INHOUD

Voorwoord

Het vak Struktuur van Programmeertalen, vakkode 211050, verkeert d.d. najaar 1983 nog in een experimentele fase. Voor u liggen de aantekeningen van een groot gedeelte van de stof die op de kursussen in voor- en najaar van 1983 behandeld is. Ingrijpende wijzigingen qua inhoud en vormgeving zijn niet uitgesloten. Alle op- en aanmerkingen zijn van harte welkom; zij kunnen van invloed zijn op de uiteindelijke vaststelling van dit vak.

De doelstelling van dit vak luidt: verdieping en uitbreiding van de kennis van de struktuur van programmeertalen, opdat het aanleren van nieuwe (toekomstige) talen alsmede het vergelijken van talen en ook het programmeren vergemakkelijkt wordt. Lange tijd waren er hiervoor geen goede leerboeken. Daarom ben ik enige jaren terug begonnen aan de hand van eigen aantekeningen dit vak te geven. Maar bijna tegelijkertijd zijn er boeken verschenen, die mijns inziens wel aan de doelstelling toekomen. Met name zijn dat (Tennent 1981), (Ghezzi & Jazayeri 1982) en (Bauer & Wössner 1982). Van deze drie is de laatste minder geschikt voor zelfstudie door de onoverzichtelijke presentatie en de overweldigende hoeveelheid van informatie. De eerste twee zijn wel geschikt; met name (Ghezzi & Jazayeri 1982) bevat veel informatie over bestaande programmeertalen en is zeer leesbaar.

Waarom dan toch een eigen syllabus? Ten eerste vind ik dat het aspekt van typering in genoemde boeken onbevredigend (onvoldoende en soms onjuist) wordt behandeld. Ten tweede wordt er mijns inziens veel te weinig aandacht aan beschrijvende (=funktionele, applikatieve) programmeertalen besteed. Het valt te verwachten dat in de toekomst beschrijvende programmeertalen een hoge vlucht nemen. Daarenboven kunnen ze een geweldige steun bieden tijdens het ontwerpproces van gebiedende (=imperatieve, konventionele) programma's. Enig inzicht in de struktuur van dergelijke talen (of subtalen) is dus gewenst.

Een ander aspekt waarin deze syllabus met (Tennent 1981) en (Ghezzi & Jazayeri 1982) verschilt is het volgende. Zij gaan uit van bestaande programmeertalen en presenteren zo mogelijk geïdealiseerde versies daarvan. Maar wij gaan uit van een ideale*) notatie en presenteren zonodig beperktere versies daarvan; die beperkingen zijn soms nodig voor een doelmatige verwerking van de programma's op de hedendaagse machines.

---

*) niet qua syntactische vormgeving, wel qua semantische inhoud.

In deze syllabus zullen wij allereerst de struktuur van beschrijvende talen uiteenzetten en twee implementatiemodellen schetsen, en vervolgens steeds meer gebiedende aspekten toevoegen. Het onderwerp typering wordt elders uitvoerig behandeld en komt hier in het geheel niet aan bod. Het verdient aanbeveling om naast deze syllabus ook genoemde boeken te raadplegen, en om in een beschrijvende taal te programmeren.

Ik dank Gerrit van der Hoeven voor leerzame gesprekken, Ed Brinksma voor uitvoerig kommentaar op de eerste versie van deze syllabus, Joost Engelfriet voor vele suggesties tot verbetering in de Inleiding en Hl t/m 4, Hans van Berne en Wim van Oosterom voor diverse opmerkingen en Jolanda van Laar voor het prima type-werk en de vele uren die zij heeft besteed aan het herstel van mijn onzorgvuldigheden.

Enschede, juni 1984

Bij de tweede druk

Behoudens de herschrijving van enkele passages zijn er alleen notationele en opmaak-wijzigingen aangebracht.

Enschede, januari 1985

Notationele en andere afspraken.

1. Algoritmen. Om onze informatika-afkomst niet te verloochenen zullen we algoritmen in een soort programmeertaal presenteren. De gebruikte programmeertaal behoeft over het algemeen geen uitleg, hopen we; het is een soort pseudo-Pascal, of pseudo-Algol. Voorts gebruiken we de zg. guarded commands en guarded expressions: (B staat voor Boolean, E voor expressie, S voor Statement)

if B1 -> E1 | --- | Bn -> En fi

if B1 -> S1 | --- | Bn -> Sn fi

do B1 -> S1 | --- | Bn -> Sn od

De evaluatie van een if-konstruktie vereist dat tenminste een der Bi's tot true evalueert; een bijbehorende Ei resp. Si wordt dan vervolgens geevalueerd. De evaluatie van een do-konstruktie gaat als volgt. Indien alle Bi's tot false evalueren is de evaluatie voltooid; zo niet, dan wordt een of andere Si geevalueerd (waarvan de Bi tot true evalueert), en wordt daarna dit proces herhaald. De do-konstruktie is dus een nondeterministische veralgemening van de while-konstruktie.

De bewakingen (guards) Bi mogen willekeurige kondities zijn. Wij zullen echter ook de volgende vormen gebruiken voor een bewaakte expressie of statement:

expr ? = --idf--idf'-- -> ----idf---idf'---

De test slaagt als er waarden zijn voor de identifiers in het rechterlid (hier aangeduid met idf, idf') zodat de gelijkheid geldt; in dat geval duiden die identifiers rechts van de pijl desbetreffende waarden aan. Bijvoorbeeld, als x en y natuurlijke getallen moeten zijn, dan slaagt de test van

4+5 ?= (3 * x)+y -> x+y

met x=0,y=9 of x=1,y=6 of x=2,y=3 of x=3,y=0 en is de uitkomst dus 9 of 7 of 5 of 3.

2. Afbeelding versus Funktie. Wanneer wij het woord afbeelding gebruiken, dan bedoelen wij het funktiebegrip zoals dat in de wiskunde geldig is. Het woord funktie duidt een proces, voorschrift of berekeningsmethode aan dat bij gegeven

argumenten een resultaat oplevert. Vaak is een funktie ook een afbeelding, maar soms is dat heel moeilijk in te zien: formeel is een afbeelding f bepaald door twee (wiskundig goed gedefiniëerde) verzamelingen A en B en een relatie R deel van A x B zo dat

$$(x,y1):R,\ (x,y2):R \quad \Rightarrow \quad y1=y2$$

En wanneer (x,y):R dan zeggen en schrijven we f(x)=y. Wanneer wij klakkeloos funkties met afbeeldingen identificeren krijgen we een paradox; het vermijden hiervan vergt een niet-triviale wiskundige theorie, zie (Scott 1976).

3. Expressies. We gebruiken het woord expressie als synoniem voor programma, programmafragment, eenheid van uitdrukking, enzovoorts. We sluiten dus niet bij voorbaat neveneffekten uit wanneer we over expressies spreken. Eveneens gebruiken we de woorden evaluatie, exekutie, elaboratie en reduktie als synoniemen.

4. Metavariabelen. We gebruiken de letters x, y en z als (meta)variabele over X, de verzameling van zogenaamde variabelen. Dat wil zeggen, x en y en z duiden altijd een element van X aan. Net zo laten we e variëren over E, de verzameling van expressies; e is dus een (meta)variabele over E. Ook versieren we e met achtervoegsels en aksenten: e, e', e", ef, ea, eb, ... De achtervoegsels f, a, b, g, l zijn mnemonisch voor: funktie, argument, body (=romp), groep, lijst. Verder is k een metavariabele over K, de zogenaamde konstanten; dat zijn bijzondere expressies. Voorts zijn m en n metavariabelen over de verzameling natuurlijke getallen {0,1,2, ...}. Met m, n en m+n enzovoorts duiden wij de decimale notatie van de getallen m, n, m+n enzovoorts aan.

5. Oefeningen. De oefeningen gemerkt met het uitroepteken ! dienen als hulp bij de bestudering van de leerstof. Beheersing van dat soort oefeningen is noodzakelijk (en voldoende) voor een voldoende beoordeling bij tentaminering over deze syllabus. De overige oefeningen vragen (te) veel werk, gaan (te) diep in de leerstof of vergen een (te) grote vindingrijkheid.

INLEIDING

De doelstelling van dit vak luidt: verdieping en uitbreiding van de kennis van de struktuur van programmeertalen, opdat het aanleren van nieuwe (toekomstige) talen alsmede het vergelijken van talen en ook het programmeren vergemakkelijkt wordt. Met "struktuur" wordt zoiets bedoeld als "het wezenlijke" of "de kern". Wat niet bedoeld wordt is bijvoorbeeld de schrijfwijzen die er voor eenzelfde begrip in verschillende programmeertalen gangbaar zijn (de syntaxis), of de definitie-methoden voor programmeertalen (grammatikale systemen, axiomatische of denotationele semantiek), of sterk machine-afhankelijke implementatietechnieken. De struktuur van een programmeertaal is dus: die programmeertaal op een voldoend hoog abstraktienivo bezien; syntaktische en implementationele aspekten worden niet beschouwd, en waar mogelijk worden zinvolle veralgemeningen doorgevoerd en beperkingen opgeheven. Aldus zou het wezenlijke te voorschijn moeten komen van begrippen zoals

- variabelen en toekenning
- zichtbaarheidsregels (scope)
- levensduur (extent, dangling references)
- parametermechanismen
- datatype definities (classes, modules)
- typering en typen
- evaluatie-strategiën
- synchronisatie (parallellisme, concurrency, co-operation, communication)
- nondeterminisme
- bewijsregels voor korrektheid

enzovoort.

Daarnaast willen we ook het wezenlijke van de zogenaamde beschrijvende (funktionele of applikatieve) talen uiteenzetten. Deze worden weliswaar nog nauwelijks gebruikt, maar onze verwachting is dat dat in de toekomst zeker wel het geval zal zijn. We proberen twee vliegen in een klap te slaan door aan de hand van een beschrijvende taal zoveel mogelijk van bovenstaande, en andere, begrippen toe te lichten.

We gaan als volgt te werk. We definiëren steeds een minitaal aan de hand

minitalen zijn een uitbreiding of kleine wijziging van de eerste, HOF genaamd. Omdat we inzicht willen krijgen in de struktuur van de taal, staat de doorzichtigheid van de syntaxis voorop, en lijken de talen nogal gebruikersonvriendelijk. (Het gaat er niet om om te programmeren in zo'n taal, maar om te praten over programma's in de taal). De gelijkenis met konventionele talen zoals Pascal en Algol 68 is in eerste instantie ver te zoeken; althans dat lijkt zo. We hopen dat aan het einde van deze syllabus duidelijk is waar de verschillen en overeenkomsten liggen. De gelijkenis met een taal zoals SASL of KRC moet toch wel direkt opvallen.

De onderwerpen synchronisatie en non-determinisme zullen in de syllabus niet aan de orde komen. Meer door ruimte- en tijdgebrek, dan enig andere oorzaak. Het onderwerp typering wordt (voorlopig?) afzonderlijk buiten deze syllabus behandeld.

We vestigen er de nadruk op dat de expressies niet gezien moeten worden als de letterlijke weergave van wat een programmeur schrijft of intikt, maar als het resultaat na ontleding en eventuele type-kontrole van de ingevoerde programmatekst. Expressies zijn dus eigenlijk boomstrukturen met aan de knopen een kenmerk voor de samenstellingswijze, aan de takken kenmerken voor de onderscheiden samenstellende delen, en aan de bladen alleen variabelen en konstanten. Door het overvloedig gebruik van haakjes kunnen dergelijke boomstrukturen gemakkelijk lineair worden weergegeven. Leesbare lineaire representaties kunnen verkregen worden door allerlei ontledingsafspraken; zoals voorrangsregels (* boven +), associatieregels (- associeert naar links: 12-4-3 = (12-4)-3 en niet 12-(4-3), en opsplitsing van operator-symbolen (if b then t else e en niet ifthenelse (b,t,e)). Wij zijn bij 'Struktuur van Programmeertalen' niet geïnteresseerd in methoden om dergelijke "abstrakte" expressies konkreet weer te geven, en daarbij leesbaarheid, beknoptheid enzovoorts te bereiken. De afspraken die wij zullen maken voor de lineaire weergave zijn vrij willekeurig.

Overigens is de konkrete syntaktische representatie natuurlijk wel medebepalend voor het welslagen van een programmeertaal. De overmaat aan haakjes in LISP wordt bijvoorbeeld vaak als storend ervaren, en een goed gekozen verzameling infix-operatoren doet misschien afbreuk aan de syntaktische eenvoud maar komt het gebruiksgemak zeker ten goede. Deze aspecten liggen in deze syllabus buiten de belangstellingssfeer.

Literatuur bij de Inleiding

Algemene leerboeken over Struktuur (Aspekten, Principes, Koncepten etc) van Programmeertalen zijn (Tennent 1981), (Ghezzi & Jazuayeri 1982), (Bauer & Wössner 1982), (MacLennan 1983), (Ledgard & Marcotty 1981), (Pratt 1975), (Horowitz 1984) en de in deze boeken opgesomde literatuur. Een grote bron van inspiratie voor deze syllabus is (Landin 1966) geweest. (Hororwitz 1983) bevat 24 historische en invloedrijke verhalen van diverse vooraanstaande auteurs. Voorts noemen we (Reynolds 1981b),(Ledgard 1971).

Leerboeken over Funktioneel Programmeren en Funktionele Programmeertalen zijn (Burge 1975) (met een taal die sterk lijkt op SASL) en (Henderson 1980). Het boek (Darlington et al 1982) bevat een aantal interessante verhalen, o.a. (Turner 1982) met een behandeling van KRC. Opmerkelijke funktionele programma's worden gepresenteerd in (Henderson 1982), (Henderson 198?) en (Joosten 1985). Een veel verwezen artikel is (Backus 1978). SASL wordt gedefinieerd in (Turner 1976); een aanverwante taal, TWENTEL, in (van der Hoeven 1984). Er bestaan talloze goede leerboeken over LISP; we noemen slechts (Sussman 1984).

HOOFDSTUK 1

DE PROGRAMMEERTAAL HOF

In dit hoofdstuk definiëren wij de miniprogrammeertaal HOF. Veruit het belangrijkste bestanddeel van HOF is de zogenaamde lambda-expressie; daarmee kunnen Hogere Orde Funkties worden uitgedrukt. Een andere karakteristiek is dat HOF beschrijvend is, ook wel funktioneel of applikatief genoemd; dat wil zeggen dat een HOF-programmatekst in beginsel geen volgorde vastlegt waarin de verscheidene onderdelen van de tekst geëvalueerd moeten worden. Assigneerbare programmavariabelen ontbreken dus, evenals andere vormen van neveneffekten.

In de komende paragrafen zullen we allereerst een informele uitleg geven van lambda-expressies, dan de syntaxis en tenslotte de semantiek van HOF definiëren. Belangrijke begrippen die aan de orde komen zijn: lambda-abstraktie, partiële parameterisatie, binding en substitutie, reduktie (=evaluatie), reduktiestrategie en konfluentie.

Nota bene. Waar wij in deze syllabus fn schrijven, wordt traditioneel en in de literatuur de griekse letter lambda geschreven. Om typografische redenen wijken wij daarvan af. Het verdient aanbeveling om in handgeschreven teksten wel een lambda te schrijven in plaats van fn en eventueel ook een delta in plaats van df.

Par. 1.1 Lambda-expressies

We geven nu een informele uitleg van de vorm en de betekenis van informeel gebruikte lambda-expressies. Eigenlijk is het heel eenvoudig: een lambda-expressie is louter een schrijfwijze voor een totale of partiële afbeelding of funktie, net zoals er schrijfwijzen zijn voor getallen (denk aan de decimale, binaire en romeinse schrijfwijzen). Met een lambda-expressie wordt een afbeelding of funktie aangeduid zonder hem een naam te geven; dit verschijnsel van anonieme funkties komt in programmeertalen en in de wiskunde maar weinig voor. Hier volgen een paar voorbeelden.

1. De totale afbeelding over N gedefinieerd door f(x)=x+x kan worden geno teerd door

fn x. x+x

of fn x:N. x+x

2. De partiële afbeelding over R gedefinieerd door f(x)=17/x kan worden genoteerd door

fn x. 17/x

of fn x:R. 17/x

3. De totale afbeelding over R\{0} gedefinieerd door f(x)= 17/x kan worden genoteerd door

fn x:R\{0}. 17/x

of fn x:x≠0. 17/x

4. Zij V de verzameling van totale afbeeldingen over N. Zij f gedefinieerd door f(g,x)=2*g(x) voor g in V en x in N. Dan kan f worden genoteerd door

fn (g, x). 2*g(x)

of fn (g:V, x:N). 2*g(x)

Zonodig kan ook het bereik van de afbeelding in de notatie worden opgenomen. Maar meestal laten we het bereik en het domein weg uit de notatie, als die uit de kontekst bekend zijn.

In deze voorbeelden is de zinvolheid van de lambda-expressie duidelijk. Maar het is op de voorhand niet zeker of alle lambda-expressies een afbeelding aanduiden, net zoals er opeenvolgingen van symbolen I,V,X,L,C,D,M zijn die in het romeinse stelsel geen getal aanduiden (bijvoorbeeld XMIM). Toch zullen we in HOF ook expressies zoals fn x. x(x) toestaan. Het is de vraag of "de/een afbeelding aangeduid door fn x. x(x)" bestaat, wat zou bijvoorbeeld het domein en het bereik zijn? Zonodig is met typering het gebruik van zulke expressies te voorkomen, zie (Fokkinga 1983). Wanneer we lambda-expressies niet interpreteren als afbeeldingen, maar als funkties ofwel berekeningsvoorschriften, dan leveren dergelijke expressies geen moeilijkheden op: de betekenis van een lambda-expressie is dan een voorschrift hoe bij gegeven argument de berekening voor het resultaat moet geschieden.

De syntaktische handeling om bij gegeven expressie e en naam x (die mogelijk in e voorkomt) de expressie fn x. e te vormen, wordt lambda-abstractie of funktie-abstraktie genoemd, kortweg abstraktie. Het woord abstraktie in deze betekenis heeft niets te maken met abstraktie in de zin van het afzien van sommige aspekten. Naast lambda-abstraktie zijn er ook andere vormen van abstraktie. Bijvoorbeeld de verzamelingsabstraktie: vorm van een predikaatexpressie P en naam x de verzamelingsexpressie {x|P}. Aldus wordt de expressie {x|x is priem and

x>100} verkregen door abstraktie uit ´x is priem and x>100´ met betrekking tot x.

De argumenten en het resultaat van afbeeldingen kunnen zelf ook weer afbeeldingen zijn. En net zo voor funkties. Bijvoorbeeld, differentiatie is een afbeelding die bij argument f de afgeleide f´ oplevert. Dergelijke afbeeldingen worden van hogere orde genoemd. We zouden kunnen definiëren: een afbeelding die geen afbeeldingen als argument of resultaat heeft is van nulde orde; en als het maximum van de orden der argumenten en resultaat n is, dan is de afbeelding zelf van orde n+1. De notatie van hogere orde afbeeldingen geeft geen moeilijkheden. Bijvoorbeeld

fn x. (fn y. x+y)

duidt een afbeelding f aan zodat f(3) = (fn y. 3+y) = de vermeerdering met 3, en dus f(3) toegepast op 4 levert 7. In het algemeen geldt voor die f dat f(x) = (fn y. x+y). Aan dit voorbeeld zien we tevens dat we de afbeelding fn (x,y). x+y met twee veranderlijken kunnen nabootsen door een hogere orde afbeelding met één veranderlijke: voor alle m en n geldt

(fn (x,y). x+y) (m,n) = ((fn x. (fn y. x+y))(m))(n)

Met andere woorden, iedere afbeelding met n veranderlijken kunnen we opvatten als een afbeelding met 1 veranderlijke die als resultaat een afbeelding geeft met n-1 veranderlijken:

(fn (x,y,...z). ---) = fn x. (fn (y,...z). ---)
en (fn (x,y,...z). ---) = fn x. (fn y. ... (fn z. ---)...)

Deze omzetting wordt wel Currying genoemd (naar Curry, die dit als een der eersten formuleerde). Als een programmeertaal toelaat dat funkties met n veranderlijken worden toegepast op slechts een argument, en Currying impliciet plaatsvindt, dan zeggen we dat partiële parameterisatie is toegestaan.

* * *

Het moge inmiddels duidelijk zijn dat lambda-expressies de funkties en procedures uit programmeertalen modelleren. Daarenboven is in de literatuur bewezen dat met lambda-expressies alles geprogrammeerd kan worden wat met behulp van machinaal uitvoerbare algoritmen uitgedrukt kan worden. Ook datatypen zoals de ge-

tallen met de rekenkundige bewerkingen zijn in lambda-expressies uit te drukken; zie Hoofdstuk 6. Bovendien kan alle naamgeving in beginsel verklaard worden in termen van lambda-abstrakties. Naamgeving doet zich voor bij declaraties, definities, parameterisatie, with-konstruktie in Pascal, case- en for- konstrukties in Algol 68, enzovoort. We zullen dit nog tegenkomen. En tenslotte kunnen alle gewenste zichtbaarheidsregelingen (nodig voor het afschermen van aspekten, dus voor "abstraktie") met lambda-expressies worden uitgedrukt; zie Hoofdstuk 9. Om al deze redenen beginnen wij de uiteenzettingen over de struktuur van programmeertalen met de taal HOF waarin de lambda-expressies zonder beperkingen zijn opgenomen.

In de meeste konventionele programmeertalen waaronder Pascal en Algol 68 worden lambda-expressies niet in hun volle algemeenheid toegelaten. De reden daarvoor is dat zij niet met een stapelorganisatie (voor de opslag van waarden tijdens de exekutie) te implementeren zijn. Dit wordt uitgelegd in Hoofdstuk 4.

Par. 1.2 De syntaxis van HOF

Voor de bestudering van een beschrijvende taal met hogere orde funkties zouden we kunnen volstaan met de opname in HOF van alleen lambda-expressies. Dan zou de presentatie verreweg het eenvoudigst zijn. Maar om het programmeren in HOF te vergemakkelijken en de overeenkomst met bestaande talen te verduidelijken zullen we ook andere expressievormen in HOF opnemen. Met name expressievormen voor definities en rekursie, voor getallen en waarheidswaarden, en voor groepen (records, structures) en lijsten. Het zal overigens blijken dat er semantisch geen verschil is tussen groepen en lijsten. De konkrete syntaktische representaties verschillen wel een beetje, maar in abstracto niet. We nemen ze beide op om twee redenen. Ten eerste om de lezer meer voorbeelden te verschaffen van expressievormen en bijhorende syntaktische en semantische handelingen, en ten tweede om later met typering eventueel wel onderscheid te maken: oneindige lijsten staan we dan wel toe maar oneindig geneste groepen niet.

Bij informeel gebruik van HOF expressies zullen we lambda-expressies met verscheidene parameters toelaten, en groepen met verscheidene komponenten, en blokken met verscheidene definities. In de formele uiteenzettingen laten we slechts één parameter per lambda-expressie toe, slechts twee komponenten per groep en slechts één definitie per blok. De veralgemeningen zijn nergens moeilijk, maar zouden de presentatie wel aanzienlijk lastiger maken. Om dezelfde reden hebben groepen geen komponentnamen (field identifiers), maar gebruiken we 1 en 2 als selektoren.

Definitie Expressies

Zij X een (aftelbaar oneindige) verzameling van dingen die we variabelen zullen noemen en als identifiers zullen schrijven. We gebruiken x, y en z als metavariabele over X, (en soms ook wel als element van X). De verzameling E van expressies wordt als volgt met een kontekstvrije grammatika gedefinieerd. We gebruiken e, ef, ea, eb, e1, e2, eg enzovoorts als metavariabele over E.

```
e ::= x | (fn x. e) | (e e)
      | (df x = e. e)
      | (rec x. e)
      | <e, e> | (e.1) | (e.2)
      | 0 | 1 | 2 | ... | (e+e) | (e-e) | e*e) | (e eqn e)
      | true | false | (e and e) | (e or e) | (not e)
      | (if e then e else e)
      | nil | (e: e) | (hd e) | (tl e) | (eqnil e)
```

Terminologie

1. Sommige niet-samengestelde expressies die geen variabelen zijn worden konstanten genoemd. We duiden met K de verzameling konstanten aan, en laten k variëren over K. Voor HOF geldt

```
k ::= 0 | 1 | 2 | ... | true | false | nil
```

2. We noemen (df x = ex. eb) een blok, x = ex zijn definitie, ex zijn definiërende expressie en eb zijn romp; (fn x. eb) een funktie-expressie of abstraktie, x zijn parameter en eb zijn romp; (rec x. eb) een rekursie-expessie en eb zijn romp; <e1, e2> een groepexpressie en e1, e2 zijn komponenten; nil en (eh: et) een lijstexpressie en eh resp. et zijn kop- resp. staartexpressie; (ef ea) een applikatie-expressie en ef resp. ea zijn funktie- resp. argumentdeel; enzovoorts. Als er geen misverstand waarschijnlijk is, zeggen we wel groep in plaats van groepexpressie en funktie in plaats van funktie-expressie, enzovoorts.
3. De samenstellende delen van een expressie die geen blok, funktie- of rekursie-expressie is, noemen we operanden, en de samenstellende symbolen de operator-(symbolen). Dus <,> en .1 en .2 en +, -, *, eqn, and, or, not, if then else, :, hd, tl en eqnil zijn operatorsymbolen. Al naar gelang ze vóór of ná de enige operand staan, of tussen de twee operanden, of verspreid tussen verscheidene operanden, noemen we ze prefix, postfix, infix en distfix operatorsymbolen. (Distfix komt van distributed-fix). (We zullen de df, fn, rec, en konstanten ook wel operatorsymbolen noemen). We zullen ook spreken van de

niet-geschreven applikatie-operator van (ef ea).

4. De zuivere lambda-expressies zijn de expressies voortgebracht door

   e ::= x | (fn x. e) | (e e)

   Desgewenst kunnen we (df x = e. e) daar nog aan toevoegen.
5. Let erop dat in HOF (hd e) geen applikatie is; (hd e) is een expressie met slecht één samenstellende deelexpressie. Als hd een variabele is, is (hd e) wel een applikatie; de samenstellende onderdelen zijn dan de expressies hd en e. Soortgelijks geldt ook voor (tl e), (not e) enzovoorts.

Notatie

Als er geen misverstand waarschijnlijk is laten we de haakjes uit expressies weg. In het bijzonder spreken we daarbij het volgende af.

1. Applikatie associeert naar links, dat wil zeggen (e1 e2 e3) staat voor ((e1 e2) e3) en niet voor (e1 (e2 e3)).
2. De romp van een blok, funktie- en rekursieexpressie moet zo groot mogelijk gekozen worden. Dus (fn x. e1 e2) staat voor (fn x. (e1 e2)) en niet voor ((fn x. e1) e2) en (fn x. fn y. eb) staat voor (fn x. (fn y. eb)).
3. De postfix operatoren hebben een grotere prioriteit dan de prefix operatoren, en die hebben weer voorrang boven de infix operatoren. Dus (hd e1.2 : e2) staat voor ((hd (e1.2)) : e2) en niet voor ((hd e1).2 : e2) of (hd (e1.2 : e2)). Applicatie heeft de hoogste voorrang: f n+1 staat voor (f n)+1 en niet f (n+1).
4. De infixoperator : (spreek uit: op kop van) associeert naar rechts, de overige infixoperatoren naar links.

Voorts gebruiken we de volgende afkortingen

(fn x y z. eb)	voor	(fn x. fn y. fn z. eb) i.e. (fn x.(fn y.(fn z. eb)))
(df x=ex; y=ey; z=ez. eb)	voor	(df x=ex. df y=ey. df z=ez. eb)
(df x rec= ex. eb)	voor	(df x = (rec x. ex). eb)

Als m en n getallen aanduiden, dan noteren we met m, n en m+n enzovoorts de decimale notatie van die getallen. Dus (3+4) is een samengestelde expressie, maar (3+4) is een notatie voor de konstante (enkelvoudige expressie) 7.

