt deze methode nog veelvuldig toegepast, o.a. in PASCAL. (N.B. Strikt genomen zijn laatstgenoemde postume verwijzingen van een ander soort dan die die we al eerder tegenkwamen. De reeds eerder besproken postume verwijzingen zijn van een semantische aard, doordat ze ontstaan zijn door een door de semantiek voorgeschreven vernietiging van een object. De postume verwijzingen waar we hier mee te maken hebben zijn van een implementatie-technische aard, doordat ze ontstaan zijn door het beroven van een object van het aan hem toegewezen geheugen, terwijl het object semantisch gezien gewoon doorleeft.)

De hierboven beschreven oplossing is niet alleen onveilig, zij draagt in feite een deel van de taak van de geheugenbeheerder over aan de gebruiker, hetgeen onwenselijk is. Een tweede oplossing gaat uit van de volgende grondgedachte. De machine kan een object alleen gebruiken (d.w.z. zich toegang verschaffen tot het object) als het de beschikking heeft over zijn verwijzing. Als geen van de objecten in het geheugen meer een

verwijzing naar het object bevat, staat het dus vast dat het object niet meer gebruikt wordt (dit geldt echter niet voor de wortel!). Dit leidt tot een methode waarbij de geheugenbeheerder bijhoudt hoeveel verwijzingen er nog naar een object bestaan [6]. Dit gebeurt door het bijhouden van een tellertje (Eng. "reference count") in ieder object. Bij het copiëren van een verwijzing naar het object moet dit tellertje opgehoogd worden, en bij vernietiging van een verwijzing naar het object moet het tellertje afgelaagd worden. Zodra het tellertje nul wordt bestaan er geen verwijzingen meer naar het object en kan het aan het object toegewezen geheugen vrijgegeven worden. Deze methode is weliswaar veilig, maar heeft als voornaamste nadeel dat zij vrij veel extra tijd en ruimte kost, zelfs als nooit geheugen vrijgegeven wordt (of behoeft te worden). Bovendien faalt de methode indien circulaire verwijzingen kunnen voorkomen [21]. Zo zullen in afbeelding 6, indien buiten de objecten X en Y geen verwijzingen naar X en Y bestaan, de tellers van X en Y voor eeuwig op 1 blijven staan. Dientengevolge zal het geheugen bezet door X en Y nooit vrijgegeven worden, alhoewel X en Y nooit meer gebruikt (kunnen) worden.



Afbeelding 6

De oorzaak van het manco van bovenstaande methode ligt in het feit dat er min of meer van uitgegaan wordt dat zolang er nog een verwijzing naar een object bestaat, dit object nog toegankelijk is. Dit is echter niet waar, zoals uit afbeelding 6 blijkt. Afgezien van de wortel is een object alleen maar toegankelijk als er een verwijzing naar het object bestaat, die bevat is in een ander toegankelijk object. Alleen dan immers kan de machine deze verwijzing te pakken krijgen. Aangezien de verwijzing van de wortel de enige verwijzing is waarover de machine direct kan beschikken, houdt dit in minder recursieve termen in dat een object alleen toegankelijk is als er een toegangspad vanuit de wortel naar het object bestaat. Objecten waarvoor een dergelijk pad bestaat zullen we "bereikbaar" noemen.

Het begrip "bereikbaarheid" vormt de grondslag van een derde methode om het in onbruik raken van objecten vast te stellen. Deze methode wordt "sanering" (Eng. "garbage collection") genoemd, en vormt het eigenlijke onderwerp van deze lezing (de eerste beschrijving van deze methode is te vinden in [22]; zie ook [23]). Zij gaat er van uit dat de geheugenbeheerder een speciale werknemer in dienst heeft, de zgn. "saneerder" (Eng. "garbage collector"), wiens enige taak het is om vast te stellen of objecten bereikbaar zijn of niet, en zo nee het aan deze objecten toegewezen geheugen vrij te geven. Het werkschema van deze werknemer was traditioneel als volgt. Zolang de geheugenbeheerder nog voldoende vrij geheugen heeft, staat de saneerder op non-actief. Pas als de geheugenbeheerder door kwistig uitdelen door zijn vrij geheugen raakt, wordt de saneerder ingeschakeld. Het is zijn taak om alle onbereikbare objecten op te sporen en het geheugen toegewezen aan deze objecten vrij te geven. Als de saneerder zijn taak volbracht heeft, kan de geheugenbeheerder zijn werk met een verse (en hopelijk voldoend grote) voorraad vrij geheugen voortzetten.

Sanering is een veilige methode die, in tegenstelling tot de methode van het tellen van verwijzingen, alle in onbruik geraakte objecten opspoort. Bovendien kost deze methode in zijn traditionele vorm alleen extra tijd als het vrije geheugen ook werkelijk een keer uitgeput raakt. Het grootste bezwaar dat kleeft aan althans de traditionele manier van sanering is, dat gedurende het saneringsproces de executie van het programma stil ligt. Gezien de tijd die dit proces kost, kan dit in interactieve toepassingen leiden tot onaanvaardbaar lange reactietijden. Dit probleem kan ondervangen worden door het werkschema van de saneerder te veranderen en in plaats van hem in te schakelen wanneer de nood aan de man komt, hem in continu-dienst te laten werken. Dit houdt in dat de saneerder en het programma parallel aan elkaar werken. Ten einde te voorkomen dat saneerder en programma elkaar voor de voeten lopen, is echter een vrij uitgebreid synchronisatie-apparaat noodzakelijk. De kosten hiervan zijn alleen gerechtvaardigd als ofwel de continuïteit van de executie van programma's essentieel is, ofwel deze kosten opgevangen kunnen worden door de beschikbaarheid van "toegewijde apparatuur".

In tegenstelling tot de traditionele sanering kan het onderwerp parallelle sanering nog geenszins beschouwd worden als een afgerond geheel, reden waarom we ons hier zullen beperken tot eerstgenoemde. De

geïnteresseerde lezer zij verwezen naar [9, 24, 28, 33]. Een andere methode om het langdurig onderbreken van de executie van een programma voor een sanering te voorkomen, bestaat uit het "uitsmeren" van het sanerings-proces over het proces van geheugen-toewijzing. Iedere keer dat een stuk geheugen wordt toegewezen worden dan een paar stapjes van een sanering uitgevoerd. Een algoritme die hierop is gebaseerd staat beschreven in [1] (voornaamste bezwaar: er is een dubbel geheugen nodig). Een andere interessante mogelijkheid bestaat uit het combineren van de methode van het tellen van verwijzingen met de methode van sanering. Men zie hiervoor [2, 7, 36].

In de praktijk is de taak van de saneerder ingewikkelder dan hierboven is beschreven. Allereerst is het vaak zo dat de saneerder niet de enige werknemer is die de geheugenbeheerder in dienst heeft. Gebruikmakend van de verschillende eigenschappen van de levensduur van objecten, kan de geheugenbeheerder een aantal werknemers (zoals "stapelaars") in dienst hebben, die ieder het geheugenbeheer voor een bepaalde klasse objecten voor hun rekening nemen. Ten einde te voorkomen dat de saneerder roet in het eten van de andere werknemers gooit, zal duidelijk omschreven moeten worden wat tot de bevoegdheid van de saneerder behoort en wat niet. Voor de eenvoud zullen we dit probleem hier echter afdoen met de aanname dat sanering de enige strategie is die gevolgd wordt bij het vrijgeven van geheugen.

Verder is het zo dat vele geheugenbeheerders er van uitgaan dat het vrije geheugen compact is. Dit vereenvoudigt niet alleen de nodige administratie, het vereenvoudigt tevens het toewijzen van geheugen. Aannemende dat objecten compacte locaties krijgen toegewezen (hetgeen redelijk is) kan men immers eenvoudig een locatie toewijzen aan een object door van een van de uiteinden van het vrije geheugen een voldoende groot stuk af te snijden, hetgeen de compactheid van het vrije geheugen niet verstoort. Echter, het vrije geheugen zoals dat er na een sanering uitziet zal in de regel niet compact zijn. De locaties die door de saneerder worden vrijgegeven kunnen door het hele geheugen verspreid liggen, met als gevolg een sterke fragmentatie van het vrije geheugen. Een saneerder wordt daarom zeer vaak gecombineerd met een "compactificeerder", die het vrije geheugen "compactificeert", d.w.z. de objecten in het geheugen zodanig "verschuift" dat een compact vrij geheugen resulteert. Het duo saneerder & compactificeerder wordt een "compactificerende saneerder" (Eng.

"compact(ify)ing garbage collector") genoemd.

In implementaties waar we met virtueel geheugen te maken hebben, heeft sanering in feite alleen zin als dit gecombineerd wordt met compactificatie. De reden hiervoor is dat in een dergelijke omgeving vrij geheugen niet echt schaars is. Evenwel, hoe groter de omvang van het in gebruik zijnde geheugen is, des te meer zullen de toegangstijden toenemen (door "page faults"). Een sanering gevolgd door een compactificatie kan dan toegepast worden om de omvang van het in gebruik zijnde geheugen, en dus de toegangstijden, te verkleinen.

3. (COMPACTIFICERENDE) SANERING

In het voorgaande is het onderwerp sanering besproken als een van de mogelijke oplossingen voor het probleem van het geheugenbeheer. We zullen ons nu bezighouden met (compactificerende) sanering als probleem op zich. Dit houdt in dat we een aantal van de belangrijkste algoritmen voor de oplossing van dit probleem zullen bespreken. In plaats van een formeel model voor de beschrijving van deze algoritmen te introduceren, zullen we ons daarbij baseren op de begrippen en aannames die in het voorgaande op min of meer informele wijze besproken zijn. Indien nodig zullen hieraan nog nieuwe begrippen en aannames toegevoegd worden. De algoritmen zullen beschreven worden in een soort ABN (Algoritmisch Beschaafd Nederlands), waarvan de semantiek over het algemeen zonder meer duidelijk zal zijn.

Het saneringsprobleem zou triviaal zijn, ware het niet dat de saneerder en (eventueel) de compactificeerder onder een zware geheugenbeperking moeten werken. Zoals iedere algoritme hebben de saneerder en de compactificeerder nl. een zekere (en in de regel niet voorspelbare) werkruimte nodig. Op het moment dat de saneerder aangeroepen wordt is er echter een gebrek aan vrij geheugen (want dat was juist de reden dat de saneerder aangeroepen werd). Deze omstandigheid heeft geleid tot het ontwerp van ingenieuze saneerders en compactificeerders, die hun werkruimte op allerlei slimme manieren in het geheugen coderen zonder dat dit veel extra ruimte kost. Dergelijke coderingstrucs leiden er echter wel toe dat een beschrijving van de betreffende algoritmen ondoorzichtig, om niet te zeggen onleesbaar wordt. We zullen proberen dit te voorkomen door de coderingstruc en de onderliggende algoritme van elkaar te scheiden. De

"abstracte" algoritme die men aldus verkrijgt is meestal eenvoudig te begrijpen, terwijl door een simpele transformatie met behulp van de coderingstruc de "concrete" algoritme te verkrijgen is.

Het onderwerp valt op min of meer natuurlijke wijze uiteen in sanerings- en compactificatie-algoritmen. Vandaar dat we deze in twee aparte secties zullen behandelen.

3.1. Sanerings-algoritmen

Het is de taak van de saneerder om voor ieder object te bepalen of dit object nog bereikbaar is en zo nee de aan dit object toegewezen locatie vrij te geven. Het is niet moeilijk in te zien, dat de enige manier om de onbereikbaarheid van een object te bepalen is om de verzameling van alle bereikbare objecten te genereren en te kijken of het betreffende object daar in zit of niet. Aangezien het genereren van deze verzameling een dure aangelegenheid is, is het natuurlijk niet verstandig om deze operatie voor ieder object opnieuw uit te voeren. Het is het beste om deze verzameling één keer te genereren en bereikbare objecten als zodanig te "merken". Dit leidt tot de volgende sanerings-algoritme:

Sanerings-algoritme 1
Merk alle bereikbare objecten.
Geef alle locaties bezet door ongemerkte objecten vrij.

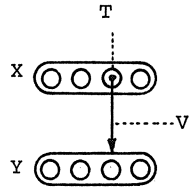
Uit bovenstaande komen de eerste extra kosten die een saneerder in een implementatie vergt naar voren: de "merk-informatie". Deze merk-informatie kan op zeer vele manieren geïmplementeerd worden. Het meest gebruikelijk is om een bitje in iedere locatie te reserveren dat aangeeft of het zich op deze locatie bevindende object gemerkt is of niet. Een andere veel gebruikte mogelijkheid, met name als er geen plaats is voor een merkbit in een locatie, is om een compact deel van het geheugen te reserveren als "bit-afbeelding" (Eng. "bit map"). Iedere locatie wordt dan afgebeeld op een bit uit dit stuk geheugen, die aangeeft of het object dat zich op deze locatie bevindt gemerkt is of niet.

Zoals te zien is valt de sanerings-algoritme uiteen in een "merk-algoritme" en een "ontruimings-algoritme". De laatste algoritme hangt in

sterke mate af van het soort administratie dat de geheugenbeheerder voert. Gebruikt deze b.v. een lijst van vrije locaties om het vrije geheugen te administreren, dan komt de algoritme neer op het aan deze lijst toevoegen van de locaties waar zich een ongemerkt object bevindt. In het algemeen zal de ontruimings-algoritme vrij triviaal zijn, reden waarom we er geen verdere aandacht aan zullen besteden.

3.1.1. Merk-algoritmen

Minder eenvoudig ligt het met merk-algoritmen, waarvan er een groot aantal bekend zijn. In wezen echter is iedere merk-algoritme niets meer dan een regelrechte toepassing van de definitie van het begrip bereikbaarheid en werkt als volgt. Aanvankelijk is geen van de objecten gemerkt. De algoritme begint met de wortel te merken. Vervolgens wordt een of ander gemerkt (en dus bereikbaar) object X "bezocht" (zie afbeelding 7). Tijdens dit bezoek wordt gekeken of X componenten heeft die een verwijzing bevatten; deze componenten zullen we de "vertakkingen" van het object noemen. Bevat X inderdaad vertakkingen dan wordt er zo'n vertakking T "uitgekozen". De verwijzing V bevat in T wordt gebruikt om het object Y waarnaar V verwijst "op te sporen" en vervolgens te merken (indien dit niet reeds gebeurd is). Uitgaande van het feit dat X bereikbaar is, is het immers duidelijk dat ook Y bereikbaar is. Daarna wordt opnieuw een gemerkt object bezocht, etc. Dit proces van het bezoeken van objecten wordt voortgezet totdat alle vertakkingen van gemerkte objecten minstens één maal zijn uitgekozen.



Afbeelding 7

Uit de definitie van bereikbaarheid volgt vrijwel meteen dat na afloop van het bovenstaande proces de verzameling gemerkte objecten gelijk is aan de verzameling bereikbare objecten. Wat daar niet uit volgt is dat het proces ook termineert. Om daarvoor te zorgen moet het bezoeken van objecten

en het uitkiezen van vertakkingen enigszins aan banden gelegd worden. Wat zijn in dit verband redelijke beperkingen? Een in algemene zin redelijke beperking die aan een algoritme opgelegd kan worden is, dat de algoritme iets niet twee keer doet als één keer voldoende is. Toegepast op dit geval komt dat er op neer dat we het volgende eisen:

(1) Zodra bekend is dat alle vertakkingen van een object reeds een keer zijn uitgekozen, wordt het object niet meer bezocht.
(2) Een vertakking van een object wordt hooguit één maal uitgekozen.

Deze beperkingen zorgen er niet alleen voor dat het merkproces termineert, maar tevens dat de tijd die dit proces vergt lineair is in het aantal bereikbare objecten en het totaal aantal vertakkingen van bereikbare objecten, hetgeen het beste is dat men kan bereiken. Met soepeler beperkingen zijn nog vele andere algoritmen denkbaar dan die welke we hier zullen bespreken. Vrijwel zonder uitzondering zijn deze echter ondraaglijk inefficiënt (zie b.v. algoritme A op blz. 414 in [18]).

Voor het afdwingen van bovenstaande beperkingen moeten zekere extra kosten gemaakt worden. De saneerder moet namelijk precies weten welke objecten nog bezocht mogen worden en welke vertakkingen van deze objecten nog uitgekozen mogen worden. Deze informatie, die we de "status-informatie" zullen noemen, zorgt voor de tweede soort extra kosten die een saneerder met zich mee brengt. We zullen de status-informatie representeren door een (variabele) verzameling S van objecten. Het feit dat het object X in S zit wil zeggen dat X nog bezocht mag worden. Bovendien associëren we met ieder object X in S een (variabele) verzameling vertakkingen van X, aangeduid door K(X). Het feit dat een vertakking T in K(X) zit wil zeggen dat T nog uitgekozen mag worden.

Gebruikmakend van de verzameling S en de afbeelding K kan nu eenvoudig de volgende merk-algoritme afgeleid worden, die aan de bovengenoemde beperkingen voldoet (aangenomen wordt dat aanvankelijk alle objecten ongemerkt zijn):

```
Merk-algoritme 1
S := ∅.
Merk W.
K(W) := {T | T is een vertakking van W}.
Stop W in S.
Zolang S ≠ ∅
| Pak een X uit S.
| Als K(X) ≠ ∅
| | Pak een T uit K(X).
| | Stop X in S.
| | Laat V = inhoud(T).
| | Laat Y = object(V).
| | Als Y niet gemerkt is
| | | Merk Y.
| | | K(Y) := {T | T is een vertakking van Y}.
| | | Stop Y in S.
```

Hierin is "W" de wortel en "object(V)" het object waarnaar de verwijzing V verwijst. De operatie "Pak een element uit een verzameling" heeft het effect dat dit element uit de verzameling verwijderd wordt.

Uit bovenstaande algoritme komt een derde kostenpost naar voren die de saneerder vergt (of kan vergen). De saneerder moet namelijk in staat zijn van een gegeven object te bepalen wat zijn vertakkingen zijn. De hiertoe benodigde informatie zullen we de "type-informatie" noemen en de met een bepaald object geassocieerde type-informatie zullen we het "type" van het object noemen. Hoewel aanwezig in het geheugen van de abstracte machine, hoeft deze type-informatie, anders dan ten behoeve van de saneerder, meestal niet in het geheugen van de concrete machine aanwezig te zijn. Om redenen van efficiëntie zal de implementator namelijk proberen om zo min mogelijk informatie uit het geheugen van de abstracte machine op te nemen in het geheugen van de concrete machine. Aangezien de saneerder uiteindelijk echter op het geheugen van de concrete machine moet werken, moet er een manier zijn voor de saneerder om de type-informatie te pakken te krijgen. Er zijn zeer vele methoden om dit te bewerkstelligen.

Allereerst, als alle objecten dezelfde structuur hebben (zoals in puur LISP), dan is het type van alle objecten bij implementatie van de taal bekend en brengt de type-informatie geen extra kosten met zich mee. In zgn. "sterk getypeerde" talen (zoals ALGOL 68) is het type van (de meeste) objecten weliswaar niet bij implementatie van de taal bekend, maar wel vlak voor de executie van het programma ("statisch"). Bovendien bestaat er in deze talen een vast verband tussen het type van een object X en het type van een object Y waarnaar de inhoud van een vertakking T van X verwijst. Door dit verband op te nemen in de type-informatie, kan gegeven het type van de wortel het type van ieder bereikbaar object afgeleid worden. Afgezien van de volledig statische type-informatie, is het enige wat dit extra kost het feit dat voor ieder object in de verzameling S het type bijgehouden moet worden. In alle andere gevallen komt men er niet onderuit om expliciet type-informatie in het geheugen op te nemen. De meest voor de hand liggende methode is om bij ieder object aan te geven wat het type is van dat object, b.v. door een verwijzing op te nemen in het object die verwijst naar een brok type-informatie. Een andere methode is om bij iedere vertakking aan te geven wat het type is van het object waarnaar de inhoud van deze vertakking verwijst (hetgeen alleen kan als het type van objecten niet verandert). Nog een andere methode bestaat uit het van een speciaal merkteken voorzien van vertakkingen van objecten, zodat ze onderscheidbaar zijn van de andere componenten.

Laten we nog eens naar merk-algoritme 1 kijken. Zoals te zien is het aantal malen dat een bereikbaar object bezocht wordt gelijk aan het aantal vertakkingen van het object. Een optimalisatie die voor de hand ligt is om al deze bezoeken tot één bezoek te combineren. Daar we in de keuze van een object uit de verzameling S in merk-algoritme 1 geheel vrij zijn, kan dit inderdaad ook. Het is dan overbodig om voor ieder object in S bij te houden welke vertakkingen nog uitgekozen mogen worden en welke niet (ze mogen allemaal uitgekozen worden), zodat de afbeelding K overbodig wordt. Aldus kan de volgende merk-algoritme direct uit merk-algoritme 1 afgeleid worden:

Merk-algoritme 1.1

```
S := ∅.
Merk W.
Stop W in S.
Zolang S ≠ ∅
| Pak een X uit S.
| Voor iedere vertakking T van X
| | Laat V = inhoud(T).
| | Laat Y = object(V).
| | Als Y niet gemerkt is
| | | Merk Y.
| | | Stop Y in S.
```

Met name als objecten gemiddeld veel vertakkingen hebben is bovenstaande algoritme qua tijd erg efficiënt. Efficiënte implementatie van S is b.v. mogelijk door in ieder object een extra component op te nemen die een verwijzing kan bevatten. Met behulp hiervan kan S geïmplementeerd worden als een gelinkte lijst. Bovendien kan de merk-informatie dan op een slimme manier in deze extra component gecodeerd worden. In laatstgenoemde vorm komt merk-algoritme 1.1 dan overeen met de merk-algoritme uit de zgn. "LISP 2 garbage collector" (zie de oplossing van opgave 2.5.33 in [18]).

In merk-algoritme 1.1 is de status-informatie (bestaande uit de verzameling S en de afbeelding K) gereduceerd tot alleen de verzameling S. Het is echter ook mogelijk om S te elimineren, zodat alleen K overblijft. Om dit te demonstreren zullen we merk-algoritme 1 in een iets andere (concretere) vorm gieten. Allereerst nemen we aan dat alle vertakkingen van een object X vanaf 1 genummerd zijn en dat de vertakkingen in deze volgorde uit K(X) gepakt worden. K(X) kan dan gerepresenteerd worden door een tellertje k(X), waarbij de volgende invariant geldt:

$$K(X) = \{\text{i-de vertakking van } X \mid k(X) < i \leq \text{graad}(X)\}$$

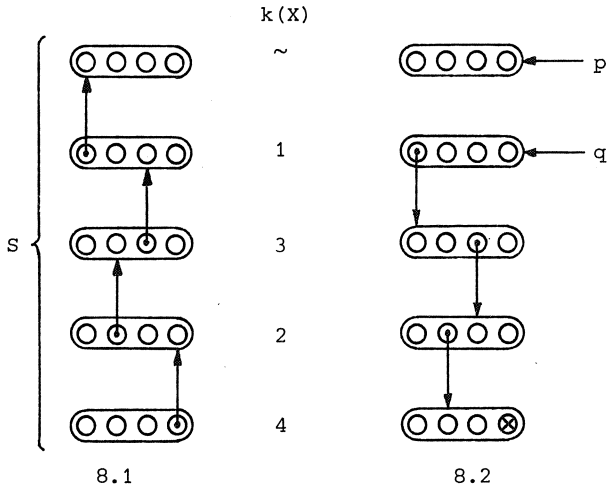
Hierin is "graad(X)" het aantal vertakkingen van X. Verder kiezen we als representatie voor S een stapel. Het is bij stapels de gewoonte om aan te nemen dat de top van de stapel vrij toegankelijk is, dus zonder dit element eerst van de stapel te hoeven halen. Hiervan gebruikmakend kan merk-algoritme 1 nu als volgt herschreven worden:

Merk-algoritme 1.2

```
Merk W.
k(W) := 0.
S := <W>.
Zolang S ≠ <>
| Laat X = top(S).
| Als k(X) = graad(X)
| | Verlaag S.
| zoniet
| | k(X) := k(X) + 1.
| | Laat T = vertakking(X, k(X)).
| | Laat V = inhoud(T).
| | Laat Y = object(V).
| | Als Y niet gemerkt is
| | | Merk Y.
| | | k(Y) := 0.
| | | Stapel Y op S.
```

Hierin is "<W>" de stapel met W als enige element en "<>" de lege stapel; "vertakking(X, i)" is de i-de vertakking van X.

Uit bovenstaande algoritme is eenvoudig op te maken dat telkens wanneer een object X op de top van de stapel staat en daar een object Y bovenop gezet wordt, de k(X)-e vertakking van X een verwijzing naar Y bevat. Dit leidt ertoe dat de stapel S er tussen twee slagen van het merkproces uitziet zoals in afbeelding 8.1 (waarbij aangenomen wordt dat ieder object uit precies vier vertakkingen bestaat). Met behulp van twee variabele verwijzingen p en q kan deze situatie zonder verlies van informatie omgezet worden in die van afbeelding 8.2. Hierin is het kruisje in de vierde vertakking van het onderste object een dummy verwijzing, die we zullen aanduiden met "nil". Het voordeel van de situatie in afbeelding 8.2 is echter dat zij de stapel S overbodig maakt, of beter gezegd, afbeelding 8.2 schetst een mogelijke implementatie van de stapel S zonder dat dit extra ruimte kost (afgezien van de twee variabele verwijzingen p en q). De op deze truc gebaseerde merk-algoritme staat bekend als de "Schorr-Waite merk-algoritme" en werd voor het eerst beschreven in [27].



Afbeelding 8

De Schorr-Waite merk-lgoritme kan op eenvoudige wijze afgeleid worden uit merk-algoritme 1.2 door de stapeloperaties op S uit te drukken in termen van de implementatie van S zoals die geschetst is in afbeelding 8.2. We hebben in merk-algoritme 1.2 te maken met de volgende operaties, waarvan de meeste direct te vertalen zijn:

```
S := <W>:
     p, q := verwijzing(W), nil.

S ≠ <>:
     p ≠ nil.

Laat X = top(S):
     Laat X = object(p).

Verlaag S:
     Als q = nil
     | p := nil
     zoniet
     | Laat Y = object(q).
     | Laat T = vertakking(Y, k(Y)).
     | p, q, inhod(T) := q, inhoud(T), p.
```

```
Stapel Y op S:
    p, q, inhoud(T) := inhoud(T), p, q.
```

Hierin is "verwijzing(W)" de verwijzing van de wortel W. Door een eenvoudige substitutie krijgen we aldus de Schorr-Waite merk-algoritme:

```
Merk-algoritme 1.2.1
Merk W.
k(W) := 0.
p, q := verwijzing(W), nil.
Zolang p ≠ nil
| Laat X = object(p).
| Als k(X) = graad(X)
| | Als q = nil
| | | p := nil.
| | zoniet
| | | Laat Y = object(q).
| | | Laat T = vertakking(Y, k(Y)).
| | | p, q, inhoud(T) := q, inhoud(T), p.
| zoniet
| | k(X) := k(X) + 1.
| | Laat T = vertakking(X, k(X)).
| | Laat V = inhoud(T).
| | Laat Y = object(V).
| | Als Y niet gemerkt is
| | | Merk Y.
| | | k(Y) := 0.
| | | p, q, inhoud(T) := inhoud(T), p, q.
```

Deze algoritme (in een aantal variaties) is de laatste tijd min of meer de standaard geworden om methoden om de correctheid van programma's te bewijzen uit te testen [12, 19, 26, 32]. Daarbij wordt meestal zonder meer uitgegaan van de concrete algoritme, hetgeen de begrijpelijkheid niet bevordert. De hier gevolgde methode om uitgaande van een eenvoudige abstracte algoritme stap voor stap de concrete algoritme af te leiden heeft dit bezwaar veel minder. De correctheid van de concrete algoritme volgt daarbij direct uit de correctheid van de abstracte algoritme en de correctheid van de "onderweg" gekozen implementaties. Het bewijs van de

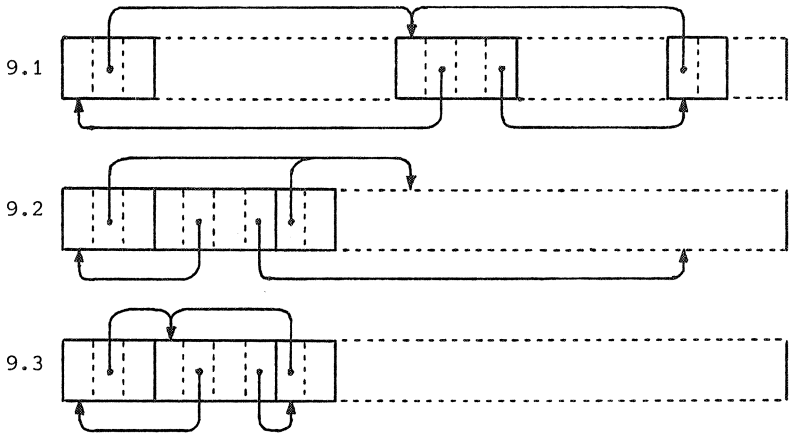
correctheid van de abstracte algoritme en de gekozen implementaties voor de Schorr-Waite algoritme zullen we hier echter niet geven. Het zal de lezer niet moeilijk vallen dit zelf te construeren.

Merk-algoritme 1.2.1 zal door de daarin gebruikte coderings- en decoderings-operaties minder snel zijn dan merk-algoritme 1.2 waarbij S als een "externe" stapel geïmplementeerd wordt. Dit kan enigszins overkomen worden door merk-algoritme 1.2 te gebruiken met een (kleine) eindige stapel en wanneer deze vol mocht raken over te gaan op merk-algoritme 1.2.1 [27]. Merk-algoritme 1.2.1 zal helemaal een stuk langzamer zijn dan merk-algoritme 1.1. De vraag is dan wanneer merk-algoritme 1.2.1 toch de voorkeur verdient boven merk-algoritme 1.1. Wat de benodigde ruimte voor de verschillende soorten informatie betreft bestaat het enige verschil tussen merk-algoritme 1.1 en merk-algoritme 1.2.1 uit het feit dat in eerstgenoemde voor de status-informatie een stapel S benodigd is, terwijl in laatstgenoemde een teller k(X) per object X nodig is. Zoals gezegd kan de stapel S geïmplementeerd worden door in ieder object een extra component te introduceren die een verwijzing kan bevatten. Het introduceren van een teller in een object kost eveneens een extra component, zij het dat deze teller niet verder hoeft te tellen dan het aantal vertakkingen van het object. Als dit aantal klein is (b.v. 2 zoals in LISP) kan dit tellertje vaak in een paar verloren bitjes gecodeerd worden. Als dit aantal groot is of zelfs potentieel oneindig (zoals in ALGOL 68) dan zal echter een volle extra component per object nodig zijn. In de regel kan men dus stellen dat merk-algoritme 1.2.1 alleen de voorkeur verdient boven merk-algoritme 1.1 als snelheid niet voorop staat en het aantal vertakkingen per object dusdanig klein is dat het benodigde tellertje geen of zeer weinig extra ruimte kost.

Alle hier besproken algoritmen zijn afgeleid van merk-algoritme 1. Het is echter ook mogelijk om andere abstracte algoritmen die aan de gestelde beperkingen voldoen als uitgangspunt te kiezen (b.v. een abstracte versie van de algoritme op blz. 556 in [30]), waaruit nog andere merk-algoritmen afgeleid kunnen worden. In de regel zijn deze algoritmen echter minder efficiënt dan die die we hier besproken hebben, reden waarom we er niet verder op in zullen gaan.

3.2. Compactificatie-algoritmen

Het is de taak van de compactificeerder om de objecten in het geheugen van de concrete machine zodanig te verschuiven dat een compact vrij geheugen resulteert. Op het eerste gezicht mag dit een vrij triviale zaak lijken. De reden dat dit niet zo is heeft te maken met de manier waarop verwijzingen in de concrete machine gerepresenteerd worden. Een object wordt zoals gezegd gerepresenteerd door de aan hem toegewezen locatie. Een verwijzing wordt, hiermee in overeenstemming, gerepresenteerd door een "wijzer" naar de locatie van het object. Wijzers zijn dus in feite adressen, mogelijk aangevuld met enige extra informatie. Voor het gemak zullen we hier aannemen dat een wijzer, die een verwijzing naar een object X representeert, gelijk is aan het adres van de eerste cel van de locatie van X. De situatie zoals de compactificeerder die aantreft kan daarom schematisch worden weergegeven zoals in afbeelding 9.1. Verder zullen we aannemen dat de compactificeerder het vrije geheugen in het meest rechtse deel van het geheugen positioneert. Ten einde te voorkomen dat de wortel daarbij verschoven wordt, zullen we aannemen dat deze een vaste locatie in het meest linkse deel van het geheugen heeft.



Afbeelding 9

Zou de compactificeerder nu eenvoudig de objecten zoveel mogelijk naar links verschuiven (de inhoud van locaties daarbij copiërend), dan zou de situatie van afbeelding 9.2 ontstaan. Duidelijk is nu te zien dat de

wijzers "verkeerd staan": ze wijzen nog naar de oude locaties van objecten, terwijl ze uiteraard voor een correct functioneren van het programma naar de nieuwe locaties moeten wijzen. De compactificeerder zal daarom alle wijzers in het geheugen moeten "aanpassen", zodat de situatie van afbeelding 9.3 ontstaat. Het zal duidelijk zijn dat voor het aanpassen van wijzers de compactificeerder enige extra informatie moet bijhouden, die we zullen aanduiden als de "aanpassings-informatie". Het bijhouden van deze informatie zullen we "boekhouden" noemen.

Conceptueel valt het werk van de compactificeerder nu uiteen in drie fasen:

(1) Boekhouden.
In deze fase wordt de aanpassings-informatie opgebouwd.
(2) Aanpassen.
Gebruikmakend van de aanpassings-informatie worden alle wijzers aangepast.
(3) Verschuiven.
Alle objecten worden naar hun nieuwe locaties verschoven.

Dit is zoals gezegd alleen een "conceptuele decompositie". In de praktijk kunnen deze fasen op talloze manieren met elkaar vermengd worden, zoals we straks zullen zien.

Een van de problemen waarmee de compactificeerder te maken heeft is het feit dat door het verschuiven van objecten locaties elkaar niet mogen gaan overlappen. De meest voor de hand liggende manier om dit probleem op te lossen is om objecten in de volgorde van links naar rechts te verschuiven. Dus, eerst wordt het meest linkse object gepakt en zo ver mogelijk naar links verschoven, dan wordt het op één na meest linkse object gepakt en zo ver mogelijk naar links verschoven, etc. Dit is een veilige manier om objecten te verschuiven, die bovendien de plezierige eigenschap heeft dat de objecten in dezelfde volgorde in het geheugen blijven liggen; de methode is zgn. "genetische volgorde behoudend (GVB)" (Eng. "genetic order preserving") [29]. Dit heeft niet alleen voordelen bij het gebruik van virtueel geheugen (vermindert de kans op "thrashing"), maar stelt de implementator tevens in staat essentieel gebruik te maken van het feit dat de objecten in een bepaalde volgorde in het geheugen liggen, zonder bang te

hoeven zijn dat de compactificeerder deze volgorde verstoort. Neemt men bepaalde dingen aan omtrent de grootte van objecten of neemt men aan dat objecten naar een apart (vrij) stuk geheugen verschoven worden, dan zijn nog wel andere zij het niet-GVB manieren denkbaar om objecten op een veilige manier te verschuiven (zie b.v. [3, 5, 10, 14]). Hier zullen we echter aannemen dat objecten in de volgorde van links naar rechts verschoven worden.

De voornaamste extra kosten van een compactificeerder worden veroorzaakt door de aanpassings-informatie. Daarnaast speelt in iedere compactificeerder ook informatie een rol die vergelijkbaar is met de type- en status-informatie in een merk-algoritme. Wat de benodigde type-informatie betreft kan de compactificeerder echter zonder meer gebruik maken van de voor de merk-algoritme benodigde type-informatie. Dit laatste geldt eveneens voor de ruimte die nodig is voor de status-informatie, maar vaak is het zo dat voor deze informatie niet eens meer dan een paar tellertjes nodig is, zodat de eventueel bestaande ruimte voor de status-informatie van de merk-algoritme voor andere doeleinden gebruikt kan worden (b.v. om de aanpassings-informatie in op te slaan). We zullen daarom de compactificatie-algoritmen indelen naar de vorm die de aanpassings-informatie daarin heeft. Dit leidt tot een indeling van de bestaande compactificatie-algoritmen in twee klassen die hieronder besproken zullen worden.

3.2.1. Compactificatie met een representatie-afbeelding

In de eerste compactificatiemethode die we zullen beschouwen wordt de aanpassings-informatie gerepresenteerd door een (variabele) afbeelding R die de oude representatie van een verwijzing afbeeldt op zijn nieuwe representatie. De drie fasen van een compactificeerder die gebruik maakt van de "representatie-afbeelding" (Eng. "relocation map") R zien er als volgt uit:

Compactificatie-algoritme 1

```
Boekhouden:
    t := links.
    Voor ieder object X v.l.n.r.
    |  Laat a = adres(X).
    |  R(a) := t.
    |  t := t + grootte(X).
Aanpassen:
    Voor iedere vertakking T
    |  Laat c = cel(T).
    |  Laat a = inhoud(c).
    |  inhoud(c) := R(a).
Verschuiven:
    t := links.
    Voor ieder object X v.l.n.r.
    |  Verschuif X naar adres t.
    |  t := t + grootte(X).
```

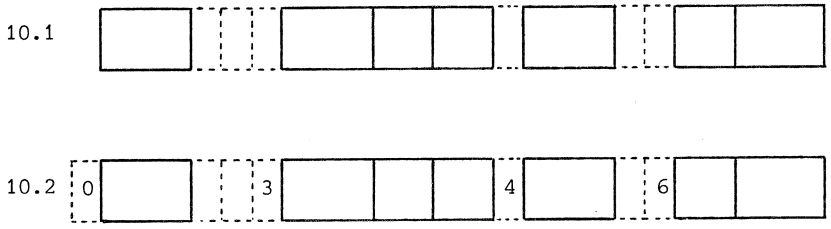
Hierin is "t" een variabele adresteller, "links" het adres van de meest linkse cel in het geheugen en "adres(X)" het adres van de eerste cel van de (oude) locatie van X. Verder is "grootte(X)" het aantal cellen dat de locatie van X in beslag neemt en "cel(T)" de cel waar vertakking T zich bevindt, waarbij we voor het gemak aannemen dat iedere vertakking een cel beslaat (oftewel dat wijzers in precies één woord passen). Te zien is dat in de aanpassingsfase alle vertakkingen van objecten afgelopen worden. Als objecten geen type-informatie bevatten of deze informatie niet statisch bekend is, dan moet dit aflopen in principe op dezelfde manier als in de merk-algoritme gebeurt (via het bijhouden van status- en type-informatie). In alle andere gevallen is het echter mogelijk eenvoudig de objecten van links naar rechts af te lopen (zoals in de andere fasen) en bij aankomst bij een object al zijn vertakkingen af te lopen, hetgeen in de regel veel efficiënter is.

Er zijn verschillende manieren om de representatie-afbeelding R zonder al te veel extra ruimte te implementeren. Een eerste methode (toegepast in b.v. de reeds genoemde "LISP 2 garbage collector") is om in ieder object X met oud adres a een cel te reserveren waarin de waarde van R(a) opgeslagen wordt. Dit hoeft geen extra ruimte te kosten als deze cel in het merkproces

toch gebruikt werd om b.v. status-informatie in op te slaan (en dit tijdens de compactificatie niet nodig is). Voorwaarde bij gebruik van deze methode is uiteraard wel dat een object niet verschoven wordt voordat alle wijzers naar het object aangepast zijn. Hoewel ik het nooit gezien heb, is het mogelijk om bij gebruikmaking van deze methode de aanpassingsfase met zowel de boekhoudfase als de verschuiffase te combineren, daarmee het aantal keren dat het geheugen doorlopen wordt tot twee reducerend.

Een tweede methode om R te implementeren [3, 5, 10, 14], die alleen toepasbaar is wanneer de oude locaties van objecten niet overschreven worden, lijkt enigszins op de vorige. Daarin worden de boekhoud- en verschuiffase met elkaar gecombineerd. Na het verschuiven van een object X met oud adres a wordt de waarde van R(a) in de oude locatie van X gezet. In de afzonderlijke aanpassingsfase worden dan alle wijzers aangepast. Gezien de niet-overschrijvingseis vergt deze methode in feite een dubbel geheugen en is derhalve alleen zinvol in toepassingen met virtueel geheugen.

Bovenstaande twee methoden komen in feite neer op het implementeren van R als een array (nl. het geheugen zelf) en zijn niet altijd zonder gebruik van extra ruimte toepasbaar. Een derde methode om R te implementeren [35] heeft dit laatste bezwaar niet. De grondslag van deze methode is de volgende. Het deel van het geheugen dat bezet is door objecten kan gezien worden als een verzameling compacte "blokken", waartussen zich één of meer vrije cellen bevinden (zie afbeelding 10.1). Het is niet moeilijk in te zien dat iedere wijzer a naar een object dat zich in een bepaald blok bevindt dezelfde "verschuiving" a - R(a) heeft. Hiervan, en van het feit dat zich tussen twee blokken minstens één vrije cel bevindt, kan men gebruik maken om R in deze vrije cellen te coderen. Dit kan door b.v. in de eerste vrije cel links van een blok de unieke met dit blok geassocieerde verschuiving op te slaan (daarbij aannemend dat zich links van de wortel een vrije cel bevindt; zie afbeelding 10.2). Het bepalen van de waarde van R(a) bestaat nu uit het zoeken van de eerste vrije cel links van de cel met adres a en het aftrekken van de inhoud ervan van a. Dit zoeken zorgt ervoor dat compactificatie met behulp van deze methode meestal niet in lineaire tijd gaat. Het zoeken kan versneld worden b.v. door de cellen waarin zich de verschuivingen bevinden in een binaire boom te rangschikken (als daar tenminste voldoende ruimte voor is; zie b.v. [29]). Het combineren van fasen bij deze methode is lastig, zij het niet onmogelijk.



Afbeelding 10

In de vierde en laatste methode [13] die we hier zullen bespreken wordt eveneens van verschuivingen gebruik gemaakt, zij het dat R nu geïmplementeerd wordt als een tabel. Deze tabel bevat paren (a, s), waarin a het adres van de eerste cel van een blok is en s de met dit blok geassocieerde verschuiving. De paren in de tabel zijn op hun eerste element gesorteerd, zodat de waarde van R(a) met behulp van een binair zoekproces gevolgd door een aftrekking bepaald kan worden. Hieruit blijkt al dat compactificatie met deze methode in de regel niet in lineaire tijd gaat. Evenals bij de derde methode staat hier echter tegenover dat, mits een paar (a, s) uit de tabel in een cel past, deze methode geen extra ruimte kost. Het is nl. mogelijk de boekhoudfase en de verschuiffase zodanig te combineren dat de tabel tijdens het verschuiven van objecten in het tussenliggende vrije geheugen wordt opgebouwd. De tabel "rolt" daarbij als het ware door het geheugen. Hoewel dit rollen in principe een lineair proces is [13], zorgt dit er wel voor dat de paren in de tabel niet in volgorde blijven, zodat een extra sorteerproces uitgevoerd moet worden. Dit is een tweede reden waarom deze methode niet in lineaire tijd werkt. Door gebruik te maken van de extra vrije ruimte buiten de tabel is het mogelijk deze "ordeverstoring" te beperken en tevens het zoeken in de tabel te versnellen [11, 34].