Oefening 1.1! Welke van de volgende symboolrijen stellen HOF-expressies voor? Geef zo mogelijk ook de onverkorte notatie.

a. (df x=0. (df y=1. df x=2). df f=(fn z. x). f x)
b. (df x=0. df x=(df y=1. 2). df f= (fn z. x). f x
c. (df x= (df x= (df y=1. 2). 0). df f= (fn z. x). f x)

d. (df x= (df x= (df y=1. 2)). df f=fn z.x. f x)
e. (fn x y x. eb 1 2) 3
f. fn x. (fn y. fn x. eb 1 2) 3
g. fn x. ((fn y. fn x. eb 1 2) 3)

-Noteer de volgende HOF-expressie zo kort mogelijk en geef ook de volledig onverkorte notatie.

h. fn z. (z fnx. fn y. y ((fn x. fn y. fn z. ((y ((x y) z))) (x (fn x y. x)))
(x (fn x. fn y. x))) (fnz. z (fn x y. y) (fn x. fn y. y))) (fn x. fn y. y)
[Dit is een zinvolle expressie; zie Oefening 6.6]

Informele semantiek

De informele semantiek zal voor de meeste expressies wel duidelijk zijn. De expressies (fn x. eb) en (ef ea) zijn in Par. 1.1 toegelicht. Een blok (df x = ea. eb) duidt de waarde van eb aan, waarbij de in eb voorkomende x'en staan voor (de waarde van) ea. Rekursie-expressies duiden rekursief gedefinieerde objekten aan, zonder ze een naam te geven (net zo als abstrakties anonieme funkties aanduiden). Bijvoorbeeld, het zal blijken dat (rec x. 1:x) staat voor de (oneindige!) lijst x die voldoet aan x = 1:x, dus voor de lijst 1:1:1: ... Het is overigens de vraag of (rec x. 1:x) wel tot eindigende berekeningen leidt; hier komen we nog uitgebreid op terug. De interpretatie van de overige expressies is hopelijk duidelijk. Merk overigens op dat er géén algemeen toepasbare gelijkheidstest is. Er zijn alleen de gelijkheidstesten eqn (equal number) en eqnil (equal to nil). In het algemeen kunnen funkties als waarde van expressies optreden (of als onderdeel van zo'n waarde) en een gelijkheidstest voor funkties is niet machinaal realiseerbaar. Immers, twee funkties zijn gelijk als zij voor willekeurig argument gelijke resultaten opleveren; er is geen algoritme die dit aan de hand van funktie-expressies kan beslissen.

Meervoudige expressies Omwille van de leesbaarheid van voorbeelden gebruiken we ook meervoudige variabelen-introdukties en meervoudige expressies. Dus extra expressievormen worden gedefinieerd als volgt

```
e::= ...
   | (fn (x1, ... xn). eb) | (ef (ea1, ... ean))
   | (df (x1, ... xn) = (ea1, ... ean). eb)
   | (rec (x1, ... xn). (eb1, ... ebn))
   | enzovoorts
```

Wanneer n=0, is (e1, ... en) de nulvoudige expressie (), net zo voor (x1,.., xn). Onze uiteenzettingen over HOF-expressies zullen we eenvoudigheidshalve ner-

gens veralgemenen tot bovenstaande vormen; die formalisering laten we over aan de lezer. Bovendien schrijven we (df x=ex, y=ey, z=ez. eb) voor (df (x,y,z) = (ex,ey,ez). eb), en (dfrec x=ex, y=ey, z=ez. eb) voor (df (x,y,z) = (rec (x,y,z). (ex,ey,ez)). eb).

* * *

Tot zover de syntaxis van HOF. Nu definiëren we een paar belangrijke syntaktische begrippen en handelingen: binding, gelijkheid behoudens naamgeving en substitutie.

In het algemeen duidt in de informatika binding het vastleggen aan van semantische eigenschappen van namen. Bijvoorbeeld, x:=3 bindt 3 (dynamisch en tijdelijk) aan x, var x: integer bindt (bij blokingang) een nieuwe lokatie aan x, en const x=3 bindt 3 (statisch en permanent) aan x. Bij funktie-applikatie wordt de argumentwaarde gebonden aan de formele parameter. Naar aanleiding hiervan onderscheiden we in expressies gebonden en vrije voorkomens van variabelen. Voor de gebonden voorkomens ligt de betekenis als het ware al vast, en is niet meer vrij te kiezen; dat is wel het geval voor de vrije voorkomens. (Nota bene, onderscheid de voorkomens (=plaatsen) van een variable, van de variable zelf. En onderscheid ook de voorkomens van een deelexpressie van die deelexpressie zelf).

Definitie Bindingsexpressies
De expressievormen (fn x. eb), (df x=ea. eb) en (rec x. eb) heten bindingsexpressies. De symbolen fn, df en rec heten binder, de variabele x heet de gebonden variabele van de bindingsexpressie. Het deel eb heet het bereik van de binding. Het deel fn x, df x, rec x van zo'n expressie heet de binding (of introductie, definitie) van x en het deel x daarin het definierend (of introducerend, bindend) voorkomen.

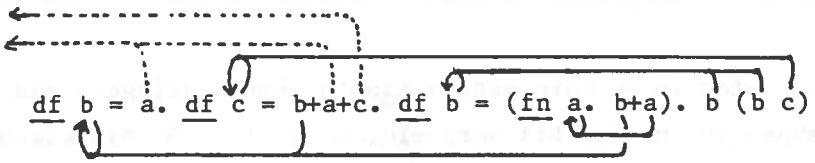
Definitie Vrije en gebonden voorkomens en variabelen
Een voorkomen van x heet vrij in e als dat niet ligt in het bereik van een of andere introduktie van x, en ook niet zelf een introducerend voorkomen is. Een voorkomen van x heet gebonden in e als het niet vrij is in e. Een bindend voorkomen van x bindt de vrije voorkomens van x in het bereik van de binding en bindt ook zichzelf.

Een variabele heet vrij/gebonden in e als hij een vrij/gebonden voorkomen heeft in e. De verzameling van vrije/gebonden variabelen van e zullen we -heel soms- noteren met VV(e) resp BV(e).

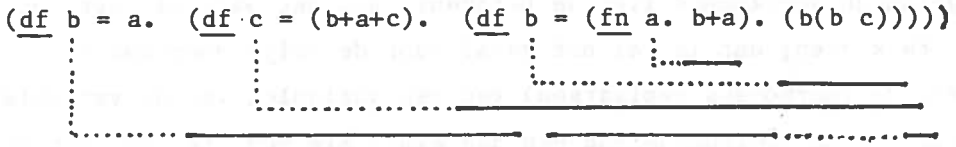
Definitie Gezichtsveld (scope)

Het gezichtsveld (engels: de scope) van een binding of definitie van x is het bereik ervan met weglating van ieder daarin voorkomende andere bindend voorkomen van x en het daarbij horende bereik. (Een gezichtsveld is dus een expressie met gaten erin; die gaten nemen de positie van introducerende voorkomens en subexpressies in. Een bindend voorkomen van x bindt ieder voorkomen van x in zijn gezichtsveld.)

Soms hebben de woorden globaal en lokaal exakt de betekenis van vrij resp. gebonden. Wij zullen die woorden nauwelijks gebruiken voor dit doel. Ter illustratie tekenen we hieronder voor ieder voorkomen een pijl naar zijn bindend voorkomen en geven we ook de gezichtsvelden aan.



De bindingsrelatie tussen de variabele-voorkomens



De gezichtsvelden van de introdukties

In deze expressie zijn a en c zowel vrije als gebonden variabelen. Merk op dat in (df x=ea. eb) het deel ea niet in het bereik van de binder ligt; zie het vrije voorkomen van c hierboven. Hadden we dit wel zo laten zijn, dan hadden we (zonder rekursie-expressies!) rekursieve definities gehad. Merk ook op dat we voor een gebonden voorkomen van x als volgt "zijn" bindend voorkomen kunnen vinden, dat wil zeggen het bindend voorkomen dat hem bindt: zoek de kleinste sub-expressie met een introduktie van x en een bereik dat het beschouwde voorkomen bevat; de x in die introduktie is dan het bindend voorkomen.

Oefening 1.2! Bepaal voor ieder voorkomen van een variabele of het gebonden dan wel vrij is, en bepaal -indien van toepassing- ook zijn bindend voorkomen en het gezichtsveld.

a. df f = (fn x. (fn f. f) x + x * f). f (df f = x. x+x)

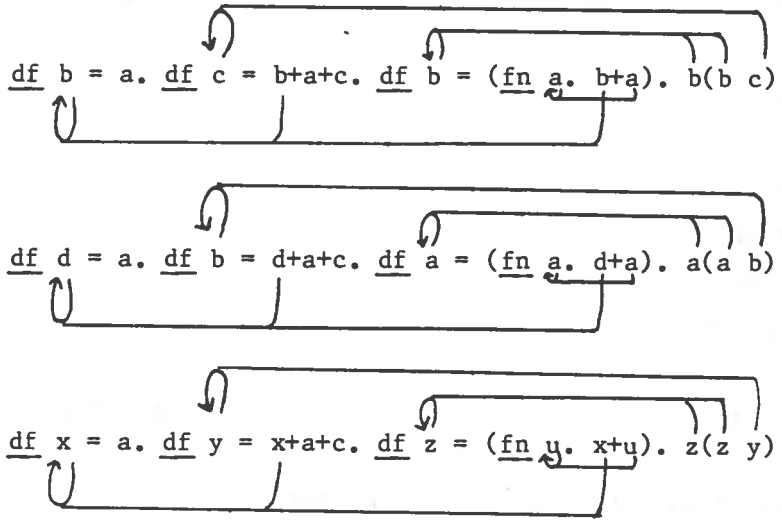
b. (fn p. (fn q. (fn p. p (p q)) (fn r. p+r)) (p+4)) 2

c. verzin zelf wat leuks.

Het belang van het begrip binding zit hem in het volgende. "Lokale naamgeving" speelt in talen met zg. statische scope (zoals HOF) (dit begrip komt elders nog uitvoerig aan bod) géén rol in de betekenis van een expressie. "Lokale naamgeving" behelst de keuze van de variabele op een bindend, en alle daardoor gebonden voorkomens. We definiëren nu wanneer twee expressies gelijk zijn behoudens lokale naamgeving, en zullen de semantische gelijkwaardigheid van zulke expressies in de volgende paragraaf afdwingen.

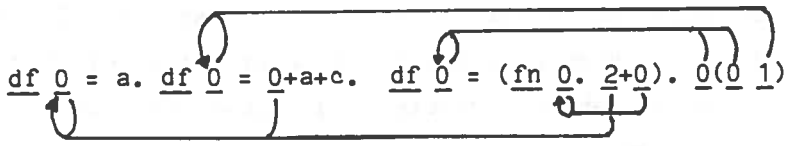
Definitie Gelijkheid behoudens lokale naamgeving
Twee expressies zijn gelijk behoudens lokale naamgeving (alfabetische varianten van elkaar, onderling alfa-konvertibel) wanneer zij op (de namen van) de gebonden variabelen na gelijk zijn en bovendien dezelfde bindingsrelatie tussen de gebonden voorkomens bepalen.

We herhalen ter illustratie het eerder gegeven voorbeeld, en geven daarvan alfabetische varianten. De pijlen geven de bindingsrelatie tussen de gebonden voorkomens aan. (Ga ook na dat de informele interpretatie van al deze expressies gelijk is).



Door variabelen als suggestieve identifiers te schrijven kunnen zij de lezer van een expressie erg behulpzaam zijn. Maar volgens de syntaxis dienen zij alleen om de bindingsrelatie lineair weer te kunnen geven. Dit kan ook op andere, minder

leesbare, manieren. Bijvoorbeeld als volgt. Vervang ieder voorkomen van een variabele door het verschil in nestingsdiepten (van bereiken) tussen dat voorkomen en zijn bindend voorkomen; met andere woorden, i.p.v. een identifier komt er een onderstreept getal n_x te staan dat aangeeft "aan hoeveel binders, van binnen uit geteld, dat voorkomen zich onttrekt". Bovenstaande expressie wordt dan



Bij een implementatie van de taal kan hiervan gebruik worden gemaakt, zie Hoofdstuk 4, par. 4.2.

Oefening 1.3! Welke van de volgende expressies zijn alfabetische varianten van elkaar?

a. (fn x. fn y. x y), (fn y. fn x. x y), (fn y. fn x. y x);

b. (fn x. x), (fn x. x);

c. (fn x. x + x), (fn x. 2 * x);

d. (df x = x+y+z. df y = 2*x+2*y+2*z. df z = 3*(x+y+z). x+y+z),
(df y = x+y+z. df z = 2*y+2*z+2*x. df x = 3*(y+z+x). y+z+x).

Oefening 1.4! Wat zijn in Pascal en Algol 68 bindingskonstrukties? En wat zijn de bereiken van de binders? Neem hierbij als leidraad dat alfabetische varianten semantisch gelijkwaardig zijn. Geef in het bijzonder de bindingsrelatie van de volgende fragmenten.

a.
```
type t = integer;
procedure p;
     type pt = ↑t;
          t = real;
     var x: t; y: pt;
     begin ---- end{p}
---
```

b.
```
function f(x: t): t;                                   NB. dit is geen Pascal!
     var x: t;
     begin --- x:=x+1 --- f(x) --- f:= --- end{f}
```

c.
```
var i: integer;                                  NB. dit "mag" niet in Pascal
begin --- i:=i+1 ---
     for i:=i+10 to i+20 do
     begin --- i:=i+1 --- end
end
```

```
d. int i; --- i:=i+10 ---
   for i from i+1 to i+10 while i<j do --- i --- od;
   --- i ---
```

Oefening 1.5! Geef enige bindingskonstrukties die in de omgangs- of wiskundetaal gebruikt worden. (Neem als leidraad dat alfabetische varianten hetzelfde betekenen.) Wenk: denk aan de kwantoren "voor-alle" en "er-is"; denk aan definities, en "notationele afspraken", aan differentiëren en integreren, enzovoort.

In de praktijk laten de bindingsrelaties zich soms minder gemakkelijk formaliseren dan in de taal HOF. Dit moge bijvoorbeeld blijken uit de volgende notaties.

```
a. var x: integer;
       z: record x, y: real end;
   begin --- z -- z.x ---- z.y --- x --- y ---
       with z do --- x -- y --- od
   end
```

b. Laat I staan voor het integraalteken, en I(a,b) voor het integraalteken met grenzen a en b.

```
   I(0,1) x dx = I(0,1) y dy
   I x dx = 0.5*x**2
```

In voorbeeld a fungeert z. en with z do als een soort binder van x en y. In voorbeeld b lijkt de dx een binder te zijn, maar de x is toch vrij in de uitdrukking voor de onbepaalde integraal. In feite is de bindingsrelatie binnen een expressie afhankelijk van de kontekst waarin die expressie voorkomt.

Tenslotte nog enige woorden over substitutie. Dat is een syntaktische handeling die we veel zullen gebruiken. Bijvoorbeeld, we zullen definiëren dat het resultaat van fn x. 3*x+(sqr x) toegepast op 4 in eerste instantie uitgedrukt wordt door 3*4+(sqr 4), dat is: het argument gesubstitueerd voor alle vrije voorkomens van x in de romp 3*x+(sqr x). Stel nu dat de romp (df y=3. y*x+(sqr x)) luidt. Stel voorts dat het argument niet 4 is maar de expressie (y+y); dat zou kunnen als dat argument en dus de hele applikatie in de scope van een binding df y=2 ligt. Letterlijke vervanging van het argument e' = (y+y) voor de vrije voorkomens van x in de romp e = (df y=3. y*x+(sqr x)) levert dan

```
df y=3. y*(y+y) + (sqr (y+y))
```

En dit is duidelijk ongewenst: de bindingsrelatie van het argument is gewijzigd, we spreken dan van een naamkonflikt (name clash). Dit kan voorkomen worden door

eerst een geschikte alfabetische variant e" van de romp e te nemen: e" = (df y´=3. y´*x+(sqr x)) ontstaat uit e door de y in y´ te veranderen. Dan krijgen we als resultaat van de substitutie

df y´=3. y´*(y+y) + (sqr (y+y))

Hiermee is gegarandeerd dat de vrij voorkomende y in het argument ook vrij blijft na substitutie. Deze handeling leggen we vast in de volgende definitie.

Definitie Substitutie
De substitutie van e´ voor alle voorkomens van x in e, genoteerd e[x/e´], wordt als volgt uitgevoerd. Neem een alfabetische variant e" van e zo dat voor alle vrije variabelen y van e´ geldt:
ieder vrij voorkomen van x in e" ligt niet in het bereik van een binding van y. Vervang vervolgens ieder vrij voorkomen van x in e" letterlijk door e´. (De aldus resulterende expressie is behoudens lokale naamgeving uniek bepaald).

Par. 1.3 De semantiek van HOF

In deze paragraaf zullen we de semantiek van HOF definiëren. Dat wil zeggen, we definiëren de betekenis ofwel interpretatie van expressies, of in andere woorden, de waarde aangeduid door expressies. De definitie van de semantiek kan op velerlei manieren gebeuren, variërend van heel abstrakt tot heel konkreet. Wij zullen een methode kiezen die redelijk abstrakt is, omdat er nauwelijks andere begrippen bij gebruikt worden dan expressies; met name zien we helemaal af van aspekten die optreden bij machinale verwerking (die komen in Hoofdstuk 3 en 4 aan de orde). De methode is ook operationeel, omdat de waarde aangeduid door een expressie wordt gedefinieerd door een berekeningsproces. Dit staat in tegenstelling tot denotationeel, waarbij er een afbeelding (geen proces of algoritme) wordt gedefinieerd die bij iedere expressie zijn waarde geeft.

Voor de programmeur is het belangrijk om korrektheidsbeweringen over zijn programma´s te kunnen bewijzen. We zullen in Hoofdstuk 5 een stelsel voor het doen van korrektheidsbewijzen uit de semantiekdefinitie afleiden. Voorts laten we, zoals gezegd, in Hoofdstukken 3 en 4 zien hoe op systematische wijze een implementatie uit de semantiekdefinitie afgeleid kan worden.

Vraag Op welke schriftelijke manier is u de semantiek van Algol 68, Pascal (en Basic) bijgebracht? Hoe luiden de officiele semantiekdefinities van die talen? Zijn die definities meer gericht op de implementatie of op het bewijzen van de

korrektheid van programma's?

De definitiemethode

We zullen middels een stelsel van herschrijfregels en een herschrijfstrategie aangeven hoe een expressie herhaaldelijk herschreven kan worden totdat er een expressie resulteert (die niet verder herschrijfbaar is en) die we de waarde van de oorspronkelijke expressie noemen. Dit is geheel analoog aan de manier waarop rekenkundige uitdrukkingen door herhaalde vereenvoudigingen tot het uiteindelijke antwoord omgevormd worden. Bijvoorbeeld, voor n=7

```
(1/2)*n*(n+1) =  (1/2)*7*8
              => (1/2)*56
              => 28
```

Het zal blijken dat de strategie volgens welke de afzonderlijke herschrijfregels worden toegepast, nauwelijks ter zake doet voor wat het eindresultaat betreft, maar wel voor het aantal benodigde herschrijfstappen. Ook dit verschijnsel doet zich voor bij het vereenvoudigen van rekenkundige uitdrukkingen: het loont vaak de moeite om in een breuk éérst de teller en noemer van gelijke faktoren te ontdoen. Daarom zullen we de herschrijfregels en de herschrijfstrategieën afzonderlijk presenteren.

De herschrijfregels

We geven nu een opsomming van de herschrijfregels; een toelichting volgt daarna. De regels zijn genoemd naar de samenstellingswijze van de herschreven expressie.

(app)	((fn x. eb) ea)	-> eb[x/ea]
(df)	(df x=ea. eb)	-> eb[x/ea]
(rec)	(rec x. eb)	-> eb[x/(rec x. eb)]
(.1)	(<e1, e2>.1)	-> e1
(.2)	(<e1, e2>.2)	-> e2
(+)	(m + n)	-> m + n
(-)	(m - n)	-> m - n voor m≥n
(*)	(m * n)	-> m * n
(eqn)	(m eqn n)	-> true voor m=n
	(m eqn n)	-> false voor m≠n
(and)	(true and false)	-> false, etcetera

(or) (true or false) -> true , etcetera
(not) (not true) -> false
(not false) -> true
(if) (if true then e1 else e2) -> e1
(if false then e1 else e2)-> e2
(hd) (hd (eh: et)) -> eh
(tl) (tl (eh: et)) -> et
(eqnil)(eqnil nil) -> true
(eqnil (eh: et)) -> false

Toelichting

1. Regel (app) wordt traditioneel de beta regel genoemd. Hij formaliseert precies de zogenaamde "body replacement rule" waarmee in het Algol 60 Rapport de semantiek van procedures wordt uitgelegd.
2. Regel (df) laat zien dat er een zekere overeenkomst is tussen definitie en parameterisatie: het lijkt erop dat (df x=ea. eb) en ((fn x. eb) ea) onderling uitwisselbaar zijn. We zullen ervoor zorgen dat dit zoveel mogelijk het geval is, en noemen dat

Het Beginsel Van Overeenkomst tussen Definitie en Parameterisatie

3. Regel (rec) formaliseert rekursie. Door herhaaldelijk die regel op (rec x. 1:x) toe te passen krijgen we (1:(1:(1: ---(rec x. 1:x) ---))). De fakulteitsfunktie kan als volgt gedefinieerd worden:

df fac rec = (fn x. if x eqn 0 then 1 else x * fac(x-1))

i.e. df fac = (rec f. fn x. if x eqn 0 then 1 else x * f(x-1))

We kunnen ook louter de definiërende expressie van de tweede definitie nemen: dat is een expressie voor de fakulteitsfunktie zonder hem een naam te geven. Eenmaal toepassen van (rec) op die expressie levert

(fn x. if x eqn 0 then 1 else x * (rec f. ---) (x-1))

dus de rekursieve aanroep is vervangen door de definierende expressie zelf.

4. Merk op dat er geen herschrijving is gedefinieerd voor bijvoorbeeld (6-17). Onze keuze is vrij willekeurig. We hadden ook wel met gehele in plaats van natuurlijke getallen kunnen werken, en we hadden die expressie ook nog wel tot 0 kunnen laten herschrijven. De motivatie voor deze keuze is dat we ook partiële operaties willen hebben in de taal, vergelijk underflow en overflow in programmeertalen. In Hoofdstuk 9 zullen we foutstoppen behandelen.
5. In plaats van regels (+), (-) en (*) hadden we ook ietwat andere regels kunnen geven, zoals de volgende.

(+') (0 + e) -> e

(m + n) -> m+n

of

(+") (0 + e) -> e
(e + 0) -> e
(m + n) -> m+n

Soortgelijks geldt ook voor (*), (and), (or) en (if). Een alternatief voor (if) is met name

(if´) (if e then e0 else e0) -> e0
(if true then e0 else e1) -> e0
(if false then e0 else e1)-> e1

De alternatieven zoals (+´) voor (+) hadden wij ook nog wel kunnen kiezen zonder enige wezenlijke verandering. Maar de alternatieven (+") en (if´) vereisen een soort parallellisme in de implementatie van de herschrijving (of "evaluatie"), om namelijk bij nonterminerende e toch eventueel de expressie zelf te herschrijven. Zo´n parallellisme is niet gebruikelijk bij programmeertalen, en daarom verwerpen we die alternatieven.

Definitie Reduktie
De herschrijfregels heten ook wel reduktieregels. Een expressie die als linkerlid optreedt in een van de reduktieregels heet redex (reduceerbare expressie, meervoud: redices); het bijbehorende rechterlid heet contractum. Het vervangen van een redex door zijn contractum, in een omvattende expressie, heet kontraktie of reduktiestap en wordt genoteerd met $\overset{1}{=>}$. Herhaaldelijk, nul of meer keer, kontraheren heet reduceren of reduktie en wordt genoteerd met =>.

De -of een- semantiek voor HOF wordt vastgelegd door een bepaalde strategie te kiezen volgens welke de reduktie moet plaatsvinden. Een strategie bestaat uit de keuze om de reduktie te beëindigen, dan wel uit de keuze van de eerstvolgend toe te passen reduktiestap, dus uit de keuze van een subexpressie en een daarop toe te passen reduktieregel. Bij bovenstaande reduktieregels is een subexpressie de redex van hoogstens één reduktieregel, dus kan met de keuze van een subexpressie volstaan worden. Een kontraktie en reduktie volgens een strategie S noemen we een S-kontraktie resp. S-reduktie en noteren we met $\overset{1}{\underset{S}{=>}}$ resp. $\underset{S}{=>}$.

Definitie Stokken, termineren en resultaat van redukties

Een S-reduktie termineert als op de laatst verkregen expressie volgens S geen kontraktie meer moet worden toegepast. Die laatst verkregen expressie, en al zijn alfabetische varianten, heet dan het S-resultaat (of de S-waarde) van de oorspronkelijke expressie en van de reduktie. Een S-reduktie stokt als S een subexpressie kiest die geen redex is (van de tevens gekozen reduktieregel). Als een reduktie oneindig veel stappen heeft, spreken we van nonterminatie of divergentie. (NB. Stokken ≠ stoppen = termineren.)

Hier volgen enige voorbeelden van S-redukties; we zullen S niet nader specificeren maar de toepaste reduktieregels steeds aangeven.

a. ((0 + 1) + (2 + (3 + 4)))
(+) => (1 + (2 + (3 + 4)))
(+) => (1 + (2 + 7))
(+) => (1 + 9)
(+) => 10 terminatie met 10 als resultaat.

b. ((0 + 1) - (2 + (3 + 4)))
(+) => (1 - (2 + (3 + 4)))
(+) => (1 - (2 + 7))
(+) => (1 - 9) stokkende reduktie bij strategieen zoals U en C (maar terminatie bij N)
(-) => ??

c. (df f = (fn x. (hd x)). f (1: 0: nil))
(df) => (fn x. (hd x)) (1: 0: nil))
(app) => (hd (1: 0: nil))
(hd) => 1 terminatie met 1 als resultaat.

d. (df f = (fn x. (hd x)). f <1, 0>)
(df) => ((fn x. hd x) <1, 0>)
(app) => (hd <1, 0>) stokt, bij strategieen zoals U en C (maar terminatie bij N)
(hd) => ??

Alvorens sommige strategieën te definiëren (en daarmee evenzovele semantieken?), zullen we eerst de mogelijke verschillen tussen strategieen bekijken.

Gelijkwaardigheid van strategieën

We vragen ons af in hoeverre een strategie van invloed kan zijn op een reduktie of zijn resultaat. Het is duidelijk dat afhankelijk van de strategie een

reduktie al of niet kan termineren. Beschouw bijvoorbeeld de expressie ((fn x. 1) (rec x. x)). Hierin zijn twee redices: de hele expressie zelf kan gekontraheerd worden volgens (app), en het argument kan gekontraheerd worden volgens (rec). De ene kontraktie geeft de irreducibele expressie 1, de andere kontraktie geeft de expressie zelf weer als resultaat en kan dus oneindig vaak herhaald worden. Op zich is de mogelijkheid van nonterminatie niet zo bezwaarlijk, hij is hoogstens ongewenst. Het zou wel bezwaarlijk zijn als verschillende strategieën tot verschillende irreducibele expressies kunnen leiden, bijvoorbeeld e=>1 en e=>2. Enerzijds zou dat het vetrouwen in de "juistheid" van de reduktieregels ondermijnen. Anderzijds betekent het dat de strategie een minstens even belangrijke rol gaat spelen als de reduktieregels zelf. En dat is jammer, want dan verliezen we een stuk potentiele eenvoud in de semantiek van de taal.