3.2.2. Compactificatie met vertakkings-verzamelingen

De tweede compactificatiemethode die we zullen bespreken verschilt volledig van de vorige. Werd in de eerste methode de aanpassings-informatie gerepresenteerd door een representatie-afbeelding, in deze methode wordt de aanpassings-informatie gerepresenteerd door met ieder object X een

(variabele) verzameling B(X) van vertakkingen te associëren, en wel die vertakkingen die een verwijzing naar X bevatten. Aannemende dat aanvankelijk B(X) = ∅ voor ieder object X ziet een compactificeerder die gebruik maakt van deze "vertakkings-verzamelingen" er als volgt uit:

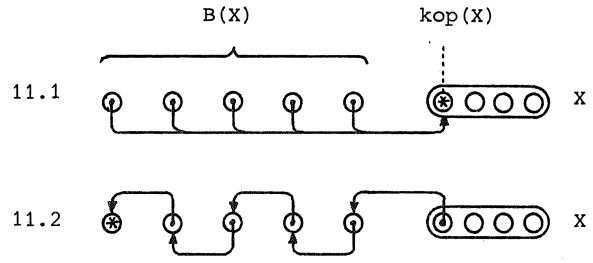
Compactificatie-algoritme 2

```
Boekhouden:
    Voor ieder vertakking T
    | Laat V = inhoud(T).
    | Laat X = object(V).
    | Stop T in B(X).
Aanpassen:
    t := links.
    Voor ieder object X v.l.n.r.
    | Zolang B(X) ≠ ∅
    | | Pak een vertakking T uit B(X).
    | | Laat c = cel(T).
    | | inhoud(c) := t.
    | t := t + grootte(X).
Verschuiven:
    t := links.
    Voor ieder object X v.l.n.r.
    | Verschuif X naar adres t.
    | t := t + grootte(X).
```

Zoals te zien is worden nu in de boekhoudfase alle vertakkingen afgelopen. Voor dit aflopen van de vertakkingen geldt hetzelfde als opgemerkt bij compactificatie-algoritme 1 (waar dit aflopen in de aanpassingsfase gebeurde).

Op het eerste gezicht mag het lijken dat het implementeren van de vertakkings-verzamelingen niet zonder een aanzienlijke hoeveelheid extra ruimte mogelijk is. Toch is dit wel zo. Om dit te demonstreren kiezen we in elk object X uit de aanwezige componenten een willekeurige component, die we zullen aanduiden met "kop(X)". Beschouw nu vlak na de boekhoudfase een object X met zijn bijbehorende vertakkings-verzameling B(X). Dan hebben we te maken met de situatie zoals die is geschetst in afbeelding 11.1. Deze situatie kunnen we zodanig transformeren dat we de vertakkingen in B(X) een

gelinkte lijst laten vormen, waarbij de nieuwe inhoud van kop(X) naar de kop van de lijst verwijst en de oude inhoud van kop(X) als lijstafsluiter fungeert (zie afbeelding 11.2). (Merk op dat we stilzwijgend verwijzingen naar vertakkingen in plaats van objecten geïntroduceerd hebben.) Deze transformatie is voor alle objecten tegelijkertijd en zonder verlies van informatie alleen mogelijk als kop(X) niet zelf een vertakking van X is en de originele inhoud van kop(X) te onderscheiden is van een verwijzing. Laten we voorlopig aannemen dat aan deze voorwaarden voldaan is. Dat is overigens geenszins een irreële veronderstelling. Zoals reeds besproken is het vaak nodig type-informatie op te nemen in objecten. Een daartoe in ieder object X geïntroduceerde component is dan een goede kandidaat voor kop(X).



Afbeelding 11

Afbeelding 11.2 schetst een implementatie van de vertakkingsverzamelingen als gelinkte lijsten zonder dat dit extra ruimte kost. De truc waarop dit gebaseerd is schijnt reeds lang bekend te zijn, hoewel de literatuur hierover van vrij recente datum is [8, 15, 16, 20, 31]. Waar men bij gebruik van deze truc op moet letten is het feit dat in een werkelijke implementatie deze gelinkte lijsten niet bestaan uit vertakkingen (wat immers abstracte dingen zijn) maar uit cellen (nl. de cellen waar deze vertakkingen zich bevinden). Dit houdt in dat een object X nooit verschoven mag worden als een van zijn vertakkingen zich nog in de vertakkingsverzameling B(Y) van een object Y bevindt (met mogelijk X = Y). Herschrijft men compactificatie-algoritme 2 simpelweg met gebruikmaking van bovenstaande truc, dan krijgt men een driefasen-compactificeerder die op triviale wijze aan deze eis voldoet. Wil men echter fasen combineren, dan moet wel degelijk met deze eis rekening gehouden worden. De volgende algoritme combineert de aanpassingsfase met zowel de boekhoudfase als de

verschuiffase en voldoet inderdaad aan deze eis (N.B. aanvankelijk is B(X) = ∅ voor ieder object X):

```
Compactificatie-algoritme 2.1
t := links.
Voor ieder object X v.l.n.r.
|  Zolang B(X) ≠ ∅
|  |  Pak een vertakking T uit B(X).
|  |  Laat c = cel(T).
|  |  inhoud(c) := t.
|  Voor iedere vertakking T van X
|  |  Laat V = inhoud(T).
|  |  Laat Y = object(V).
|  |  Stop T in B(Y).
|  t := t + grootte(X).
t := links.
Voor ieder object X v.l.n.r.
|  Zolang B(X) ≠ ∅
|  |  Pak een vertakking T uit B(X).
|  |  Laat c = cel(T).
|  |  inhoud(c) := t.
|  Verschuif X naar adres t.
|  t := t + grootte(X).
```

Net als bij de Schorr-Waite merk-algoritme kunnen we hieruit nu, gebruikmakend van de hierboven beschreven truc, de verzamelingen B(X) via een eenvoudig substitutieproces "elimineren". Compactificatie is aldus gereduceerd tot een tweefasen-proces dat geen extra ruimte kost. Bovendien gebruikt deze methode een tijd die lineair is in het aantal objecten en vertakkingen. Voor een uitvoeriger discussie van deze algoritme zie [16].

Hierboven werd ervan uitgegaan dat ieder object X een component kop(X) bevat, welke geen vertakking van X is. Het is echter mogelijk de beschreven implementatietruc toe te passen zelfs als kop(X) zelf een vertakking van X is, mits het mogelijk is componenten van objecten individueel te "merken" (waarbij het merkteken dan als lijstafsluiter fungeert). Het probleem daarbij is ervoor te zorgen dat niet tegelijkertijd kop(X) in een of andere B(Y) zit en B(X) niet leeg is. Om dit te bereiken is het combineren van

fasen absoluut noodzakelijk. Een tweefasen-algoritme die hieraan voldoet staat beschreven in [20]. Deze algoritme is echter gecompliceerder dan compactificatie-algoritme 2.1. Zo worden alle vertakkingen daar twee keer in plaats van één keer afgelopen en wordt het geheugen in de twee fasen in verschillende richtingen doorlopen.

Tot slot nog een opmerking over de combinatie van compactificatie-algoritme 2 met een merk-algoritme. Zoals te zien is worden in de eerste fase van compactificatie-algoritme 2 alle vertakkingen afgelopen, iets wat in een merk-algoritme ook al gebeurt. Het is daarom mogelijk de eerste fase van compactificatie-algoritme 2 reeds tijdens het merkproces uit te voeren. Mits daarbij aan de reeds eerder genoemde voorwaarden voldaan is, is de implementatietruc voor de vertakkings-verzamelingen daarbij nog steeds toepasbaar. Een compactificerende saneerder volgens dit principe is voor het eerst beschreven in [31]; zie verder [8, 15]. Voorwaarde bij gebruik van deze methode is natuurlijk wel dat voor het begin van het merkproces bekend is of een compactificatie uitgevoerd moet worden en zo ja welke objecten bij het compactificatieproces betrokken zijn en welke niet.

4. SLOT

Alhoewel in de loop der tijd een groot aantal artikelen over het onderwerp sanering verschenen is en saneerders in een groot aantal implementaties gebruikt worden, ontbreekt het in de literatuur aan een werkelijk systematische behandeling van dit onderwerp. Dit in tegenstelling tot een aantal andere onderdelen van het vak "implementatie van programmeertalen" (zoals het ontleden van programmateksten). Als reden hiervoor zou men kunnen aanvoeren dat het onderwerp dermate triviaal is, dat een dergelijke behandeling overbodig is. De betrouwbaarheid van de gemiddelde saneerder doet echter anders vermoeden. Binnen het beperkte kader van deze lezing is gepoogd duidelijk te maken dat een dergelijke behandeling inderdaad mogelijk is. Een uitvoeriger behandeling van het onderwerp sanering zal worden gegeven in [17]. In tegenstelling tot deze lezing zal daarbij uitgegaan worden van het algemene geval van elkaar willekeurig overlappende objecten.

LITERATUUR

[1] BAKER, H.G., Jr., *List processing in real time on a serial computer,* Communications of the ACM 21 (1978), 280-294.

[2] BARTH, J.M., *Shifting garbage collection overhead to compile time,* Communications of the ACM 20 (1977), 513-518.

[3] BOBROW, D.G., *Storage management in LISP,* in [4].

[4] BOBROW, D.G. (Ed.), *Symbol Manipulation Languages and Techniques,* North-Holland Publishing Co., Amsterdam (1968).

[5] CHENEY, C.J., *A nonrecursive list compacting algorithm,* Communications of the ACM 13 (1970), 677-678.

[6] COLLINS, G.E., *A method for overlapping and erasure of lists,* Communications of the ACM 3 (1960), 655-657.

[7] DEUTSCH, L.P. & D.G. BOBROW, *An efficient, incremental, automatic garbage collector,* Communications of the ACM 19 (1976), 522-526.

[8] DEWAR, R.B.K. & A.P. McCANN, *MACRO SPITBOL - a SNOBOL4 compiler,* Software-Practice and Experience 7 (1977), 95-113.

[9] DIJKSTRA, E.W., L. LAMPORT, A.J. MARTIN, C.S. SCHOLTEN & E.M.F. STEFFENS, *On-the-fly garbage collection: an exercise in cooperation,* Communications of the ACM 21 (1978), 966-975.

[10] FENICHEL, R.R. & J.C. YOCHELSON, *A LISP garbage-collector for virtual-memory computer systems,* Communications of the ACM 12 (1969), 611-612.

[11] FITCH, J.P. & A.C. NORMAN, *A note on compacting garbage collection,* The Computer Journal 21 (1978), 31-34.

[12] GRIES, D., *The Schorr-Waite graph marking algorithm,* Acta Informatica 11 (1979), 223-232.

[13] HADDON, B.K. & W.M. WAITE, *A compaction procedure for variable-length storage elements,* The Computer Journal 10 (1967), 162-165.

[14] HANSEN, W.J., *Compact list representation: definition, garbage collection, and system implementation,* Coomunications of the ACM 12 (1969), 499-507.

[15] HANSON, D.R., *Storage management for an implementation of SNOBOL4,* Software-Practice and Experience 7 (1977), 179-192.

[16] JONKERS, H.B.M., *A fast garbage compaction algorithm,* Information Processing Letters 9 (1979), 26-30.

[17] JONKERS, H.B.M., *A survey of garbage collection,* To appear, Mathematical Centre, Amsterdam.

[18] KNUTH, D.E., *The Art of Computer Programming,* Vol. 1: *Fundamental Algorithms,* Addison-Wesley, Reading, Mass. (1968).

[19] KOWALTOWSKI, T., *Data structures and correctness of programs,* Journal of the ACM 26 (1979), 283-301.

[20] LOCKWOOD MORRIS, F., *A time- and space-efficient garbage compaction algorithm,* Communications of the ACM 21 (1978), 662-665.

[21] McBETH, J.H., *On the reference counter method,* Communications of the ACM 6 (1963), 575.

[22] McCARTHY, J., *Recursive functions of symbolic expressions and their computation by machine,* part I, Communications of the ACM 3 (1960), 184-195.

[23] McCARTHY, J. et al., *LISP 1.5 Programmer's Manual,* MIT Press, Cambridge, Mass. (1965).

[24] MULLER, K.G., *On the feasilibility of concurrent garbage collection,* Ph.D. Thesis, Technical University, Delft, The Netherlands (1975).

[25] RANDELL, B. & L.J. RUSSELL, *ALGOL 60 Implementation,* Academic Press, London (1964).

[26] ROEVER, W.P. DE, *On backtracking and greatest fixpoints,* in: *Formal descriptions of programming concepts,* E.J. Neuhold (Ed.), North-Holland Publishing Co., Amsterdam (1978), 621-639.

[27] SCHORR, H. & W.M. WAITE, *An efficient machine-independent procedure for garbage collection in various list structures,* Communications of the ACM 10 (1967), 501-506.

[28] STEELE, G.L., Jr., *Multiprocessing compactifying garbage collection,* Communications of the ACM 18 (1975), 495-508.

[29] TERASHIMA, M. & E. GOTO, *Genetic order and compactifying garbage collectors,* Information Processing Letters 7 (1978), 27-32.

[30] THORELLI, L., *Marking algorithms,* BIT 12 (1972), 555-568.

[31] THORELLI, L., *A fast compactifying garbage collector,* BIT 16 (1976), 426-441.

[32] TOPOR, R.W., *The correctness of the Schorr-Waite list marking algorithm,* Acta Informatica 11 (1979), 211-221.

[33] WADLER, P.L., *Analysis of an algorithm for real time garbage collection,* Communications of the ACM 19 (1976), 491-500.

[34] WAITE, W.M., *Implementing software for non-numeric applications,* Prentice-Hall, Englewood Cliffs, N.J. (1973).

[35] WEGBREIT, B., *A generalised compactifying garbage collector,* The Computer Journal 15 (1972), 204-208.

[36] WISS, D.S. & D.P. FRIEDMAN, *The one-bit reference count,* BIT 17 (1977), 351-359.

RELATIES TUSSEN TAALDEFINITIE EN TAALIMPLEMENTATIE

C. HEMERIK
Technische Hogeschool, Eindhoven

1. INLEIDING

> "In the relevant literature there exist various examples relating language definition to implementations (...). A comprehensive elaboration of this subject would be of great tutorial and practical value and would in fact complement the existing material on the subject of syntax definition and parsing, which, in contrast, is well understood".
>
> P. Lucas.

Deze syllabus heeft geenszins de pretentie een "comprehensive elaboration of great tutorial and practical value" te zijn; het is wel onze bedoeling aandacht te schenken aan een onderwerp dat in de literatuur over implementatie nogal verwaarloosd wordt, terwijl het in feite de kern van de implementatieproblematiek vormt, namelijk hoe bij een gegeven definitie van een programmeertaal een correcte implementatie van die taal te construeren. Stel dat we een programmeertaal B tot onze beschikking hebben, op al dan niet formele wijze gedefinieerd in een al dan niet leesbaar rapport. Met behulp van die definitie kunnen we ons een beeld vormen van de mogelijke taalconstructies van B en hun betekenis, en vervolgens programma's in B schrijven. Als we die programma's willen laten uitvoeren, zullen we moeten beschikken over een machine M_B, die een B-programma kan accepteren en het door de taaldefinitie van B bepaalde effect van dat programma kan bewerkstelligen. Maar meestal is zo'n machine M_B niet beschikbaar en zullen we ons moeten behelpen met een wel beschikbare machine M_D, die slechts programma's geschreven in de taal D kan uitvoeren, maar die we kunnen gebruiken om M_B te simuleren. Een dergelijke simulatie wordt de implementatie van taal B op machine M_D genoemd, en kan in principe op een groot aantal manieren plaatsvinden. We zullen in deze syllabus alleen implementatie

d.m.v. vertaling beschouwen, d.w.z. dat een programma in taal B eerst vertaald wordt naar een programma in taal D, dat vervolgens uitgevoerd wordt op machine M_D en daarbij hetzelfde effect bewerkstelligt als uitvoering van het oorspronkelijke B-programma op M_B.

Maar wat betekent "hetzelfde effect"? Dat kunnen we alleen uitdrukken in termen van iets gemeenschappelijks, en het enige dat daarvoor in aanmerking komt is het "externe gedrag" van M_B en M_D, zoals zich dat manifesteert in de relatie tussen de door de machines geaccepteerde invoer en geproduceerde uitvoer. In principe is het dan denkbaar dat een in B geschreven programma voor het sorteren van een rij getallen volgens de bubble-sort methode wordt vertaald naar een D-programma dat sorteert volgens de quick-sort methode. In het algemeen is iets dergelijks echter niet realiseerbaar en zal een vertaler zich beperken tot het omzetten van B-constructies naar D-constructies volgens min of meer vaste patronen.

In deze syllabus willen we ingaan op de vraag hoe we kunnen aantonen dat een vertaling van een willekeurig B-programma volgens bepaalde patronen altijd resulteert in een D-programma met hetzelfde effect. Het zal duidelijk zijn dat we in een dergelijke beschouwing tenminste de volgende aspecten moeten betrekken:

1. definitie van B
2. definitie van D
3. specificatie van de afbeelding van B naar D
4. aantonen van de correctheid van die afbeelding.

We willen echter ook de bruikbaarheid voor deze beschouwingen toetsen van twee formele definitiemethoden, te weten denotationele semantiek en axiomatische semantiek, en komen aldus tot de volgende opzet van deze syllabus:

- In hoofdstuk 2 geven we een beschouwing over definitiemethoden van programmeertalen en een motivatie voor onze keuze van denotationele en axiomatische semantiek.
- In hoofdstuk 3 presenteren we ten dele informele beschrijvingen van een taal B met een ALGOL-achtige structuur, van een taal D die overeenkomt met de instructies van een stapelmachine, en van een vertaling C van B naar D.
- In hoofdstuk 4 geven we denotationele definities van B en D, en vervolgens een correctheidsbewijs van C, gebaseerd op deze denotationele

definities.

- In hoofdstuk 5 geven we axiomatische definities van B en D, en vervolgens een correctheidsbewijs van C, gebaseerd op deze axiomatische definities.
- In hoofdstuk 6 geven we enige conclusies aangaande het gepresenteerde.
- In hoofdstuk 7 geven we een overzicht van literatuur, die op dit gebied beschikbaar is.

2. OVER TAALDEFINITIES

Het zal duidelijk zijn dat de mogelijkheid tot het geven van een correctheidsbewijs voor een vertaler ten nauwste samenhangt met de manier waarop brontaal en doeltaal zijn gedefinieerd. Daarom is het misschien goed even stil te staan bij het onderwerp taaldefinitie. Onze opvattingen over programmeren hebben in de loop der jaren een ontwikkeling doorgemaakt, die treffend gekarakteriseerd wordt door het volgende citaat:

> "It used to be the program's purpose to instruct our computers; it became the computer's purpose to execute our programs".
>
> E.W. Dijkstra (1976)

In samenhang met deze ontwikkeling is men ook programmeertalen steeds minder gaan beschouwen als hulpmiddel bij het instrueren van computers, en steeds meer als wiskundige objecten, die kunnen dienen als expressiemiddel voor algoritmen. Een dergelijke benadering vereist echter een formele definitie, en er is de afgelopen vijftien jaar dan ook veel onderzoek verricht naar formele definitiemethoden. Een tijd lang is de situatie op dit gebied nogal onoverzichtelijk geweest (in [9] wordt zelfs gesproken over een Toren van Metababel), maar de laatste tijd is het zicht dusdanig opgeklaard, dat we duidelijk drie stromingen kunnen herkennen:

1. Operationele semantiek.
   Operationele methoden maken gebruik van een min of meer abstract machinemodel en definiëren de betekenis van taalconstructies d.m.v. hun effect op de toestanden waarin dat model kan verkeren. De betekenis van een programma wordt dan gegeven door de rij van toestanden die doorlopen wordt bij de uitvoering van dat programma. Bekende voorbeelden van operationele definities zijn de Euler definitie, de Vienna

Definition Language en SEMANOL.

2. Denotationele semantiek.
In de denotationele semantiek wordt een verdere abstractie doorgevoerd door niet meer de afzonderlijk doorlopen toestanden te beschouwen, maar een programma te beschouwen als een functie van toestanden naar toestanden, die gedefinieerd is als de compositie van functies die corresponderen met de deelconstructies van het programma. De denotationele methode wordt in de eerste plaats geassocieerd met de namen Scott en Strachey, maar vormt ook de basis van de Vienna Development Method.

3. Axiomatische semantiek.
In de axiomatische semantiek komt het begrip toestand helemaal niet meer voor, maar gebruikt men als objecten de formules van een of ander logisch systeem, en definieert men de taalconstructies d.m.v. de relaties die zij tussen dergelijke formules bewerkstelligen. De bekendste methodes zijn hier de "inductive assertions" van Floyd, de methode van Hoare en de "weakest preconditions" van Dijkstra.

Operationele definities worden doorgaans gerechtvaardigd met de uitspraak, dat ze het meest tegemoet komen aan de problemen van de implementator. Het is echter de vraag of de implementator gediend is met een gedetailleerde beschrijving van het effect van taalconstructies op een abstract machinemodel. Dergelijke beschrijvingen worden altijd beïnvloed door de eigenschappen van het model, waardoor het moeilijk wordt essenties en bijzaken van elkaar te scheiden en de weg naar een totaal andere implementatie vaak afgesneden wordt. De taalontwerper mag de implementator best suggesties doen wat betreft een mogelijke implementatie, maar dergelijke suggesties horen niet thuis in een taal*definitie*, die, zoals opgemerkt in [5], een "contract" behoort te zijn tussen ontwerper en implementator. In de rest van deze syllabus zullen we daarom operationele definities buiten beschouwing laten, en bestuderen hoe denotationele en axiomatische definities de implementator in staat stellen aan zijn contractuele verplichtingen te voldoen.

3. BESCHRIJVING VAN B, D EN C

In deze paragraaf geven we een beschrijving van de talen B en D, die als brontaal en doeltaal van de eveneens hier te beschrijven vertaler C fungeren. Ten einde het formaat van alle nog volgende beschrijvingen binnen de perken te houden zijn B en D heel klein gekozen, maar toch met voldoende "features" om een indruk te krijgen van de problemen die het bewijzen van de correctheid van vertalers met zich brengt. Hoewel we in hoofdstuk 1 gezegd hebben dat de relatie tussen invoer en uitvoer de basis zou vormen voor onze correctheidsbeschouwingen, zullen we in B en D wel output statements opnemen, maar geen input statements. Deze beperking is niet wezenlijk maar dient alleen om te voorkomen dat de I/O een onevenredig groot deel van de aandacht opeist, ten koste van interessantere zaken.

3.1. De brontaal B

De structuur van B wordt beschreven door de volgende produktieregels:

```
S : print E
  | V<i> := E
  | if R then S else S fi
  | while R do S od
  | S ; S

R : E ≥ E | ...

E : V<i> | C<v> | E + E | ...
```

Hierin staan de nonterminals S, R en E respectievelijk voor statements, relaties en expressies, de terminal V<i> voor een variabele met naam i en de terminal C<v> voor een constante met waarde v.

De alternatieven S ; S en E + E leiden wel tot syntactische ambiguïteit maar niet tot semantische ambiguïteit zoals later zal blijken. De gebruikte constructies zijn verder zo bekend, dat ze hier geen nadere toelichting behoeven. Hun semantiek zal formeel gedefinieerd worden in secties 4.2 en 5.2.

3.2. De doeltaal D

De structuur van programma's in D wordt beschreven door de volgende productieregels:

```
P : [ R ]

R : ε | ℓ<i> : R | I ; R

I : P
  | LDV a       { plaats inhoud van geheugenplaats a op stapel }
  | LDC c       { plaats constante c op stapel }
  | STO a       { berg bovenste waarde van stapel op op geheugenplaats a }
  | FJP ℓ       { spring naar label ℓ als bovenste waarde van stapel
                  false is }
  | UJP ℓ       { spring naar label ℓ }
  | ADD         { vervang bovenste twee stapelelementen door hun som }
  | GEQ         { vervang bovenste twee stapelelementen door het re-
                  sultaat van hun vergelijking d.m.v. ≥ }
  | OUT         { stuur de bovenste waarde van de stapel naar de output-
                  file }
```

Een programma is een mogelijk lege rij van instructies en labels, omsloten door rechte haken. Een instructie is zelf weer een programma, of een primitieve instructie. De betekenis van de primitieve instructies wordt aangegeven door de commentaren in de productieregels. De rechte haken om een rij instructies dienen ter afbakening van de scopes van labels. Binnen de haken geplaatste labels zijn buiten de haken niet zichtbaar, en omgekeerd evenmin. In plaats van haken te gebruiken zouden we ook alle labels in het programma verschillend kunnen kiezen, maar de gekozen vorm vereenvoudigt de beschrijving en benadrukt de lokaliteit van sprongen in doelprogramma's. De semantiek van D-programma's zal formeel gedefinieerd worden in secties 4.3 en 5.3.

3.3. De vertaler C

Bij de beschrijving van de vertaler zullen we aannemen dat we de beschikking hebben over een functie

$$\alpha : \mathrm{Id} \xrightarrow{1\text{-}1} \mathrm{Addr}$$

die namen van variabelen afbeeldt op adressen. Meestal wordt een dergelijke afbeelding geconstrueerd aan de hand van de declaraties in het bronprogramma, maar in ons geval zou dat teveel afleiden van de essenties. Goed beschouwd is een vertaler nu een functie

$$C : [Id \to Addr] \to [B \to D]$$

maar omdat α door ons verhaal heen onveranderd blijft, zullen we C verder beschouwen als

$$C : B \to D$$

waarbij C gedefinieerd wordt door de volgende regels:

(3.3.1) $C(\underline{\text{print}}\ E) = C(E)\ ;\ OUT$

(3.3.2) $C(V\langle i\rangle := E) = C(E)\ ;\ STO\ \alpha(i)$

(3.3.3) $C(\underline{\text{if}}\ R\ \underline{\text{then}}\ S_1\ \underline{\text{else}}\ S_2\ \underline{\text{fi}}) = [\ C(R)\ ;\ FJP\ \ell_1\ ;\ C(S_1)\ ;\ UJP\ \ell_2\ ;\ \ell_1 :\ C(S_2)\ ;\ \ell_2 :\ ]$

(3.3.4) $C(\underline{\text{while}}\ R\ \underline{\text{do}}\ S\ \underline{\text{od}}) = [\ell_1 :\ C(R)\ ;\ FJP\ \ell_2\ ;\ C(S)\ ;\ UJP\ \ell_1\ ;\ \ell_2 :\ ]$

(3.3.5) $C(S_1\ ;\ S_2) = C(S_1)\ ;\ C(S_2)$

(3.3.6) $C(C\langle v\rangle) = LDC\ v$

(3.3.7) $C(V\langle i\rangle) = LDV\ \alpha(i)$

(3.3.8) $C(E_1 + E_2) = C(E_1)\ ;\ C(E_2)\ ;\ ADD$

(3.3.9) $C(E_1 \geq E_2) = C(E_1)\ ;\ C(E_2)\ ;\ GEQ$

4. IMPLEMENTATIECORRECTHEID IN TERMEN VAN DENOTATIONELE SEMANTIEK

4.1. Inleiding

In de denotationele semantiek wordt een semantische interpretatie van een formele taal gegeven door het definiëren van afbeeldingen van de syntactische constructies van de taal naar hun "betekenis" in een geschikt model. Dat model is gebaseerd op volgens bepaalde regels geconstrueerde domeinen en op functies die op die domeinen gedefinieerd zijn en aan zekere continuïteitseisen voldoen. Dankzij de zo in functies en domeinen aangebrachte structuur is het mogelijk algemene bewijsregels voor dergelijke modellen op te stellen, waarbij inductieve methodes een belangrijke rol spelen. De methode is voortgekomen uit werk van C. Strachey op het gebied van taaldefinitie d.m.v. lambda-expressies, en van een theoretische onderbouw voorzien door D. Scott. Het zou te ver voeren om hier dieper in te gaan op de theorie van de denotationele semantiek. We verwijzen hiervoor naar een inleidend artikel van R.D. Tennent [21], dat op zijn beurt een uitgebreide bibliografie bevat.

In dit hoofdstuk zullen we laten zien hoe we op basis van denotationele definities van de brontaal B en de doeltaal D een correctheidsbewijs van de vertaler C kunnen construeren. In secties 4.2 en 4.3 geven we denotationele definities van B resp. D; in sectie 4.4 stellen we de criteria op waaraan een vertaler van B naar D moet voldoen, en in sectie 4.5 geven we tenslotte het bewijs, dat C aan deze criteria voldoet.

4.2. Denotationele definitie van B

Bij de definitie van B maken we gebruik van de volgende domeinen:

Val	domein van waarden
Id	domein van variabelen
File = Val^*	de elementen van File zijn rijen waarden
Store = Id → Val	de elementen van Store zijn afbeeldingen van Id → Val
State = Store × File	de elementen van state zijn paren (σ,f), met $\sigma \in$ Store en $f \in$ File

Vervolgens introduceren we de functies die de syntactische constructies van B afbeelden op hun respectieve betekenissen:

$$\mathcal{P} : S \to File$$
$$\mathcal{M} : S \to [State \to State]$$
$$\mathcal{V} : E \to [Store \to Val]$$
$$\mathcal{T} : R \to [Store \to \{true, false\}]$$

Met behulp van deze functies kunnen we de betekenis van de syntactische constructies als volgt definiëren:

(4.2.1) $\mathcal{P}(S) = (\mathcal{M}(S)(\perp,\varepsilon))[2]$

(4.2.2) $\mathcal{M}(\underline{\text{print}}\ E)(\sigma,f) := (\sigma, f.\mathcal{V}(E)\sigma)$

(4.2.3) $\mathcal{M}(V\langle i\rangle := E)(\sigma,f) = (\sigma\{\mathcal{V}(E)\sigma/i\}, f)$

(4.2.4) $\mathcal{M}(\underline{\text{if}}\ R\ \underline{\text{then}}\ S_1\ \underline{\text{else}}\ S_2\ \underline{\text{fi}})(\sigma,f) = \underline{\text{if}}\ \mathcal{T}(R)\sigma\ \underline{\text{then}}\ \mathcal{M}(S_1)(\sigma,f)\ \underline{\text{else}}\ \mathcal{M}(S_2)(\sigma,f)$

(4.2.5) $\mathcal{M}(\underline{\text{while}}\ R\ \underline{\text{do}}\ S\ \underline{\text{od}})(\sigma,f) = \underline{\text{if}}\ \mathcal{T}(R)\sigma\ \underline{\text{then}}\ (\mathcal{M}(\underline{\text{while}}\ R\ \underline{\text{do}}\ S\ \underline{\text{od}}) \circ \mathcal{M}(S))(\sigma,f)\ \underline{\text{else}}\ (\sigma,f)$

(4.2.6) $\mathcal{M}(S_1\ ;\ S_2) = \mathcal{M}(S_2) \circ \mathcal{M}(S_1)$

(4.2.7) $\mathcal{V}(C\langle v\rangle)\sigma = v$

(4.2.8) $\mathcal{V}(V\langle i\rangle)\sigma = \sigma(i)$

(4.2.9) $\mathcal{V}(E_1 + E_2)\sigma = \mathcal{V}(E_1)\sigma + \mathcal{V}(E_2)\sigma$

(4.2.10) $\mathcal{T}(E_1 \geq E_2)\sigma = \mathcal{V}(E_1)\sigma \geq \mathcal{V}(E_2)\sigma$

De opzet van deze definitie is betrekkelijk standaard. Toestanden worden voorgesteld als paren (σ,f), waarbij σ een afbeelding is van variabelen naar waarden en f een rij van waarden. De denotatie van een statement S is dan een afbeelding van toestanden naar toestanden, en de denotatie van een expressie E is een afbeelding van stores naar waarden. De denotatie $\mathcal{P}(S)$ van een programma S is dan de tweede component van de toestand die

verkregen wordt door $M(S)$ toe te passen op een initiële toestand, bestaande uit de overal ongedefinieerde store en de lege file. De notatie f.v betekent concatenatie van de waarde v aan de file f en de notatie $\sigma\{v/i\}$ wordt gedefinieerd door: $\sigma\{v/i\}(j) =$ if i = j then v else $\sigma(j)$.

4.3. Denotationele definitie van D

De denotationele definitie van D biedt wat meer problemen. In het algemeen zijn denotationele definities gebaseerd op het principe, dat de betekenis van een samengestelde constructie is samengesteld uit de betekenissen van de delen, bijv. $J(i_1;i_2) = J(i_2) \circ J(i_1)$. Deze regel gaat echter niet op indien i_1 zelf een volgende uit te voeren actie specificeert, zoals bijv. bij een sprongopdracht het geval is. De gebruikelijke oplossing voor dit probleem is die met behulp van *continuations* en bestaat ruwweg hieruit, dat de "normale" voortzetting $J(i_2)$ als parameter aan $J(i_1)$ wordt meegegeven zodat binnen $J(i_1)$ de mogelijkheid geschapen wordt, die normale voortzetting te negeren en een andere voortzetting toe te passen. We zullen deze methode toepassen bij de definitie van D.

We gaan uit van twee primitieve domeinen, nl.:

Addr	domein van adressen
Val	domein van waarden (net als bij B)

Voor het karakteriseren van de toestand introduceren we de volgende domeinen:

File = Val^*	domein van filewaarden (net als bij B)
Mem = Addr → Val	domein van geheugens
Stack = Val^*	domein van stapels (een stapel is een rij waarden)
Status = Mem × Stack × File	domein van machinetoestanden.

We zullen nu om te beginnen aan ieder label een betekenis toevoegen. De betekenis van een label is het element van Status → Status dat overeenkomt met de uitvoering van het programma, beginnende bij dat label. Vandaar de volgende domeinen:

Continuation = Status → Status
Labelenv = Label → Continuation

Voorts voeren we de volgende twee functies in:

$$L : P \rightarrow \text{Labelenv}$$
$$J : R \rightarrow [\text{Labelenv} \times \text{Continuation} \rightarrow \text{Continuation}]$$

L associeert met ieder programma een label-environment, die de betekenis van ieder in dat programma voorkomend label geeft. Die label-environment wordt bepaald door uit te gaan van de overal ongedefinieerde label-environment, en aan ieder in het programma voorkomend label een waarde toe te kennen, zijnde de betekenis van de op dat label volgende instructiereeks.

J associeert met iedere instructiereeks een functie, die als parameters een label-environment en een continuation heeft. De label-environment geeft de betekenis van alle labels van het programma waarin de instructiereeks voorkomt, en de continuation is de toestandstransformatie die overeenkomt met de op de instructiereeks volgende actie.

We geven nu de formele definitie van L en J:

(4.3.1) $L(p) =$

$\underline{\text{let}}\ [\ell_1 : s_1\ ;\ \ell_2 : s_2\ ;\ \ldots\ ;\ \ell_n : s_n\ ;\ \ell_{n+1} :] = p$

$\underline{\text{in}}$

$$\perp\ \{J(s_1)(L(p),\ L(p)(\ell_2))/\ell_1\}$$
$$\{J(s_2)(L(p),\ L(p)(\ell_3))/\ell_2\}$$
$$\vdots$$
$$\{J(s_n)(L(p),\ L(p)(\ell_{n+1}))/\ell_n\}$$
$$\{I/\ell_{n+1}\}$$

waarin $I = \lambda c.c$ de identieke functie op continuation is.

(4.3.2) $J([r])(e,c) = c \circ J(r)(L([r]),I)$

(4.3.3) $J(\ell : r)(e,c) = J(r)(e,c)$

(4.3.4) $J(i\ ;\ r)(e,c) = J(i)(e,J(r)(e,c))$

(4.3.5) $J(\text{LDC v})(e,c)(m,s,f) = c(m,v.s,f)$

(4.3.6) $J(\text{LDV a})(e,c)(m,s,f) = c(m,\dot{m}(a).s,f)$

(4.3.7) $J(\text{STO a})(e,c)(m,v.s,f) = c(m\{v/a\},s,f)$

(4.3.8) $J(\text{OUT})(e,c)(m,v.s,f) = c(m,s,f.v)$

(4.3.9) $J(\text{UJP } \ell)(e,c) = e(\ell)$

(4.3.10) $J(\text{FJP } \ell)(e,c)(m,v.s,f) = \underline{\text{if}}\ v\ \underline{\text{then}}\ c(m,s,f)\ \underline{\text{else}}\ e(\ell)(m,s,f)$

(4.3.11) $J(\text{ADD})(e,c)(m,v_1.v_2.s,f) = c(m,(v_2+v_1).s,f)$

(4.3.12) $J(\text{GEQ})(e,c)(m,v_1.v_2.s,f) = c(m,(v_2 \geq v_1).s,f)$

We merken bij de definitie van J het volgende op:

1. $J([r])$ met een willekeurige label-environment e wordt gedefinieerd in termen van $J(r)$ met als label-environment $L([r])$, waaruit blijkt dat niet-lokale labels geen invloed hebben op $J(r)$.
2. In regel (4.3.4) zien we hoe de normale voortzetting $J(r)(e,c)$ van $J(i)$ als parameter aan $J(i)$ wordt meegegeven.
3. Bij een gewone instructie, bijv. in regel (4.3.5), wordt na een toestandswijziging de normale voortzetting c toegepast, maar bij een sprong ((4.3.8), (4.3.9)) wordt c genegeerd en in plaats daarvan de bij ℓ horende toestandstransformatie toegepast, die gevonden wordt door de label-environment e toe te passen op ℓ.

Tenslotte introduceren we nog een functie R, die de betekenis van een volledig D-programma geeft:

$$R : P \to \text{File}$$

gedefinieerd door:

(4.3.13) $R(p) = (J(p)(\perp,I)(\perp,\varepsilon,\varepsilon))[3].$

4.4. Correctheidscriteria voor C

Nu we de denotationele definities van B en D gegeven hebben, zijn we in staat de eigenschappen te formuleren waaraan C moet voldoen. We hebben in hoofdstuk 1 reeds gezegd dat ons enige criterium voor de correctheid van C is, dat een B-programma b en zijn vertaling $C(b)$ bij uitvoering hetzelfde effect hebben, maar het zal duidelijk zijn dat we dat in het algemeen niet kunnen bewijzen zonder gebruik te maken van zekere correspondenties tussen de door b en $C(b)$ teweeggebrachte toestandsveranderingen, en van de eigenschappen van vertalingen van deelconstructies. We voeren daartoe de volgende predikaten in, met een vooralsnog informele betekenis:

cvp(S) : "programma S is correct vertaald"
cvs(S) : "statement S is correct vertaald"
cve(E) : "expressie E is correct vertaald"
cvr(R) : "relatie R is correct vertaald"

cvp(S) laat zich eenvoudig formuleren als het feit dat S en $C(S)$ hetzelfde effect hebben, in casu dat ze gelijke files produceren:

(4.4.1) $\quad cvp(S) \stackrel{def}{=} P(S) = R(C(S))$.

De formulering van cvs(S) is iets ingewikkelder, omdat die betrekking heeft op toestandsveranderingen. We zullen daarom gebruik moeten maken van correspondenties tussen toestanden van het B-programma en het D-programma, en eisen dat die correspondenties door uitvoering van S resp. $C(S)$ niet verstoord worden. We zullen daarom eerst een definitie van correspondentie moeten geven. Aangezien C m.b.v. α variabelen 1-1 duidig op geheugenplaatsen heeft afgebeeld, is een voor de hand liggende formulering, dat variabelen en daarmee corresponderende geheugenplaatsen dezelfde waarden moeten hebben, en voorts dat de files gelijke waarden moeten hebben. In formele notatie:

(4.4.2) $\quad (\sigma,f_1) \overset{\alpha}{\approx} (m,s,f_2) \stackrel{def}{=} \sigma = m \circ \alpha \wedge f_1 = f_2$.

Met behulp van de correspondentie $\overset{\alpha}{\approx}$ kunnen we cvs(S) nu als volgt formuleren:

(4.4.3) $\mathrm{cvs}(S) \stackrel{\mathrm{def}}{=}$

$$s_B \overset{\alpha}{\approx} s_D \Rightarrow M(S)s_B \overset{\alpha}{\approx} J(C(S))(e,I)s_D.$$

Bij cve(E) liggen de zaken weer anders, omdat evaluatie van de expressie in het B-programma niet leidt tot toestandverandering, maar in het D-programma wel. Het is namelijk de bedoeling dat door uitvoering van de code $C(E)$ de waarde van de expressie op de stapel geplaatst wordt, hetgeen we kunnen formuleren als:

(4.4.4) $\mathrm{cve}(E) \stackrel{\mathrm{def}}{=}$

$$\sigma = m \circ \alpha \Rightarrow J(C(E))(e,I)(m,s,f) = (m,V(E)\sigma.s,f).$$

De formulering van cvr(R) wordt op analoge wijze gegeven door:

(4.4.5) $\mathrm{cvr}(R) \stackrel{\mathrm{def}}{=}$

$$\sigma = m \circ \alpha \Rightarrow J(C(R))(e,I)(m,s,f) = (m,T(R)\sigma.s,f).$$

Tenslotte is er nog een belangrijke eis waaraan C moet voldoen, namelijk dat de voor een constructie b gegenereerde code $C(b)$ geconcateneerd mag worden met andere code, zonder dat daardoor de betekenis van $C(b)$ wordt gewijzigd. Dit zou kunnen gebeuren indien $C(b)$ "losse" labels of sprongin-structies bevat, en zou in strijd zijn met het compositiebeginsel. We kunnen de gewenste regulariteit van een stuk D-code formuleren op de volgende wijze:

(4.4.6) $\mathrm{reg}(d) \stackrel{\mathrm{def}}{=} J(d)(e,c) = c \circ J(d)(e,I)$

voor alle $e \in$ Labelenv, $c \in$ Continuation.

4.5. Bewijs van de correctheid van C

In deze paragraaf zullen we bewijzen dat C inderdaad ieder B-programma correct vertaalt, ofwel: $\forall S$: cvp(S). We zullen dit aantonen d.m.v. inductie naar de syntactische structuur van B-programma's, door te bewijzen dat de primitieve constructies correct vertaald worden, en dat uit de correcte vertaling van de deelconstructies van een constructie de correctheid van de vertaling van de constructie volgt. In dit inductiebewijs zullen we tevens de regulariteit van iedere constructie aantonen, omdat die essentieël is voor volgende inductiestappen. Maar allereerst geven we drie lemmata,

die het rekenwerk in de overige bewijzen enigszins verkorten.

LEMMA 4.5.1. *Indien* J(i)(e,c)(m,s,f) = c(F(m,s,f)), *waarbij* F *een functie van* m,s *en* f *is, waarin* c *niet voorkomt, dan is* i *regulier.*

BEWIJS.

(1)	J(i)(e,c)(m,s,f) = c(F(m,s,f))	{gegeven }
(2)	J(i)(e,I)(m,s,f) = F(m,s,f)	{(1) met c = I}
(3)	J(i)(e,c)(m,s,f) = c(J(i)(e,I)(m,s,f))	{(1) en (2) }
(4)	J(i)(e,c) = c ∘ J(i)(e,I)	{(3), (4.4.6) }

q.e.d.

LEMMA 4.5.2. *De instructies* [r], LDC, LDV, STO, OUT, ADD *en* GEQ *zijn regulier.*

BEWIJS. Voor [r] onmiddellijk uit (4.3.2) voor de overige instructies onmiddellijk uit hun definitie in sectie 4.3 en Lemma 4.5.1.

LEMMA 4.5.3. reg(i) ⇒ J(i ; r)(e,c) = J(r)(e,c) ∘ J(i)(e,I).

BEWIJS.

(1)	J(i ; r)(e,c) = J(i)(e,J(r)(e,c))	{(4.3.4)}
(2)	reg(i) ⇒ J(i)(e,J(r)(e,c)) = J(r)(e,c) ∘ J(i)(e,I)	{(4.4.6)}.

Lemma 4.5.3 lijkt misschien wat onnozel, maar komt toch goed van pas doordat het ons een eenvoudiger rekenregel geeft in het geval dat i regulier is.

We zijn nu gereed voor het inductieve bewijs van de correctheid van C.

Geval E : C<v>

Gegeven : σ = m ∘ α

Te bewijzen: cve(E), reg(C(E))

Bewijs:

	J(C(C<v>))(e,c)(m,s,f)	
=	J(LDC v)(e,c)(m,s,f)	{(3.3.6)}
=	c(m,v.s,f)	{(4.3.5)}
=	c(m,V(C<v>)σ.s,f)	{(4.2.7)}

Uit c = I volgt cve(E); uit Lemma 4.5.1 volgt reg(C(E)).

Geval E : V<i>

Gegeven : $\sigma = m \circ \alpha$

Te bewijzen: cve(E), reg($\mathcal{C}$(E))

Bewijs:

$$
\begin{aligned}
& \mathcal{J}(\mathcal{C}(V\langle i\rangle))(e,c)(m,s,f) \\
&= \mathcal{J}(\text{LDV } \alpha(i))(e,c)(m,s,f) && \{(3.3.7)\} \\
&= c(m,m(\alpha(i)).s,f) && \{(4.3.6)\} \\
&= c(m,\sigma(i).s,f) && \{\text{gegeven}\} \\
&= c(m,\mathcal{V}(V\langle i\rangle)\sigma.s,f) && \{(4.2.8)\}
\end{aligned}
$$

Uit c = I volgt cve(E); uit lemma 4.5.1 volgt reg($\mathcal{C}$(E)).