Gelukkig kan men voor HOF bewijzen dat alle irreducibele expressies waartoe een expressie resulteert, alfabetische varianten zijn van elkaar. En sterker zelfs, ieder tweetal expressies waartoe een gegeven expressie reduceert, zijn op hun beurt verder te reduceren tot een gemeenschappelijk resultaat. Deze eigenschap wordt de Church-Rosser eigenschap genoemd en ook wel Konfluentie. In formules:

Als e=>e1 en e=>e2
dan zijn er e´ en e" (alfabetische varianten van elkaar)
zodat e1=>e´ en e2=>e".

Dus alle strategieën zijn gelijkwaardig in de zin dat ieder tweetal tussenresultaten uit een S- en een S´-reduktie van een gegeven expressie tot een gemeenschappelijke expressie gereduceerd kunnen worden; dit geldt ook voor de eindresultaten indien zij bestaan. Dit wordt verder uitgewerkt in Hoofdstuk 5.

De Church-Rosser eigenschap gaat verloren als we foutstoppen met de afhandeling daarvan toevoegen aan de taal, zie Hoofdstuk 9, of als we assigneerbare variabelen toevoegen, zie Hoofdstuk 7, of bijvoorbeeld funktiekonstanten fct en arg met de volgende reduktieregels

(f) (fct (ef ea)) -> ef
(a) (arg (ef ea)) -> ea

Het is niet moeilijk om hiermee een expressie te vinden die onder verschillende strategieën tot verschillende irreducibele expressies reduceert. Een enigszins zinvolle expressie is de volgende. Laat (tn e) staan voor t(t(t(...e))) (met n

maal een t), en denk bij t aan (fn f. fn x. f(f x)). Dan geldt

(fct ((fn f. t (t^n f)) t))	(f) =>	(fn f. t (t^n f)) = (fn f. t^{n+1} f)
(fct ((fn f. t (t^n f)) t))	(app)=>	(fct (t (t^n t))) (f)=> t
(arg ((fn f. t (t^n f)) t))	(a) =>	t
(arg ((fn f. t (t^n f)) t))	(app)=>	(arg (t (t^n t))) (a)=> (t^n t)

De Church-Rosser eigenschap is een plezierige eigenschap voor een programmeertaal. De geldigheid ervan wordt wel gebruikt als kriterium om een taal beschrijvend (funktioneel, applikatief) te noemen. (Omdat fct en arg hierboven net werken als car en cdr in Lisp, zien we dat Lisp volgens deze opvatting niet beschrijvend is!) Het biedt eén implementeur een geweldige mate van vrijheid. Het maakt het, bijvoorbeeld ook in korrektheidsbewijzen, mogelijk om een subexpressie naar willekeur te reduceren onder behoud van "semantiek" van de omvattende expressie, zie Hoofdstuk 5.

Reduktiestrategieën

Allereerst merken we op dat er een triviale strategie mogelijk is: pas nooit een kontraktie toe. Zo'n strategie is makkelijk te implementeren, maar overigens weinig zinvol. Het resultaat van bijvoorbeeld de expressie 1*2*3*4*5 is die expressie zelf, en niet 120 wat je zou wensen.

De normaalvormstrategie N

Als andere uiterste is er de strategie waarbij de (uniek bepaalde) irreducibele expressie, indien die bestaat, door de reduktie bereikt wordt. De irreducibele expressie waartoe een expressie e reduceerbaar is heet de normaalvorm van e en deze strategie heet de normaalvormstrategie N. De N-strategie moet dus voorkomen dat een non-terminerende reduktie wordt gekozen wanneer er wel een terminerende reduktie met een irreducibel resultaat bestaat. Dat kan door slechts een "strikt noodzakelijke" subexpressie ter kontraktie te kiezen, dat wil zeggen een redex die bij kontrakties van willekeurig andere redices niet kan verdwijnen en dus noodzakelijk toch ooit eens gekozen zou moeten worden. De keuze kan daarom allereerst al beperkt worden tot zogenaamde maximale redices, dat zijn de redices die geen deel zijn van een andere redex ; want door kontrakties in de onderdelen kan zo'n redex zelf niet verdwijnen. Bovendien moet de maximale redex ook niet in een then- of else-deel gekozen worden, tenzij het if-deel al irreducibel is {en dus verschilt van true en false}; zo'n then- of else-deel zou namelijk door kontrakties in het if-deel toch onderdeel kunnen worden van een omvattende redex en dan alsnog kunnen verdwijnen. En net zo moet de maximale re-

dex ook niet in een argumentdeel gekozen worden tenzij het funktiedeel al irreducibel is {en dus geen abstraktie (fn. eb) is}; zo'n argumentdeel zou namelijk door kontrakties in het funktiedeel toch onderdeel kunnen worden van een omvattende redex en dan alsnog verdwijnen.
Alle overige maximale redices kunnen nooit door reduktie in de kontekst deel worden van een omvattende redex en dan alsnog verdwijnen; van het moet er dus één ter kontraktie worden gekozen.

Omdat we in onze lineaire notatie van expressies het if-deel links van het then- en else-deel schrijven en ook het funktie-deel links van het argument-deel, kunnen we de N-strategie verrassend eenvoudig formuleren:

Kies ter kontraktie de meest linker maximale redex.

Dit houdt overigens ook in dat (e+e') enzovoort van links naar rechts worden gereduceerd; dat is strikt genomen niet noodzakelijk (maar wel bij de alternatieve regel (+')). Hier is een voorbeeld van een N-reduktie.

```
        ((1+2) + ((fn x. 3 + 4) (rec x. 5 + x)))
(+)  => (3 + ((fn x. 3 + 4) (rec x. 5 + x)))
(app)=> (3 + (3 + 4))
(+)  => (3 + 7 ))
(+)  => 10
```

Let wel, de N-reduktie van e = (rec f. fn x. if x then 0 else 1 + f x) stopt niet vanwege de niet verdwijnende rec-expressie:

```
    e = (rec f. fn x. if x then 0 else 1 + f x)
 (rec)=>(fn x. if x then 0 else 1 + e x)
 (rec)=>(fn x. if x then 0 else 1 +
                (fn x. if x then 0 else 1 + e x) x)
 (app)=>(fn x. if x then 0 else 1 +
                (if x then 0 else 1 + e x))
 etc  =>(fn x. if x then 0 else 1 +
               (if x then 0 else 1 +
                 (if x then 0 else 1 +

                   (if x then 0 else 1 + e x)...)))
```

Daarentegen stopt de N-reductie van (e true) na drie stappen!

Bovenstaande reduktiestrategie is voornamelijk van theoretisch belang; de nu volgende strategieën worden in de praktijk gebruikt, d.w.z. ze modelleren bestaande evaluatiestrategieën. Er wordt daarbij niet naar gestreefd om altijd een irreducibele vorm als resultaat te krijgen. Met name zijn we voor funkties niet geïnteresseerd in de expressie waarmee die wordt genoteerd, maar alleen in het resultaat na toepassing op een argument. Zie bijvoorbeeld de expressie e hierboven; het stelt een funktie voor maar heeft geen irreducibele vorm, terwijl (e true) dat wel heeft. Bovendien, anders dan bij getallen, waarheidswaarden, groepen en lijsten is er voor funkties geen voor de hand liggende unieke expressie. (Bedenk ook dat het in het algemeen niet beslisbaar is of twee expressies eenzelfde afbeelding aanduiden, of tot eenzelfde funktie-expressie reduceerbaar zijn). Voorts zijn we niet geïnteresseerd in expressies waarin zoiets als (<-,-> e) of (hd <-,->) of (3 - 7) op een kardinale plaats voorkomt; als zo'n vorm tijdens een reduktie optreedt, dan willen we dat als een "dynamische (run-time) fout" beschouwen. Ook vrije variabelen wensen we liever niet te zien in een eindresultaat; bijvoorbeeld wat heb je aan zoiets als (if x then e1 else e2)? Moet je dan zowel e1 en e2 verder reduceren, of helemaal niets meer? Dit overwegende komen we tot de definitie van expressies die we als resultaat wensen.

Definitie Resultaatvorm
De expressies r in resultaatvorm worden voortgebracht door

r::= k | (r: r) | <r, r> | (fn x. e)

met e een willekeurige expressie. (Een expressie in resultaatvorm bevat dus geen redices tenzij die in een funktieexpressie liggen). De expressies r' oppervlakkig in resultaatvorm worden gedefinieerd door

r' ::= k | (e: e) | <e, e> | (fn x. e)

met e een willekeurige expressie. Iedere resultaatvorm is dus ook oppervlakkig in resultaatvorm.

De uitstelstrategie U

De uitstel-strategie U volgt in beginsel dezelfde werkwijze om een resultaatvorm te bereiken als de N-strategie volgt om de irreducibele vorm te bereiken. Er zijn slechts twee verschillen. Ten eerste wordt er binnen een resultaatvorm nooit gereduceerd; dus de U-reduktie van e = (rec f. fn x. if x then 0 else 1 + f x) stopt na één (rec) -kontraktie met als resultaat (fn x. if x then 0 else 1 + e x) terwijl de N-reduktie niet eindigt, zoals we gezien hebben. Ten

tweede stokt de U-reduktie zodra een dynamische type-fout optreedt en dus geen resultaatvorm bereikt kan worden; de N-reduktie gaat dan gewoon door om de rest nog te reduceren. Bijvoorbeeld, (if 3+4 then 5+6 else 7+8) reduceert volgens N tot (if 7 then 11 else 15), terwijl de U-reduktie ervan na de (+)-kontraktie stokt met als "tussenresultaat" (if 7 then 5+6 else 7+8). En net zo stokt de U-reduktie van ((fn x. x (2+4)) <1,3>) na de (app)-kontraktie met als "tussenresultaat" (<1,3> (2+4)).

Een formele definitie van de U-strategie luidt als volgt. (Het begrip reduciendum (= "moetende gereduceerd worden") definiëren we verderop.)

Kies ter kontraktie de meest linker maximale reduciendum die niet in een resultaatvorm ligt.

Inderdaad, er zijn slechts twee verschillen met de N-strategie. In plaats van redices worden nu reduciendi gekozen, zodat de reduktie stokt wanneer de reduciendum geen redex blijkt te zijn. Bovendien wordt er niet binnen resultaatvormen, met name niet binnen funktie-expressies, gereduceerd. Overigens zijn de U- en N- strategie gelijk. De U-strategie levert een resultaatvorm op indien de expressie tot een resultaatvorm te reduceren is. Rest ons nog het begrip reduciendum te definiëren. Ruwweg gezegd is een reduciendum, ofwel een redex ofwel een redex-achtige expressie met een duidelijke fout aan de oppervlakte. Hier volgen een paar voorbeelden van reduciendi.

- ((fn x. eb) ea) en ook (<e1,e2> ea), ((eh: et) ea), (k ea)
- <e1, e2>.1 en ook (fn x. eb).1, (eh: et).1, k.1
- (m + n) en ook (fn x. eb) + n, (eh: et) + n, m + (fn x. eb),...

enzovoorts.

Een grammatika die reduciendi r" voortbrengt is

```
r" ::= (r' e) | (df x=e. e) | (rec x. e)
     | (r'.1 | (r'.2)
     | (r' + r') | (r' - r) | (r' * r') | (r' eqn r')
     | (r' and r') | (r' or r') | ( not r')
     | (if r' then e else e)
     | (hd r') | (tl r') | (eqnil r')
```

met r' oppervlakkig in resultaatvorm en e willekeurig. Dus een reduciendum r" is een expressie die ontstaat uit een redex door daarin ieder samenstellend deel van voorgeschreven oppervlakkige resultaatvorm te vervangen door een willekeurige oppervlakkige resultaatvorm r'.

Ter illustratie van de U-strategie geven we de reduktie van (e 3) met e = (rec f. fn x. if x eqn 0 then 1 else x * f(x-1)), resulterend in de fakulteit van 3.

(e 3)
(rec) => (e' 3) met e'= (fn x. if ....else x * e (x-1))
(app) => if 3 eqn 0 then 1 else 3 * e (3-1)
(eqn) => if false then 1 else 3 * e (3-1)
(if) => 3 * e (3-1)
(rec) => 3 * e' (3-1)
(app) => 3 * if (3-1) eqn 0 then 1 else (3-1) * e ((3-1)-1)
(-)   => 3 * if 2 eqn 0 then 1 else (3-1) * e ((3-1)-1)
(eqn) => 3 * if false then 1 else (3-1) * e ((3-1)-1)
(if) => 3 * ((3-1) * e ((3-1)-1))
(-)   => 3 * (2 * e ((3-1)-1))
(rec) => 3 * (2 * e' ((3-1)-1))
(app) => 3 * (2 * if ((3-1)-1) eqn 0 then 1 else ((3-1)-1) * e (((3-1)-1)-1))
enzovoorts.

Deze U-reductie is tevens een N-reduktie. Dat is niet zo in onderstaand voorbeeld.

(fn x y. x y) (fnz. z 3) <5+6, 7+8>
(app) => (fn y. (fn z. z 3) y) <5+6, 7+8>
(app) => (fn z. z 3) <5+6, 7+8>
(app) => <5+6, 7+8> 3
stokt; de N-reduktie gaat verder met twee (+)-kontrakties.

De uitstelstrategie U wordt wel "output driven" genoemd. Steeds wordt die reduktiestap uitgevoerd die "allereerst noodzakelijk" is om de gewenste uitvoer {een expressie in resultaaatvorm} te bereiken. Met name betekent dit dat het funktiedeel ef van een applikatie (ef ea) slechts zover gereduceerd wordt totdat het oppervlakkig in resultaatvorm is; is dat dan van de vorm(fn x. eb) dan vindt daarna de (app)-kontraktie plaats {en anders stokt de reduktie} zonder dat er in eb of ea ook maar één reduktiestap is gedaan. En in een selektie eg. l wordt het groepdeel eg slechts zover gereduceerd totdat het oppervlakkig in resultaatvorm staat; is dat dan van de vorm <e1, e2> dan vindt daarna de (.1)-kontraktie plaats {en anders stokt de reduktie} zonder dat er in e1 of e2 ook maar één reduktiestap is gedaan. En net zoiets gebeurt er bij de reduktie van (hd e), (tl e), (equil e), (if e then e` else e") (e+e'). (Deze beschrijving is met uitsluiting van hetgeen er tussen akkolades staat ook geldig voor een N-reduktie!)

De kompositionele strategie C

De volgende strategie is de kompositionele strategie C. Hierbij wordt, behoudens twee uitzonderingen, een redex pas gekontraheerd als zijn samenstellende delen al tot resultaatvorm zijn gereduceerd. (Wij noemen een afbeelding kompositioneel als het beeld van een samenstelling een samenstelling van de beelden der onderdelen is). De uitzonderingen zijn deze. De then- en else-tak van een if-expressie worden niet alvast tevoren tot resultaatvorm gereduceerd (maar één van hen komt aan bod na kontraktie van de if-expressie), en ook*) het bereik van een bindingsexpressie wordt pas gereduceerd na kontraktie van die bindingsexpressie zelf, zo dat de variabelen in het bereik (gebonden door betreffende binding) zijn weggewerkt alvorens delen van dat bereik tot resultaatvorm worden gereduceerd; (een variabele is zelf geen resultaatvorm!*). Bijgevolg zijn ((fn x. eb) ea) en (df x=ea. eb) nog steeds onderling uitwisselbaar: het Beginsel van Overeenkomst tussen Definitie en Parameterisatie!. De kompositionele strategie luidt dus:

kies ter kontraktie (ja zelfs, de meest linker) een minimale non-resultaatvorm die niet in een resultaatvorm ligt, en niet in een then- of else-tak noch in het bereik van een bindingsexpressie.

De keuze van "de meest linker" doet voor HOF nauwelijks ter zake. Maar wanneer we later assigneerbare variabelen en sequentie toevoegen (Hoofdstuk 7), dan wordt die volgorde kruciaal. (Als de gekozen subexpressie geen redex blijkt te zijn, dan stokt de reduktie).

De kompositionele strategie wordt ook wel de applikatieve strategie genoemd. Dat vinden wij echter een verwarrende naamgeving, omdat het juist de strategie is waarmee imperatieve talen worden geëvalueerd.

Terminerende redukties volgens de kompositionele strategie resulteren in expressies in resultaatvorm, net als bij de uitstelstrategie. Maar soms termineert een C-reduktie niet terwijl de U-reduktie dat wel doet. Dat komt doordat bij niet-terminerende C-reduktie van ea, en dus niet-terminerende C-reduktie van ((fn x. eb) ea), de U-reduktie van die applikatie mogelijk wel termineert. Bijvoorbeeld als x niet vrij voorkomt in eb, en dus eb[x/ea] = eb, of als de vrije voorkomens van x in eb uiteindelijk niet "gebruikt" worden in de U-reduktie.

---

*) Hierop komt in Hfd.7 een uitzondering

de C-reduktie minder stappen telt dan de U-reduktie. Dat komt doordat een substitutie eb[x/ea] pas wordt uitgevoerd bij de C-strategie als ea al in resultaatvorm staat; eventuele duplikaties van ea ten gevolge van die substitutie leiden dus niet tot herhaalde redukties van ea.
Als voorbeeld daarvan weer de berekening van de fakulteit van 3, nu volgens de C-strategie.

Zij e = (rec f. fn x. if x eqn 0 then 1 else x * f(x-1)),
en e´= (fn x. if x eqn 0 then 1 else x * e(x-1)).

```
(e 3)
(rec) => (e´ 3)
(app) => if 3 eqn 0 then 1 else 3 * e(3-1)
(eqn) => if false then ...
(if)  => 3 * (e(3-1))
(rec) => 3 * (e´(3-1))
(-)   => 3 * (e´ 2)
(app) => 3 * if 2 eqn 0 then 1 else 2 * e(2-1)
(eqn) => 3 * if false then ...
(if)  => 3 * (2 * e(2-1))
(rec) => 3 * (2 * e´(2-1))
(-)   => 3 * (2 * (e´ 1))
...
      => 3 * (2 * (1 * 1))
(*)   => 3 * (2 * 1)
(*)   => 3 * (2 * 1)
(*)   => 3 * 2
(*)   => 6
```

Hetzelfde eindresultaat als bij de U- en N-reduktie, maar nu bereikt met veel minder (-)-kontrakties en overigens dezelfde stappen.

Hier volgt de C-reduktie van een expressie die al eerder werd beschouwd.

```
          ((1+2)+((fn x. 3+4) (rec x. 5+x))
(+)   =>  (  3  +((fn x. 3+4) (rec x. 5+x))
(rec) =>  (  3  +((fn x. 3+4) (5+(rec x. 5+x)))
(rec) =>  .
          .
          .
(rec) =>  (  3  +((fn x. 3+4) (5+(5+(5...+(rec x. 5+x)...)))))
          .
          .
```

Het C-resultaat is onbepaald; het U-resultaat is 10.

Een reduktiestrategie die de doelmatige aspekten van de U- en de C-strategie kombineert is de luie strategie (lazy evaluation), L. Deze wordt uitvoerig in Hoofdstuk 2 en 3 beschreven; we zullen hem hier globaal en met een voorbeeld beschrijven. De te kontraheren subexpressie wordt precies als bij de U-strategie bepaald. Duplikatie van reduktie ten gevolge van de duplikaties van ea in eb[x/ea], wordt voorkomen door de bedoelde voorkomens van ea gedeeld (shared) te representeren. Dus wanneer ooit zo´n voorkomen van ea wordt gereduceerd, dan vindt dat tevens plaats voor alle andere voorkomens. Met andere woorden, eenmaal uitgevoerde redukties worden onthouden voor toekomstig gebruik. Ter illustratie volgt hier de L-reduktie van (e 3)
met e = (rec f. fn x. if x eqn 0 then 1 else x * f(x-1)).
Zij e´ = (fn x. if x eqn 0 then 1 else x * e(x-1)).

(e 3)
(rec) => (e´ 3)
(app) => if 3 eqn 0 then 1 else 3 * e(3-1)
(eqn) => if false ...
(if) => 3 * e(3-1)
(rec) => 3 * e´(3-1)
(app) => 3 * if (3-1) eqn 0 then 1 else (3-1) * e((3-1)-1)

worden gedeeld gerepresenteerd!

(-) => 3 * if 2 eqn 0 then 1 else 2 * e(2-1)
(if) => 3 * (2 * e(2-1))
(rec, app) => 3 * (2 * if (2-1) eqn 0 then 1 else (2-1) * e ((2-1)-1))

worden gedeeld gerepresenteerd

(-) => 3 * 2 * if 1 eqn 0 then 1 else 1 * e(1-1)
...

Ten aanzien van het aantal kontrakties in een reduktie is de L-strategie doelmatiger dan zowel de U alsook de C-strategie. Maar dit wil niet zeggen dat de L-strategie in alle aspekten doelmatiger is dan met name de C-strategie. We komen hier nog op terug in Hoofdstuk 2.

Er zijn ook strategieën waarbij de kontrakties niet sequentieel maar parallel plaatsvinden, en er dus geen sprake is van een deterministische volgorde van de reduktiestappen. Zoiets is alleen maar zinvol als de Church-Rosser eigenschap

geldt, dus als de taal beschrijvend is. De mate van deze parallelliteit is ongetwijfeld veel groter dan in een gebiedende taal middels uitdrukkelijke besturing door de programmeur mogelijk is. De parallelle strategieen zijn nog volop in onderzoek.

Er zijn ook variaties op de U-, L- en C-strategieen. Bijvoorbeeld deze. Pas een kontraktie pas toe als de samenstellende delen van de redex al in resultaatvorm staan, met uitzondering van then- en else-takken en bindingsbereiken (zoals bij C) en (zoals bij U) van de argumenten van een applikatie. De kontraktie van bijvoorbeeld <e, e´>.1 en (hd (e:e´)) vindt dus pas plaats als zowel e alsook e´ al in resultaatvorm staan, terwijl de kontraktie van ((fn x. ---) <e, e´>) onmiddellijk gebeurt. Deze strategie modelleert de zg. call-by-name strategie van Algol 60. Overigens, om de overeenkomst tussen ((fn x. eb) ea) en (df x=ea. eb) te behouden, moeten we ook voor blokken hun kontraktie toestaan zonder dat de definiërende expressie in resultaatvorm staat.

Tot zover onze uiteenzetting over de mogelijke reduktiestrategieën. Iedere strategie bepaalt een semantiek voor HOF. Maar vanwege de Church-Rosser stelling zijn ze in zekere zin toch allemaal gelijkwaardig: indien zowel de U-reduktie als ook de C-reduktie termineren, en de een z´n resultaat is een getal- of waarheidskonstante (of een geneste groep of lijst over getal- of waarheidskonstanten), dan is dat resultaat identiek aan het resultaat van de andere reduktie. Het verschil in de strategieën uit zich dus voornamelijk in het al of niet bepaald zijn van de resultaten.

Belangrijke onderwerpen uit Hoofdstuk 1

funktioneel = applikatief = beschrijvend = "zonder neveneffekten";
hogere orde funktie, Currying, partiële parameterisatie;
syntaxis van HOF
  funktie-expressie, rekursie-expressie etc.;
  variabele = identifier;
  konventies voor het weglaten van haakjes;
  konventies voor afkortingen en veralgemeningen;
  ontbreken van een algemene gelijkheidstest;
  syntaktische binding etc., alfabetische variatie;
  gezichtsveld = scope, bereik van een binding;
  substitutie, naamkonflikt;
semantiek(en) van HOF
  reduktie = herschrijving, kontraktie, redex, contractum;
  resultaat = waarde;
  Beginsel van Overeenkomst tussen Definitie en Parameterisatie;
  terminatie, stokken, nonterminatie = divergentie;
  Church-Rosser eigenschap = konfluentie;
  reduktiestrategie = evaluatievolgorde, N, U, C, L;
  output-driven = demand driven, kompositioneel = applikatief.

Literatuur bij Hoofdstuk 1

Over de Lambda-Calculus. (Barendregt 1984) vormt een prima inleiding. Een bijna alles omvattend boek --voor gevorderden-- is (Barendregt 1981).

Over Formalismen voor Programmeertaaldefinities. Aanbevolen is (Pagan 1981); deze bevat eem uitgebreide literatuurbeschrijving. Voor de zgn. Denotationele beschrijvingswijzen zie (Gordon 1979), (Tennent 1976), (Stoy 1977) en (Milne & Strachey 1976). Voor de zgn. Weense Methode zie (Bjorner & Jones 1978). (Plotkin 1981) heeft als inspiratiebron gediend voor onze operationele aanpak.

Oefeningen

1.6 Simuleer lijsten met groepen, en omgekeerd. Ga de korrektheid na aan de hand van redukties.

1.7 Stel dat HOF uitgebreid zou worden met (een beperkt soort) assigneerbare variabelen, hoe zouden dan daarvoor de reduktieregels moeten luiden? (Zie Hoofdstuk 7).

1.8 Waar vindt U de U- en C-strategie terug in Pascal of Algol 68? (Denk aan while, for, sequentie, if, case, procedures, and, subskriptie, with, enzovoorts).

1.9! Geef een rekursief algoritme dat de C-reduktie beschrijft. Bijvoorbeeld zo:

```
proc reduceer (e) =
   if e ?= (fn x. eb) -> klaar
    | e ?= (ef ea) ->
         reduceer (ef); reduceer (ea); kontraheer-app;
         reduceer (e {= resultaat na vorige redukties})
     ...
```

1.10 Probeer ook dergelijke algoritmen te formuleren voor de N- en U-reduktie. Wenk: bepaal eerst een algoritme die een expressie volgens de N- of U-strategie reduceert totdat hij oppervlakkig in resultaatvorm staat. In termen hiervan zijn die andere twee eenvoudig uit te drukken.

1.11! Geef de N-, U- en C-redukties voor (mult 2 3) waarbij

```
mult = rec f. fn x y. if x eqn 0 then 0 else y + f(x-1) y
```

Geef ook de redukties van (mult 2) en van mult zelf, dus zonder argument.
NB. herinner je dat fn x y. een afkorting is van fn x. fn y.

1.12 We hebben verscheidene expressievormen gedefinieerd die we eenvoudshalve niet formeel in HOF hebben opgenomen, zoals (dfrec x=ex, y=ey. eb), (fn x y. eb), etc. Geef voor hun ook de definities van binding en van de reduktieregels. (Wenk. Definieer simultane substitutie e[x/e'] voor vektoren x en e').

1.13! Definieer in HOF de funkties div (gehele deling), mod (rest na deling), diffn (verschilt van, voor getallen), eqln (gelijkheid voor lijsten van getallen), append (de ++ van SASL), geq (groter of gelijk), leq (kleiner of gelijk),

gr (groter), less (kleiner), enzovoorts enz.

1.14 Volgend op de definitie van binding is een alternatieve representatie van expressies gegeven: ieder gebonden voorkomen is vervangen door een (onderstreept) getal dat aangeeft aan hoeveel binders dat voorkomen zich onttrekt. Geef een (informele) rekursieve funktie rewrite (e, ...) die die alternatieve representatie van e oplevert. (Wenk. Er zijn nog twee extra parameters nodig: een parameter n die aangeeft "binnen hoeveel binders" de subexpressie e zelf ligt, en een parameter d (een funktie) die voor iedere x aangeeft "binnen hoeveel binders" het definiërend voorkomen van x ligt. ("Binnen hoeveel binders" = "de nestingsdiepte")).

1.15 In hoeverre vind je dat in Pascal en Algol 68 aan het Beginsel van Overeenkomst tussen Definitie en Parameterisatie wordt voldaan? (Voor Pascal: zie (Tennent 1981, 1977)). Zijn (df x=ea. eb) en ((fn x. eb) ea) ook uitwisselbaar ten aanzien van Typepolymorfie (syllabus over Typering, Hoofdstuk 5)? (Antwoord: neen).