Geval E : $E_1 + E_2$

Gegeven : $\sigma = m \circ \alpha$, cve(E_1), cve(E_2), reg($\mathcal{C}(E_1)$), reg($\mathcal{C}(E_2)$)

Te bewijzen: cve(E), reg(E)

Bewijs:

$$
\begin{aligned}
& \mathcal{J}(\mathcal{C}(E_1+E_2))(e,c)(m,s,f) \\
&= \mathcal{J}(\mathcal{C}(E_1) \ ; \ \mathcal{C}(E_2) \ ; \ \text{ADD})(e,c)(m,s,f) && \{(3.3.8)\} \\
&= (c \circ \mathcal{J}(\text{ADD})(e,I) \circ \mathcal{C}(E_2)(e,I) \circ \mathcal{C}(E_1)(e,I))(m,s,f) && \{\text{reg}(\mathcal{C}(E_1)), \\
& && \ \text{reg}(\mathcal{C}(E_2)), \\
& && \ \text{reg(ADD)}, \\
& && \ \text{lemma 4.5.3}\} \\
&= (c \circ \mathcal{J}(\text{ADD})(e,I) \circ \mathcal{C}(E_2)(e,I))(m,\mathcal{V}(E_1)\sigma.s,f) && \{\text{cve}(E_1)\} \\
&= (c \circ \mathcal{J}(\text{ADD})(e,I))(m,\mathcal{V}(E_2)\sigma.\mathcal{V}(E_1)\sigma.s,f) && \{\text{cve}(E_2)\} \\
&= c(m,\mathcal{V}(E_1+E_2)\sigma.s,f) && \{(4.3.11)\}
\end{aligned}
$$

Uit c = I volgt cve(E); uit lemma 4.5.1 volgt reg($\mathcal{C}$(E)).

Geval R : $E_1 \geq E_2$

Bewijs volledig analoog aan dat van geval E : $E_1 + E_2$.

Geval S : V<i> := E

Gegeven : $(\sigma,f_1) \overset{\alpha}{\simeq} (m,s,f_2)$, cve(E), reg($\mathcal{C}$(E))

Te bewijzen: cve(V<i> := E), reg($\mathcal{C}$(V<i> := E))

Bewijs:

(1) $M(\text{V<i> := E})(\sigma,f_1) = (\sigma\{V(E)\sigma/i\},f_1)$ {(4.2.3)}

(2) $J(C(\text{V<i> := E}))(e,c)(m,s,f_2)$

$= J(C(E)\ ;\ \text{STO}\ \alpha(i))(e,c)(m,s,f_2)$ {(3.3.2)}

$= (c \circ J(\text{STO}\ \alpha(i))(e,I) \circ J(C(E))(e,I))(m,s,f_2)$ {reg(STO), reg(C(E))}

$= (c \circ J(\text{STO}\ \alpha(i))(e,I))(m,V(E)\sigma.s,f_2)$ {cve(E)}

$= c(m\{V(E)\sigma/\alpha(i)\},s,f_2)$ {(4.3.7)}

(3) $(\sigma,f_1) \overset{\alpha}{\sim} (m,s,f_2)$

$\Rightarrow$ $\sigma = m \circ \alpha$

$\Rightarrow$ $\sigma\{V(E)\sigma/i\} = m\{V(E)\sigma/\alpha(i)\} \circ \alpha$

$(\sigma\{V(E)\sigma/i\},f_1) \overset{\alpha}{\sim} (m\{V(E)\sigma/\alpha(i)\},s,f_2)$

Uit (1), (2), (3), c = I volgt cvs(S); uit (2) en lemma 4.5.1 volgt reg(C(S)).

Geval S : print E

Gegeven : $(\sigma,f_1) \overset{\alpha}{\sim} (m,s,f_2)$, cve(E), reg(C(E))

Te bewijzen: cvs(S), reg(C(S))

Bewijs:

(1) $M(\underline{\text{print}}\ E)(\sigma,f_1) = (\sigma,f_1.V(E)\sigma)$ {(4.2.2)}

(2) $J(C(\underline{\text{print}}\ E))(e,c)(m,s,f_2)$

$= J(C(E)\ ;\ \text{OUT})(e,c)(m,s,f_2)$ {(3.3.1)}

$= (c \circ J(\text{OUT})(e,I) \circ J(C(E))(e,I))(m,s,f_2)$ {reg(C(E)), reg(OUT)}

$= (c \circ J(\text{OUT})(e,I))(m,V(E)\sigma.s,f_2)$ {cve(E)}

$= c(m,s,f_2.V(E)\sigma)$ {(4.3.8)}

(3) $(\sigma,f_1.V(E)\sigma) \overset{\alpha}{\sim} (m,s,f_2.V(E)\sigma)$ {volgt uit gegeven}

Uit (1), (2), (3), c = I volgt cvs(S); uit (2) en lemma 4.5.1 volgt reg(C(S)).

Geval S : S_1 ; S_2

Gegeven : $(\sigma,f_1) \overset{\alpha}{\sim} (m,s,f_2)$, $cvs(S_1)$, $cvs(S_2)$, $reg(C(S_1))$, $reg(C(S_2))$

Te bewijzen: cvs(S), reg(C(S))

Bewijs:

(1) $M(S_1 \ ; \ S_2) = M(S_2) \circ M(S_1)$ {(4.2.6)}

(2) $J(C(S_1 \ ; \ S_2))(e,c)$
$= J(C(S_1) \ ; \ C(S_2))(e,c)$ {(3.3.5)}
$= c \circ J(C(S_1))(e,I) \circ J(C(S_2))(e,I)$ {reg($C(S_1)$), reg($C(S_2)$)}

Uit (1), (2), cvs(S_1), cvs(S_2), $c = I$ volgt cvs(S); uit (2) en lemma 4.5.1 volgt reg($C(S)$).

Geval S : if R then S_1 else S_2 fi

Gegeven : $(\sigma,f_1) \overset{\alpha}{\simeq} (m,s,f_2)$, cvr(R), cvs($S_1$), cvs($S_2$), reg($C(R)$), reg($C(S_1)$), reg($C(S_2)$)
te bewijzen : cvs(S), reg($C(S)$)
Bewijs:

(1) $M(\underline{\text{if}}\ R\ \underline{\text{then}}\ S_1\ \underline{\text{else}}\ S_2\ \underline{\text{fi}})(\sigma,f_1) =$
$\underline{\text{if}}\ T(R)\sigma\ \underline{\text{then}}\ M(S_1)(\sigma,f_1)\ \underline{\text{else}}\ M(S_2)(\sigma,f_1)$ {(4.2.4)}

(2) $J(C(\underline{\text{if}}\ R\ \underline{\text{then}}\ S_1\ \underline{\text{else}}\ S_2\ \underline{\text{fi}}))(e,c)(m,s,f_2)$
$= J([C(R)\,;\ \text{FJP}\ \ell_1\ ;\ C(S_1)\,;\ \text{UJP}\ \ell_2\ ;\ \ell_1 : C(S_2)\ ;\ \ell_2 :])(e,c)(m,s,f_2)$ {(3.3.3)}
$= (c \circ J(C(R)\ ;\ \text{FJP}\ \ell_1\ ;\ C(S_1)\ ;\ \text{UJP}\ \ell_2\ ;\ \ell_1 : C(S_2)\ ;\ \ell_2 :)$
$(L(p),I))(m,s,f_2)$ {(4.3.2)}

waarin

(*) $L(p) = \perp\{J(C(S_2))(L(p),L(p)(\ell_2))/\ell_1\}$ {(4.3.1)}
$\{I/\ell_2\}$
$= (c \circ J(\text{FJP}\ \ell_1\,;\ C(S_1)\ ;\ \text{UJP}\ \ell_2\ ;\ \ell_1 : C(S_2)\ ;\ \ell_2 :)(L(p),I))$
$(m,T(R)\sigma.s,f_2)$ {reg($C(R)$), lemma 4.5.3, cvr(R)}
$= (c \circ J(\text{FJP}\ \ell_1)(L(p),J(C(S_1)\ ;\ \text{UJP}\ \ell_2\ ;\ \ell_1 : C(S_2)\ ;\ \ell_2 :)(L(p),I))$
$(m,T(R)\sigma.s,f_2)$ {(4.3.4)}
$= c(\underline{\text{if}}\ T(R)\sigma$ {(4.3.10)}
$\underline{\text{then}}\ J(C(S_1)\ ;\ \text{UJP}\ \ell_2\ ;\ \ell_1 : C(S_2)\ ;\ \ell_2 :)(L(p),I)(m,s,f_2)$
$\underline{\text{else}}\ L(p)(\ell_1)(m,s,f_2))$
$= c(\underline{\text{if}}\ T(R)\sigma$
$\underline{\text{then}}\ (J(\text{UJP}\ \ell_2\,;\ \ell_1 : C(S_2)\,;\ \ell_2 :)(L(p),I) \circ J(C(S_1))(L(p),I))$
(m,s,f_2) {reg($C(S_1)$)}

$\underline{\text{else}}\ J(C(S_2))(L(p),I)(m,s,f_2))$ {(*)}

$= c(\underline{\text{if}}\ T(R)\sigma$

$\underline{\text{then}}\ J(C(S_1))(L(p),I)(m,s,f_2)$ {(4.3.4);(4.3.9), (*)}

$\underline{\text{else}}\ J(C(S_2))(L(p),I)(m,s,f_2))$.

Uit (1), (2), c = I, cvs(S_1), cvs(S_2) volgt cvs(S);
uit (2) en lemma 4.5.1 volgt reg(C(S)).

Geval: S : $\underline{\text{while}}$ R $\underline{\text{do}}$ S_1 $\underline{\text{od}}$

Gegeven : $(\sigma,f_1) \overset{\alpha}{\simeq} (m,s,f_2)$, cvr(R), cvs($S_1$), reg($C$(R)), reg($C(S_1)$)
Te bewijzen: cvs(S), reg(C(S))
Bewijs:

(1) $M(\underline{\text{while}}\ R\ \underline{\text{do}}\ S_1\ \underline{\text{od}})(\sigma,f_1) =$
$\underline{\text{if}}\ T(R)\sigma\ \underline{\text{then}}\ (M(\underline{\text{while}}\ R\ \underline{\text{do}}\ S_1\ \underline{\text{od}}) \circ M(S_1))(\sigma,f_1)\ \underline{\text{else}}\ (\sigma,f_1)$
{(4.2.5)}

(2) $J(C(\underline{\text{while}}\ R\ \underline{\text{do}}\ S_1\ \underline{\text{od}}))(e,c)(m,s,f_2)$
$= J([\ell_1 : C(R)\ ;\ \text{FJP}\ \ell_2\ ;\ C(S_1)\ ;\ \text{UJP}\ \ell_1\ ;\ \ell_2 :])(e,c)(m,s,f_2)$
{(3.3.4)}
$= (c \circ J(\ell_1 : C(R)\ ;\ \text{FJP}\ \ell_2\ ;\ C(S_1)\ ;\ \text{UJP}\ \ell_1\ ;\ \ell_2 :)(L(p),I))$
(m,s,f_2) {(4.3.2)}

waarin

(*) $L(p) = \perp\{J(C(R)\ ;\ \text{FJP}\ \ell_2\ ;\ C(S_1)\ ;\ \text{UJP}\ \ell_1\ ;\ \ell_2 :)$
$(L(p),L(p)(\ell_2))/\ell_1\}\{I/\ell_2\}$
$= (c \circ J(\text{FJP}\ \ell_2\ ;\ C(S_1)\ ;\ \text{UJP}\ \ell_1\ ;\ \ell_2 :)(L(p),I))(m,T(R)\sigma.s,f_2)$
{reg(C(R)), lemma 4.5.3, cvr(R)}
$= c(\underline{\text{if}}\ T(R)\sigma$ {(4.3.4);(4.3.10)}
$\underline{\text{then}}\ J(C(S_1)\ ;\ \text{UJP}\ \ell_1\ ;\ \ell_2 :)(L(p),I)(m,s,f_2)$
$\underline{\text{else}}\ L(p)(\ell_2)(m,s,f_2))$
$= c(\underline{\text{if}}\ T(R)\sigma$
$\underline{\text{then}}\ (J(\text{UJP}\ \ell_1\ ;\ \ell_2)(L(p),I) \circ J(C(S_1))(L(p),I))(m,s,f_2)$
{reg(C(S))}
$\underline{\text{else}}\ (m,s,f_2))$ {(*)}
$= c(\underline{\text{if}}\ T(R)\sigma$
$\underline{\text{then}}\ (L(p)(\ell_1) \circ J(C(S_1))(L(p),I))(m,s,f_2)$ {(4.3.9)}

$$\underline{\text{else}}\ (m,s,f_2))$$

$$= c(\underline{\text{if}}\ T(R)\sigma$$

$$\underline{\text{then}}\ (J(C(R)\ ;\ \text{FJP}\ \ell_2\ ;\ C(S_1)\ ;\ \text{UJP}\ \ell_1\ ;\ \ell_2\ :)$$

$$(L(p),I) \circ J(C(S_1))(L(p),I))(m,s,f_2) \qquad \{(*)\}$$

$$\underline{\text{else}}\ (m,s,f_2))$$

$$= c(\underline{\text{if}}\ T(R)\sigma$$

$$\underline{\text{then}}\ (J(C(\underline{\text{while}}\ R\ \underline{\text{do}}\ S_1\ \underline{\text{od}}))(L(p),I) \circ J(C(S_1))(L(p),I)(m,s,f_2)$$

$$\underline{\text{else}}\ (m,s,f_2)$$

Uit (1), (2), $cvs(S_1)$, I volgt nu m.b.v. fixed point induction, dat cvs(S); uit (2) en lemma 4.5.1 volgt $reg(C(S))$.

Geval: cvp(S)

Gegeven : cvs(S)

Te bewijzen: cvp(S)

Bewijs:

(1)	$P(S) = (M(S)(\perp,\varepsilon))[2]$	{(4.2.1)}
(2)	$R(C(S)) = (J(C(S))(\perp,I)(\perp,\varepsilon,\varepsilon))[3]$	{(4.3.13)}
(3)	$(\perp,\varepsilon) \overset{\alpha}{\simeq} (\perp,\varepsilon,\varepsilon)$	{(4.4.2)}
(4)	$M(S)(\perp,\varepsilon) \overset{\alpha}{\simeq} J(C(S))(\perp,I)(\perp,\varepsilon,\varepsilon)$	{(3),cvs(S)}
(5)	$P(S) = R(C(S))$	{(1),(2),(4),(4.4.2)}

Met dit laatste geval is het correctheidsbewijs van C voltooid.

5. IMPLEMENTATIECORRECTHEID IN TERMEN VAN AXIOMATISCHE SEMANTIEK

5.1. Inleiding

De axiomatische methoden zijn vooral ontwikkeld met het doel correctheidsbewijzen van programma's mogelijk te maken. De oudste representant van deze klasse is de door R.W. FLOYD geïntroduceerde methode van de "inductive assertions" [4] die gebaseerd is op het plaatsen van asserties in een stroomdiagram, en een bewijs dat langs iedere weg door het stroomdiagram aan die asserties voldaan is. De essentie van deze methode werd door C.A.R. HOARE [5,6] gevangen in een formeel systeem, dat gebaseerd is op het principe dat statements relaties tussen asserties specificeren. Voor primitieve statements worden die relaties d.m.v. axioma's gegeven, en voor

samengestelde statements d.m.v. deductieregels. Deze methode werd door E.W. DIJKSTRA verder ontwikkeld tot de methode van de "weakest preconditions" [3], waarbij het accent is verschoven van de verificatie van programma's naar de constructie van correcte programma's. Kenmerkend voor al deze methoden is de hoge mate van abstractie die bereikt wordt door het ontbreken van enig model. De asserties kunnen wel beschouwd worden als karakterisering van toestanden waarin een model kan verkeren, maar zo'n interpretatie is geenszins noodzakelijk.

In dit hoofdstuk zullen we laten zien hoe we op basis van axiomatische definities van B en D een correctheidsbewijs van de vertaler C kunnen geven. In secties 5.2 en 5.3 geven we axiomatische definities van B resp. D m.b.v. Hoare-bewijsregels; in sectie 5.4 stellen we de criteria op waaraan een vertaler van B naar D moet voldoen, en in sectie 5.5 geven we het bewijs dat C aan deze criteria voldoet. In sectie 5.6 bespreken we tenslotte, hoe we de beëindiging van D-programma's kunnen bewijzen.

5.2. Axiomatische definitie van B

De axiomatisering van de in B voorkomende constructies is langzamerhand zo bekend, dat hier nauwelijks toelichting vereist is. De enige opmerking die we dan ook maken is, dat een print statement opgevat kan worden als een assignment aan de file variabele f.

(5.2.1) $P[i \leftarrow E]\{V\langle i\rangle := E\}P$

(5.2.2) $P[f \leftarrow f.E]\{\underline{\text{print}}\ E\}P$

(5.2.3) $$\frac{P \wedge R\{S_1\}Q,\ P \wedge \neg R\{S_2\}Q}{P\{\underline{\text{if}}\ R\ \underline{\text{then}}\ S_1\ \underline{\text{else}}\ S_2\ \underline{\text{fi}}\}Q}$$

(5.2.4) $$\frac{P \wedge R\ \{S\}\ P}{P\{\underline{\text{while}}\ R\ \underline{\text{do}}\ S\ \underline{\text{od}}\}P \wedge \neg R}$$

(5.2.5) $$\frac{P\{S_1\}Q,\ Q\{S_2\}R}{P\{S_1;\ S_2\}R}\ .$$

5.3. Axiomatische definitie van D

Het geven van een axiomatische karakterisering van D lijkt in eerste instantie een hopeloze opgave, maar blijkt erg mee te vallen als we de volgende operationele beschrijving van de instructies in gedachten houden:

LDC v : s:= Push(v, s)
LDV a : s:= Push(m(a), s)
STO a : m(a), s:= Top(s), Pop(s)
OUT : f, s:= f.Top(s), Pop(s)
UJP ℓ : goto ℓ
FJP ℓ : if $\neg$Top(s) then s:= Pop(s); goto ℓ else s:= Pop(s) fi
ADD : s:= Push(Top(Pop(s)) + Top(s), Pop(Pop(S)))
GEQ : s:= Push(Top(Pop(s)) $\geq$ Top(s), Pop(Pop(s)))

Allereerst geven we een axiomatische definitie van de stack. Aangezien deze definitie het standaardvoorbeeld is van een abstracte typespecificatie, is nadere toelichting overbodig.

Empty : Stack
Push : Val $\cup$ {true, false} $\times$ Stack $\rightarrow$ Stack
Top : Stack $\rightarrow$ Val $\cup$ {true, false}
Pop : Stack $\rightarrow$ Stack

(5.3.1) Top(Push(v, s)) = v

(5.3.2) Pop(Push(v, s)) = s

Definitie van Top(Empty) en Pop(Empty) is overbodig.

Vervolgens geven we de definities van de instructies.

(5.3.3) P[s $\leftarrow$ Push(v, s)] {LDC v} P

(5.3.4) P[s $\leftarrow$ Push(m(a), s)] {LDV a} P

(5.3.5) P[m(a) $\leftarrow$ Top(s), s $\leftarrow$ Pop(s)] {STO a} P

(5.3.6) P[f $\leftarrow$ f.Top(s), s $\leftarrow$ Pop(s)] {OUT} P

(5.3.7) assertion(ℓ) {UJP ℓ} false

(5.3.8) $$\frac{P \wedge \neg Top(s) \Rightarrow assertion(\ell)[s \leftarrow Pop(s)],\ P \wedge Top(s) \Rightarrow Q[s \leftarrow Pop(s)]}{P\{FJP\ \ell\}Q}$$

(5.3.9) P[s $\leftarrow$ Push(Top(Pop(s)) + Top(s), Pop(Pop(s)))] {ADD} P

(5.3.10) P[s $\leftarrow$ Push(Top(Pop(s)) $\geq$ Top(s), Pop(Pop(s)))] {GEQ} P

(5.3.11) $$\frac{P\{r\}Q,\ P \text{ en } Q \text{ bevatten niets van de gedaante "assertion}(\ell)\text{"}}{P\{[r]\}Q}$$

(5.3.12) $$\frac{P\{r\}Q}{P\{\ell\colon r\}Q}$$

(5.3.13) $$\frac{P\{r_1\}Q,\ Q\{r_2\}R,\ reg(r_1),\ reg(r_2)}{P\{r_1;\ r_2\}R,\ reg(r_1;\ r_2)}$$

(5.3.14) $reg(r) \stackrel{def}{=}$ "r bevat geen losse labels of spronginstructies"

We merken bij deze definitie het volgende op:

1. In tegenstelling tot 4.4.6 heeft het predikaat reg(r) hier betrekking op de syntactische structuur, vanwege het feit dat bovenstaande bewijsregels alle betrekking hebben op syntactische eenheden, en niet op denotaties daarvan. Het is zeker mogelijk een formelere definitie van reg(r) te geven, maar dat is in dit geval wel wat zwaar geschut voor iets dat we in een oogopslag kunnen zien.
2. Met assertion(ℓ) wordt de assertie aangeduid die geldt op de plaats van het label ℓ. De bewijsvoering vindt plaats door bij elk label een assertie te plaatsen en vervolgens m.b.v. de bewijsregels aan te tonen dat langs elke weg door het programma aan de labelasserties voldaan is.
3. De meeste regels zijn met betrekkelijk weinig moeite te begrijpen en te gebruiken; de enige die wat problemen geeft is regel (5.3.8). De regel kan begrepen worden door uit te gaan van de operationele beschrijving van FJP, en daarop het assignment-axioma, het sprong-axioma en de if-then-else-fi regel toe te passen. Het gebruik van regel (5.3.8) is ook niet zo eenvoudig, omdat we in de meeste gevallen bij gegeven postconditie en labelassertie een preconditie zullen moeten verzinnen, die aan beide premissen voldoet.