1.16 Hoe zouden de reduktieregels moeten luiden als we een expressievorm (if e->e | ... | e->e fi) aan HOF toevoegen? Zie het hoofdstuk over Notationele afspraken. (Zo'n expressie introduceert nondeterminisme, verstoort de Church-Rosser eigenschap, en maakt het begrip van semantische gelijkwaardigheid van Hoofdstuk 5 ingewikkelder).

1.17 Definieer een reduktiestrategie zo dat iedere terminerende reduktie een irreducibele expressie oplevert (zoals bij de normal order strategie) die bovendien in resultaatvorm is (zoals bij U en C). (Wanneer zoiets niet bereikbaar is moet de reduktie dus stokken of niet-termineren).

1.18! Hebben (fn x. x*2) en (fn x. x+2) een gemeenschappelijk reduktieresultaat? Zou je die expressies "semantisch gelijkwaardig" willen noemen? (Dit wordt formeel gedaan in Hoofdstuk 5; voor bovenstaande expressies hebben we typering nodig!).

1.19 Probeer eens een expressie te verzinnen, zonder rec erin, waarvan de reduktie oneindig voortloopt, niet stopt. Doe dit ook voor de U en C strategieën. (Dit is niet eenvoudig. Een oplossing wordt behandeld in Hoofdstuk 6).

1.20! Verzin een representatie in HOF voor gehele (zowel positieve als negatieve) getallen, en definieer daarbij de gebruikelijke rekenkundige

bewerkingen zoals minus, maal, gelijk, plus, deel, rest, enzovoorts.

1.21! Verzin een representatie in HOF voor rationale getallen, en definieer daarbij de gebruikelijke bewerkingen.

1.22! Verzin een expressie met een stokkende C-reduktie maar terminerende U-reduktie. Kan de C-reduktie van een expressie termineren terwijl de U-reduktie stokt?

1.23 Voeg een expressie (for x to e prefix e) toe aan HOF (met formele definities van bindingsrelaties en de reduktieregels) zó dat (for x to m prefix e) => (e[x/m]: e[x/m-1]: ...: e[x/0]: nil). Schrijf ook een HOF funktie f zó dat (f e0 (fn x. e)) tot datzelfde resultaat reduceert. Is er enig verschil tussen de S-reduktie (S = U of C) van (for x to e0 prefix e) en (f e0 (fn x. e))?

1.24! De elementen van een lijst kunnen zelf ook weer een lijst vormen; we spreken in dat geval over een geneste lijst. Kun je in HOF een funktie definiëren die de nestingsdiepte van een lijst berekent?

1.25! Voeg aan HOF de expressievorm (isfct e) toe, met reduktieregels

```
(isfct (fn. x. eb)) -> true
(isfct <e0, e1>)    -> false
(isfct (eh: et))    -> false
(isfct  nil)        -> false
(isfct  true)       -> false
(isfct  false)      -> false
(isfct  n)          -> false
```

En net zo iets voor (islist e), (isnbr e), (isgrp e), (isbool e). Doe nu nogmaals opgave 1.24.

1.26! Geef een voorbeeld van een zinvol gebruik van de definitievorm (df x=---x---. eb), (herinner je dat dit een niet rekursieve definitie is!). (Wenk: herdefinieer een globaal gegeven funktie; bijvoorbeeld in Pascal: een eigen procedure new die de rol van de standaard procedure new overneemt). Hoe kan men in Pascal of Algol 68 zo'n niet-rekursieve definitie df x=---x--- realiseren?

1.27 Bedenk een semantiek behoudende vertaling van SASL naar HOF-met-de-L-strategie, en omgekeerd.

1.28! Bewijs dat een reduktie geen vrije variabelen introduceert, dat wil zeggen e => e' impliceert VV(e) ⊇ VV(e'). Geldt ook BV(e) = BV(e')?

1.29 Geef een formeel afleidingssysteem (axioma's en afleidingsregels) voor formules van de vorm e ==> e' zo dat e ==> e' afleidbaar is precies wanneer e => e' volgens de C-strategie.

1.30 Doe Oefening 1.29 ook met de U- of N-strategie in plaats van de C-strategie. Wenk: zie de wenk in Oefening 1.10.

1.31 Definieer de Collaterale C-strategie CC als volgt. "Kies ter kontraktie een of andere, niet noodzakelijk meest linker, minimale non-resultaatvorm die niet in een then- of else-tak, bereik van een bindingsexpressie of resultaatvorm ligt." Is voor HOF het CC-resultaat altijd identiek aan het C-resultaat, indien beide bestaan? Is het zo dat beide hetzelfde terminatie/ nonterminatie gedrag vertonen? Geef zonodig voorbeelden ter illustratie van het verschil.

1.32 Geef expressies e1, e2, e3, ... zodat de N- en U-reduktie van ei tot aan de i-de kontraktie identiek zijn en daarna verschillen maar wel beide termineren.

1.33! Laat zien dat in het algemeen een expressie tot verschillende resultaat-vormen gereduceerd wordt (anders dan louter een verschil in naamgeving) al naar gelang C of U gebruikt wordt. Geef voorbeelden.

1.34! Kan een U-reduktie termineren terwijl de N-reduktie niet termineert? En omgekeerd?

1.35! Geef de gezichtsvelden van de introdukties in

```
    df x1 = e1, ..., xn = en. eb
    df x1 = e1; ...; xn = en. eb
dfrec x1 = e1, ..., xn = en. eb
```

1.36! Vertaal enige SASL- (of TWENTEL of KRC of LISP) programma's in HOF.

HOOFDSTUK 1A

STATISCHE VERSUS DYNAMISCHE SCOPE

De meeste programmeertalen, o.a. HOF, Algol, Pascal en FORTRAN, hebben zogenaamde statische scope. Sommige talen, o.a. Lisp en APL, hebben dynamische scope. Volgens ons is dynamische scope een grove ontwerpfout, die voortkomt uit een daardoor mogelijk gemaakte eenvoudige implementatie. We zullen beide begrippen hier omschrijven.

Statische scope

We zeggen dat een voorkomen van een variabele statisch gebonden is als het bindend voorkomen ervan éénduidig vastligt en algoritmisch (berekenbaar uit de tekst, dus statisch ofwel at compile-time) bepaald is. We zeggen dat een taal statische scope heeft als alle voorkomens van variabelen statisch gebonden zijn en alfabetische varianten semantisch gelijkwaardig zijn. Bijvoorbeeld, de volgende vier alfabetische varianten reduceren alle tot eenzelfde resultaat, nl. 2.

```
e1 = (df x=1; f=(fn y. x). (df x=2. f 0) + (df x=3. f 0))
e2 = (df x=1; f=(fn y. x). (df z=2. f 0) + (df z=3. f 0))
e3 = (df x=1; f=(fn y. x). (df z=2. f 0) + (df x=3. f 0))
e4 = (df x=1; f=(fn y. x). (df x=2. f 0) + (df z=3. f 0))
```

Dynamische scope

Heeft een taal geen statische scope, dan spreken we van "een of andere dynamische scope". Van de vele mogelijkheden die er dan nog zijn, staat er één in het bijzonder bekend als (de) dynamische scope. Een algemeen geldige omschrijving is moeilijk te geven omdat die afhangt van de semantiek, en vooral van de manier waarop de semantiek geformuleerd is. We geven eerst een voorbeeld. Beschouw wederom de expressies e1,..., e4. De x in (fn y. x) wordt volgens dynamische scope niet gebonden door een binding van x in de kontekst van die abstraktie (dat zou statische scope zijn), maar door een binding van x in de omgeving waarin die abstraktie wordt aangeroepen (en bijgevolg de x buiten het afschermende schild van de lambda komt door de (app)regel). Dus e1 reduceert dan tot 2+3 = 5, e2 tot 1+1 = 2, e3 tot 1+3 = 4 en e4 tot 2+1 = 3.

Het zijn alleen de vrije voorkomens binnen abstrakties die aldus dynamisch gebonden worden. Bijvoorbeeld, in

e1´ = (df x=1; f=x. (df x=2. f) + (df x=3. f))

wordt de x in de definiërende expressie voor f statisch gebonden. Dus e1´ en al zijn alfabetische varianten reduceren tot 1+1 = 2. En als laatste voorbeeld,

e5 = (df x=1; f=(fn y1. fn y2. x). df x=2; g=(f 0). df x=3. g 0)

reduceert onder dynamische scope tot 3; de x in de abstraktie f komt pas ten gevolge van de aanroep (g 0) buiten het schild van afschermende lambda´s.

In veel gevallen is het effekt van dynamische scope te simuleren met statische scope; met name als alle gebruik van funktie-expressies als funktiedeel in een applikatie te achterhalen is, (en dat is in het algemeen niet het geval). Geef daartoe iedere abstraktie een stel extra parameters, en wel deze: iedere variabele die in die abstraktie een dynamisch gebonden voorkomen heeft. Voorts zet je bij iedere toepassing van zo´n abstraktie die variabelen zelf ook nog als extra argumenten. Dus e1, .., e4 worden nu:

e1" = df x=1; f=(fn(y,x).x). (df x=2. f(0,x))+(df x=3. f(0,x))
e2" = df x=1; f=(fn(y,x).x). (df z=2. f(0,x))+(df z=3. f(0,x))
e3" = df x=1; f=(fn(y,x).x). (df z=2. f(0,x))+(df x=3. f(0,x))
e4" = df x=1; f=(fn(y,x).x). (df x=2. f(0,x))+(df z=3. f(0,x))

en

e5" = df x=1; f=(fn (y1,x). fn (y2,x).x). df x=2; g=f(0,x). df x=3. g(0,x)

Let vooral op e5"! (Merk op dat e1", ..,e5" geen dynamisch gebonden voorkomens hebben, en dus onder statische scope en dynamische scope tot hetzelfde resultaat leiden.

Historisch is dynamische scope ontstaan bij de definitie van programmeertalen in termen van het omgevingenmodel. Daarin is dynamische scope namelijk iets eenvoudiger te implementeren dan statische scope. We zullen dat nog tegenkomen in Hoofdstuk 4, Par. 4.2.;daar staat ook een formele definitie van dynamische scope. (Een eenvoudiger implementatie betekent overigens niet dat de evaluatie

efficienter gaat. Het tegendeel lijkt hier het geval te zijn.)

Wij vinden dynamische scope een ontwerpfout, omdat het indruist tegen alle informele bindingsregels in wiskundige en andere teksten. Met name heeft dynamische scope als nadeel dat alfabetische varianten niet meer gelijkwaardig zijn. Dus lokale naamgeving, waarbij geen rekening gehouden hoeft te worden met de naamgeving in de kontekst, is niet mogelijk.

Oefeningen

1A.1! Laat zien dat er bij dynamische scope geen parameters nodig zijn. Wenk: vervang (fn x. eb) door (fn (). eb) en de applikaties ((fn x. eb) ea) door (df x=ea. (fn(). eb)).

1A.2! Stel dat de volgende programma's met dynamische scope geevalueerd worden.

a. df f = (fn x. 1). (fn g. df f = (fn x. 2). g 3) (fn y. f y)

b. df f = (fn x. 1). (fn g. df f = (fn x. 2). g 3) f

c. Verzin zelf iets nieuws

Wijzig ze zodanig dat met statische scope dezelfde resultaten worden verkregen; voeg daartoe aan abstrakties en applikaties extra parameters en argumenten toe, zoals in dit hoofdstuk beschreven is.

d. df f = (fn x.1); h = (fn x. f x). (fn g. df f = (fn x.2). g 3) h

e. df x = 0. df f = <fn y. y+x, fn y. y-x>. df x=3. (f.1)2

f. df x = 0. df f = (fn y.x). df g = (fn (h,y). h y). df x=1. g(f,2)

1A.3! Verzin een expressie die onder statistische scope stokt en onder dynamische scope niet.

HOOFDSTUK 2

VERGELIJKING VAN REDUKTIESTRATEGIEEN

Vanwege de gelijkwaardigheid van reduktiestrategieën voor een beschrijvende programmeertaal, zie Hoofdstuk 1, lijkt het erop dat de strategieën alleen verschillen in het al of niet bepaald zijn van het eindresultaat van een reduktie. Qua "semantiek" is dit inderdaad het enige verschil, maar qua "pragmatiek" vloeien hier nog andere verschillen uit voort. We zullen de pragmatische voor- en nadelen van de U-, L- en C-strategie uiteenzetten, en vervolgens aangeven hoe (of in hoeverre) de ene strategie geprogrammeerd kan worden wanneer de reductie volgens een andere strategie plaatsvindt.

Par. 2.1 Voor- en nadelen van de strategieen

De Uitstel- en de Luie strategie

De verschillen hiertussen zijn alleen maar pragmatisch: de U- en L-reduktie van een expressie termineren beide wel, of beide niet. {Er lijkt zelfs te gelden dat iedere reduktiestap in de L-reduktie korrespondeert met een aantal opvolgende reduktiestappen in de U-reduktie zodat voor overeenkomstige tussenresultaten eL en eU geldt eU=>eL, d.w.z. eL is minstens zover gereduceerd als eU. In het bijzonder geldt dat voor de eindresultaten.} We kunnen de L-strategie daarom opvatten als een efficiënte implementatie van de U-strategie.

Oneindige datastrukturen

Bij de U- en L-strategie zijn "oneindige" datastrukturen eenvoudig uit te drukken. Bijvoorbeeld, in de scope van

df a = (rec x. 1: x)

"staat a voor" (1: (1: (1: .....)))), de oneindige lijst van enen, of: de willekeurig ver gereduceerde expressie (1: (1: (1: (1: ...(rec x. 1: x).....)))). Wanneer a in het geheel niet wordt gebruikt, wordt ook zijn definiërende expressie niet gereduceerd. Wordt ooit (hd tl tl a) gereduceerd, dan wordt de definierende expressie slechts zover gereduceerd als nodig is:

```
          (hd tl tl (rec x. 1: x))
(rec)  => (hd tl tl (1: (rec x. 1: x)))
(tl)   => (hd tl (rec x. 1: x))
(rec)  => (hd tl (1: (rec x. 1: x)))
(tl)   => (hd (rec x. 1: x))
(rec)  => (hd (1: (rec x. 1: x)))
(hd)   => 1
```

Bij de Luie strategie is er slechts één expressie voor alle voorkomens van a, en die expressie zou door de reduktie van (hd tl tl a) driemaal aan de (rec)-contraktie onderworpen worden. Dus voor alle andere voorkomens van a is de expressie nu ook (1: (1: (1: (rec x. 1: x)))) geworden. De volgende redukties van a hoeven die drie eerste (rec)-kontrakties niet te herhalen. Overigens, in Hoofdstuk 3 geven we een implementatie waarbij nóóit een (rec)-kontraktie wordt gedaan, omdat rec-expressies "zonder rec" (maar als het ware al oneindig ver uitgereduceerd) gerepresenteerd worden. Een interessanter voorbeeld is het volgende.

```
df fib = ((rec ext. fn a b. a: (ext b (a+b))) 1 1)
```

In het bereik hiervan staat fib voor de oneindige lijst van Fibonacci-getallen:

```
fib => 1: 1: 2: 3: 5: 8: ... a: ((rec----) b (a+b))
```

(voor opeenvolgende Fibonaccigetallen a en b). Wanneer het voor het bereiken van een eindresultaat niet noodzakelijk is om fib geheel uit te reduceren, dan wordt die oneindig voortlopende reduktie inderdaad uitgesteld.

Coroutines

Bij de L-strategie kunnen ingewikkelde berekeningen in afzonderlijke fasen (slagen, passes) worden uitgedrukt waarbij (grote) datastrukturen als "tussenresultaat" lijken op te treden, terwijl die fasen van de berekening toch verweven --en niet de een na de ander-- plaatsvinden. In feite treedt steeds een (klein) onderdeel van de datastruktuur als tussenresultaat op. Aldus wordt het gemak van programmering gekombineerd met de efficiëntie van geheugengebruik.

Een heel eenvoudig voorbeeld is dit. Stel dat het eerste Fibonacci-getal gevraagd wordt dat een veelvoud is van 117. We kunnen dan eerst "in een afzon-

derlijke fase" de (oneindige) rij van Fibonaccigetallen definiëren, en "in de tweede fase" uit die rij het kleinste 117-voud zoeken. Dus

```
(df fib = ((rec ext. fn a b. a: ext b (a+b)) 1 1)
 ;  f rec= (fn 1. if ((hd 1) rem 117) eqn 0
                    then (hd 1) else f (tl 1))
 .  f fib
 )
```

Merk op dat fib maar één keer gebruikt wordt (voorkomt); bijgevolg "verdwijnt", zelfs bij luie reduktie, het beginstuk van de rij waarvan al gekonstateerd is dat het geen 117-voud heeft.

Een meer realistisch voorbeeld is het volgende. Gegeven een tekst, bestaande uit regels van verschillende lengten. Gevraagd de tekst als volgt te transformeren. Ten eerste moeten de regels alle precies 80 karakters lang worden; ten tweede moeten alle spatiegroepen zo mogelijk tot één spatie of een regelovergang samengevoegd worden; ten derde moeten hoofdletters door overeenkomstige kleine letters worden vervangen; en tenslotte moeten woorden (van minder dan 81 karakters) niet afgebroken worden, regels moeten zonodig met spaties worden aangevuld. Als we de vier vereisten in afzonderlijke fasen kunnen programmeren, dan is een programma snel geschreven. De optredende datastrukturen zijn de steeds meer getransformeerde versies van de tekst. Met de luie strategie zijn die nooit geheel (in resultaatvorm) aanwezig, maar worden zij slechts beetje bij beetje voortgebracht.

Precies ditzelfde verschijnsel kan in konventionele talen ook met coroutines worden bereikt. Coroutines verschillen van subroutines (procedures en funkties), doordat zij na het opleveren van een resultaat later weer "doorgestart" kunnen worden op het punt waar zij voor het laatst gebleven waren. Die achtereenvolgens opgeleverde (tussen)resultaten vormen tesamen dan zo'n datastruktuur, zo als de getransformeerde versies van de tekst in bovenstaand voorbeeld. In een ad-hoc notatie kunnen we met behulp van coroutines het eerste Fibonaccigetal dat een 117-voud is als volgt vinden.

```
df fib = coroutine
           var a, b := 1, 1; return a;
           while true
           do a, b := b, a+b; return a od
         end;
```

```
        f = (fn c.
              process ℓ = activate c;
              var z := call ℓ;
              while z geen 117-voud do z := call ℓ od;
              z)
    . f fib
```

Volgens de kompositionele reduktie worden in een applikatie de argumenten eerst tot resultaatvorm gereduceerd, alvorens de romp van de funktie wordt bekeken. De redukties van argument en funktie vinden niet verweven plaats. Er moeten dan speciale maatregelen getroffen worden om bovenstaand effekt te bereiken; bijvoorbeeld de introduktie van coroutines.

Beëindiging van herhalingen

Er is bij de U- en L-strategie geen noodzaak om "extra lus-uitgangen" (in HOF: rekursie-beëindigingen) aan te geven. Beschouw bijvoorbeeld de funktie all die bepaalt of alle elementen van een lijst true zijn:

```
    df and = (fn x y. if x then y else false)
     ; all rec= (fn z. if eqnil z then true
                       else and (hd z) (all (tl z)) )
     ;   a = (false: true: true:....)
```

De applicatie (all a) zal bij de kompositionele strategie aanleiding geven tot een "rekursiediepte" gelijk aan de lengte van a. Maar bij de uitstel- en luie strategie zal all niet rekursief worden geaktiveerd omdat in

```
    and (hd a) (all (tl a))
```

de uitgestelde reduktie van (all (tl a)) wordt afgesteld:

```
    and (hd a) (all (tl a))
(app) => if (hd a) then (all (tl a)) else false
(hd)  => if false then (all (tl a)) else false
(if)  => false
```

Eliminatie van konstante berekeningen

De luie strategie heeft, anders dan de uitstel- en de kompositionele stra-

tegie, een optimaliserend effekt. Zo is er soms geen noodzaak om konstante berekeningen buiten een lus of rekursie te halen. Dat komt doordat bij de luie strategie (beter: de luie implementatie van de uitstel-strategie), expressies nooit worden gedupliceerd maar altijd gedeeld worden door de betreffende voorkomens. Beschouw bijvoorbeeld

```
df f = (fn x. (3+4+5)+x). (f 1)+(f 2)
```

Na reduktie van (f 1) tot resultaatvorm is de romp van f ook gereduceerd tot 12+x. Bij de reduktie van (f 2) wordt dus de reduktie van (3+4+5) tot 12 niet herhaald. De programmeur hoeft dus geen transformaties op zijn tekst uit te voeren om dat soort berekeningen buiten de herhalingen te krijgen. (De noodzaak zou bij een rekursieve funktie natuurlijk groter zijn dan bij de niet-rekursieve funktie hierboven).

Dit optimaliserend effekt van de luie strategie verschijnt ook nog in een ogenschijnlijk heel andere gedaante: de extra kosten voor het gebruik van hogere orde funkties vallen na het eerste gebruik ervan geheel weg. De funktie all van enige alinea's terug zouden we ook met de "standaard"funktie litr kunnen definiëren.

```
df litr = fn f a. rec g. fn l.
            if eqnil l then a else f (hd l) (g (tl l))
 ; all  = litr and true
```

Bij de eerste reduktie van (all ....) zal de definiërende expressie voor all, (litr and true), gereduceerd worden tot exakt dezelfde expressie die wij al eerder gaven. Volgende redukties van (all ....) vinden dan voor all de expliciete definiërende expressie; de reduktie van (litr and true) vindt niet meer plaats.

Kompositionaliteit

De uitstel- en luie redukties zijn niet kompositioneel. Het blijkt voor de programmeur en de menselijke lezer heel moeilijk te zijn zich een beeld te vormen van de volgorde waarin de reduktiestappen plaatsvinden. Zo een beeld is niet nodig voor een korrektheidsargumentatie, maar wel om bij foutmeldingen tijdens de reduktie (exekutie) het verloop van de berekening en daarmee de oorzaak van de fout te achterhalen. Afhankelijk van je standpunt ten aanzien van ontluizen (debuggen) is dit in het voor- of nadeel van een niet-kompositionele reduktiestrategie.

Ook blijkt het héél erg lastig te zijn om een greep op de prestatie te krijgen bij niet-kompositionele strategieën, in het bijzonder bij de luie strategie. Misschien is dit onvermogen slechts van tijdelijke aard; anno 1983 wordt er nog naarstig onderzoek gepleegd naar onder andere deze aspekten van de reduktiestrategieen.

Ruimtebeslag

In een implementatie van de reduktie dient ieder tussenresultaat op de een of andere manier gerepresenteerd te zijn. Daar zijn natuurlijk allerlei mogelijkheden voor, zelfs mogelijkheden die we nu misschien nog niet kennen. Toch kunnen we enige voorspellingen doen ten aanzien van de ruimte die in beslag genomen wordt voor de representatie van tussenresultaten. Beschouw de expressie

```
(df sum rec= fn s n. if n eqn 0 then s else sum (s+n)(n-1)
  . sum 0 1000
)
```

De C- resp. L-reduktie zien er als volgt uit. Hierbij staat sum' voor (rec sum. fn s n. .....) en sum'' voor (fn s n. ---- sum' (s+n)(n-1)).

```
expr                                  expr
⇒ sum' 0 1000                         ⇒ sum' 0 1000
=> sum' (0+1000)(1000-1)              => sum' (0+1000)(1000-1)
=>                                    => sum' (0+1000+999)(999-1)
=> sum' (1000+999)(999-1)             => sum' (0+1000+999+998)(998-1)
=>                                    .
=> sum' (1999+998)(998-1)             .
=>                                    .
.                                     => sum' (0+1000+999+998+...+1)(1-1)
.                                     => (0+1000+999+998+...+1)
.                                     => (1000+999+998+...+1)
.                                     => (1999+998+...1)
.                                     => (2997+...+1)
.                                     .
.                                     .
=> 500500                             => 500500
```

Het aantal reduktiestappen is voor beide redukties gelijk, en het eindresultaat natuurlijk ook. In de C-reduktie is de lengte van de optredende tussenresultaten onafhankelijk van de toevallig gekozen argumenten 0 en 1000. Inderdaad is het in

beginsel mogelijk de evaluatie van dat programma volgens de C-strategie in konstante opslagruimte uit te voeren. Dus wat de benodigde opslagruimte betreft gedraagt dat programma zich onder de C-strategie net zo als het gebiedende programma

```
s, n := 0, 1000;
while not n eqn 0 do s, n := s+n, n-1 od;
s
```

Maar het is heel anders gesteld met de L-reduktie. Daar wordt de reduktie van het eerste argument uitgesteld totdat de waarde daarvan nodig blijkt wanneer namelijk het tweede argument eqn 0 is. Totdat moment groeit de expressie voor dat eerste argument enorm aan; op het hoogtepunt (dieptepunt? middelpunt?) van de reduktie luidt die

```
(...(((0+1000)+999)+998)...+1)
```

De benodigde opslagruimte bij tweede argument n neemt dus lineair toe met n.

Soms zal de lineaire aangroeiing van de benodigde opslagruimte bij de reduktie van een rekursieve funktie niet bezwaarlijk zijn: na voltooiing komt die ruimte weer vrij. Maar stel nu dat van een zeer groot bestand met persoonsgegevens de som der salarissen berekend moet worden. Bij reduktie volgens de U- en L-strategie kan de ruimte dan volkomen uitgeput raken. In zo´n berekening zal op de een of andere manier de C-strategie afgedwongen moeten worden.

Het verschijnsel van de aangroeiende tussenresultaten in een L-reduktie betekent tevens een afzwakking van onze uitspraken die wij deden in de subparagraaf ´coroutines´. Daar beweerden we dat de oneindige datastrukturen slechts beetje-bij-beetje gevormd worden en niet helemaal uitgereduceerd aanwezig zijn. Maar beschouw nu eens het volgende programma

```
(df down rec= fn n. if n eqn 0 then nil else n: down (n-1)
; sum´ rec= fn s 1. if eqnil 1 then s else sum´ (s+hd 1) (tl 1)
. sum´ 0 (down 1000)
)
```

De L-reduktie van (sum´ 0 (down 1000)) verloopt bijna net zo als die van (sum 0 1000); het tweede argument van sum´ heeft de vorm (down(n-1)) als die van sum de vorm (n-1) heeft. Dus weliswaar is (down 1000) geen moment aanwezig in de vorm

1000: 999: 998: ...: 1: nil ,

zijn elementen zijn er wel allemaal in het tussenresultaat

sum' (((0+1000)+999)+...+1) (down (1-1))

Dit lijkt een niet te verwaarlozen nadeel te zijn van de L- (en zeker van de U-) strategie.

Par. 2.2 Programmering van reduktiestrategieën

In het voorgaande hebben we gezien dat wat sommige pragmatische aspekten betreft de L- (of U-)strategie te verkiezen is boven de C-strategie, en wat andere aspekten betreft de C- boven de L- (of U-)strategie. We laten nu zien hoe de ene strategie geprogrammeerd of nagebootst kan worden in de andere. Hiervoor zijn soms nieuwe expressievormen nodig; we zullen de reduktieregels wel geven maar de aanpassing van de formele strategie-definities niet.

Programmering van de U-strategie in C

Dankzij het feit dat zelfs volgens de C-strategie nooit binnen abstrakties wordt gereduceerd, kunnen we als volgt het uitstel van redukties afdwingen. Plaats iedere expressie e waarvan de reduktie moet worden uitgesteld, in een abstraktie met geen of een loze parameter: e wordt (fn (). e) of (fn x. e) met x niet vrij in e. Maak voorts van iedere expressie e waarvan de reduktie weer op gang gebracht moet worden, een applikatie met geen of een loos argument: e wordt (e ()) of (e 0). Een voorbeeld moge dit verduidelijken.

We willen een funktie if definiëren zo dat (if e1 e2 e3) eenzelfde resultaat heeft als (if e1 then e2 else e3). De volgende poging faalt.

(df if = (fn x y z. if x then y else z). ...(if e1 e2 e3)...)