5.4. Correctheidscriteria voor C

Evenals in sectie 4.4 zullen we ook hier de eisen moeten formuleren, waaraan C moet voldoen. Daarbij zullen we weer gebruik maken van predikaten cvp, cvs, cve en cvr, met dezelfde intuïtieve betekenis als in sectie 4.4, maar uiteraard met een heel andere formele definitie. Het predikaat cvp(S) zal weer tot uitdrukking moeten brengen dat S en C(S) gelijke files produceren, maar we zullen tevens eisen dat C(S) op ordentelijke wijze met de stapel omspringt:

(5.4.1) $\quad cvp(S) \stackrel{def}{=} f_1{=}\varepsilon\ \{S\}\ Q(f_1) \Rightarrow s{=}Empty \wedge f_2{=}\varepsilon\ \{C(S)\}\ s{=}Empty \wedge Q(f_2)$,

waarbij Q(f) een predikaat is dat alleen betrekking heeft op de file f.

Het bewijs zal dit keer geleverd worden in termen van correspondenties tussen asserties betreffende het B-programma resp. D-programma. Gezien het 1-1 duidige verband tussen variabelen en adressen zullen een assertie P betreffende het B-programma en een assertie P^* betreffende het D-programma met elkaar corresponderen indien P^* uit P verkregen kan worden door daarin elke variabele i te vervangen door de term $m(\alpha(i))$, zijnde de waarde van de geheugenplaats die overeenkomt met i. De correcte vertaling van statements laat zich dan formuleren als:

(5.4.2) $\quad cvs(S) \stackrel{def}{=} P\{S\}Q \Rightarrow P^* \wedge s{=}Empty\ \{C(S)\}\ Q^* \wedge s{=}Empty.$

Wat betreft cve(E) is de eis weer, dat door uitvoering van C(E) de waarde van de expressie E, of, daarmee gelijkwaardig, de waarde van de expressie E^*, op de stapel geplaatst wordt, terwijl voor het overige de toestand ongewijzigd blijft, hetgeen we kunnen formuleren als:

(5.4.3) $\quad cve(E) \stackrel{def}{=} P[s \leftarrow Push(E^*, s)]\ \{C(E)\}\ P.$

Tenslotte wordt cvr(R) weer analoog geformuleerd als:

(5.4.4) $\quad cvr(R) \stackrel{def}{=} P[s \leftarrow Push(R^*, s)]\ \{C(R)\}\ P.$

In sectie 5.5 zullen we aantonen dat C inderdaad aan bovengestelde eisen voldoet. We moeten daarbij wel bedenken dat deze eisen alleen het verband tussen de *partiële* correctheid van B- en D-programma's tot uitdrukking brengen. Op het verband tussen de beëindiging van B- en D-programma's zullen we verder ingaan in sectie 5.6.

5.5. Bewijs van de partiële correctheid van C

In deze paragraaf zullen we bewijzen dat C ieder partieel correct B-programma vertaalt in een partieel correct D-programma, formeel: $\forall S$: cvp(S). Het bewijs wordt gevoerd d.m.v. inductie naar de syntactische structuur van B-programma's. Strict genomen moeten we ook de regulariteit van iedere constructie bewijzen, maar die bewijzen zijn zo triviaal dat we ze achterwege

laten. We zullen de bewijzen zoveel mogelijk uniform structureren, ook al gaat dit in een enkel geval ten koste van de beknoptheid. We zullen steeds, indien we moeten bewijzen dat voor de vertaling van een constructie b de eigenschap $X\{C(b)\}Y$ geldt, in een eerste stap de vertaling van b schrijven als $C(b) = \{a_0\}i_1; \{a_1\}i_2; \ldots \{a_{n-1}\}\ i_n\ \{a_n\}$, vervolgens $a_n = Y$ stellen en daarna in een aantal stappen de asserties a_{n-1} t/m a_0 berekenen, waarbij als laatste stap zal moeten blijken dat $a_0 = X$. Tenslotte merken we nog op, dat indien voor een constructie b de waarheid van de vorm $P\{b\}Q$ alleen vastgesteld kan worden m.b.v. een deductieregel (zoals bij *if* en *while*), we de premissen van die deductieregel als extra aannamen in ons bewijs mogen gebruiken.

Geval E : C<v>

Te bewijzen: cve(E), i.c. $P[s \leftarrow Push(C\langle v\rangle^*, s)]\ \{C(C\langle v\rangle)\}\ P$

Bewijs:

(1) $C(C\langle v\rangle) = \{a_0\}$ LDC v $\{a_1\}$ {(3.3.6)}

(2) $a_1 = P$

(3) a. $C\langle v\rangle^* = v$

b. $P[s \leftarrow Push(v, s)]\ \{LDC\ v\}\ P$ {(5.3.3)}

c. $a_0 = P[s \leftarrow Push(C\langle v\rangle^*, s)]$ {(2),(3a),(3b)}

q.e.d.

Geval E : V<i>

Te bewijzen: cve(E), i.c. $P[s \leftarrow Push(V\langle i\rangle^*, s)]\ \{C(V\langle i\rangle)\}\ P$

Bewijs:

(1) $C(V\langle i\rangle) = \{a_0\}$ LDV $\alpha(i)$ $\{a_1\}$ {(3.3.7)}

(2) $a_1 = P$

(3) a. $V\langle i\rangle^* = m(\alpha(i))$

b. $P[s \leftarrow Push(m(\alpha(i)), s)]\ \{LDV\ \alpha(i)\}\ P$ {(5.3.4)}

c. $a_0 = P[s \leftarrow Push(V\langle i\rangle^*, s]$ {(2),(3a),(3b)}

q.e.d.

Geval E : $E_1 + E_2$

Gegeven : cve(E_1), cve(E_2)

Te bewijzen: cve(E), i.c. $P[s \leftarrow Push((E_1 + E_2)^*, s)]\ \{C(E_1 + E_2)\}\ P$

Bewijs:

(1) $C(E_1 + E_2) = \{a_0\}\ C(E_1);\ \{a_1\}\ C(E_2);\ \{a_2\}$ ADD $\{a_3\}$ {(3.3.8)}

(2) $a_3 = P$

(3) $a_2 = P[s \leftarrow Push(Top(Pop(s)) + Top(s), Pop(Pop(s)))]$ {(5.3.9)}

(4) $a_1 = a_2[s \leftarrow Push(E_2^*, s)]$ {cve(E_2)}
$= P[s \leftarrow Push(Top(Pop(Push(E_2^*, s))) + Top(Push(E_2^*, s)),$
$Pop(Pop(Push(E_2^*, s))))]$
$= P[s \leftarrow Push(Top(s) + E_2^*, Pop(s))]$ {(5.3.1),(5.3.2)}

(5) $a_0 = a_1[s \leftarrow Push(E_1^*, s)]$ {cve(E_1)}
$= P[s \leftarrow Push(Top(Push(E_1^*, s)) + E_2^*, Pop(Push(E_1^*, s)))]$
$= P[s \leftarrow Push(E_1^* + E_2^*, s)]$ {(5.3.1),(5.3.2)}
$= P[s \leftarrow Push((E_1 + E_2)^*, s)]$
q.e.d.

Geval R : $E_1 \geq E_2$

Bewijs volledig analoog aan dat van geval E : $E_1 + E_2$.

Geval S: V<i>:= E

Gegeven : cve(E)

Te bewijzen: cvs(S), i.c. $P^*[m(\alpha(i)) \leftarrow E^*] \wedge s=Empty\ \{C(S)\}\ P^* \wedge s=Empty$

Bewijs:

(1) $C(V<i>:= E) = \{a_0\}\ C(E);\ \{a_1\}$ STO $\alpha(i))\ \{a_2\}$ {(3.3.2)}

(2) $a_2 = (P^* \wedge s=Empty)$

(3) $a_1 = a_2[m(\alpha(i)) \leftarrow Top(s), s \leftarrow Pop(s)]$ {(5.3.5)}
$= P^*[m(\alpha(i)) \leftarrow Top(s)] \wedge Pop(s) = Empty$

(4) $a_0 = a_1[s \leftarrow Push(E^*, s)]$ {(cve(E)}
$= P^*[m(\alpha(i)) \leftarrow Top(Push(E^*, s))] \wedge Pop(Push(E^*, s)) = Empty$
$= P^*[m(\alpha(i)) \leftarrow E^*] \wedge s=Empty$ {(5.3.1),(5.3.2)}
q.e.d.

Geval S: print E

Gegeven : cve(E)

Te bewijzen: cvs(S), i.c. $P^*[f \leftarrow f.E^*] \wedge s=Empty\ \{C(S)\}\ P^* \wedge s=Empty$

Bewijs:

(1) $C(\underline{\text{print}}\ E) = \{a_0\}\ C(E);\ \{a_1\}\ \text{OUT}\ \{a_2\}$ {(3.3.1)}

(2) $a_2 = (P^* \wedge s=\text{Empty})$

(3) $a_1 = a_2[f \leftarrow f.\text{Top}(s),\ s \leftarrow \text{Pop}(s)]$ {(5.3.6)}
$= P^*[f \leftarrow f.\text{Top}(s)] \wedge \text{Pop}(s) = \text{Empty}$

(4) $a_0 = a_1[s \leftarrow \text{Push}(E^*, s)]$ {cve(E)}
$= P^*[f \leftarrow f.\text{Top}(\text{Push}(E^*, s))] \wedge \text{Pop}(\text{Push}(E^*, s)) = \text{Empty}$
$= P^*[f \leftarrow f.E^*] \wedge s=\text{Empty}$ {(5.3.1),(5.3.2)}
q.e.d.

Geval S: $S_1;\ S_2$

Triviaal.

Geval S: $\underline{\text{if}}\ R\ \underline{\text{then}}\ S_1\ \underline{\text{else}}\ S_2\ \underline{\text{fi}}$

Gegeven : cvr(R), cvs(S_1), cvs(S_2)
Extra aannamen: $P \wedge R\{S_1\}Q$, $P \wedge \neg R\{S_2\}Q$
Te bewijzen : cvs(S), i.c. $P^* \wedge s=\text{Empty}\ \{C(S)\}\ Q^* \wedge s=\text{Empty}$
Bewijs:

(1) $C(\underline{\text{if}}\ R\ \underline{\text{then}}\ S_1\ \underline{\text{else}}\ S_2\ \underline{\text{fi}}) =$
$[\{a_0\}\ C(R);\ \{a_1\}\ \text{FJP}\ \ell_1;\ \{a_2\}\ C(S_1);\ \{a_3\}\ \text{UJP}\ \ell_2;\ \{a_4\}\ \ell_1:$
$C(S_2);\ \{a_5\}\ \ell_2:]$

(2) $a_5 = Q^* \wedge s=\text{Empty}$

(3) a. $P^* \wedge \neg R^* \wedge s=\text{Empty}\ \{C(S_2)\}\ Q^* \wedge s=\text{Empty}$ {cvs(S_2), aanname 2}
b. $a_4 = P^* \wedge \neg R^* \wedge s=\text{Empty}$ {(2),(3a)}

(4) a. assertion (ℓ_2) = a_5
b. false $\Rightarrow a_4$
c. $a_5\ \{\text{UJP}\ \ell_2\}\ a_4$ {(4a),(4b),(5.3.7)}
d. $a_3 = a_5$

(5) a. $P^* \wedge R^* \wedge s=\text{Empty}\ \{C(S_1)\}\ Q^* \wedge s=\text{Empty}$ {cvs(S_1), aanname 1}
b. $a_2 = P^* \wedge R^* \wedge s=\text{Empty}$ {(4d),(5a)}

(6) a. assertion (ℓ_1)[s ← Pop(s)]
$= a_4[s \leftarrow \text{Pop}(s)]$
$= P^* \wedge \neg R^* \wedge \text{Pop}(s) = \text{Empty}$

b. $a_2[s \leftarrow Pop(s)] = P^* \wedge R^* \wedge Pop(s) = Empty$

c. $P^* \wedge Top(s) = R^* \wedge Pop(s) = Empty \wedge \neg Top(s) \Rightarrow$
assertion $(\ell_1)[s \leftarrow Pop(s)]$ {(6a)}

d. $P^* \wedge Top(s) = R^* \wedge Pop(s) = Empty \wedge Top(s) \Rightarrow a_2[s \leftarrow Pop(s)]$ {(6b)}

e. $a_1 = P^* \wedge Top(s) = R^* \wedge Pop(s) = Empty$ {(6c),(6d),(5.3.8)}

(7) $a_0 = a_1[s \leftarrow Push(R^*, s)]$ {cvr(R)}
$= P^* \wedge Top(Push(R^*, s)) = R^* \wedge Pop(Push(R^*, s)) = Empty$
$= P^* \wedge s=Empty$ {(5.3.1),(5.3.2)}
q.e.d.

Geval S: <u>while</u> R <u>do</u> S_1 <u>od</u>

Gegeven : cvr(R), cvs(S_1)
Extra aanname: $P \wedge R \{S_1\} P$
Te bewijzen : cvs(S), i.c. $P^* \wedge s=Empty \{C(S)\} P^* \wedge \neg R^* \wedge s=Empty$
Bewijs:

(1) $C($<u>while</u> R <u>do</u> S_1 <u>od</u>$) =$
$[\ell_1: \{a_0\}\ C(R);\ \{a_1\}$ FJP $\ell_2;\ \{a_3\}\ C(S_1);\ \{a_4\}$ UJP $\ell_1;\ \{a_5\}\ \ell_2:]$
We beginnen nu met het geven van assertion(ℓ_1) en assertion(ℓ_2), en laten zien dat a_0, berekend als preconditie van $C(R)$, gelijk is aan assertion(ℓ_1).

(2) assertion(ℓ_1) = $(P^* \wedge s=Empty)$

(3) a_5 = assertion(ℓ_2) = $P^* \wedge \neg R^* \wedge s=Empty$

(4) a. $false \Rightarrow a_5$
b. $P^* \wedge s=Empty \{$UJP $\ell_1\}\ a_5$ {(2),(4a),(5.3.7)}
c. $a_4 = (P^* \wedge s=Empty)$

(5) $a_3 = (P^* \wedge R^* \wedge s=Empty)$ {(4c),cvs(S_1), aanname}

(6) a. assertion(ℓ_2)$[s \leftarrow Pop(s)] = P^* \wedge \neg R^* \wedge Pop(s) = Empty$ {(3)}
b. $a_3[s \leftarrow Pop(s)] = P^* \wedge R^* \wedge Pop(s) = Empty$ {(5)}
c. $P^* \wedge Top(s) = R^* \wedge Pop(s) = Empty \wedge \neg Top(s) \Rightarrow$
assertion(ℓ_2)$[s \leftarrow Pop(s)]$ {(6a)}
d. $P^* \wedge Top(s) = R^* \wedge Pop(s) = Empty \wedge Top(s) \Rightarrow a_3[s \leftarrow Pop(s)]$ {(6b)}
e. $a_1 = (P^* \wedge Top(s) = R^* \wedge Pop(s) = Empty)$ {(6a),(6b),(5.3.8)}

(7) $a_0 = a_1[s \leftarrow Push(R^*, s)]$ {cvr(R)}
$= P^* \wedge Top(Push(R^*, s)) = R^* \wedge Pop(Push(R^*, s)) = Empty$
$= P^* \wedge s=Empty$ {(5.3.1),(5.3.2)}
$= assertion(\ell_1)$ {(2)}
q.e.d.

Geval cvp(S)

Gegeven : cvs(S)

Te bewijzen: cvp(S)

Bewijs: volgt onmiddellijk uit de definities van cvp en cvs {(5.4.1),(5.4.2)}

Met dit laatste geval is het partiële correctheidsbewijs van C voltooid.

5.6. Terminatie

Tot nu toe hebben we ons alleen beziggehouden met partiële correctheid van het doelprogramma, d.w.z. als het eindigt, dan in een toestand die correspondeert met de toestand waarin het bronprogramma eindigt. We moeten echter ook laten zien dat een eindigend bronprogramma wordt vertaald naar een eindigend doelprogramma. Zonder deze eis zou het maken van een vertaler triviaal zijn, omdat we dan ieder bronprogramma kunnen vertalen als een niet-eindigend doelprogramma:

$$C(p) = \ell\text{: UJP } \ell$$

immers:

(1) $assertion(\ell)$: P^*

(2) P^* {UJP ℓ} false

(3) false $\Rightarrow Q^*$

(4) P^* {ℓ: UJP ℓ} Q^*

Voor een taal als B wordt het eindigen van een while statement gewoonlijk bewezen door gebruik te maken van een terminatiefunctie t, die door de herhaalde statement in waarde verlaagd wordt, maar naar beneden begrensd is. De deductieregel voor de while statement krijgt dan de volgende vorm:

$$\frac{P \wedge R \ \{S\}\ P,\ P \Rightarrow t \geq 0,\ P \wedge R \wedge t=c \ \{S\}\ t \leq c-1}{P \ \{\underline{\text{while}}\ R\ \underline{\text{do}}\ S\ \underline{\text{od}}\}\ P \wedge \neg R}$$

We kunnen voor D-programma's een soortgelijke techniek gebruiken, door met

ieder label een grootheid te associëren, die aangeeft hoe vaak het label bezocht is. De bewijsregels krijgen dan de volgende vorm:

$$\text{assertion}(\ell)[n\ell \leftarrow n\ell+1]\ \{\ell:\}\ \text{assertion}(\ell)$$

$$\text{assertion}(\ell)[n\ell \leftarrow n\ell+1]\ \{\text{UJP}\ \ell\}\ \text{false}$$

Het bewijs dat een D-programma eindigt wordt dan geleverd door aan te tonen dat alle $n\ell$'s naar boven begrensd zijn. In het geval van de vertaling van de if statement is dat bewijs bijzonder eenvoudig. In het geval van de while statement gaan we uit van het bestaan van een t voor het bronprogramma en gebruiken de vertaalde versie t^* om aan te tonen dat $n\ell_1$ naar boven begrensd is door T+1, waarbij T de initiële waarde van t^* is. We zullen hier geen volledig bewijs geven, maar alleen de relevante asserties laten zien:

$$\{P^* \wedge n\ell_1 = 0 \wedge t^* \leq T - n\ell_1\}$$
$$\ell_1:$$
$$\{P^* \wedge t^* \leq T - n\ell_1 + 1\}$$
$$C(R)$$
$$\text{FJP}\ \ell_2$$
$$\{P^* \wedge R^* \wedge t^* \leq T - n\ell_1 + 1\}$$
$$C(S)$$
$$\{P^* \wedge t^* \leq T - n\ell_1\}$$
$$\text{UJP}\ \ell_1$$
$$\ell_2:$$
$$\{P^* \wedge t^* \leq T - n\ell_1 + 1\}$$

Aangezien $P^* \Rightarrow t^* \geq 0$, geldt tevens $P^* \Rightarrow n\ell_1 \leq T + 1$, zodat uit de beëindiging van het bronprogramma de beëindiging van het doelprogramma volgt.

6. CONCLUSIES

Wat kunnen we nu leren van de in de voorgaande hoofdstukken gegeven definities en correctheidsbewijzen? In de eerste plaats natuurlijk dat dergelijke bewijzen best uitvoerbaar zijn, ook zonder de op sommige plaatsen in de literatuur [8,14] onmisbaar geachte "powerful automatic verification systems". Men kan hier natuurlijk tegen aanvoeren dat de gebruikte

bron- en doeltaal erg primitief zijn, en dat de gepresenteerde bewijsmethoden niet toepasbaar zijn op "echte" talen en "echte" vertalers, maar daarbij willen we het volgende aantekenen. De bewijzen zijn gebaseerd op een correctheidspredikaat per nonterminal en een bewijsstap per produktieregel, en bijgevolg recht evenredig met de grootte van de grammatica, en dat is het beste dat we mogen verwachten. Indien de methoden niet toepasbaar zijn op een "echte" taal, dan is dat veeleer te wijten aan de onregelmatigheden in die taal, die een modulaire bewijsstructuur in de weg staan. Dit geldt trouwens niet alleen voor implementatiebewijzen; het is een bekend feit dat onregelmatigheden in een taal aanleiding geven tot sterke uitdijing van de definitie en van de vertaler, en dat het gebruik van formele definitiemethoden een heilzaam effect heeft op de structuur en omvang van de gedefinieerde taal. Wat betreft de doeltaal, is het goed te bedenken dat geen enkele vertaler code genereert voor een echte machine, ook al lijkt dit vaak wel zo. De doelmachine die de implementator voor ogen staat is een abstracte machine, waarin code en data gescheiden zijn, en waarin sprake is van datastructuren als bloksegmenten en stapels en van acties als blokbinnenkomst en blokverlating. Die abstracte doelmachine moet natuurlijk gerealiseerd worden m.b.v. de faciliteiten van de concrete machine, maar dat is een volgende implementatiestap, waarvoor de bekende bewijstechnieken voor abstracte datatypen gebruikt kunnen worden.

Wat betreft de bewijzen zelf lijkt het dat de axiomatische methode te prefereren is boven de denotationele methode, niet alleen vanwege de geringere hoeveelheid dom rekenwerk, maar vooral vanwege het grotere inzicht dat de axiomatische bewijzen bieden. Deze eigenschap wordt natuurlijk van groter belang naarmate we taalconstructies en vertalingen beschouwen die minder standaard zijn. We moeten ons ook realiseren dat we in werkelijkheid niet zoals in deze syllabus de correctheid van een gegeven vertaling moeten bewijzen, maar aan de hand van de definities van bron- en doeltaal een correcte vertaling moeten construeren, en daarbij zijn de axiomatische formuleringen van grotere heuristische waarde dan de denotationele.

7. BIBLIOGRAFIE

Tot slot geven we in dit hoofdstuk een kort overzicht van publicaties op het gebied van correctheid van implementaties, geclassificeerd naar de gebruikte semantische definitiemethode (alhoewel de scheidslijnen niet

altijd duidelijk te trekken zijn).

Operationeel

Het oudste correctheidsbewijs voor een implementatie is te vinden in [10], waarin een vertaler voor arithmetische expressies behandeld wordt. In [17] worden in essentie dezelfde methoden gebruikt voor een wat uitgebreidere taal. In [7] wordt de correctheid van implementatietechnieken bewezen door de equivalentie van interpretatoren aan te tonen. Een soorgelijke benadering is te vinden in [11]. PAGAN [16] beschrijft het verband tussen een definitie in de Vienna Definition Language en een in ALGOL 68 geschreven interpretator.

Denotationeel

Het verband tussen denotationele definities en implementatie werd het eerst beschreven door MILNE [12]. Het denotationele bewijs in hoofdstuk 4 van deze syllabus werd geïnspireerd door [18], waarin de correctheid van een interpretator en een compiler voor een eenvoudig taaltje bewezen wordt. STOY [20] illustreert o.a. een aantal bewijsmethoden aan de hand van een vertaler voor arithmetische expressies. Wie de moed kan opbrengen moet ook maar eens kijken in [13].