Immers, bij de applikatie (if e1 e2 e3) worden volgens de C-strategie alle argumenten gereduceerd alvorens de applikatie wordt gekontraheerd. Bij (if e1 then e2 else e3) is dat niet zo. De volgende poging slaagt.

(df if = (fn x y z. if x then y() else z()).
...if e1 (fn (). e2) (fn (). e3)...)

De argumenten (fn (). e2) en (fn (). e3) staan al in resultaatvorm.

Op deze manier kunnen we bijvoorbeeld ook oneindige strukturen, zoals lijsten, representeren. Een oneindige lijst wordt dan gerepresenteerd als een funktie met een loos argument die bij applikatie de kop van de lijst tesamen met de representatie van de oneindige staart geeft. Het eerste Fibonaccigetal dat een 117-voud is kan dan als volgt gevonden worden.

```
(df fibVanaf rec= (fn a b. fn (). <a, fibVanaf b (a+b)>)     *)
  ; fib  = fibVanaf 1 1
  ; f rec= (fn ℓ. (df k = ℓ().0, s = ℓ().1
                   . if k is 117-voud then k else f s))
  . f fib
  )
```

*) Bij voor de hand liggende typeringen zou (fn (). a: fibVanaf b (a+b)) niet goedgetypeerd zijn, vandaar onze kodering m.b.v. groepen. Volgens de reduktieregels zou het goed gaan met (fn (). a: fibVanaf b (a+b)).

Let wel, zo als het er nu staat wordt "per aanroep van f" de applikatie (ℓ()) tweemaal gereduceerd. Het effekt van de L-strategie wordt dus met deze nabootsing niet bereikt. Dat doel vereist nog meer aanpassingen in de programmatekst. In bovenstaand voorbeeld gaat dat nog vrij eenvoudig; in het algemeen misschien niet meer.

Het uitstellen en weer op gang brengen van de reduktie wordt ook wel, in sommige programmeertalen, gekoppeld aan speciaal daartoe gemerkte parameters en de overeenkomende argumenten: de zogenaamde call-by-name van Algol 60. In HOF zouden wij dat als volgt vorm kunnen geven.

- (fnname x. eb) is een nieuwe expressievorm, overeenkomend met (fn x. eb[x/x()])
- een applikatie (ef ea) waarbij ef reduceert tot (fnname x. eb), staat eigenlijk voor (ef (fn (). ea)).

Dus de reduktie van argumenten called-by-name wordt impliciet uitgesteld, terwijl gebruik van die argumenten in de romp van de funktie de uitgestelde

reduktie impliciet weer op gang brengt. Nota bene, om de Overeenkomst tussen Parameterisatie en Definitie in stand te houden moeten we ook "define-by-name" invoeren:

(dfname x=ea. eb)

Deze expressie staat dan voor (df x=(fn (). ea). eb[x/x()]).

Programmering van de L-strategie in C.

Zojuist hebben we gezien hoe een uitstel van reduktie geprogrammeerd kan worden onder de C-strategie. Wat nu nog nodig is, is dat redukties niet onnodig worden herhaald, ofwel dat expressies gedeeld worden gerepresenteerd. Zoiets is niet te programmeren, dat is een implementatie-aspekt. In wat nu volgt gaan we ervan uit dat de substitutie eb[x/ea] wordt geëffektueerd door de ea slechts eenmaal op te slaan en de bedoelde voorkomens van x daarnaar te laten verwijzen. (Deze techniek wordt uitvoerig behandeld in Hoofdstuk 4).

Een algemene methode om naar wens een luie reduktie te bereiken is er niet. In sommige gevallen kan herhaalde reduktie, van een expressie e, voorkomen worden door een definitie van df x=e in te lassen op het punt waar e voor het eerst gereduceerd zou worden; in ~~het~~ de ~~bereik~~ scope van die definitie staat x dan voor het resultaat van reduktie van e. Maar in veel gevallen lukt dit niet, doordat het niet beslisbaar is of e wel of niet gereduceerd zal worden. Daarom worden er nieuwe expressie-vormen ingevoerd. We beschouwen twee mogelijkheden.

Ten eerste zijn er de expressievormen (delay e) en (force e). De vorm (delay e) heeft ongeveer hetzelfde effekt als (fn (). e); in feite wordt van e een voorschrift gemaakt waarin staat hoe de resultaatvorm van e gevonden kan worden, (of wat de resultaatvorm is). De vorm (force e) heeft ongeveer hetzelfde effekt als (e ()); in feite wordt volgens het voorschrift dat door e wordt aangeduid de resultaatvorm verkregen, en wordt tevens dat voorschrift gewijzigd zo dat voor volgende force's het resultaat al klaar staat. Hiermee kan het eerste Fibonaccigetal dat een 117-voud is als volgt gevonden worden.

```
(df fibVanaf rec= (fn a b. delay (a: fibVanaf b (a+b)) )
 ;  fib = fibVanaf 1 1
 ;  f rec= (fn ℓ. (df k = (hd (force ℓ)), s = (tl (force ℓ))
                  . if k is 117-voud then k else f s))
 . f fib
```

)

Van de twee forces op l is er één die in het voorschrift aangeduid door l het resultaat al klaar vindt; de ander zal eerst de delay "wegreduceren" totdat het resultaat (a: delay (b: fibVanaf..)) gevormd is. Merk op dat de introduktie van de nieuwe expressies gepaard moet gaan met een toevoeging aan de definitie van de C-strategie: de te kontraheren nonresultaatvorm wordt niet binnen een delay-expressie gekozen.

De tweede mogelijkheid is als volgt. Het uitstellen en weer op gang brengen wordt weer gekoppeld aan een speciaal daartoe gemerkte parameter, en de daarmee overeenkomende argumenten: de zogenaamde call-by-need. In HOF zouden wij dat als volgt vorm kunnen geven.

- (fnneed x. eb) staat voor (fn x. eb[x/force x])
- (ef ea) met ef reducerend tot (fnneed x. eb) staat voor (ef (delay ea))

Dus geheel analoog aan call-by-name. Het verschil met call-by-name is dat argumenten called-by-name meermalen gereduceerd kunnen worden (in HOF: met steeds hetzelfde resultaat), terwijl argumenten called-by-need hoogstens éénmaal worden gereduceerd, en dat resultaat wordt dan ook voor de overige voorkomens genomen. (Omwille van de Overeenkomst zouden we nu ook de vorm (dfneed x=ea. eb) moeten introduceren. Deze staat dan voor (df x=(delay ea). eb[x/force x]).)

Programmering van de C-strategie in L of U

Om naar wens sommige expressies eerder te laten reduceren dan volgens de L- of U-strategie het geval zou zijn, is een nieuwe expressievorm nodig. Allereerst is er de volgende mogelijkheid. Kies als nieuwe expressievormen (fnval x. e) en (dfval x=e. e) met reduktieregels

((fnval x. eb) ea) -> eb[x/ea] mits ea in resultaatvorm
(dfval x=ea. eb) -> eb[x/ea] " " " "

En wijzig nu nog de diverse definities zo dat zelfs volgens de L- en U-strategie het argumentdeel van ((fnval x. eb) ea) eerst tot resultaatvorm wordt gereduceerd alvorens bovenstaande reduktieregel wordt toegepast. Analoog voor (dfval x = ea. eb). (Je kunt nog kiezen of je ea op zich weer wel volgens de L- of U- strategie laat evalueren, of direkt maar volgens de C-strategie). (Dit lijkt op de zg. call-by-value parametertransmissie van Algol 60; zonder de aanwezigheid van

van assigneerbare variabelen is een belangrijk aspekt van die call-by-value hiermee nog niet aangegeven).

De vervroegde reduktie hoeft niet per se aan parameterisatie en definities gekoppeld te zijn. Laten we aannemen dat substituties eb[x/ea] geïmplementeerd worden door éénmaal de expressie ea op te slaan en alle voorkomens van x in eb daarnaar te laten verwijzen (zoals in de implementatie modellen van Hoofdstukken 3 en 4). Kies dan als nieuwe expressievorm (value e in eb) met reduktieregel

(value e in eb) -> eb mits e in resultaatvorm

Voordat (value e in eb) wordt gekontraheerd, moet e al gereduceerd zijn. We kunnen dan in plaats van (fnval x. eb) ook schrijven (fn x. value x in eb); en omgekeerd, in plaats van (value e in eb) ook ((fnval x. eb) e).

Als illustratie programmeren we wederom het sommeren van de lijst (down 1000) = (1000: 999: ...: 1: nil).

```
(df down rec= (fn n. if n eqn 0 then nil else n: down(n-1))
  ; sum" rec= fn (ℓ, val s).
              if eqnil ℓ then s else sum"(tl ℓ , s + hd ℓ )
  . sum" (down 1000, 0)
  )
```

In de reduktie van (sum" (down 1000, 0)) zal de hulpdatastruktuur (down 1000) nooit in resultaatvorm aanwezig zijn, terwijl bovendien het tweede argument van sum" niet zal aangroeien tot (0+1000+998+...) maar steeds onmiddellijk tot een getalkonstante wordt gereduceerd. Vergelijk dit met wat we konstateerden in de subparagraaf ´Ruimtebeslag´.

Belangrijke Onderwerpen uit Hoofdstuk 2

voordelen luie strategie: eenvoudige programmering van
  oneindige datastrukturen
  coroutine-effekt
  eliminatie van konstante berekeningen
  kommunicerende processen
  beëindiging van herhalingen
nadelen luie strategie:
  ruimtebeslag
  (redeneren over efficiëntie en run-time fouten)
U-simulatie d.m.v.
  parameterloze funkties
  call-by-name
L-simulatie d.m.v.
  delay en force
  call-by-need
C-simulatie d.m.v.
  call-by-value

Literatuur bij Hoofdstuk 2

Zie hiervoor de verwijzingen naar literatuur over Funktioneel Programmeren (bij de Inleiding van deze syllabus) en over Lazy evaluation (bij Hoofdstuk 3).

Oefeningen bij Hoofdstuk 2

2.1! Programmeer in HOF onder de Luie strategie oneindige lijsten zoals: (i) alle priemgetallen, (ii) alle getallen die alleen faktoren 2, 3 en 5 hebben (in opklimmende volgorde), (iii) alle woorden (in volgorde van opklimmende lengte en per lengte in alfabetische volgorde), enzovoorts.

2.2 Geef zo getrouw mogelijke simulaties van de programma's van Oefening 2.1 in Pascal en/of Algol 68.

2.3 Programmeer het "realistische voorbeeld" uit de subparagraaf 'Coroutines' in HOF, Pascal, Algol 68 en een taal met Coroutines.

2.4 Neem een willekeurig door U zelf geschreven programma, en ga na hoeveel assignments of definities alleen tot doel hebben om konstante berekeningen niet herhaald te laten plaatsvinden.

2.5! Geef de L- en C-redukties van (df f=(fn x. (3+4+5)+x). f 1 + f 2) en van ((fn x. (3+4+5)+x) 1) + ((fn x. (3+4+5)+x) 2). Geef alle sharing (deling) aan.

2.6! Geef de L-reduktie van (fib 10) resp (fib' 10 1 1) waarbij gedefinieerd is

```
fib rec= fn n. if n eqn 0 then 1 else
               if n eqn 1 then 1 else
               fib(n-1) + fib(n-2)

fib'rec= fn n a b.
            if n eqn 0 then a else fib' (n-1) b (a+b)
```

2.7! Geef de L-reduktie van (sum''' 0 (down 1000)) waarbij

```
sum'''rec= fn (val s, val 1). if eqnil 1 then s else sum(s + hd 1, tl 1)
```

en vergelijk met (sum 0 (down 1000)) uit de subparagraaf 'Ruimtebeslag' en sum"(down 1000, 0) uit de subparagraaf 'Programmering C in L', en met de C- reduktie.

2.8! Definieer in HOF met call-by-name een funktie cand zó dat cand(false, e) => false zonder e te evalueren, en voorts cand(true, k)=>k voor k=true en k=false. Definieer of simuleer zo'n funktie ook in Pascal of Algol 68. (Zo'n funktie kan

bijvoorbeeld gebruikt worden in een test i<n cand a[i+1]/=x wanneer 1:n het indexbereik van a is).

2.9 Programmeer in HOF uitgebreid met islist (zie opgave 1.25) een funktie die test of twee getalbomen eenzelfde bladrij hebben. Hierbij definiëren we

- een getalboom is een lijst met elementen die een getal of wederom een getalboom zijn;
- de bladrij van een getalboom is de rij getallen die je krijgt door in een voor de hand liggende lineaire notatie, bijvoorbeeld m.b.v. lijsten, alles behalve de getallen weg te strepen.

Doe zoiets ook in Pascal of Algol 68, of in een taal met coroutines.

2.10 Programmeer een linear search in HOF. Bij argumenten (x1: x2: ... xN: nil) en x moet de kleinste index i opgeleverd worden zodat x eqn xi, of —indien zo'n i niet bestaat— 0. Hoe wordt de rekursie beeindigd wanneer de index gevonden is? Hoe gaat dat in een gebiedende taal zoals Pascal en Algol 68? (Gebruik ter vergelijking de assignment a:=tla in de gebiedende taal).

2.11! Beschouw de volgende Pascal procedure

```
procedure while (be: boolean; procedure stmnt);
  begin if be then
        begin stmnt; while (be, stmnt) end
  end
```

Is de aanroep while (be, stmnt) in het algemeen equivalent met de repetitie while be do stmnt? Geef voorbeelden van equivalentie en niet-equivalentie. Verander de procedure zo dat de aanroep wel altijd equivalent is. Geef een voorbeeld met geschikte be en stmnt.

2.12! Stel dat Pascal is uitgebreid met call-by-name. Beschouw de volgende definitie.

```
function sum (a,b: integer; var i: integer;
              name expr: integer): integer;
var s: integer;
begin i:=a-1; s:=0;
      while i≠b do
      begin i:=i+1; s:=s+expr end;
      sum:=s
```

end

Waartoe evalueert sum(1, 10, k, a[k]) in de scope van var k: integer; a: array [1..10] of integer? Wat is het resultaat van die aanroep wanneer we name vervangen door value respektievelijk need?

2.13! Hoe kun je in Pascal zonder enige uitbreiding de function sum uit Oefening 2.12 zo direkt mogelijk simuleren? Geef een voorbeeld voor de sommatie van a[1] tot en met a[10]. (Doe die ook in Algol 68.)

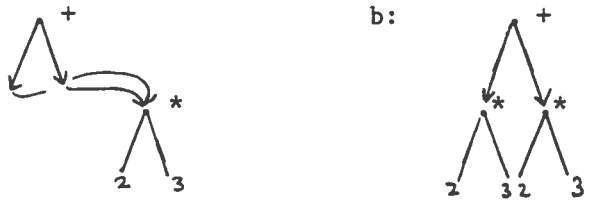
HOOFDSTUK 3

EEN IMPLEMENTATIEMODEL: GRAAFREDUKTIE

In dit hoofdstuk beschrijven we de essentie van een veelbelovende implementatietechniek. Deze is anno 1983 nog volop onderwerp van onderzoek, en wordt nog slechts op kleine schaal gebruikt. In beginsel is deze methode geschikt voor alle reduktiestrategieën; maar met name de luie reduktie kan uitstekend in dit model gerealiseerd worden.

Het wezenlijke idee is dat expressies door grafen gerepresenteerd worden. Iedere knoop van de graaf representeert een subexpressie; de uitgaande takken van zo'n knoop verwijzen naar de representatie van de samenstellende delen van de subexpressie. Zodoende is het mogelijk gelijke subexpressies slechts eenmaal op te slaan, en duplikatie ervan te voorkomen. Reduktie van zo'n gedeelde representatie betekent dus in feite een reduktie op verscheidene voorkomens van de gerepresenteerde subexpresssie. Bijvoorbeeld, na (app)-kontraktie van ((fn x. x+x) (2*3)) zal representatie (a) en niet (b) ontstaan:

a: b:



Reduktie van de linker summand in (a) houdt nu tevens in dat de rechter summand gereduceerd wordt; dat is in (b) niet het geval. Het is wel zo dat (a) en (b) beide een representatie van ((2*3)+(2*3)) zijn.

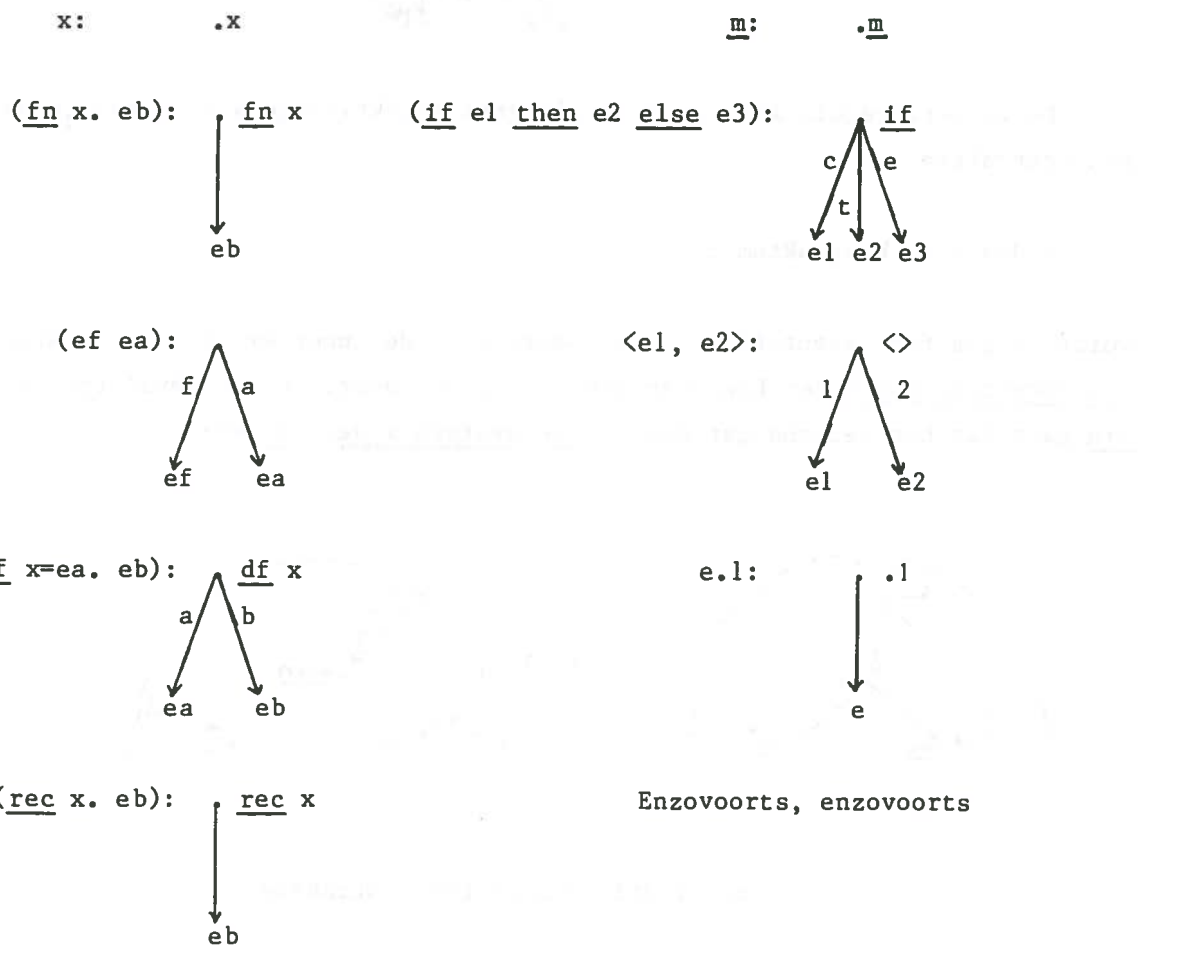
In Par. 3.1 lichten we bovenstaande wat uitvoeriger toe. Met name de realisatie van de (app)-regel krijgt de aandacht. We zullen hem zo definiëren dat delen van de funktieromp die niet van de parameter afhangen ook door de verscheidene applikaties worden gedeeld. In Par. 3.2 laten we zien hoe in beginsel de graafreduktie in zo eenvoudige stapjes kan worden gerealiseerd, dat die stapjes eenduidig met machine-instrukties (zouden) kunnen overeenkomen. Enige slotopmerkingen volgen in Par. 3.3.

Par. 3.1 De graafrepresentatie en graafreduktie

In deze paragraaf schetsen we hoe expressies met grafen gerepresenteerd kunnen worden, en hoe de kontrakties op grafen gerealiseerd worden.

De, of een, graafrepresentatie voor een expressie ziet er als volgt uit. Iedere subexpressie wordt door een knoop gerepresenteerd, en iedere knoop zal de representatie van een subexpressie zijn. Iedere knoop is gemerkt met een aanduiding voor de samenstellingswijze van de expressie en heeft gemerkte uitgaande kanten (pijlen, verwijzingen) naar de representaties van de samenstellende subexpressies. We zullen zien dat er ten gevolge van rec-expressies lussen in de graaf komen.

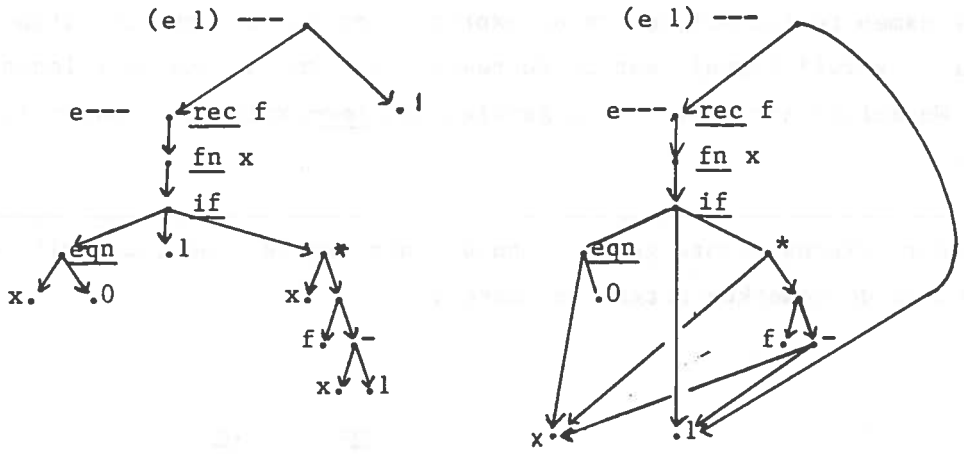
Voor iedere expressievorm geven we nu de vorm van de knoop, dat wil zeggen het merkteken en de gemerkte uitgaande kanten.



De applikatie-knopen hebben geen merkteken, omdat we in expressies ook geen uit-

drukkelijke applikatieoperator hebben; zie (ef ea). Meestal laten we in de graaftekeningen de merktekens bij de uitgaande kanten achterwege, als dat geen misverstanden zal opleveren.

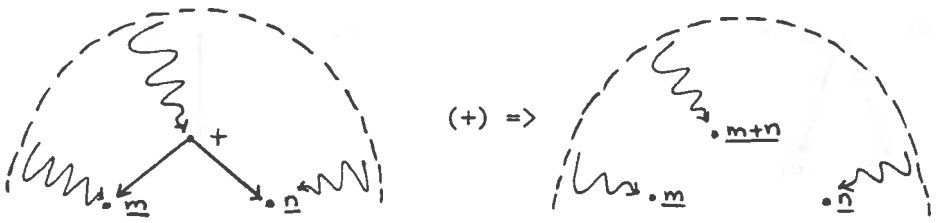
Een voorbeeld. In beide onderstaande grafen representeert de met e--- aangegeven knoop de expressie e = (rec f. fn x. if x eqn 0 then 1 else x * f(x-1)).



De reduktieregels zijn nu alle als graafreduktieregels te interpreteren. Een kontraktie

redex r -> kontraktum c

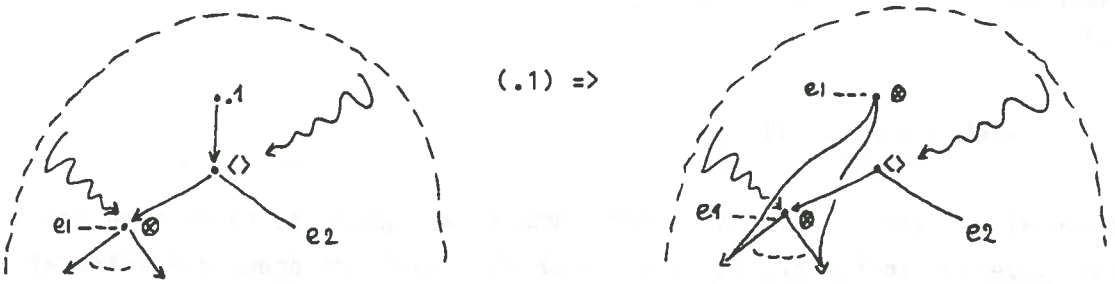
wordt op graafrepresentaties gerealiseerd door de knoop kr die r representeert "te vervangen door" een knoop kc die c representeert. In het geval (m + n) -> m+n gaat dat het eenvoudigst door kr te overschrijven als kc:



realisatie van de (+)-kontraktie

De symbolen  staan voor de verwijzingen vanuit de rest van de graaf. De knopen die de onderdelen van de redex representeren mogen in het algemeen

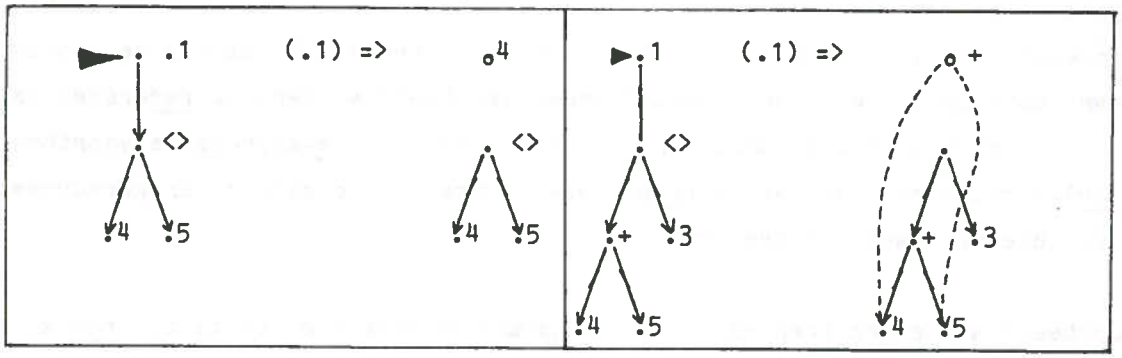
niet weggelaten worden (na de kontraktie), omdat er daarnaar verwezen kan worden vanuit andere knopen. In het geval <e1, e2>.1 -> e1 is er in de te reduceren graaf al een knoop ke die e1 representeert. Ook nu kan kr als ke worden overschreven, terwijl de oorspronkelijke knoop ke intakt blijft:



realisatie van de (.1)-kontraktie

(Pas op. Door "kr te overschrijven als ke" wordt in feite knoop ke geduplieerd en derhalve misschien ook sommige reductiestappen.)

Een andere mogelijkheid voor "de vervanging van kr door ke" is om alle uitgaande kanten die in kr eindigen in ke te laten eindigen; maar dit ziet er niet zo doelmatig uit omdat dan alle knopen van de graaf bekeken en zo nodig gewijzigd moeten worden. We geven nu als voorbeeld van de (.1)-kontraktie de kontraktie van <4,5>.1 en van <4+5, 3>.1. De redexknoop kr geven we aan met ► en de kontraktumknoop ke tekenen we gestippeld en met open rondjes.



twee voorbeelden van een (.1)-kontraktie

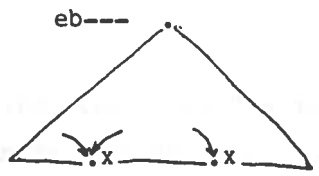
We geven nu een toelichting op de (df) en (rec)-regel; daarna komt de (app)regel aan de beurt. Bij de (df)regel (df x=ea. eb) -> eb[x/ea] kan de knoop die (df x=ea. eb) representeert overschreven worden als de knoop die eb representeert, waarbij tevens de knopen voor x vervangen worden door de knoop die ea representeert. Wanneer eenmaal alle mogelijke (df)-kontrakties zijn toegepast, is er geen subexpressie van de vorm (df x=ea. eb) meer over, en die worden ook niet door andere reduktieregels voortgebracht. Het eenmalig toepassen van alle

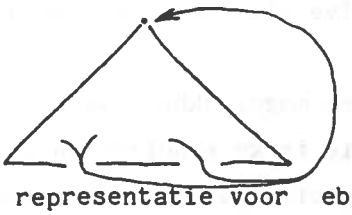
(df)-kontrakties kan geschieden tijdens de opbouw van de graaf die een gegeven te evalueren expressie representeert. Dus in feite is de (df)-regel tijdens reduktie niet meer nodig: we gaan er verder vanuit dat expressies zonder df-knopen gerepresenteerd worden.

Nu de (rec)-regel, (rec x. eb) -> eb[x/rec x. eb]. Door deze oneindig vaak toegepast te denken zien we dat (rec x. eb) reduceert tot de "oneindige expressie"

eb[x/eb[x/eb[x/...]]]

En hoewel "oneindige expressie" een gevaarlijk begrip is, is er toch een voor de hand liggende eindige representatie voor dit geval: de graaf die ontstaat uit de representatie voor eb door daarin de knopen voor x te vervangen door de knoop voor eb zelf. Aldus ontstaan er lussen in de graaf.

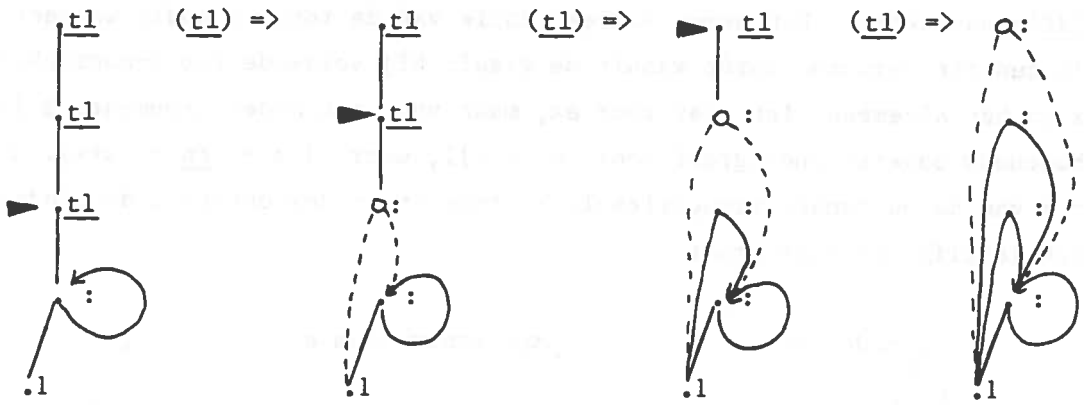




representatie voor eb[x/eb[x/...]]
dus voor (rec x. eb)

(In bovenstaande figuur is "vervanging van x door eb" gerealiseerd door "verlenging van de pijlen naar x tot pijlen naar het geheel".) Net als bij de df-expressie kan iedere subexpressie van de vorm (rec x. eb) al bij de opbouw van de graaf door zo'n cyklische subgraaf gerepresenteerd worden. De rec-regel is dan verder niet meer nodig, omdat er geen regel is die rec-expressies voortbrengt. De (rec)-expressies zijn wat de (rec)-regel betreft al oneindig ver gereduceerd. We geven hiervan twee voorbeelden.

Voorbeeld a. De graafrepresentatie en graafreduktie van (tl tl tl (rec x. 1: x)) zijn als volgt.

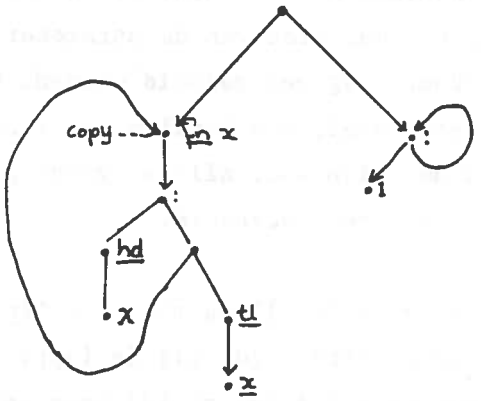


Wanneer er niet naar de twee onbereikbaar geworden knopen gewezen wordt, vanuit de rest van de graaf, mogen zij weggelaten worden. Er zijn inderdaad slechts 3 reduktiestappen nodig en er komt geen (rec)-kontraktie meer voor (want er zijn geen (rec)-knopen!).

Voorbeeld b. Beschouw het volgende programmafragment.

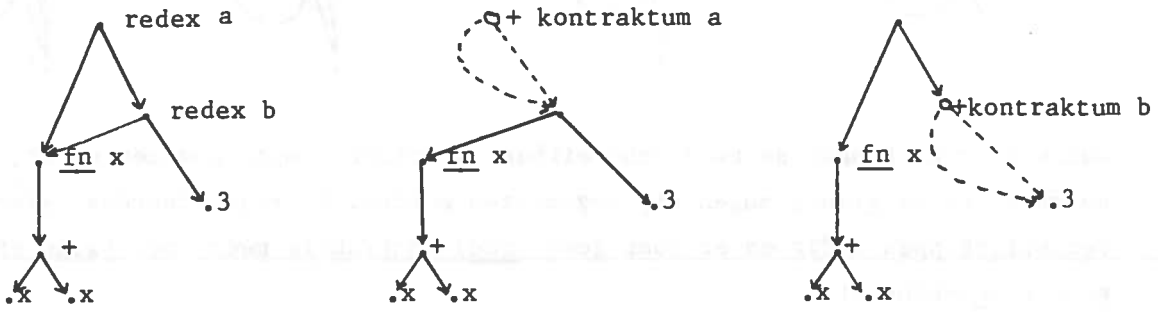
```
(df copy rec= fn x. hd x : copy (tl x)
  ;    x rec= 1: x
  . copy x)
```

De graafrepresentatie (zonder df-knopen en rec-knopen) ziet er als volgt uit. De reduktie zullen we verderop nog geven.



Nu een toelichting op de (app)-regel ((fn x. eb) ea) -> eb[x/ea]. De knoop kc die het kontraktum eb[x/ea] representeert kan gevormd worden door een kopie te nemen van de subgraaf die de romp van (fn x. eb) representeert en daarin alle knopen voor x te vervangen door de al bestaande knoop voor ea (zoals bij de

(df)-kontraktie). Het nemen van een kopie van de romp is nodig wanneer er naar de funktie verwezen wordt vanuit de graaf; bij volgende (app)-kontrakties staat x in het algemeen niet meer voor ea, maar voor een ander argument. Bijvoorbeeld, beschouw onderstaande graaf voor (e (e 3)), waarbij e = (fn x. x+x). Na kontraktie van de buitenste respektievelijk binnenste redex ontstaat de middelste respektievelijk rechter graaf.



representatie van (e (e 3))

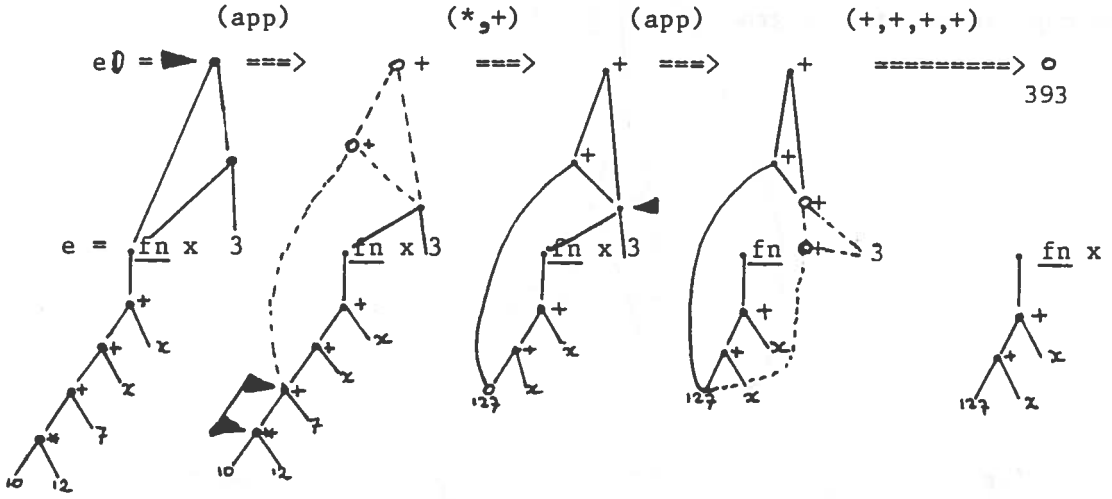
representatie van ((e 3) + (e 3))

representatie van (e (3 + 3))

De representatie voor (e (e 3)) ontstaat bijvoorbeeld als resultaat van ((fn f. f(f 3)) e) en van ((fn f. fn x. f(f x)) e 3); ga maar na.

Hierboven stelden we dat bij de (app)-kontraktie ((fn x. eb) ea) -> eb[x/ea] de hele romp eb gekopieerd moet worden (als er vanuit de graaf naar de funktie van de applikatie verwezen wordt). Maar dat kan zuiniger. Elk deel van de representatie van (fn x. eb) dat niet van de parameter afhangt, dat wil zeggen waarin x niet vrij voorkomt, mag wel gedeeld worden. Het voordeel hiervan is dat eventuele redukties op zo'n deel, ten gevolge van redukties op het resultaat van de applikatie, ook voelbaar zijn voor alle volgende gebruiken van die abstraktie. We lichten dit toe met een voorbeeld.

Beschouw de expressie e0 = (e (e 3)) waarbij e = (fn x. 10*12+7+x+x). Wanneer het deel 10*12+7 niet gekopieerd wordt bij de (app)-kontrakties, maar gedeeld wordt, dan vindt de reduktie 10*12+7 => 127 maar eenmaal in plaats van tweemaal plaats. De graafreduktie volgens de U- (en dus L-) strategie ziet er als volgt uit. De redex geven we aan met ► en het kontraktum tekenen we gestippeld en met open rondjes.



In de laatste graaf is de representatie voor de abstraktie (fn x. 127+x+x) geheel onbereikbaar geworden (en mag weggelaten worden).

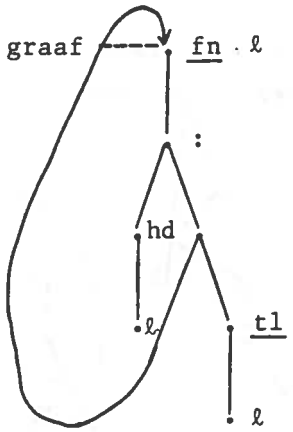
Deze realisatie van de (app)-regel ((fn x. eb) ea) -> eb[x/ea] luidt dus als volgt.

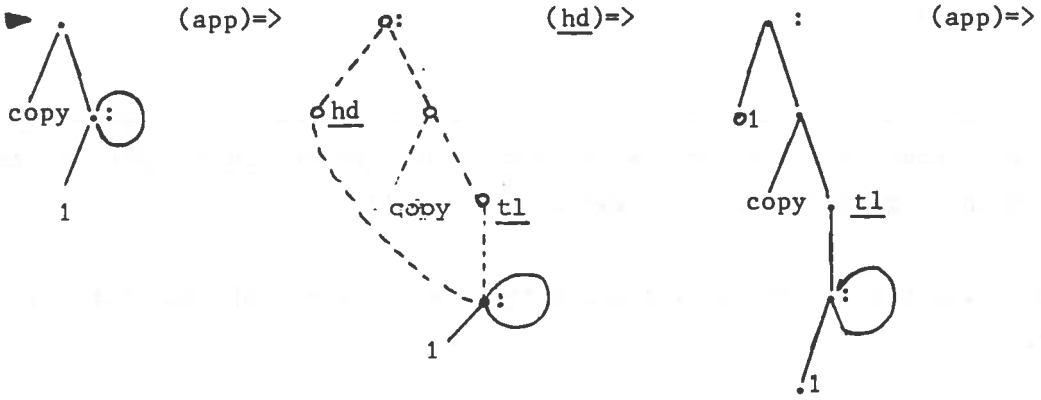
Maak een representatie voor eb door
- voor iedere subexpressie waarin de x niet vrij voorkomt de al bestaande representatie (in de redex) te nemen, en
- voor iedere subexpressie waarin de x wel vrij voorkomt een nieuwe knoop aan te maken.

Vervang in de aldus gevormde representatie voor eb alle knopen x door de knoop voor ea (die in de redex gegeven is).

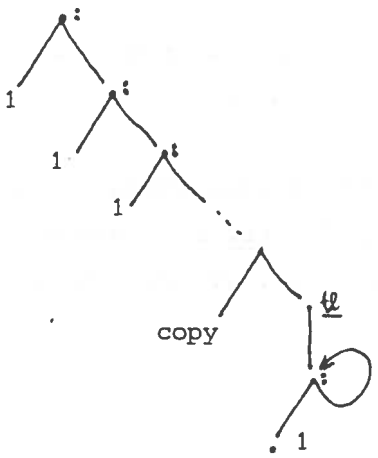
We geven nu nog de alreeds aangekondigde reduktie van (copy (rec x. 1: x)) met copy = (rec f. fn l. hd l: f(tl l)). Steeds wordt de meest linker maximale reduciendum ter reduktie gekozen; we tonen dus de luie strategie.

We laten copy staan voor de graaf





----- =>



Wanneer de kontrakties volgens de bovenbeschreven methode gerealiseerd wor-

den, en er steeds de meest linker maximale (=zo hoog mogelijk gelegen) reduciendum gekontraheerd wordt (mits die niet in een abstraktie ligt), dan is dat precies een reduktie volgens de luie strategie. De voor- en nadelen daarvan zijn beschreven in Hoofdstuk 2. In de komende paragrafen geven we nog enige opmerkingen voor een efficiënte implementatie van deze luie graafreduktie. (NB. Konstante- knopen mogen best geduplicëerd worden, want dat geeft toch geen duplicatie van reductiestappen.)

Par. 3.2 Opsplitsing van substitutie

Op één na hebben alle kontrakties slechts een lokaal effekt op de graaf. Zij kreeëren en overschrijven slechts een (klein) vast(!) aantal knopen. Daarom kunnen zij in beginsel met machine-instrukties van een "graafreduktiemachine" geïdentificeerd worden. De ene uitzondering is de (app)-kontraktie. Wij laten nu zien hoe in beginsel ook de (app)-kontraktie met zulke eenvoudige kontrakties te realiseren is. Het komt erop neer dat de expressies zodanig gewijzigd worden (tijdens een kompilatiefase, dus voorafgaande aan de reduktie-fase) dat de substitutie eb[x/ea] niet in één keer voltooid wordt, maar in een variërend aantal kleine stappen. (Die stappen komen overeen met de stappen in een definitie van substitutie met induktie naar de opbouw van eb).

Aldus is een graafrepresentatie van een expressie op te vatten als de kode van die expressie; de knopen van de graaf zijn de instrukties. Exekutie van zo'n kode (=reduktie van de graaf) heeft tot gevolg dat de kode wordt gewijzigd. We hebben dus te maken met een zichzelf wijzigende kode. Maar let op: de wijzigingen behouden per definitie de semantiek en kunnen dus geen onverwachte of onvoorziene fouten introduceren! Dit staat in tegenstelling tot een zichzelf wijzigende kode die rechtstreeks door een programmeur in machinetaal of assembler wordt geschreven.

Wie niet geïnteresseerd is in de details kan de rest van deze paragraaf overslaan.

De wijziging van de expressies gaat als volgt. Beschouw een willekeurige abstraktie; deze heeft een van de volgende vormen.

(a) (fn x. e) met x niet vrij in e,
of (b1) (fn x. x)
(b2) (fn x. samenstelling van e1, .., eN)
(b3) (fn x. fn y. eb).

We nemen (fn x. e+e´) als typisch voorbeeld van (b2), en we laten voorlopig geval (b3) buiten beschouwing. De abstrakties uit (a) en (b1) hoeven niet gewijzigd te worden, omdat zij bij kontraktie al een uiterst lokaal effekt op de graaf hebben:

(a´) ((fn x. e) ea) -> e voor x niet vrij in e
(b1´) ((fn x. x) ea) -> ea

In beide regels is de knoop voor het kontraktum al in de redex aanwezig.

De abstraktie (fn x. e+e´) van geval (b2) wijzigen we in (fn x. ((fn x. e) x) + ((fn x. e´) x)); met andere woorden, de operanden e en e´ zijn gewijzigd in ((fn x. e) x) en ((fn x. e´) x). Bij kontraktie van een applikatie krijgen we nu het volgende.

(b2´) ((fn x. e) x + (fn x, e´) x) ea)
$\overset{1}{=}>$ ((fn x. e) ea) + ((fn x. e´) ea)
=> e[x/ea] + e´[x/ea]
= (e + e´)[x/ea]

Dus enerzijds is het resultaat na verscheidene reduktiestappen gelijk aan het resultaat van ((fn x. e+e´) ea), en anderzijds zien we dat de eerste kontraktie ook slechts een lokaal effect op de graaf heeft: knopen voor de onderdelen (fn x. e), (fn x. e´) en ea in het kontraktum zijn al in de redex aanwezig. Wanneer we alle abstrakties, en ook de nieuw voortgebrachte, eenmaal aan deze wijziging onderwerpen, dan hebben dus alle (app)-kontrakties zo´n lokaal effekt. En dat is al bijna wat we hebben wilden. Het enige wat nog nodig is, is de gevallen (a) tot en met (b2) syntaktisch te onderscheiden, zodat onmiddellijk duidelijk is welke van de speciale regels (a´), (b1´) of (b2´) moet worden toegepast.

We geven nu een uitbreiding van de syntaxis (en van de reduktieregels) zo dat de gevallen (a)..(b2) syntaktisch onderscheiden aangeduid kunnen worden. Er is een nieuwe konstante I, een nieuwe expressievorm (K e), dus K is een operator, en er zijn veel nieuwe expressievormen die ontstaan door operatoren (zoals hd, tl, +, *, <>, .0, eqnil enz.) van een aksent te voorzien. De abstractie uit geval (a) schrijven we als (K e); dus de reduktieregel moet luiden

(K) ((K e) ea) -> e

De abstraktie uit geval (b1) schrijven we als I; dus de reduktieregel moet lui-

den:

(I) (I ea) -> ea

De abstraktie (fn x. (fn x. e)x + (fn x. e´)x) schrijven we als ((fn x. e) +´ (fn x. e´)); dus de reduktieregel moet luiden:

(+´) ((ef +´ eg) ea) -> (ef ea) + (eg ea)

En net zo voor ander operatoren, waaronder ook de niet geschreven applikatie-operator.

Er rest ons nu nog om geval (b3) te behandelen. Maar dat is eenvoudig. Als we volgens bovenstaande werkwijze eerst de (fn y. eb) wijzigen en met behulp van de nieuwe konstante I en de nieuwe expressievormen herschrijven, dan verdwijnt de lambda en verschijnt er een van de gevallen (a) .. (b2). De dan volgende herschrijving heeft wel tot gevolg dat de nieuwe operatoren K, +´, *´ enz. ook van een aksent worden voorzien. Dus in feite komen er oneindig veel operatoren (eigenlijk: expressievormen) bij. Reduktieregel (+´) luidt dan in het algemeen:

(op´) ((ef op´ eg) ea) -> (ef ea) op (eg ea)
voor op = +, *, +´, *´, +", *", enzovoorts

Aldus is het mogelijk om voorafgaande aan de reduktie de expressie te "kompileren" zodat alle lambda-abstrakties verdwijnen in ruil voor aksenten bij de operatoren en de verschijning van K en I. De reduktiestappen op de gekompileerde expressie hebben allen een lokaal effekt op de graafrepresentatie; zij zijn in beginsel als een elementaire machine-instruktie realiseerbaar. (Het uiteindelijk resultaat is niet of nauwelijks anders dan het resultaat van de reduktie van de oorspronkelijke expressie: ze verschillen slechts doordat in de ene sommige subexpressies de gekompileerde versie zijn van de overeenkomstige subexpressies uit de andere. Met name de resultaten die geen abstrakties bevatten zijn identiek).

Een bijkomend voordeel van de opsplitsing van substitutie is dit. Het kopiëren van de romp van de abstraktie gebeurt nu stap-voor-stap, en alleen voorzover nodig blijkt, dus soms helemaal niet. Bijvoorbeeld het (app)- kontraktum van ((fn x. if true then e else e´) ea), namelijk (if true then e else e´)[x/ea], bevat een kopie van e´ (voor zover x daarin vrij voorkomt). Maar bij de gekompileerde versie krijgen we

((if´ (fn x. true) then (fn x. e) else (fn x. e´)) ea)
(if´) $\stackrel{1}{\Rightarrow}$ if (fn x. true)ea then (fn x. e)ea else (fn x. e´)ea
(app) $\stackrel{1}{\Rightarrow}$ if true then (fn x. e)ea else (fn x. e´)ea
(if) $\stackrel{1}{\Rightarrow}$ (fn x. e)ea

en dus wordt er niets van e´ gekopieerd in de graafrepresentatie.

Een nadeel is dat er ten gevolge van de opsplitsing in kleine stappen in iedere stap een paar extra knopen in de graafrepresentatie nodig zijn, die er niet zouden zijn bij de (app)-kontraktie. Dit is de prijs die we moeten betalen om de reduktie met uniforme, eenvoudige stappen (machine-instrukties) te volvoeren. (Een bovengrens voor dat aantal extra knopen hangt lineair af van het aantal knopen in de reduktie van de oorspronkelijke expressie).

Voor de volledigheid geven we hier nog een voorbeeld. We waarschuwen de lezer dat de gekompileerde expressie niet zo "leesbaar" meer is. We kompileren de abstraktie fn x. fn y. x*y+x*3 stap voor stap, door steeds de binnenste lambda´s eerst te wijzigen (w) en dan in de nieuwe operatoren te herschrijven (h).

fn x. fn y. x*y+x*3
w: fn x. fn y. (fn y.x*y)y + (fn y.x*3)y
h: fn x. (fn y. x*y) +´ K(x*3)
w: fn x. (fn y. (fn y. x)y * (fn y. y)y) +´ K(x*3)
h: fn x. (fn y. x) *´ (fn y. y) +´ K(x*3)
h: fn x. Kx *´ I +´ K(x*3)
w: fn x. (fn x. Kx *´ I)x +´ (fn x. K(x*3))x
h: (fn x. Kx *´ I) +" (fn x. K(x*3))
w: (fn x. (fn x. Kx)x *´ (fn x. I)x) +" (fn x. K (fn x. x*3)x)
h: (fn x. Kx) *" (fn x. I) +" K´(fn x. x*3)
w: (fn x. K (fn x. x)x) *" (fn x. I) +" K´(fn x.(fn x. x)x * (fn x. 3)x)
h: K´(fn x. x) *" KI +" K´(fn x. x) *´ (fn x. 3))
h: K´I *" KI +" K´(I *´ K3)

We laten het over aan de ijverige lezer om de reduktie (graafreduktie) te geven van de toepassing van deze expressie op bijvoorbeeld 4 en 5.

Par. 3.3 Overige implementatie-details

Het moge duidelijk zijn dat de graafreduktie zoals in de vorige paragrafen geschetst op een (konventionele) machine gerealiseerd kan worden. Natuurlijk ko-

men daarbij nog een groot aantal problemen of probleempjes naar voren, waarop we in deze syllabus niet in kunnen en willen gaan. We noemen in deze paragraaf slechts twee aspekten die een iets algemener belang hebben dan loutere implementatie-details.

Allereerst is er het probleem van het beheer over de opslagruimte (voor de opslag van knopen). In het algemeen is het niet te voorspellen wanneer een knoop onbereikbaar is. Met name is het niet zo dat steeds de laatst gekreeerde knoop het eerst onbereikbaar zal worden; er is geen LIFO (Last In First Out) beheer over de opslagruimte mogelijk. Een of andere "garbage detection and collection" techniek is nodig. Dat wil zeggen, wanneer tijdens het reduktieproces de ruimte voor de opslag van knopen uitgeput dreigt te raken, moet er een algoritme gestart worden dat die cellen opspoort waarin slechts onbereikbaar geworden knopen zijn opgeslagen, en die cellen dan in een lijst van vrije cellen verzamelt. Daarna kan het reduktieproces weer doorgaan; uit de zojuist gevormde lijst van vrije cellen kan het z´n opslagruimte putten voor nieuw te kreeeren knopen. In de literatuur is veel bekend over technieken om afval (garbage) op te sporen en te verzamelen. Wij gaan hier niet verder op in.

Als tweede aspekt noemen we de vorm van de knopen. In onze uiteenzettingen tot nu toe zijn er knopen met (nul) een, twee of drie uitgaande kanten. Voor het beheer van de opslagruimte is het aantrekkelijk dat alle knopen evenveel ruimte innemen. Bovendien is het doelmatig dat een knoop niet meer ruimte inneemt dan noodzakelijk is. Beide wensen kunnen vervuld worden door alle knopen precies twee uitgaande kanten te geven. Dit kan al op expressie-nivo bereikt worden, en wel als volgt. Voer een nieuwe konstante plus in en vertaal (kompileer) een expressie (e+e´) in ((plus e) e´). De reduktieregel voor plus moet dus luiden

(plus) ((plus m) n) -> m+n

En analoog voor alle andere expressiesamenstellingswijzen anders dan applikatie en abstraktie. Aldus resulteert er na zo´n vertaling een expressie met als samenstellingswijzen alleen abstraktie en applikatie. Na eliminatie van abstraktie, zoals in de vorige paragraaf beschreven, resulteert er een expressie met applikatie (beter: de aksent-varianten van applikatie) als enige samenstellingswijze.

Belangrijke onderwerpen uit Hoofdstuk 3

graafrepresentatie van expressies
deling (sharing)
rekursie-expressies door lussen gerepresenteerd
kontraktieregels voor grafen
    i.h.b. de realisatie van de (app)-kontraktie
zichzelf wijzigende kode
luie evaluatie = graafreduktie volgens U-strategie
garbage detection and collection nodig

Literatuur bij Hoofdstuk 3

(Turner 1979) vormt een prettig leesbare uiteenzetting van de luie strategie, toegespitst op SASL. (Dijkstra 1980) was de inspiratiebron voor onze manier om alle lambda's te elimineren, zie par. 3.3; de traditionele methode gaat met de drie kombinatoren S, K en I, zie (Turner 1979).

Oefeningen bij Hoofdstuk 3

3.1! Geef de L-graafreduktie van

```
(df from rec= (fn n. n: from (n+1))
 . hd tl tl tl (from 1)
 )
```

3.2 Geef de L-graafreduktie voor (tpl (tpl (fn x. x+x)) 3) en voor (tpl tpl (fn x. x+x) 3), waarbij tpl = fn f. fn x. f(f(fx)). Wenk. Er zijn 15(app) plus 9(+)-kontrakties respektievelijk 45(app)- plus 27(+)- kontakties nodig. Maak uw tekening overzichtelijk.

3.3! Beschouw voor I = 1,2 het blok

```
(df allI = (litrI and true). --- (allI (true: true: true: false:...)) ---)
```

waarbij

```
litr1 = rec g. fn f a ℓ. if eqnil ℓ then a else f(hd ℓ) (g f a(tl ℓ))
litr2 = fn f a. rec g. fn ℓ. if eqnil ℓ then a else f (hd ℓ) (g (tl ℓ))
and   = fn x y. if x then y else false
```

Geef de L-graafreduktie voor het deel (allI (true:...)); bedenk dat allI ook nog elders in de kontekst ervan voorkomt. Hoe ziet voor I=1,2 de graaf voor allI eruit na afloop van de reduktie voor die subexpressie? In welk opzicht is de ene litr efficiënter dan de andere?