Axiomatisch

Op het gebied van axiomatische correctheidsbewijzen van implementaties is vrijwel geen literatuur beschikbaar. LYNN [8] beschrijft de automatische verificatie van een LISP compiler en van stukjes van een PL/0 compiler. RUSSELL [19] bewijst de correctheid van een aantal transformaties, die programma's in een gegeven taal omzetten naar programma's in een subset van die taal.

Algebraïsch

Een benadering die we tot nu toe niet genoemd hebben is die m.b.v. algebraïsche methoden. Deze benadering heeft veel gemeen met de denotationele semantiek, maar maakt tevens gebruik van de algebraïsche structuur die in talen en hun semantiek aanwezig is. Door de afbeeldingen tussen deze structuren te beperken tot homomorfismen, en gebruik te maken van bekende eigenschappen m.b.t. tot bestaan en de uniciteit van homomorfismen is het dan mogelijk inductieve bewijzen te vermijden. Deze aanpak werd geïntroduceerd in [1] en toegepast in [14]. De methode wordt ook beschreven in [15] en

verder uitgewerkt en toegepast op een grotere taal in [2].

LITERATUUR

[1] BURSTALL, R.M. & P.J. LANDIN, *Programs and their Proofs: an Algebraic Approach,* Machine Intelligence 4, (B. Meltzer & D. Michie, eds), Edinburgh University Press, 1969, pp.17-43.

[2] CHIRICA, L.M., *Contributions to compiler correctness,* Ph.D. Thesis, Computer Science Department, UCLA, 1976.

[3] DIJKSTRA, E.W., *A Discipline of Programming,* Prentice-Hall, 1976.

[4] FLOYD, R.W., *Assigning Meaning to Programs,* In: Mathematical Aspects of Computer Science. (J.T. Schwartz, ed.), Proc. Symposia in Applied Mathematics 19, Amer. Math. Soc., Providence (R.I.), 1967.

[5] HOARE, C.A.R. & N. WIRTH, *An Axiomatic Definition of the Programming Language Pascal,* Acta Informatica 2 (1973), pp.335-355.

[6] HOARE, C.A.R., *An Axiomatic Basis for Computer Programming,* Comm. ACM 12 (1967), pp.576-580.

[7] JONES, C.B. & P. LUCAS, *Proving Correctness of Implementation Techniques,* In: Semantics of Algorithmic Languages (E. Engeler, ed.), Lecture Notes in Mathematics 188, Springer (1971), pp.178-211.

[8] LYNN, D.S., *Interactive Compiler Proving Using Hoare Proof Rules,* Ph.D. Thesis, Information Sciences Institute, University of Southern California, 1978.

[9] MARCOTTY, M., H.F. LEDGARD & G.V. BOCHMANN, *A Sampler of Formal Definitions,* Computing Surveys 8, (1976), pp.191-276.

[10] McCARTHY, J. & J. PAINTER, *Correctness of a Compiler for Arithmetic expressions,* In: Mathematical Aspects of Computer Science (J.T. Schwartz, ed.), Proc. Symposia in Applied Mathematics 19, Amer. Math. Soc., Providence (R.I.), 1967.

[11] McGOWAN, C. & P. WEGNER, *The equivalence of sequential and associative information structure models,* Proceedings of a Symposium on Data Structures in Programming Languages, SIGPLAN Notices 6 (1971), pp.191-216.

[12] MILNE, R., *The Formal Semantics of Computer Languages and their Implementation,* Computing Laboratory, Oxford University, Technical Monograph PRG-13, 1974.

[13] MILNE, R. & C. STRACHEY, *A Theory of Programming Language Semantics,* Chapman and Hall, 1976.

[14] MILNER, R. & R. WEYHRAUCH, *Proving Compiler Correctness in a Mechanized Logic,* Machine Intelligence 7 (B. Meltzer & D. Michie, eds), Edinburgh University Press, 1972.

[15] MORRIS, F.L., *Advice on Structuring Compilers and Proving Them Correct,* Conference Record ACM Symposium on Principles of Programming Languages, Boston, 1973.

[16] PAGAN, F.G., *On Interpreter-Oriented Definitions of Programming Languages,* Computer Journal 19 (1976), pp.151-155.

[17] PAINTER, J.A., *Semantic Correctness of a Compiler for an ALGOL-like language,* Artificial Intelligence Memo 44, Stanford, 1967.

[18] RUSSELL, B.D., *Implementation Correctness Involving a Language with Goto Statements,* SIAM J. Comput. 6 (1977), pp.403-415.

[19] RUSSELL, B.D., *Correctness of Program Transformations based on Axiomatic Semantics,* Niet gepubliceerd manuscript.

[20] STOY, J., *Foundations of Mathematical Semantics,* Lect. Notes Advanced Course on Abstract Software Specifications, Copenhagen, 1979.

[21] TENNENT, R.D., *The Denotational Semantics of Programming Languages,* Comm. ACM 19 (1976), pp.437-453.

DE CYBER ALGOL 68 VERTALER

J.J.F.M. SCHLICHTING
Control Data Services B.V. Rijswijk

1. INLEIDING

De Cyber ALGOL 68 vertaler is ontwikkeld door de Nederlandse Control Data Organisatie in de jaren 1972 t/m 1975 in opdracht van de Nederlandse Universitaire Rekencentra ACCU, RUG en SARA.

De opdracht luidde volledige implementatie van het ALGOL 68 Report. Wij hebben zoveel mogelijk het toen in ontwikkeling zijnde Revised Report [1] gevolgd.

De in juli 1975 opgeleverde vertaler week op de volgende punten af van het Revised Report:

1. *Style-i-sub- en style-i-bus-symbols* worden voorgesteld door (/ en /) i.p.v. door (en).
2. VOID kan niet optreden als constituent van een united mode.
3. Het gebruik van het *flexible-symbol* leidt niet tot een andere mode, maar bepaalt slechts een attribuut van de gegenereerde waarde.
4. Het *long-symbol* kan alleen gebruikt worden in LONG REAL.
5. *Format-patterns* in transput formats zijn niet geïmplementeerd. In plaats daarvan bestaat een Hollerith pattern, waarmee een dynamisch bepaalde string in het format aangegeven kan worden.
6. In de routine ASSOCIATE is niet voorzien.
7. CONVERSION KEY is niet beschikbaar.
8. Formats worden in hun geheel gestaticized op het moment van associatie met een file.

In deze voordracht wordt na een globale beschrijving van de vertaler slechts het machine-onafhankelijke gedeelte behandeld.

2. OVERZICHT VAN DE VERTALER

2.1. Structuur van de vertaler

De vertaler bestaat uit een permanent geladen besturingsprogramma en zes overlays, die ieder een volledige scan over de te vertalen tekst uitvoeren.

De mode-onafhankelijke parser-scan wordt voorafgegaan door de zgn. lexical scan, waarin alle *mode-*, *operator-* en *priority-declarations* met hun geldigheidsbereik worden gevonden en in tabellen vastgelegd, zodat tijdens de ontleedfase van iedere indicant de soort (mode of operator) bekend is. De lexical scan verzorgt ook het afdrukken van het te vertalen programma.

De parser levert de syntactische struktuur van het programma in omgekeerde Poolse notatie.

Na de ontleedfase worden alle in het programma voorkomende modes gecontroleerd op juistheid (o.a. afwezigheid van circulariteiten) en worden identieke modes bepaald in de mode-equivalence routine.

De derde scan identificeert identifiers en operators en berekent alle coercies, die in het getransformeerde programma worden ingevoegd. Na deze scan worden in het geval van prelude-vertaling de vertaler-tabellen uitgeschreven, die bij latere vertaling beschikbaar moeten zijn.

De daarop volgende scan genereert registerloze code in triplet vorm (operator en twee operanden).

De vijfde scan alloceert registers en optimaliseert de object-code.

De laatste scan drukt de foutmeldingen af, vult adressen in en genereert het objectprogramma in standaard relokateerbare vorm. De optionele object-code listing wordt door deze scan verzorgd.

2.2. Naamlijst

De Naamlijst is een conglomeraat van interne vertaler-tabellen, waarin iedere tabel wordt voorgesteld door alle ingangen met een voor die tabel karakteristieke waarde van het in alle ingangen voorkomende veld CLAS.
Wij onderscheiden de volgende categorieën van ingangen:

a. NAME-ingangen bevatten namen, dus tags en bold tags en de interne representatie van constanten. Dit zijn de enige ingangen met een variabele lengte.

b. CORR-ingangen vertegenwoordigen voorlopige ranges, of CORRAL's, zoals bepaald in de lexical scan.

c. RNGH-ingangen vertegenwoordigen ranges in het programma.

d. Ingangen, die declaraties vertegenwoordigen:
PRIO : *priority-declaration*
OPER : *operator-declaration*
MODE : *mode-declaration*
IDEN : *identifier-declaration*
Deze ingangen zijn geketend aan de CORR- of RNGH-ingang, die hun geldigheidsbereik vertegenwoordigt.

e. Ingangen voor *mode-declarators*:
MRW-ingangen vertegenwoordigen een mode.
FIELD-ingangen vertegenwoordigen de componenten van een gestructureerde mode.
UMEM-ingangen vertegenwoordigen de constituenten van een united mode.

f. Ingangen voor constanten in het object-programma:
CONS-ingang voor absolute constanten.
ADCN-ingang voor adres-constanten.
XCON-ingang voor constanten met een aantal relokateerbare velden.
RELC-ingang defenieert een relokateerbaar veld van een XCON-ingang.

g. ERROR-ingangen:
Worden gegenereerd door detectie van fouten in het programma en bevatten foutnummer, regelnummer en eventueel additionele informatie.
Iedere scan heeft een eigen keten van ERROR-ingangen. De laatste scan verzorgt het afdrukken van alle foutmeldingen.

2.3. Implementatietaken

De vertaler is bijna geheel geschreven in SYMPL, de vertaler input-output en enige hulproutines zijn geschreven in de assemblertaal COMPASS.

2.4. Geheugenindeling van de vertaler

De vertaler-passes of overlays liggen onmiddellijk boven het besturingsprogramma. De Naamlijst begint boven de grootste overlay en groeit naar hogere adressen. De stapels van de eerste, tweede en vierde scan beginnen bij het hoogst beschikbare adres en groeien naar lagere adressen.

In de derde scan wordt de geheugenruimte boven de Naamlijst in drie gelijke delen verdeeld: het eerste deel voor uitbreiding van de Naamlijst, het tweede deel voor twee naar elkaar toegroeiende stapels en het derde deel voor een later te behandelen optimalisatie-tabel.

De vijfde en zesde scan gebruiken geen stapels. Automatische geheugenuitbreiding is niet geïmplementeerd. Als de Naamlijst en de stapel elkaar raken, wordt de vertaling afgebroken. Hervertaling met een groter beschikbaar geheugen is de enige remedie.

3. PASS1

Pass1, de eerste overlay van de vertaler, decodeert de ALGOL 68 stuurkaart en voert de lexical scan over het programma uit. De belangrijkste functies van deze scan zijn:

1. Lezen van de brontekst en het uitschrijven hiervan.
2. Herkenning van "basic tokens" en verwijdering van blanks buiten *string-denotations* en commentaar.
3. Vervanging van enkelvoudige syntactische eenheden, zoals identifiers, field-names, labels, indicants en denotations door verwijzing naar de geëigende ingangen in de Naamlijst.
4. Analyse van de haakjesstructuur en bepaling van voorlopige ranges of CORRAL's.
5. Herkenning van *priority-, operator-* en *mode-declarations*.
6. Uitschrijven van de getransformeerde brontekst op de IL1 (Intermediate Language 1) file.

Ad 1: Lezen van de brontekst en het afdrukken hiervan. Een enkele subroutine drukt de laatst ingelezen regel af, leest een nieuwe regel en zet deze om naar ASCII. Implementatie van een andere characterset (b.v. ASCII) is mogelijk door vervanging van deze subroutine door een andere met dezelfde functies en characterset-onafhankelijke interface.

Ad 3: Behandeling van enkelvoudige syntactische eenheden. Voor identifiers, field-names, labels en indicants wordt een NAME-ingang gegenereerd met behulp van een 7-bits hash-algoritme. Voor *INTREAL-* en *STRING-denotations* wordt een CONS-ingang gegenereerd, die verwijst naar een NAME-ingang voor de bijbehorende interne representatie en naar een MRW-ingang voor de mode van de denotatie. *Format-denotations* worden volgens de methode van Goos [3] omgezet in een string en een row-display. De string bevat de *format-patterns* en de row-display bevat componenten van de mode

UNION(PROC INT, PROC [ ] CHAR)

en vertegenwoordigt de dynamisch bepaalde delen van het format.

Ad 4: Analyse van de haakjesstructuur en de bepaling van CORRAL's.
In de volgende constructies worden één of meer CORRAL's gevonden, aangegeven door --.

```
          ( -- )                    BEGIN -- END

      ( -- | -- | -- )              IF -- THEN -- ELSE -- FI

( -- | -- |: -- | -- | -- )         IF -- THEN -- ELIF -- THEN
                                        -- ELSE -- FI

CASE -- IN OUT -- ESAC

CASE -- IN OUSE -- IN OUT -- ESAC.
```

Deze zijn alle constructies, die een range opleveren, waarin *mode-, operator-* of *priority-declarations* kunnen voorkomen.
Voor iedere gevonden CORRAL wordt een CORR-ingang gegenereerd, waarvan de index in de Naamlijst onmiddellijk na het openingssymbool op de IL1-file wordt geschreven.

Ad 5: Herkenning van *priority-, operator-* en *mode-declarations*.
Voor ieder van deze declaraties wordt een PRIO-, OPER- of MODE-ingang gegenereerd en aan de CORR-ingang van de lopende CORRAL gehangen.

Ad 6: Schrijven van de IL1-file.

De IL1-file is opgebouwd uit 20-bit bytes, waarvan de meest significante 3 bits het type en de overige 17 bits de waarde aangeven, als volgt.

Type	Waarde
Symbol	Basic symbol
Regelnummer	Regelnummer
CORRAL	Index van CORR-ingang
Identifier	Index van NAME-ingang
Indicant	Index van NAME-ingang
Constant	Index van CONS-ingang
Monad	Name-ingang van monadische operator
Nomad	Name-ingang van operator

4. PASS2

In de tweede scan wordt een mode-onafhankelijke parse van het programma doorgevoerd. De prioriteit-afhankelijke ontleding van formules wordt tot de derde scan uitgesteld. Het ontleedprogramma wordt gegenereerd m.b.v. een parser-generator, waarvan de input bestaat uit een SLR3-grammatica met ingevoegde semantiek en foutnummers. Het gegenereerde programma ontleedt het aangeboden programma, voert in iedere toestand de aangegeven semantische actie uit, terwijl bij detectie van fouten de recovery-routine wordt aangeroepen met het op het betreffende punt in de grammatica meegegeven foutnummer als parameter.
De volgende secties geven details over
1. opstelling van de grammatica,
2. het ontleedprogramma,
3. de invoer-routine,
4. declaraties en ranges,
5. modes.

4.1. Opstelling van de grammatica

In het Revised Report wordt ALGOL 68 gedefinieerd m.b.v. een tweeniveau grammatica; hieruit wordt een BNF-grammatica van een supertaal

afgeleid door productieregels die een onbegrensde verzameling regels produceren, door een enkele regel te vervangen. In dit proces gaan alle mode-afhankelijke restricties, en dus ook de coercies, verloren; deze worden in een volgende pass behandeld.

Iedere indicant kan met een declaratie (binnen een CORRAL) geïdentificeerd worden; het type (operator of mode) is daardoor bekend en constructies als "TAG x" kunnen dus eenduidig ontleed worden als formule of declaratie. De prioriteit van operators wordt genegeerd, zodat het aantal productieregels wordt beperkt; de prioriteit-afhankelijke ontleding van formules vindt in de volgende scan plaats. Labels in *serial-* en *enquiry-clauses* worden grammaticaal toegelaten, maar semantisch gedetecteerd.

Een *formal-parameter-plan* en een *serial-clause* kunnen in een SLR(k)-parser niet zonder meer worden onderscheiden. De invoer-routine kijkt daarom bij het inlezen van een *open-symbol* een niet statisch bekend aantal symbolen vooruit om *formal-parameter-plans* te detecteren en zonodig een speciaal symbool "formal" in de invoer te insereren. De grammatica kent dus dit terminaal symbool "formal", hoewel het op de IL1-file niet voorkomt.

4.2. Het ontleedprogramma

De parser werkt bottom-up. De invoer wordt gelezen totdat een productieregel wordt herkend, die dan vervangen wordt door een indicatie van de door die regel voorgestelde notie, meestal na aanroep van een semantische routine. Soms is aan deze notie algemene informatie verbonden, die door de semantische actie gerefereerd of bepaald wordt.

De parser levert een voorstelling van de productieboom van het programma in omgekeerde Poolse notatie, genaamd IL2. Een notie wordt in IL2 voorgesteld door een operator, zijn directe zijtakken door de bijbehorende operanden. Operanden zijn zelf weer operatoren (met operanden) of terminale symbolen. Terminale symbolen zijn getalwaarden of indices in de Naamlijst, die b.v. ranges, modes, identifiers of indicants vertegenwoordigen. Noties, die lijsten of reeksen van operanden bij zich hebben worden voorgesteld door een operator met een variabel aantal operanden. Voorbeelden hiervan zijn de COLL(n)-operator voor een *collateral-clause* met n *units*.

Enige operanden stellen niet een enkele notie voor, maar een gemeenschappelijk deel van verschillende noties. Een voorbeeld is de RANGE-operator, die optreedt overal waar een range wordt gevormd, dus b.v. in

een *choice-clause* of een *while-part*.

Ook komt het voor dat een enkele operator verschillende noties kan voorstellen; de CLOSED-operator b.v. stelt een *closed-clause* of een *collateral-clause* voor. In dergelijke gevallen vindt de verdere analyse plaats in de volgende scan.

4.3. Invoer-routine

De IL1-file wordt gelezen en de gelezen informatie wordt omgezet naar het voor de parser geëigende formaat. Bij iedere aanroep van de invoer-routine vanuit de parser wordt een symbool opgeleverd, soms vergezeld van een of twee parameters welke dan als globale informatie aan de semantische routines kunnen worden doorgegeven.

De behandeling van de onder punt 6 in hoofdstuk 3 genoemde IL1-bytes is als volgt:

1. ALGOL 68 symbolen worden ongewijzigd doorgegeven.
2. Regelnummers worden buiten de parser om behandeld.
3. CORRAL's worden niet aan de parser doorgegeven. De ingelezen CORRAL wordt geopend, d.w.z. de geassocieerde deelacties van indicants worden identificeerbaar gemaakt. Het sluiten van CORRAL's geschiedt d.m.v. in de grammatica aangegeven semantische routines.
4. Identifiers leveren een "TAG" symbool met de NAME-ingang als parameter.
5. Constanten leveren een CONS-symbool met de CONS-ingang als parameter.
6. Indicants worden geïdentificeerd met identificeerbare declaraties en leveren een "mode indicant" of een "operator indicant" met de MODE- of NAME-ingang als parameter.
7. Monad of
8. Nomad levert een "MONAD" of "NOMAD" symbool met de NAME-ingang als parameter.

4.4. Declaraties en ranges

Voor iedere declaratie wordt een ingang in de Naamlijst gegenereerd en wel een

IDEN-ingang	voor	*identity-definition,*
		variable-definition
		parameter

		specification in *conformity-clause*
		for-part of
		label-definition
OPER-ingang	voor	*operation-definition*
MODE-ingang	voor	*mode-declaration*.

Een *priority-definition* geeft slechts dan aanleiding tot een (dummy) OPER-ingang, indien in een prelude-vertaling wel een *priority-declaration*, maar geen *operation-declaration* voor de betrokken operator voorkomt. In alle andere gevallen wordt de prioriteit in de reeds bestaande OPER-ingang vastgelegd, dan wel genegeerd.

De gegenereerde ingangen worden verbonden met de RNGH-ingang van de lopende range. Een RNGH-ingang wordt gegenereerd bij de verwerking van de eerste declaratie in een CORRAL.
De structuur van CORRALS en ranges loopt grotendeels parallel; deze redundantie is nuttig i.v.m. foutherstel-procedures.

4.5. Modes

PASS2 bouwt voor iedere declarer een boomstructuur uit MRW-ingangen. Voor declarers van type PROC, UNION en STRUCT worden bovendien RNGH- en IDEN-, UMEM- en FIELD-ingangen gebruikt.

De belangrijkste velden van een MRW-ingang zijn PIVA, AUXL en MDLK, waarvan de waarden hieronder in tabelvorm worden gegeven.

De RNGH-ingang voor een proc-mode bevat een verwijzing naar de MRW-ingang van de resultaat-mode in een ketting van UMEM-ingangen die de parameters vertegenwoordigen.

UMEM- en FIELD-ingangen bevatten een verwijzing naar de MRW-ingang van de mode van de parameter, constituent of field.

Declarator	PIVA	AUXL	MDLK
int	INT	aantal	0
real	REAL	long of	0
bits	BITS	minus aantal	0
bytes	BYTES	short symbolen	0
void	VOID	0	0
bool	BOOL	0	0
char	CHAR	0	0
1) union of rows	ROWALL	0	0
2) input	INTYPE	0	0
2) output	OUTTYPE	0	0
3) error	ERROR	0	0
ref amode	REF	0	amode
[ ] amode	ROW	aantal dimensies	amode
proc	PROC	aantal parameters	RNGH-ingang
union	UNION	aantal constituenten	UMEM-ingang
struct	STRUCT	aantal velden	FIELD-ingang

1) De union van alle row-modes; dit is de mode van de operanden van de lwb en upb operatoren.
2) De zgn. primitieve transput-modes voor input resp. output.
3) Deze mode wordt gebruikt voor syntactisch incorrecte declarers etc., om propageren van foutmeldingen te voorkomen.

5. MODE-EQUIVALENCING

De mode-equivalencing routine is ondergebracht in de derde overlay, omdat de mode-equivalencing moet worden uitgevoerd tussen de tweede en derde scan en de derde overlay kleiner is dan de tweede.

De gebruikte algoritme is gebaseerd op het proefschrift van Mary Zosel [2].

Alle modes worden ingedeeld in klassen, die worden opgesplitst in niet-equivalente klassen, totdat iedere klasse equivalente modes bevat.