3.4! Geef de graaf die ongeveer halverwege de L-graafreductie onstaat van respectievelijk (sum0 (down 1000)), (sum1 0 (down 1000)) en van (sum2 0 (down 1000)) waarbij

```
down rec= fn n. if n eqn 0 then nil else n: down(n-1)
sum0 rec= fn ℓ. if eqnil ℓ then 0 else (hd ℓ) + sum0 (tl ℓ)
sum1 rec= fn x ℓ. if eqnil ℓ then x else sum1 (x + hd ℓ)(tl ℓ)
sum2 rec= fnval x. fn ℓ.    ---idem---
```

3.5! Geef de L-graafreduktie van het programma dat het kleinste Fibonaccigetal oplevert dat een veelvoud is van 117.

3.6! Geef de L-graafreduktie van ((fn f. (Wf Wf)) (fn y. 1:y)) waarbij Wf = (fn x. f(x x)).

HOOFDSTUK 4

EEN IMPLEMENTATIEMODEL:LIFO-BEHEER VAN OPSLAGRUIMTE

Par. 4.1 Inleiding

In dit hoofdstuk schetsen we een traditionele implementatietechniek voor blokgestruktureerde talen. (We noemen een taal blokgestruktureerd als het geneste bindingskonstrukties heeft en statische scope; voor statische scope, zie Hoofdstuk 1A). In deze inleiding leggen we de belangrijkste karaktertrekken informeel uit; een precieze behandeling volgt in de rest van dit hoofdstuk. Om aan te sluiten bij traditionele terminologie gebruiken we "evaluatie" als synoniem voor reduktie, en "waarde" als synoniem voor expressie in resultaatvorm (resulterend uit een reduktie).

Allereerst merken we op dat dit model zich ook leent voor een implementatie van assigneerbare variabelen, zoals de Pascal var-variabelen en de Algol 68 loc-variabelen. Maar daar zullen we in dit hoofdstuk nog geen aandacht aan besteden.

Een belangrijk begrip dat in dit model, en in uiteenzettingen over programmeertalen in het algemeen, een grote rol gaat spelen, is dat van semantische omgeving, kortweg omgeving (engels: environment). In dit model wordt een substitutie eb[x/ea] (ten gevolge van de evaluatie van een bindingskonstruktie) nooit daadwerkelijk uitgevoerd. In plaats daarvan wordt aan eb de "semantische binding: x staat voor de waarde ea" gekoppeld. (Onderscheid semantische bindingen van syntaktische bindingen; zie Hoofdstuk 1.2). Een verzameling van dergelijke bindingen, voor verschillende x, wordt omgeving genoemd. Met andere woorden, een omgeving is een afbeelding die voor elke vrije variabele x de waarde ea geeft die eigenlijk voor x gesubstitueerd moet worden. Een semantiekbeschrijving m.b.v. omgevingen noemen we een (het) omgevingenmodel.

Door de introduktie van omgevingen, ter representatie van nog uit te voeren substituties, hoeven expressies zelf niet meer gewijzigd te worden tijdens de evaluatie. Aldus wordt het mogelijk om met een vaste kode (=voorbewerkte expressie) te werken.

Dit implementatiemodel is erop gebaseerd dat de evaluatie van expressies

volgens het LIFO-regime plaats vindt (LIFO = Last In First Out). Dat wil zeggen, de laatst gestarte evaluatie van een subexpressie wordt het eerst voltooid. Dus de kompositionele strategie en call-by-name en call-by-need voldoen hier bij uitstek aan, terwijl parallellisme en coroutines hierdoor worden uitgesloten. Vanwege het LIFO-regime kunnen tussenresultaten (= waarden van subexpressies) op een stapel opgeslagen worden: de laatst verkregen waarde wordt het eerst weer gebruikt. En net zo kunnen de administratieve gegevens die aangeven tot hoever de evaluatie gevorderd is, en welke evaluaties dus nog voltooid moeten worden, op een stapel worden bijgehouden: de laatst begonnen (sub)evaluatie wordt het eerst voltooid. Voor de opslag van tussenresultaten en administratieve gegevens wordt de ruimte dus volgens het LIFO-regime bespeeld; de laatst gebruikte ruimte wordt het eerst weer vrijgegeven; (bij coroutines en parallellisme is er "per proces" zo'n stapel nodig.)

Wanneer na evaluatie van de romp van een funktie de binding van de argumentwaarde aan de parameter niet meer nodig is, kan ook de laatst gevormde binding het eerst weer worden opgeruimd. De opslag van bindingen kan dan volgens het LIFO-regime plaatsvinden. Dit is inderdaad het geval bij funkties zoals (fn x. 2*x+3); na evaluatie van (2*x+3) in de omgeving die "5 bindt aan x" is het resultaat 13 en dat is verder onafhankelijk van de binding van 5 aan x. ((Tennent 1981) gebruikt de terminologie dat "x gebonden is aan 5"). Maar bij funkties als resultaat van funkties is dit in het algemeen niet het geval. Beschouw bijvoorbeeld de funktie (fn y. (fn x. 2*x+y)) en stel dat die wordt toegepast op 3. De evaluatie hiervan is na een stap voltooid, en resulteert in de expressie (fn x. 2*x+y) waarvoor (nog steeds) de binding van 3 aan y geldig is. Deze binding mag niet verloren gaan, direkt na de evaluatie; hij moet bestaan blijven zolang het resultaat nog bestaat. Dus zonder funkties als resultaat (of als onderdeel van een resultaat) kan de ruimte voor de opslag van bindingen volgens het LIFO-regime bespeeld worden; met funkties als resultaat is dit niet altijd het geval.

Opmerking. Uit het bovenstaande konkluderen we dat het onterecht is, in dit opzicht, om in plaats van funkties met verscheidene veranderlijken, zoals (fn (y,x). 2*x+y), louter hogere orde funkties met één veranderlijke zoals (fn y.(fn x. 2*x+y)) in de taal HOF te hebben. We nodigen de lezer daarom uit om al onze uiteenzettingen te veralgemenen tot expressies zoals (fn (x,y,...z). e), (df (x,y,...z)=(ex,ey,...ez). e) enzovoorts.

Het LIFO-beheer van de opslagruimte wordt bij traditionele talen zo belangrijk gevonden, dat het taalontwerp daar zelfs op gericht is: er kunnen of mogen

geen funkties als onderdeel van een resultaat worden opgeleverd. Dit is het geval in Pascal. In Algol 68 kunnen funkties weliswaar worden opgeleverd, maar slechts onder de beperking dat het LIFO-beheer toch mogelijk is: het gevormde resultaat mag niet meer afhankelijk zijn van de binding aan de parameter, anders is het effekt van het programma niet gedefinieerd. Dus geoorloofd zijn

```
df f = (fn x. if x then sinus else cosinus). ...
df f = (fn x. if x then (fn (x,y). x+y) else (fn (x,y). x*y)). ...
```

maar niet

```
df f = (fn y.(fn x. 2*x+y)). ...
df f = (fn x. if x then (fn y. x) else (fn y. not x)). ...
```

Het LIFO-beheer van opslagruimte maakt een snelle evaluatie mogelijk. Maar het verbieden of beperken van funkties als funktieresultaat noodzaakt tot de invoering van nieuwe expressievormen die anders niet nodig waren geweest; zie bijvoorbeeld Hoofdstuk 6 en 9.

In de volgende paragrafen gaan we als volgt te werk. Uitgaande van de gegeven kompositionele evaluatie bereiken we na een aantal implementatiestappen het uiteindelijke model. Die stappen zijn achtereenvolgens

- in Par. 4.2 de formulering als een rekursieve procedure eval0 van de C-evaluatie, eval0: fct(E)W;
- in Par. 4.3 de inruil van substituties voor omgevingen, resulterend in eval1: fct(E,R)W;
- in Par. 4.4. en 4.5 de eliminatie van de R-parameter en het W-resultaat van eval1 in ruil voor stapels sR en sW,
  eval2: proc(E,R) {gebruikt var sW}
  eval3: proc(E)W {gebruikt var sR}
  eval4: proc(E) {gebruikt var sW en var sR};
- in Par. 4.6. de omzetting van eval4 tot een kode-voortbrengende procedure kode. Dit gaat gepaard met de introduktie van een stapel sT (voor Terugkeeradressen) die net als sW en sR door de voortgebrachte kode bespeeld wordt.
  kode: proc(E) "proc(){gebruikt var sW, sR, sT}"

(We gebruiken hier fct(A,B)C als een notatie voor het type (A,B->C). Het woord proc i.p.v. fct geeft aan dat er neveneffekten mogelijk zijn).

De verzameling W is de verzameling van waarden: expressies in resultaatvorm. De formuleringen eval1, .., eval4, kode heten "het omgevingenmodel" voor HOF. (Dergelijke modellen zijn ook mogelijk bij niet-kompositionele evaluatiestrategieen!).

Par. 4.2 Een rekursieve funktie voor de C-evaluatie

De evaluatie volgens de kompositionele strategie formuleren we nu als een rekursieve procedure eval0: fct(E)W. In feite brengen we dus niets nieuws.

```
eval0 rec= fct e.
if e ?= (fn x. eb)      -> (fn x. eb)
| e ?= (ef ea)          -> def wf = eval0 ef; wa = eval0 ea
                           in if wf ?= (fn x. eb) -> eval0 eb[x/wa] fi
| e ?= (df x=ea. eb)    -> def wa = eval0 ea in eval0 eb[x/wa]
| e ?= (rec x. eb)      -> eval0 eb[x/(rec x. eb)]
| e ?= k                -> k
| e ?= if e1 then e2 else e3 -> def w1 = eval0 e1
                                in if w1=true -> eval0 e2 | w1=false -> eval0 e3 fi
| e ?= (e1+e2)          -> def w1 = eval0 e1; w2 = eval0 e2
                           in if w1 ?= m and w2 ?= n -> m+n fi
| e ?= <e1, e2>         -> def w1 = eval0 e1; w2 = eval0 e2 in <w1, w2>
| e ?= eg.1             -> def wg = eval0 eg in if wg ?= <w1, w2> -> w1 fi
.
.
.
fi
```

Variabelen x zijn geen resultaatvorm; de evaluatie ervan stokt dus: er is inderdaad geen clausule voor het geval e ? = x in eval0 opgenomen.

Merk op dat de tekst voor eval0 nauwelijks als een definitie van de semantiek van HOF aangemerkt kan worden. Immers, de metataal waarin eval0 geformuleerd is, is niet precies gedefinieerd. Wanneer bijvoorbeeld de metataal volgens de uitstelstrategie wordt geëvalueerd, doet eval0 dat ook met HOF-expressies; en een kompositionele strategie voor de metataal draagt zich net zo over naar HOF. Er zijn technieken om de definitie van eval0 zo op te schrijven dat de evalua-

tiestrategie van de metataal geen invloed heeft op de evaluatiestrategie die door eval0 wordt gevolgd, (Reynolds 1972). Wij zullen daaraan geen behoefte hebben en gewoon stellen dat de metataal volgens de kompositionele strategie geëvalueerd wordt.

Par. 4.3 Substituties geïmplementeerd met omgevingen

We streven ernaar uiteindelijk de expressies onveranderd te laten. Dus in plaats van de substituties ten gevolge van de (app), (df) en (rec)-regel daadwerkelijk uit te voeren, wordt aan iedere expressie een zogenaamde (semantische) omgeving gekoppeld. Een omgeving r geeft voor iedere vrije variabele x de waarde w die voor x gesubstitueerd zou moeten worden; we zeggen r bindt w aan x (dit is een semantische binding, niet te verwarren met die van Hoofdstuk 1). Een omgeving is dus een partiële afbeelding van de variabelen naar de waarden. Wanneer een abstractie als resultaat van evaluatie optreedt, wordt daaraan ook een omgeving gekoppeld; zo'n tweetal heet een closure. Closures worden ook als gesloten expressies opgevat; zij vormen dus deel van de waardeverzameling W.

De verzameling omgevingen (environments) is dus gedefinieerd als R = X fin -> W, de afbeeldingen van eindige deelverzamelingen van X naar W. Voor afbeeldingen (met eindig domein) gebruiken wij de volgende notaties.

```
x<-v = de afbeelding f met dom(f)={x} en f(x)=v
f,g  = de afbeelding h met dom(h) = dom(f) union dom(g),
       en h(x)= f(x) als x: dom(f), g(x) als x: dom(g).
       Er moet gelden dom(f) en dom(g) zijn disjunkt.
f[g] = de afbeelding h met dom(h) = dom(f) union dom(g),
       en h(x)= g(x) als x: dom(g), f(x) anders.
```

We geven nu de definitie van evall: fct(E,R)W.

```
evall rec= fct e r.
if e ?= x              -> r x
| e ?= (fn x. eb)      -> <fn x. eb, r>
| e ?= (ef ea)         -> def wf = evall ef r; wa = evall ea r
                          in if wf ?= <fn x. eb, rf> ->evall eb rf[x<-wa] fi
| e ?= (df x=ea. eb)   -> def wa = evall ea r in evall eb r[x<-wa]
| e ?= (rec x. eb)     -> ....zie onder....
| e ?= (if e1 then e2 else e3) -> def w1 = evall e1 r
```

```
                    in if w1=true ->evall e2 r |w1=false ->evall e3 r fi
  | e ?= (e1+e2)     -> def w1 = evall e1 r; w2 = evall e2 r
                    in if w1 ?= m and w2 ?= n -> m+n fi
  | e ?= eg.1        -> def wg = evall eg r
                    in if  wg ?= <w1, w2> -> w1 fi
.
.
.
fi
```

Het verband tussen evall en eval0 luidt als volgt:

eval0 e[x/ expand(r x); voor alle x in (dom r)] = expand (evall e r)

Hierbij is expand: W->E de afbeelding die closures door gesloten abstrakties vervangt:

```
expand (<fn y. eb, rf>) = (fn y. eb)[x/expand (rf x); alle x]
expand ((eh: et))       = (expand(eh): expand(et))
expand(<e1,e2>)         = <expand(e1), expand(e2)>
expand (k)              = k
```

(Pas op: voor sommige w zou expand(w) een "oneindige expressie" kunnen leveren. We gaan niet op dit probleem in.)

Aan de definitie van evall is een gebruikelijke formulering van statische scope af te lezen, en wel deze.

> De omgeving waarin het bereik van een binding wordt geëvalueerd, is een bijstelling van de omgeving waarin de bindingskonstruktie wordt geëvalueerd. Met name is de omgeving rf´ voor de romp van een abstraktie een bijstelling van de omgeving rf waarin de abstraktie zelf werd geëvalueerd, (en is rf´ niet een bijstelling van de omgeving r waarin de toepassing van die abstraktie op een argument wordt geëvalueerd!).

We krijgen dynamische scope als we de romp van een abstractie laten evalueren in de omgeving van de toepassing. In de definitie van evall zou dat als volgt gaan.

```
evall(e,r) =
  if ...
```

```
| e ?= (fn x. eb) -> (fn x. eb)      {i.p.v. <fn x. eb, r>}
| e ?= (ef ea)    -> def wf= evall ef r; wa = evall ea r
                     in if wf ?= (fn x. eb) -> evall eb r[x/wa] fi

                {i.p.v. if wf ?= <fn x. eb, rf> -> evall eb rf[x<-wa] fi}
```

Omdat als resultaat van (fn x. eb) de tekst zelf (of een representatie daarvan) wordt opgeleverd, is dynamische scope in eerste instantie iets eenvoudiger te implementeren. Dat is dan ook de reden van zijn ontstaan geweest, want verder zijn er nauwelijks argumenten waarom dynamische scope de voorkeur zou verdienen boven statische scope. Zie ook Hoofdstuk 1A.

We moeten het geval e ?= (rec x. eb) nog definiëren. Merk op dat we de kompositionele strategie van evalueren aan het definiëren zijn. Dus om nonterminatie van de evaluatie van (rec x. eb) te voorkomen, moeten de door rec x gebonden voorkomens van x op den duur onbereikbaar zijn voor eval, dus binnen een niet toegepaste abstraktie liggen of binnen een niet gekozen then- of else-tak. Als we alleen de vorm (rec x. fn y. eb) toestaan, is dit zeker het geval. Met andere woorden, we laten alleen rekursieve funktiedefinities toe:

```
(rec x. fn y. eb), ofwel
x rec= (fn y. eb)
```

Deze beperking op het gebruik van rekursie is heel gebruikelijk bij kompositioneel geëvalueerde talen. Nu is de definitie van evall voor rec eenvoudig.

```
evall (e,r) =
if ...
|   e ?= (rec x. fn y. eb) -> <fn y. eb, rf>
                          met rf rec= r[x <- <fn y. eb, rf>]
```

We hebben rekursie van HOF gedefinieerd door in de metataal rekursie te gebruiken, voor rf. Zo dadelijk zal een eindige representatie voor rf gegeven worden. (Merk op dat rf eigenlijk gedefinieerd is als

```
rf rec= fct var. if var=x -> <fn y. eb, rf>
                 |  var≠x -> r var
                 fi
```

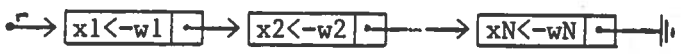
Dus ook in de metataal is rekursie alleen voor abstrakties gebruikt.)

* * *

Lijstrepresentatie voor omgevingen

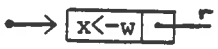
We geven nu een gebruikelijke representatie van omgevingen: geketende lijsten. Allereerst een keten van "x<-w" bindingen en daarna een keten van alleen waarden w.

De representatie van een omgeving r als een keten van "x<-w" bindingen ligt nogal voor de hand: representeer r door

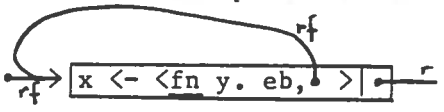


waarbij dom(r) = {x1, .., xN} en (r xi) = wi (voor alle i).
Dit leidt tot de volgende implementatie van de operaties op omgevingen.

(r x) wordt: doorloop de keten tot aan het eerste element xi<-wi met xi=x, en lever dan wi op.

r[x<-w] wordt gerepresenteerd door 

Rekursie is nu ook eenvoudig. De afbeelding rf rec= r[x <- <fn y. eb, rf>] wordt gerepresenteerd door de pointer rf van

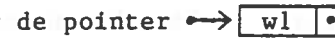


Inderdaad geldt nu dat rf de closure <fn y. eb, rf> aan x bindt, en dat ook in eb deze binding van kracht is.

Bij nadere beschouwing blijkt dat bij bovenstaande implementatie de variabelen x in de bindingen x<-w niet nodig zijn: het aantal elementen dat in de keten r doorlopen wordt bij de bepaling van (r x), is op de voorhand (compile-time) uit te rekenen voor ieder voorkomen van x. Dat aantal is namelijk het verschil tussen nestingsdiepten (van bereiken van bindingsexpressies) van dat voorkomen en zijn bindend voorkomen, of in andere woorden: "het aantal bindingen waaraan dat voorkomen zicht onttrekt", zie Hoofdstuk 1, Par. 1.2! Immers, pre-

cies wanneer het bereik van een bindingsexpressie aan evall onderworpen wordt, wordt de keten met een element uitgebreid.

Wanneer dus in een expressie, voorafgaande aan evaluatie (= onderwerping van evall), ieder voorkomen van een variabele x wordt vervangen door nx = "het aantal bindingen waaraan dat voorkomen zich onttrekt", dan kunnen omgevingen als volgt geïmplementeerd worden.

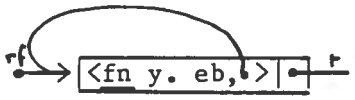
r wordt gerepresenteerd door de pointer 

(r nx) wordt: doorloop precies nx elementen van de keten en lever dan de waarde in het eerstvolgende element op.

r[nx <- w] wordt gerepresenteerd door 

(N.B. in de kontekst is nx altijd 0)

rf rec= r[x<-<fn y. eb, rf>] wordt gerepresenteerd door de pointer rf in



Par. 4.4 Stapeling van tussenresulaten

Uiteindelijk moet eval de akties van een machine beschrijven die expressies evalueert. Dus we willen alle opslag van waarden expliciet in eval beschrijven. Voor de tussenresultaten, d.w.z. de door eval opgeleverde waarden, doen we dat nu; de E- en R-parameters komen in volgende paragrafen aan bod. Het is welhaast een standaardtechniek om een rekursieve funktie zoals evall: fct(E,R)W te veranderen in een rekursieve procedure eval2: proc(E,R) die niet meer een waarde oplevert maar in plaats daarvan die waarde op een globale Waardenstapel sW zet. Die techniek passen we nu toe.

Stapelbewerkingen noteren wij als volgt. Zij s een (assigneerbare) stapelvariabele, x een assigneerbare variabele en w een waarde.

top s	de waarde op de top van s;
s :push w	breidt s op de top uit met w;
x,s :pop	haal de top van s weg en ken die toe aan x;
s :pop	haal de top van s weg

(De : duidt op een assignment, net zoals in := , en wordt uitgesproken als "wordt").

De definitie van eval2: proc(E,R) luidt nu als volgt.

```
eval2 rec= proc e r.
{var sW is een globale stapel met elementen uit W}
if e ?= x                  -> sW :push (r x)
|  e ?= (fn x. eb)         -> sW :push <fn x. eb, r>

|  e ?= (ef ea)            -> eval2 ef r; eval2 ea r;
                              wa, sW :pop; wf, sW :pop;
                              if wf ?= <fn x. eb, rf> -> eval2 eb rf[x<-wa] fi
|  e ?= (df x=ea. eb)      -> eval2 ea r; wa, sW :pop;
                              eval2 eb r[x<-wa]
|  e ?= (rec x.fn y. eb)   -> sW :push <fn y. eb, rf>
                              met rf rec= r[x<-<fn y. eb, rf>]
|  e ?= k                  -> sW :push k
|  e ?= (if e1 then e2 else e3) -> eval2 e1 r; w1, sW :pop;
                              if w1=true -> eval e2 r
                              |  w1=false -> eval2 e3 r fi
|  e ?= (e1 + e2)          -> eval2 e1 r; eval2 e2 r;
                              w1, sW :pop; w2, sW :pop;
                              if w1 ?= m and w2 ?= n -> sW :push m+n fi
|  e ?= eg.1               -> eval2 eg r; wg, sW :pop;
                              if wg ?= <w1, w2> -> sW :push w1 fi
.
.
.
fi
```

Het verband tussen eval2 en evall is eenvoudig: het netto-effekt van (eval2 e r) is dat op de stapel sW de waarde van e, dat is: (evall e r), wordt gezet:

(eval2 e r) = sW :push (evall e r)

Inderdaad wordt iedere waarde die op sW wordt gezet ten gevolge van rekursieve aanroepen van eval2, er ook weer afgehaald. Tussentijds kan sW enorm aangroeien, maar na afloop van een evaluatie is hij weer als bij aanvang maar dan met het resultaat erop gezet.

De variabelen wa, wf, w1, w2, wg enzovoorts zijn slechts zeer lokaal nodig,

en hadden we net zo goed konstanten kunnen wezen. We kunnen ook volstaan met precies twee globale variabelen, en die met registers implementeren; zij bevatten de twee bovenste elementen van sW. (Maar er zouden onbegrensd veel variabelen wf nodig zijn als we bijv. in de clausule voor (ef ea) het deel "wf,sW:pop" direkt na "eval ef" zetten.)

Par. 4.5 Stapelingen van omgevingen

We passen eval nu zo aan dat de opslag van R-argumenten expliciet geprogrammeerd wordt. Ook nu kan dat eenvoudig door een stapel sR te introduceren, die in de top de te gebruiken (de "heersende") omgeving bevat. Tussentijds kan de stapel sR enorm aangroeien, maar na afloop van een evaluatie is hij weer precies in de oorspronkelijk staat hersteld. We gaan uit van eval1 : fct(E,R)W en vormen nu dus eval3: proc(E)W {gebruikt var sR}. Het is niet moeilijk om de stapeling van omgevingen te kombineren met stapeling van tussenresultaten. Dat zou eval4: proc(E) {gebruikt var sR en var sW} leveren; we laten dit over aan de lezer.

```
eval3 rec= proc e.
{var sR is een globale stapel; op top staat de heersende omgeving}
if e ?= x                    -> ((top sR)x)
¦  e ?= (fn x. eb)           -> <fn x. eb, (top sR)>
¦  e ?= (ef ea)              -> def wf = eval3 ef; wa = eval3 ea;
                                in (1) if wf ?= <fn x. eb, rf>
                                       -> sR :push rf[x<-wa];
                                       (2) w := eval3 eb;
                                           sR :pop;
                                           w
                                       fi (3*)
¦  e ?= (df x=ea. eb)        -> def wa = eval3 ea;
                                in (sR :push (top sR)[x<-wa];
                                    w := eval3 eb;
                                    sR :pop;
                                    w
                                   )
¦  e ?= (rec x. fn y. eb)    -> <fn y. eb, rf>
                                met rf rec= (top sR)[x <- <fn y. eb, rf>]   **)
¦  e ?= k                    -> k
¦  e ?= (if e1 then e2 else e3) -> def w1 = eval3 e1;
                                   in if w1 = true -> eval3 e2
                                      ¦  w1 = false ->eval3 e3
```

```
                                                fi
|  e ?= (e1+e2)                  -> def w1 = eval3 e1; w2 = eval3 e2;
                                     in if w1 ?= m and w2 ?= n -> m+n fi
.
.
fi
```

*) N.B. de tijdstippen ①, ②, ③ noemen we "direkt voor", "direkt bij het begin",
en "direkt na afloop" van de eval3-evaluatie van de applikatie.

**) N.B. de binding x <- <fn y.eb, rf> kan eventueel "boven op sR" worden opgeslagen zonder wijzigingen van "de top" van sR. Er is bovendien een optimalisatie mogelijk van de combinatie van recursie met definitie: dfrec.

Het verband tussen eval3 en eval1 is niet moeilijk; het luidt

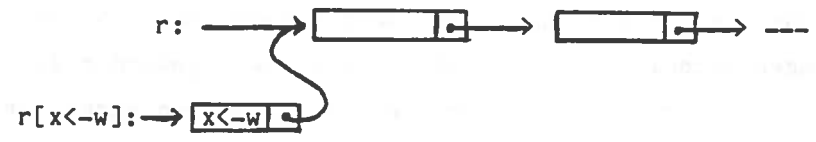
(sR :push r; w := eval3 e; sR :pop; w) = (eval1 e r)

(waarbij = betekent: heeft hetzelfde effekt behalve op w).