Dit proces wordt uitgevoerd in 5 stappen:

1. Alle modes worden ingedeeld in een klasse op basis van het PIVA-veld van de MRW-ingang.
2. De klassen met PIVA = INT,REAL,BITS,BYTES,ROW,PROC en STRUCT worden gesplitst op basis van het AUXL-veld van de MRW-ingang.
3. De klassen met PIVA = STRUCT en een gegeven AUXL-veld worden gesplitst op basis van de FIELD-namen.
4. Nummer alle klassen.
   Hergroepeer iedere klasse met PIVA = REF of PIVA = ROW op basis van het nummer van de gerefereerde mode (MDLK-veld).
   Hergroepeer iedere klasse met PIVA = PROC op basis van de nummers van de klassen van de modes van resultaat en parameters.
   Hergroepeer iedere klasse met PIVA = STRUCT op basis van de nummers van de klassen van de modes van de FIELD's.
   Hergroepeer iedere klasse met PIVA = UNION op basis van de (gesorteerde) klassen van de modes van de constituenten.
5. Herhaal stap 4 totdat geen nieuwe klassen ontstaan.

Komt een verzameling modes in verschillende vertalingen voor, dan wordt deze verzameling in iedere vertaling op dezelfde wijze geordend, ongeacht de andere modes, die buiten deze verzameling nog voorkomen. Dit betekent dat voor iedere union de constituenten altijd op dezelfde wijze worden genummerd, waardoor afzonderlijke vertaling van preludes en routines mogelijk wordt.

6. MODE CHECKING EN COERCIES

De semantische scan verwerkt de IL2-file in achterwaartse richting en leest dus effectief het programma achterstevoren in prefix Poolse notatie. De resulterende IL3-file wordt weer geschreven in postfix Poolse notatie, zodat de vierde scan de tekst in voorwaartse richting in prefix Poolse notatie kan lezen door de IL3-file achterwaarts te lezen.

De verwerking van de invoer-file IL2 is georganiseerd per IL2-operator. Voor iedere IL2-operator bevat de semantische scan 3 afzonderlijke routines, die geactiveerd worden respectievelijk bij het inlezen van de operator, na het verwerken van iedere niet-laatste operand en na het verwerken van de laatste operand.

Tijdens deze scan worden twee afzonderlijke stapels bijgehouden, nl.

de operator-stapel en de operand-stapel. De operator-stapel bevat voor iedere IL2-operator waarvan de verwerking gestart, maar niet beëindigd, is o.a. de volgende informatie: IL2-operator, het aantal operanden in de IL2-file, het aantal operanden in de operand-stapel en een operand-teller.

De operand-stapel bevat voor iedere IL2-operand een stapel-ingang van vaste lengte, waarin o.a. de volgende informatie wordt bijgehouden:

. Type o.a. collateral, balance, denotation, identifier, nil, skip, formula.
. Het aantal sub-operanden (voor operanden van type collateral of balance).
. Een index in de Naamlijst (voor operanden van type identifier of denotation).
. De mode van de (sub)-operand.
. Het adres op de IL3-file van de (sub)-operand.

Voor operanden van type collateral of balance worden de ingangen voor de suboperanden na de hoofd-ingang op de stapel geplaatst. Daar een sub-operand weer een collateral of balance kan zijn is de structuur van de stapel recursief gedefinieerd.

De eigenlijke mode-checking vindt plaats in de coercie-routine, een logische functie met drie parameters: context (strong, firm, meek, weak of soft), de index van de te coerceren operand in de operand-stapel en de a posteriori mode. De coercie-routine levert true als de operand in de gegeven context naar de a posteriori mode gecoerceerd kan worden; in dat geval worden de bijbehorende coercies op de operator-stapel geplaatst. Een hulproutine is beschikbaar om de zo gevonden coercies op de IL3-file uit te voeren.

De coercie-routine is recursief omdat de te coerceren operanden een recursieve structuur bezitten. De recursie-stapel wordt op de operand-stapel bijgehouden.

Dereferencing coercies geven aanleiding tot een volgend niveau van recursie, omdat in de uit te voeren coercie-ingangen het copiëren van iedere multipele waarde, welke een component is van een multipele waarde, afzonderlijk wordt aangegeven. De dereference-recursie-stapel wordt op de coercie-recursie-stapel geplaatst en de resulterende coercie-ingangen op de operator-stapel.

De twee stapels bevatten dus:

1) operator-stapel, coercies en dereference-coercies, en

2) operand-stapel, coercie-recursie-, dereference-coercie-stapel.

In de coercie-routine worden de volgende typen coercie-ingangen onderscheiden:

Type	
DEPROC	Deproceduring
EMPTY	SKIP naar willekeurige mode
ROWDIS1	row-display van niet-multipele waarden
ROWDIS2	row-display van multipele waarden
ROWD1	Rowing van een multipele waarde naar een multipele waarde van hogere dimensie
ROWD2	Rowing van een referentie naar een multipele waarde naar een referentie naar een multipele waarde van hogere dimensie
ROW1	Rowing van een niet-multipele waarde
ROW2	Rowing van een referentie naar een niet-multipele waarde
STRUDIS	Structure display
UNITE1	Uniting van een non-united waarde
UNITE2	Uniting van een waarde van united mode
UNROWALL	Coerceert een union van multipele waarden naar de mode ROWALL
VACUUM	Creëert een vacuum
VOID	Voiding
WIDEN1	L int naar L real
WIDEN2	L int naar L compl
WIDEN3	L real naar L compl
WIDEN4	L bits naar [ ] bool
WIDEN5	L bytes naar [ ] char
DEREF	Dereferencing
DEREF2	Secundaire ingang, die copiëren van een multipele waarde aangeeft

De coercie-algoritme bestaat uit de volgende stappen:

1. Initialisatie van recursie-stapel en coercie-stapel.
2. Is de te coerceren operand van type BALANCE, pas dan de algoritme recursief toe op de sub-operanden, rekening houdend met de context.
3. Is de te coerceren operand van type COLLATERAL, dan
   a. Is de a posteriori mode VOID, genereer een VOID-coercie;
   b. Is de a posteriori mode een gestructureerde mode, genereer dan een STRUDIS-coercie en pas de algoritme recursief toe op de sub-operanden

met context STRONG en de mode van de corresponderende FIELD van de STRUCT;

c. Is de a posteriori mode [ ] amode, genereer een ROWDIS1-coercie en pas de algoritme recursief toe op de sub-operanden met context STRONG en amode als a posteriori mode;

d. Lever resultaat false op; de coercie is niet mogelijk.

4. Behandel speciale typen operanden, zoals jump, skip, nil, vacuum.
5. Behandel de a posteriori mode ROWALL.
6. Merk de MRW-ingangen van alle modes, die door herhaalde dereferencing en/of deproceduring van de a priori mode bereikt kunnen worden.
7. Als de a posteriori mode in stap 6 gemerkt is, dan is de coercie mogelijk. Ga dan naar stap 11.
8. Indien de context STRONG is:
   test of de a posteriori mode door widening van een gemerkte mode bereikt wordt. Zo ja, genereer de geëigende widening-coercie en ga naar stap 11.
9. Indien de context STRONG is, test of de a posteriori mode van de vorm [ ] amode of ref [ ] amode is. Zo ja, genereer de geëigende coercie, vervang de a posteriori mode door amode of ref amode en ga terug naar stap 7.
10. Indien de context STRONG of FIRM is, test of de a posteriori mode een union is. Zo ja, test of de a posteriori mode door uniting van een der gemerkte modes mogelijk is. Zo ja, genereer de geëigende coercie en ga naar stap 11. Zo nee, dan is de coercie niet mogelijk.
11. Genereer de nodige dereferencing- en deprocedering-coercies en verwijder de in stap 6 gezette merktekens.

De gegenereerde coercie-ingangen bevatten alle het adres op de IL3-file van de operand waarop de coercie moet worden toegepast. Het uitschrijven van de coercies naar de IL3-file geschiedt niet in de coercie-routine, maar in de eigenlijke semantische scan, omdat in sommige gevallen, b.v. operator-identificatie, een coercie pas definitief geaccepteerd wordt als meerdere operanden met success gecoerceerd kunnen worden.

Verdere taken van de semantische scan zijn o.a. identificatie van identifiers en operatoren, controle op verwante operatoren in één range en de prioriteit-afhankelijke ontleding van formules.

Identificatie van identifiers is uiterst eenvoudig: bij binnengaan van

een range worden de IDEN-ingangen, die de declaraties in die range voorstellen, in een lijst gehangen, waarvan het begin in de NAME-ingang is geschreven. Bij verlaten van de range worden deze IDEN-ingangen uit de lijst verwijderd. De eerste IDEN-ingang in de lijst stelt dus altijd de gezochte declaratie voor.

Voor operatoren wordt een dergelijke techniek toegepast, maar in dit geval wordt de lijst van OPER-ingangen in de lijst afgezocht naar een operator met passende operand-modes.

In een latere versie van de vertaler zijn zgn. geoptimaliseerde variabelen geïmplementeerd, d.w.z. voor variabelen, waarvoor dit mogelijk is, wordt de waarde in plaats van de referentie in de stapel geplaatst. Omdat in deze implementatie alle referenties naar de heap wijzen, mag een (impliciete) referentie slechts dan naar de stapel wijzen, als geen overtreding van de scope-regels mogelijk is.

Een variabele kan slechts dan geoptimaliseerd worden als zij slechts gebruikt wordt als

a. doel van een assignatie
b. coercend van dereferencing
c. actuele parameter, als de corresponderende formele variabele geoptimaliseerd kan worden.

I.v.m. de laatste conditie worden alle paren van actuele en formele parameters in een optimalisatie-tabel geplaatst. Na verwerking van de gehele brontekst wordt m.b.v. deze tabel het niet-optimaliseerbaar predicaat gepropageerd.

LITERATUUR

[1] VAN WIJNGAARDEN, A. VAN, et al, *Revised Report on the Algorithmic Language ALGOL 68*, Acta Informatica 5 (1975), pp. 1-236.

[2] ZOSEL, M.E., *A formal grammar for the representation of modes and its application to ALGOL 68*, Ph.D. Thesis, Computer Science Group, Un. of Washington, 1971.

[3] GOOS, G., *Some problems in compiling ALGOL 68*, in: ALGOL 68 Implementation (J.E.L. Peck, ed.), Proceedings of the IFIP Working Conference on ALGOL 68 Implementation, Münich, July 20-24, 1970, North-Holland Publ. Cy., 1971.

UITGAVEN IN DE SERIE MC SYLLABUS

Onderstaande uitgaven zijn verkrijgbaar bij het Mathematisch Centrum, 2e Boerhaavestraat 49 te Amsterdam-1005, tel. 020-947272.

MCS 1.1 F. GÖBEL & J. VAN DE LUNE, *Leergang Besliskunde, deel 1: Wiskundige basiskennis*, 1965. ISBN 90 6196 014 2.

MCS 1.2 J. HEMELRIJK & J. KRIENS, *Leergang Besliskunde, deel 2: Kansberekening*, 1965. ISBN 90 6196 015 0.

MCS 1.3 J. HEMELRIJK & J. KRIENS, *Leergang Besliskunde, deel 3: Statistiek*, 1966. ISBN 90 6196 016 9.

MCS 1.4 G. DE LEVE & W. MOLENAAR, *Leergang Besliskunde, deel 4: Markovketens en wachttijden*, 1966. ISBN 90 6196 017 7.

MCS 1.5 J. KRIENS & G. DE LEVE, *Leergang Besliskunde, deel 5: Inleiding tot de mathematische besliskunde*, 1966. ISBN 90 6196 018 5.

MCS 1.6a B. DORHOUT & J. KRIENS, *Leergang Besliskunde, deel 6a: Wiskundige programmering 1*, 1968. ISBN 90 6196 032 0.

MCS 1.6b B. DORHOUT, J. KRIENS & J.TH. VAN LIESHOUT, *Leergang Besliskunde, deel 6b: Wiskundige programmering 2*, 1977. ISBN 90 6196 150 5.

MCS 1.7a G. DE LEVE, *Leergang Besliskunde, deel 7a: Dynamische programmering 1*, 1968. ISBN 90 6196 033 9.

MCS 1.7b G. DE LEVE & H.C. TIJMS, *Leergang Besliskunde, deel 7b: Dynamische programmering 2*, 1970. ISBN 90 6196 055 X.

MCS 1.7c G. DE LEVE & H.C. TIJMS, *Leergang Besliskunde, deel 7c: Dynamische programmering 3*, 1971. ISBN 90 6196 066 5.

MCS 1.8 J. KRIENS, F. GÖBEL & W. MOLENAAR, *Leergang Besliskunde, deel 8: Minimaxmethode, netwerkplanning, simulatie*, 1968. ISBN 90 6196 034 7.

MCS 2.1 G.J.R. FÖRCH, P.J. VAN DER HOUWEN & R.P. VAN DE RIET, *Colloquium Stabiliteit van differentieschema's, deel 1*, 1967. ISBN 90 6196 023 1.

MCS 2.2 L. DEKKER, T.J. DEKKER, P.J. VAN DER HOUWEN & M.N. SPIJKER, *Colloquium Stabiliteit van differentieschema's, deel 2*, 1968. ISBN 90 6196 035 5.

MCS 3.1 H.A. LAUWERIER, *Randwaardeproblemen, deel 1*, 1967. ISBN 90 6196 024 X.

MCS 3.2 H.A. LAUWERIER, *Randwaardeproblemen, deel 2*, 1968. ISBN 90 6196 036 3.

MCS 3.3 H.A. LAUWERIER, *Randwaardeproblemen, deel 3*, 1968. ISBN 90 6196 043 6.

MCS 4 H.A. LAUWERIER, *Representaties van groepen*, 1968. ISBN 90 6196 037 1.

MCS 5 J.H. VAN LINT, J.J. SEIDEL & P.C. BAAYEN, *Colloquium Discrete wiskunde*, 1968.
ISBN 90 6196 044 4.

MCS 6 K.K. KOKSMA, *Cursus ALGOL 60*, 1969. ISBN 90 6196 045 2.

MCS 7.1 *Colloquium Moderne rekenmachines, deel 1*, 1969. ISBN 90 6196 046 0.

MCS 7.2 *Colloquium Moderne rekenmachines, deel 2*, 1969. ISBN 90 6196 047 9.

MCS 8 H. BAVINCK & J. GRASMAN, *Relaxatietrillingen*, 1969.
ISBN 90 6196 056 8.

MCS 9.1 T.M.T. COOLEN, G.J.R. FÖRCH, E.M. DE JAGER & H.G.J. PIJLS, *Elliptische differentiaalvergelijkingen, deel 1*, 1970.
ISBN 90 6196 048 7.

MCS 9.2 W.P. VAN DEN BRINK, T.M.T. COOLEN, B. DIJKHUIS, P.P.N. DE GROEN, P.J. VAN DER HOUWEN, E.M. DE JAGER, N.M. TEMME & R.J. DE VOGELAERE, *Colloquium Elliptische differentiaalvergelijkingen, deel 2*, 1970.
ISBN 90 6196 049 5.

MCS 10 J. FABIUS & W.R. VAN ZWET, *Grondbegrippen van de waarschijnlijkheidsrekening*, 1970. ISBN 90 6196 057 6.

MCS 11 H. BART, M.A. KAASHOEK, H.G.J. PIJLS, W.J. DE SCHIPPER & J. DE VRIES, *Colloquium Halfalgebra's en positieve operatoren*, 1971.
ISBN 90 6196 067 3.

MCS 12 T.J. DEKKER, *Numerieke algebra*, 1971. ISBN 90 6196 068 1.

MCS 13 F.E.J. KRUSEMAN ARETZ, *Programmeren voor rekenautomaten; De MC ALGOL 60 vertaler voor de EL X8*, 1971. ISBN 90 6196 069 X.

MCS 14 H. BAVINCK, W. GAUTSCHI & G.M. WILLEMS, *Colloquium Approximatietheorie*, 1971. ISBN 90 6196 070 3.

MCS 15.1 T.J. DEKKER, P.W. HEMKER & P.J. VAN DER HOUWEN, *Colloquium Stijve differentiaalvergelijkingen, deel 1*, 1972. ISBN 90 6196 078 9.

MCS 15.2 P.A. BEENTJES, K. DEKKER, H.C. HEMKER, S.P.N. VAN KAMPEN & G.M. WILLEMS, *Colloquium Stijve differentiaalvergelijkingen, deel 2*, 1973. ISBN 90 6196 079 7.

MCS 15.3 P.A. BEENTJES, K. DEKKER, P.W. HEMKER & M. VAN VELDHUIZEN, *Colloquium Stijve differentiaalvergelijkingen, deel 3*, 1975.
ISBN 90 6196 118 1.

MCS 16.1 L. GEURTS, *Cursus Programmeren, deel 1: De elementen van het programmeren*, 1973. ISBN 90 6196 080 0.

MCS 16.2 L. GEURTS, *Cursus Programmeren, deel 2: De programmeertaal ALGOL 60*, 1973. ISBN 90 6196 087 8.

MCS 17.1 P.S. STOBBE, *Lineaire algebra, deel 1*, 1974. ISBN 90 6196 090 8.

MCS 17.2 P.S. STOBBE, *Lineaire algebra, deel 2*, 1974. ISBN 90 6196 091 6.

MCS 17.3 N.M. TEMME, *Lineaire algebra , deel 3*, 1976. ISBN 90 6196 123 8.

MCS 18 F. VAN DER BLIJ, H. FREUDENTHAL, J.J. DE IONGH, J.J. SEIDEL & A. VAN WIJNGAARDEN, *Een kwart eeuw wiskunde 1946-1971, Syllabus van de Vakantiecursus 1971*, 1974. ISBN 90 6196 092 4.

MCS 19 A. HORDIJK, R. POTHARST & J.Th. RUNNENBURG, *Optimaal stoppen van Markovketens*, 1974. ISBN 90 6196 093 2.

MCS 20 T.M.T. COOLEN, P.W. HEMKER, P.J. VAN DER HOUWEN & E. SLAGT, *ALGOL 60 procedures voor begin- en randwaardeproblemen*, 1976. ISBN 90 6196 094 0.

MCS 21 J.W. DE BAKKER (red.), *Colloquium Programmacorrectheid*, 1975. ISBN 90 6196 103 3.

MCS 22 R. HELMERS, F.H. RUYMGAART, M.C.A. VAN ZUYLEN & J. OOSTERHOFF, *Asymptotische methoden in de toetsingstheorie; Toepassingen van naburigheid*, 1976. ISBN 90 6196 104 1.

MCS 23.1 J.W. DE ROEVER (red.), *Colloquium Onderwerpen uit de biomathematica, deel 1*, 1976. ISBN 90 6196 105 X.

MCS 23.2 J.W. DE ROEVER (red.), *Colloquium Onderwerpen uit de biomathematica, deel 2*, 1976. ISBN 90 6196 115 7.

MCS 24.1 P.J. VAN DER HOUWEN, *Numerieke integratie van differentiaalvergelijkingen, deel 1: Eenstapsmethoden*, 1974. ISBN 90 6196 106 8.

MCS 25 *Colloquium Structuur van programmeertalen*, 1976. ISBN 90 6196 116 5.

MCS 26.1 N.M. TEMME (ed.), *Nonlinear analysis, volume 1*, 1976. ISBN 90 6196 117 3.

MCS 26.2 N.M. TEMME (ed.), *Nonlinear analysis, volume 2*, 1976. ISBN 90 6196 121 1.

MCS 27 M. BAKKER, P.W. HEMKER, P.J. VAN DER HOUWEN, S.J. POLAK & M. VAN VELDHUIZEN, *Colloquium Discretiseringsmethoden*, 1976. ISBN 90 6196 124 6.

MCS 28 O. DIEKMANN, N.M. TEMME (EDS), *Nonlinear Diffusion Problems*, 1976. ISBN 90 6196 126 2.

MCS 29.1 J.C.P. BUS (red.), *Colloquium Numerieke programmatuur, deel 1A, deel 1B*, 1976. ISBN 90 6196 128 9.

MCS 29.2 H.J.J. TE RIELE (red.), *Colloquium Numerieke programmatuur, deel 2*, 1976. ISBN 144 0.

* MCS 30 P. GROENEBOOM, R. HELMERS, J. OOSTERHOFF & R. POTHARST, *Efficiency begrippen in de statistiek*, . ISBN 90 6196 149 1.

MCS 31 J.H. VAN LINT (red.), *Inleiding in de coderingstheorie*, 1976. ISBN 90 6196 136 X.

MCS 32 L. GEURTS (red.), *Colloquium Bedrijfssystemen*, 1976. ISBN 90 6196 137 8.

MCS 33 P.J. VAN DER HOUWEN, *Differentieschema's voor de berekening van waterstanden in zeeën en rivieren*, 1977. ISBN 90 6196 138 6.

MCS 34 J. HEMELRIJK, *Oriënterende cursus mathematische statistiek*, ISBN 90 6196 139 4.

MCS 35 P.J.W. TEN HAGEN (red.), *Colloquium Computer Graphics*, 1977. ISBN 90 6196 142 4.

MCS 36 J.M. AARTS, J. DE VRIES, *Colloquium Topologische Dynamische Systemen*, 1977. ISBN 90 6196 143 2.

MCS 37 J.C. van Vliet (red.), *Colloquium Capita Datastructuren*, 1978. ISBN 90 6196 159 9.

MCS 38.1 T.H. KOORNWINDER (ED.), *Representations of locally compact groups with applications*, 1979. ISBN 90 6196 161 0.

MCS 38.2 T.H. KOORNWINDER (ED.), *Representations of locally compact groups with applications*, 1979. ISBN 90 6196 181 5.

MCS 39 O.J. VRIEZE & G.L. WAANROOIJ, *Colloquium Stochastische spelen, 1978*. ISBN 90 6196 167 X.

MCS 40 J. VAN TIEL, *Convexe Analyse*, 1979. ISBN 90 6196 187 4.

MCS 41 H.J.J. TE RIELE (ED.), *Colloquium Numerical Treatment of Integral Equations*, 1979. ISBN 90 6196 189 0.

MCS 42 J.C. VAN VLIET (RED.), *Colloquium Capita Implementatie Programmeertalen*, 1980. ISBN 90 6196 191 2.

De met een * gemerkte uitgaven moeten nog verschijnen.