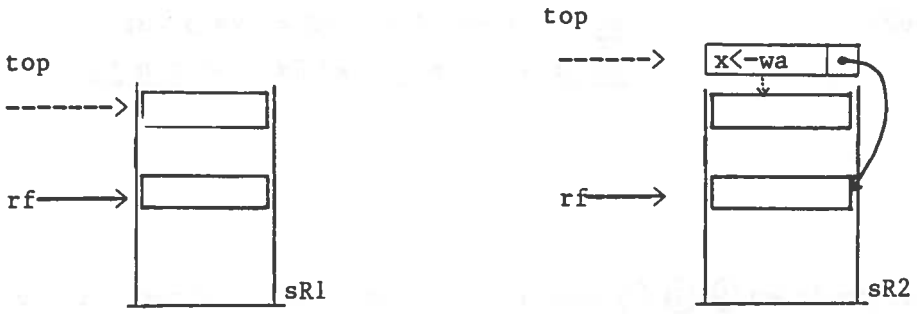
* * *

Een doelmatige implementatie van sR

Uiteraard zijn voor de omgevingen afzonderlijk nog steeds de representaties door middel van geketende lijsten mogelijk. En dank zij die representatie kunnen de representaties van verschillende omgevingen gedeeltelijk samenvallen. Met name geldt dat voor r en r[x<-w]:



Dus de bewerking (sR :push (top sR)[x<-wa]) wordt gerealiseerd door slechts (x<-wa, verwijzing naar (top sR)) op de stapel sR te zetten; dit gebeurt in de clausule voor (eval3 (df x=ea. eb)). Net zo, als rf al ergens gerepresenteerd is, bijvoorbeeld in sR, dan wordt (sR :push rf[x<-wa]) gerealiseerd door (x<-wa, de verwijzing rf) op de top van sR erbij te zetten; dit gebeurt in de clausule voor (eval3 (ef ea)):



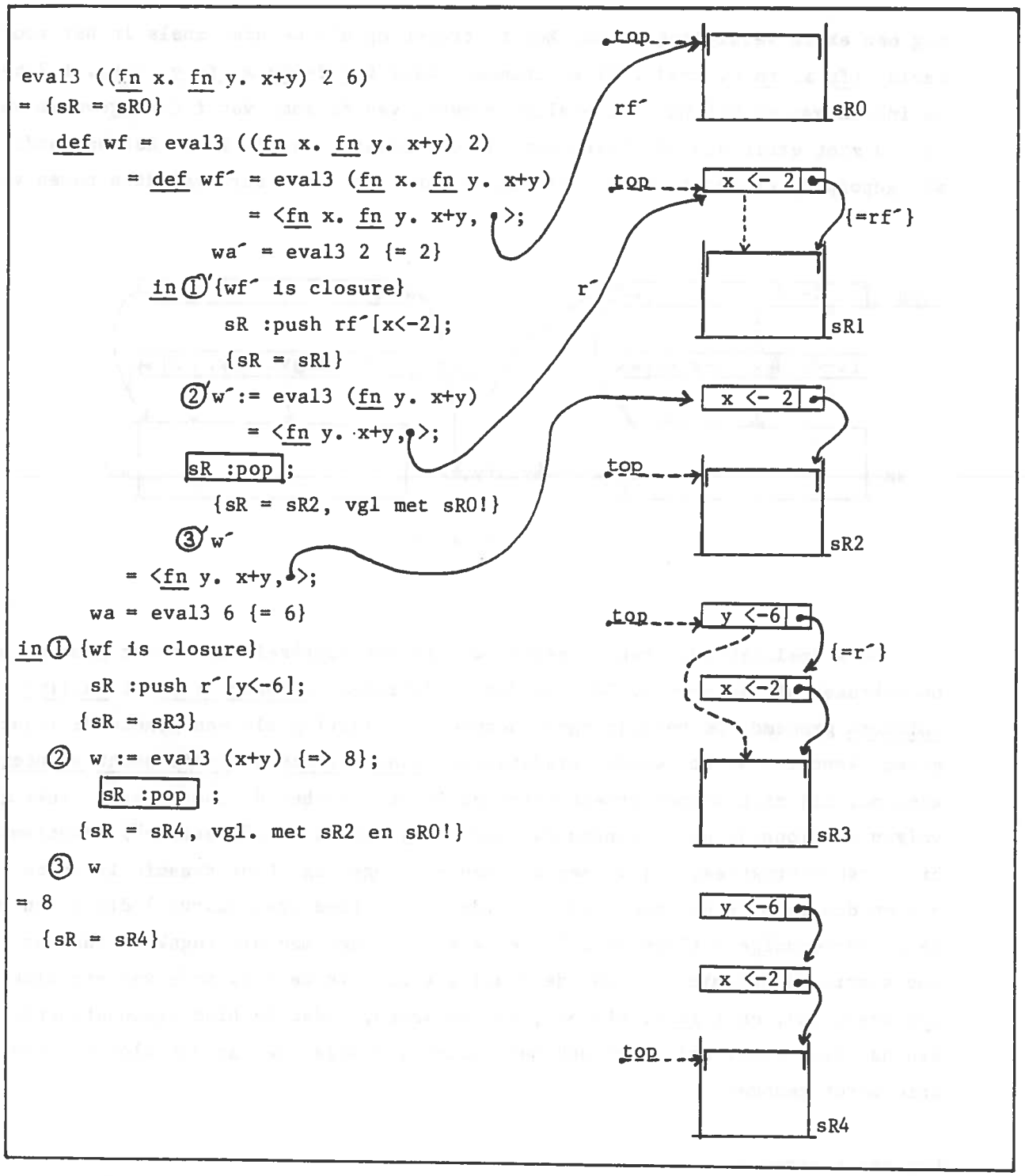
{sR=sR1} sR :push rf[x<-wa] {sR=sR2}

Maar pas op! Door de bewerking sR :pop moet weliswaar de top van de stapel worden verwijderd, maar dat element zelf mag niet verloren gaan als er daarnaar nog gewezen wordt vanuit een closure-tussenresultaat (in de stapel sW van tussenresultaten) (die in het verdere verloop nog toegepast gaat worden). Desondanks zullen we de datastruktuur sR toch een stapel blijven noemen. We geven nu een voorbeeld van dit verschijnsel.

We beschouwen de expressie ((fn x. fn y. x+y) 2 6), en de evaluatie daarvan in een omgeving (top sR0), met sR0 = de initiële inhoud van sR. We schetsen het verloop van de evaluatie, en de inhoud van sR geven we weer in plaatjes. De volgorde waarin elementen van sR ontstapeld moeten worden geven we weer met een gestippelde pijl. Deze pijlen maken dus de stapelstruktuur zichtbaar. De evaluatie gaat als volgt.

{zie de figuur op de volgende pagina}

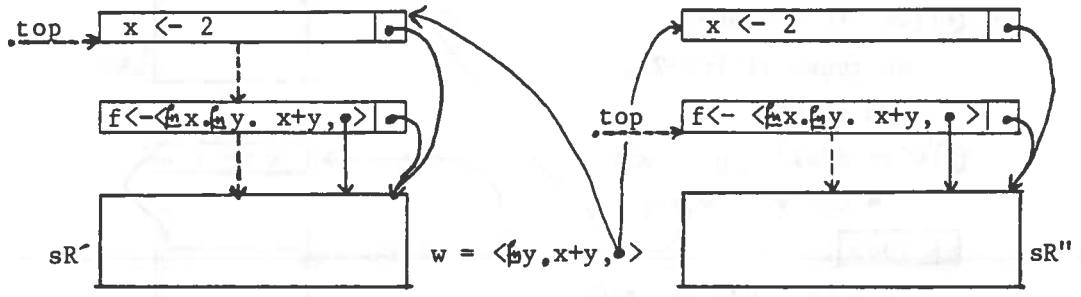
De twee bewerkingen sR :pop zijn in een rechthoek gezet, en verdienen speciale aandacht. Bij de eerste, die sR1 in sR2 verandert, moet het top-element van de stapel worden verwijderd, maar het element zelf mag niet vernietigd worden: vanuit de opgeleverde waarde w´ en dus wf wordt er nog naar die binding "x<-2" gewezen. Bij de tweede :pop moet de binding "y<-6" worden verwijderd. Nu had die binding wel volkomen mogen verdwijnen, omdat hij onbereikbaar geworden is! En na verdwijning ervan zou de binding "x<-2" ook onbereikbaar zijn, en mogen verdwijnen.



Wanneer er nooit bij ontstapeling van sR naar het ontstapelde element gewezen wordt (vanuit sW, eventueel indirekt), kan het beheer over de opslagruimte voor bindingen (omgevingen) dus volgens het LIFO-regime plaatsvinden. Dit is zeker het geval als funkties of procedures (beter: closures) niet als onderdeel van een funktieresultaat optreden, zoals in Pascal en Algol 60. Zie nogmaals de informele uiteenzetting in Par. 4.1.

Let erop dat een closure aan een variabele gebonden kan worden; dit geeft

nog een extra verwijzing in sR. Zoiets treedt op als we niet zoals in het voorbeeld ((fn x. fn y. x+y) 2 6) beschouwen, maar (df f=(fn x. fn y. x+y). f 2 6). De inhoud van sR bij het ter evaluatie nemen van de romp van f (ten gevolge van (f 2)) ziet eruit als sR´ hieronder; de opgeleverde waarde is w. Daarna wordt sR´ gepopped, en ontstaat sR" als inhoud van sR bij het ter evaluatie nemen van 6.

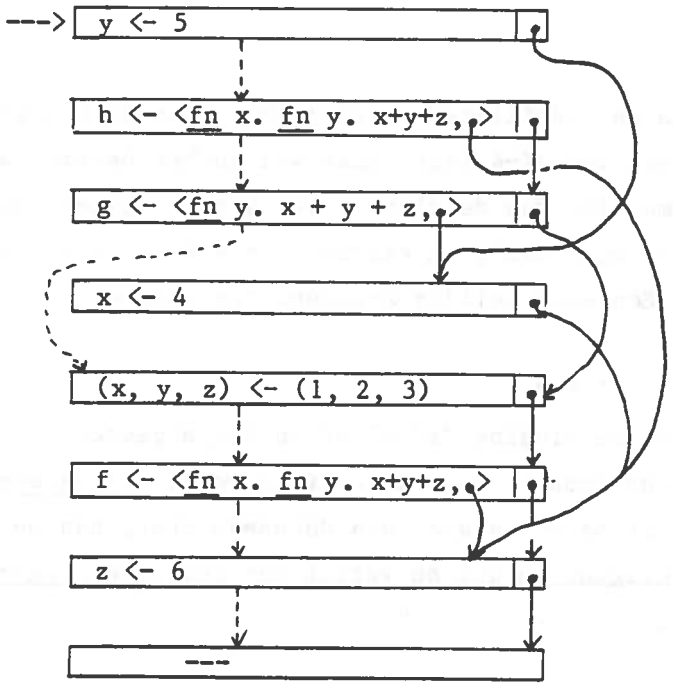


* * *

De stapelstruktuur van sR hebben wij in het voorbeeld zichtbaar gemaakt met onderbroken pijlen ----->. Deze worden traditioneel dynamic links of calling pointers genoemd. De verwijzingen waarmee een omgeving als een geketende lijst gerepresenteerd wordt, worden traditioneel static links of environment pointers genoemd; zij zijn ononderbroken getekend in het voorbeeld. Door static links te volgen doorloop je de bindingen die horen bij tekstueel ("statisch") omvattende bindingskonstrukties; zij vormen tesamen een omgeving. Door dynamic links te volgen doorloop je de omgevingen (of: de eerste bindingen daarvan) die horen bij de achtereenvolgens ("dynamisch") ter evaluatie genomen bindingskonstrukties. (De static en dynamic link van de binding voor x in de evaluatie van een blok (df x=ea. eb), en ((fn x. eb) ea), vallen samen, omdat de bindingskonstruktie die dat blok tekstueel omvat ook het laatst voorafgaande aan dat blok ter evaluatie wordt genomen).

Nog een voorbeeld

Beschouw de volgende inhoud van sR



Zonder enige moeite is hieruit af te lezen hoe de heersende omgeving eruit ziet; volg daartoe de static links vanuit het top-element:

(y<-5, x<-4, z<-6, ---)

Dus evaluatie van x+y+z in deze omgeving levert 5+4+6 = 15. Wanneer deze omgeving gepopped wordt, verschijnt de volgende omgeving op de top van sR; volg weer de static links maar nu vanuit het (volgens de dynamic links) een-na-bovenste element:

```
(h <- <fn x. fn y. x+y+z, (z<-6, ---)>,
    g <- <fn y. x+y+z, (x<-4, z<-6, ---)>,
       (x,y,z,) <- (1,2,3),
              f <- <fn x. fn y. x+y+z, (z<-6, ---)>,
                 z <- 6,
                 ---)
```

In deze omgeving zal x+y+z resulteren in 1+2+3 = 6. We kunnen ook een expressie verzinnen zo dat gedurende de evaluatie ervan sR ooit eens bovenstaande vorm heeft. Er zijn velerlei expressies mogelijk; we geven hier slechts één oplossing:

```
(df z = 6.
 df f = (fn x. fn y. x+y+z).
 df (x,y,z) = (1,2,3).
 df g = (f 4).
 df h = f.
 (g 5))
```

De delen df g = (f 4) en (g 5) zijn het moeilijkst te verzinnen. Aan de static link zien we dat g niet in het bereik van x<-4 ligt, maar wel in het bereik van (x,y,z) <- (1,2,3); voorts is de omgeving van de closure van g een bijstelling (met x<-4) van de omgeving van de closure van f en daarom is g = (f 4) een redelijke, en zelfs geslaagde, poging. Een soortgelijke argumentatie gaat ook op voor (g 5).

* * *

Het deel dat dient ter opslag van de binding "x<-w" of in het algemeen "(xl,..,xN)<-(wl,...,wN)" (dus zonder dynamic en static link) wordt wel datasegment genoemd. De stapel sR lijkt meer op een stapel van datasegmenten, dan op een stapel van omgevingen. Merk overigens op dat de variabelen zelf niet opgeslagen hoeven worden, zie Par. 4.3.

Tot slot noemen we nog twee optimalisaties van de representatie van sR, zonder daar verder op in te gaan. Ten eerste is het mogelijk om de algemene binding "(xl,..,xN)<-(wl,..,wN)" zo op te slaan dat de opzoektijd voor de aan xi gebonden waarde onafhankelijk is van i. Dit is niet-triviaal wanneer de groottes van de representaties van de waarden niet berekenbaar is (at compile time), zoals bij lijsten en bij arrays met berekende (i.p.v. konstante) grenzen. Ten tweede is het mogelijk sR zo te representeren dat de opzoektijd voor de waarde gebonden aan x onafhankelijk is van x en met name van "het verschil in nestingsdiepte (van bereiken) tussen het voorkomen van x en zijn bindend voorkomen" (zie Par. 1.2). Dit gaat met de zogenaamde display-techniek.

Par. 4.6 Kode generatie

In deze laatste stap maken we van eval4 (dat is de kombinatie van eval2 en eval3) een kode-voortbrengende funktie: (kode e) = "een niet-rekursief programma met labels en goto zo dat uitvoering daarvan hetzelfde effekt heeft op sR en sW als (eval4 e)". Het rekursieve karakter van eval4 wordt deels op triviale wijze (met induktie) weggewerkt, en deels door de introduktie van een stapel sT van Terugkeeradressen (labels) en bewerkingen daarop.

De idee is als volgt. Beschouw een typische clausule uit de definitie van eval4, zeg die voor (e1+e2):

```
eval4 (e1+e2) = eval4 e1; eval4 e2;
                w2, sW :pop; w1, sW :pop;
```

```
if w1 ?= m and w2 ?= n > sW :push m+n fi
```

In plaats van het rechterlid direkt te interpreteren kunnen we het ook opvatten als een programmatekst (de kode voor (e1+e2)) die uitgevoerd moet worden wanneer de waarde van (e1+e2) nodig is, d.w.z. op sW gezet moet worden. We moeten dan wel de delen (eval4 e1) en (eval4 e2) ook in dergelijke programmateksten uitgeschreven denken. Bij gegeven e1 en e2 lijkt dat ook wel met induktie naar de opbouw van die expressies mogelijk te zijn. Aldus ontstaat er een kode-voortbrengende funktie met als typische clausule

```
(kode (e1+e2)) = [kode e1]; [kode e2];
                 w2, sW :pop; w1, sW :pop;
                 if w1 ?= m and w2 ?= n -> sW :push m+n fi
```

Het rechterlid is een (programma)tekst; eigenlijk hadden er teksthaakjes omheen moeten staan. De delen tussen de hakenparen [en] vormen zelf niet letterlijk deel van die tekst, maar duiden wel tekstdelen aan. Dus xxx[yyy]zzz staat voor "xxx"++yyy++"zzz", waarbij ++ tekstkonkatenatie is. (Let ook op het verschil tussen ---x--- en ---[x]---; de laatste van deze twee staat voor ---aap--- of ---noot--- of ---mies--- al naar gelang metavariabele x de variabele (identifier!) "aap" of "noot" of "mies" aanduidt).

Maar er zit een addertje onder het gras. Eval4 is wel bijna helemaal, maar niet echt helemaal, met induktie naar de opbouw van expressies gedefinieerd. Beschouw de clausule voor (ef ea):

```
(eval4 (ef ea)) =
       eval4 ef; eval4 ea;
       wa, sW :pop; wf, sW :pop;
       if wf ?= <fn x. eb, rf>
             -> sR :push rf[x<-wa];
                eval4 eb;              -----(*)
                sR :pop
       fi
```

De expressie eb op regel (*) is geen onderdeel van (ef ea) en kan ook niet daaruit berekend worden. Die aanroepen van eval4 kunnen dus niet door uitschrijven, voorafgaande aan evaluatie, weggewerkt worden.

De oplossing voor bovenstaand probleem is als volgt. We zullen ervoor zor-

gen dat de romp van een abstraktie (fn x. eb) maar eenmaal aan kode onderworpen wordt, en dat ten gevolge daarvan de kode voor eb ergens wordt opgeslagen met een label, zeg lb, die het begin van die kode markeert. Een closure <fn x. eb, rf> representeren we dan door <fn x. lb, rf>. In de clausule voor kode (ef ea) kunnen we dan in plaats van (eval4 eb) op regel (*) zetten:

```
sT :push [lt]; goto lb; [lt]: sT :pop
```

waarbij lt een nieuwe nog niet eerder gebruikte label is, en sT een globale stapel van Terugkeeradressen (labels). In de kode voor eb plaatsen we aan het eind een opdracht die zorgt voor de juiste terugkeer:

```
goto (top sT)
```

Aldus is ook de rekursieve aanroep van eval4 op regel (*) weggewerkt. We houden een funktie kode (van type: fct(E) "proc(){gebruikt var sR, sW en sT}") over die helemaal met induktie naar de opbouw van expressies gedefinieerd is. Hij luidt als volgt.

```
kode rec= fct e.
if e ?= x ->                sW :push ((top sR)[x])
|  e ?= (fn x. eb) ->       sW :push <fn [x]. [lb], (top sR)>
```

waarbij bovendien nog ergens de volgende kode wordt opgeslagen (met lb een nieuwe label):

```
[lb]: [kode eb]; goto (top sT)
```

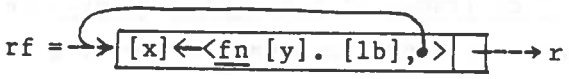
```
|  e ?= (ef ea) ->          [kode ef]; [kode ea];
                           wa, sW :pop; wf, sW :pop;
                           ① if wf ?= <fn x. lb, rf>
                           -> sR :push rf[x<-wa];
                              sT :push [lt]; ② goto lb; [lt]: sT :pop;
                              sR :pop
                           fi ③
                           (de label lt is weer een nieuwe label)
|  e ?= (df x=ea. eb) ->    [kode ea]; wa, sW :pop;
                           sR :push (top sR) [[x]<-wa];
                           [kode eb];
                           sR :pop
|  e ?= (rec x. fn y. eb) ->sW :push <fn y. [lb], rf>
```

waarbij bovendien nog ergens de volgende kode wordt opgeslagen (met lb een nieuwe label):

[lb]: [kode eb]; goto (top sT)

De omgeving rf is rekursief gedefinieerd door
rf rec= (top sR)[[x] <- <fn [y]. [lb], rf>]

(zoals uiteengezet in Par. 4.2 is de representatie hiervoor



)

```
|  e ?= (if e1 then e2 else e3) -> [kode e1]; w1, sW :pop;
                                   if w1=true -> [kode e2] | w1=false -> [kode e3] fi
|  e ?= (e1 + e2) ->              [kode e1]; [kode e2];
                                   w2, sW :pop; w1, sW :pop;
                                   if w1 ?= m and w2 ?= n -> sW :push m+n fi
.
.
fi
```

In de clausule voor (fn x. eb) en voor (rec x. fn y. eb) staat aangegeven dat "bovendien nog ergens de kode c = ([lb]: [kode eb]; goto (top sT)) moeten worden opgeslagen". Desgewenst kan c binnen de kode voor (fn x. eb) resp. (rec x. fn y. eb) worden opgenomen en wel als volgt. Omring het deel c door een sprong eroverheen; bij exekutie kan c dan niet anders binnengekomen worden dan via de beginlabel lb. Bijvoorbeeld, de clausule voor (kode (fn x. eb)) kan dan luiden:

```
goto [ℓ];
     [lb]: [kode eb]; goto (top sT);
[ℓ]:
     sW :push <fn [x]. [lb], (top sR)>
```

waarbij ℓ en lb nieuwe labels zijn.

Het stapelen en ontstapelen op sR en sT gebeurt steeds "gelijktijdig", zie de clausule voor (ef ea). Dus beide stapels kunnen feitelijk tot een stapel sRT

samengesmolten worden. Op die stapel komen dan als elementen

een groep bestaande uit
- een datasegment (voor de binding "x<-w")
- een static link (voor de rest van de omgeving)
- een terugkeeradres (voor de voortzetting van de exekutie na voltooiing van de huidige romp)
- een dynamic link (te gebruiken bij ontstapeling).

Dergelijke groepen worden ook wel stack frames of activation records genoemd. (Ghezzi, Jazayeri 1982; Ch 3.5) geeft een zeer lezenswaardige uitleg van het ook door ons beschreven implementatiemodel, geheel in termen van activation records.

Belangrijke onderwerpen uit Hoofdstuk 4

semantische binding, omgeving
closure
dynamische scope, statische scope
lijst-implementatie van omgevingen
rekursie, rekursieve omgevingen
omgevingsstapel sR, deling van omgevingen
waardenstapel sW
terugkeeradressenstapel sT
LIFO - beheer over opslagruimte,
verstoord door closures in resultaten
eval0 t/m eval4 en kode
dynamic link = calling pointer, static link = environment pointer
datasegmenten, stack frame = activation record

Literatuur bij Hoofdstuk 4

Als inspiratiebron voor dit hoofdstuk hebben (Landin 1964) en (Bjorner 1977) gediend. Het uiteindelijke model wordt ook beschreven in (Ghezzi & Jazayeri 1982) en vele andere boeken over Struktuur etc. van Programmeertalen.

Oefeningen

4.1! Stel dat HOF wordt uitgebreid met de konstante succ (een nieuwe expressievorm dus) en reduktieregel (succ m)->m+1. Geef de aanpassingen in eval (versie 0 t/m 4) en kode. Wenk: alleen de "if wf ?=..." in de clausule voor (ef ea) behoeft verandering.

4.2 Doe hetzelfde als in 4.1 maar nu met een heel stel nieuwe funktiekonstanten, zoals pred, log, exp, comp (voor funktiekompositie: (comp e1 e2 e3)->(e1 (e2 e3)) zodat (comp f g) de kompositie f na g aangeeft), plus enzovoorts. Wenk: (i) behandel (plus m) als een nieuwe konstante, zeg plusm en ga dan te werk bij succ in 4.1; (ii) geef voor dit doel een aparte funktie apply die vanuit eval wordt aangeroepen (eval zelf verandert dan nauwelijks).

4.3! Schets de evaluatie onder eval4 van de volgende drie expressies. (doe eerst oefening 4.4; neem bij twijfel eval3 i.p.v. eval4.)

```
(df f = (fn x. fn y. x+y). f 2 6)
(df f = (fn x. fn y. x+y). f 2) 6
(df f = (fn x. fn y. y+y). f 2) 6
```

Wat is het verschil tussen de expressies, en waar uit dat verschil zich bij de evaluatie? (Formuleer eerst wat je vermoedt.)

4.4! Geef de definitie van eval4; formuleer het verband met eval1, eval2 en eval3.

4.5! Verzin een voorbeeld zodat eens een element (f <- <fn x. eb, r0>, r1) op sR wordt gezet, waarbij r0 en r1 van elkaar verschillen. Doe dit ook zonder van rec gebruik te maken.

4.6! Wat zou het resultaat van ((fn x. fn y. x+y) 2 6) zijn wanneer altijd bij sR :pop de ontstapelde ruimte wordt vrijgegeven voor de opslag van andere waarden? En hoe zit dat bij ((fn x. fn y. x+y) 2 (df x=6. z))?

4.7! Behandel voorbeelden zoals (df f = (fn x. fn y. x+y). e) waarbij

```
e = (f 2 6)
e = (df g = (f 2). (g 6)+(g 7)+(g 8))
e = ((f 2)(f 6 7))
```

4.8! Behandel (df f rec= (fn (x,y). if x eqn 0 then y else f(x-1,y+1)). f(2,3)) en de fakulteitsfunktie, enzovoorts.

4.10 Geef varianten eval1', eval2', eval3', eval4' en kode' die van de oorspronkelijke verschillen doordat zij de vervanging van x door nx = "het verschil in nestingsdiepte tussen het voorkomen van x en z'n bindend voorkomen" (bij onderwerping aan de omgeving r of (top sR)) zelf uitvoeren. Wenk: geef hun de extra parameters n en d mee, zie Oefening 1.14.

4.11 (Oefening in beschrijvend programmeren) Geef een zuivere funktie kode" zonder neveneffekten, die hetzelfde resultaat oplevert als kode. Wenk: (i) zie de opmerking na de definitie van kode, en (ii) geef kode" nog een extra parameter LV (labelverzameling) mee, wellicht gerepresenteerd als een (oneindige) lijst, waaruit steeds de nieuwe labels geput kunnen worden (en laat kode" ook de nog niet gebruikte labels van LV als extra resultaat opleveren).

4.12 Formuleer de implementatie van omgevingen als geketende lijsten, en de implementatie van sR en sW en sT in de taal Pascal of Algol 68. (Gebruik new(p,v) voor de ingebruik name van een nieuw stuk opslagruimte waarin tevens v wordt opgeslagen, en waarnaar de pointervariabele p dan gaat verwijzen. In Algol 68 is dit: p:=heap:=v.) Geef dan een versie van kode (geformuleerd in SASL of de informele metataal) die als resultaat een Pascal of Algol 68 programma oplevert.

4.13! Is er verschil in evaluatie (of uitvoering van de kode) van ((fn x. eb) ea) en (df x=ea. eb)?

4.14! Is er verschil in evaluatie van (df x=ex. df y=ey. eb) en (df (x,y) = (ex,ey). eb)?

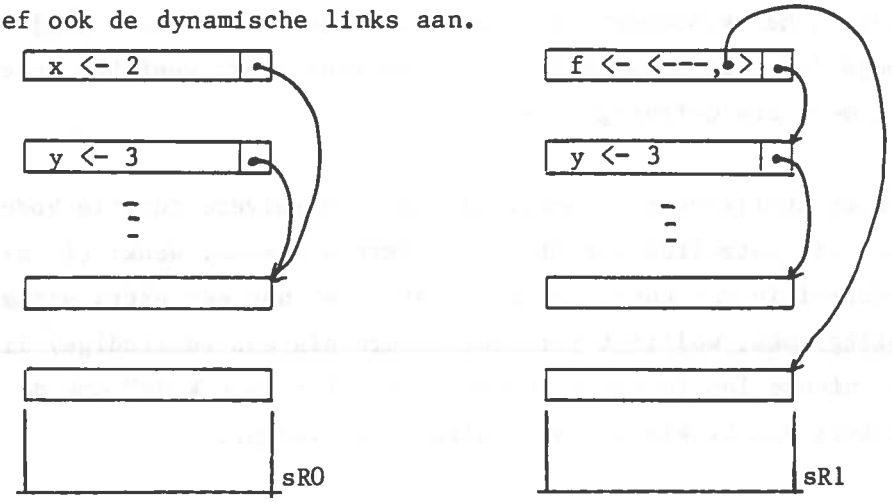
4.15! Geef de evaluatie, volgens eval4, van expressies zoals (hd (1:2:nil)), (df x=1:2:nil. hd x), ((fn x. x) (1:2:nil)), (1:2:nil), (df f = (fn x. hd x). df l = (1:2:nil). f l + f l), enzovoorts.

4.16! Ga na dat (kode (df x=ea. eb)) en (kode ((fn x. eb) ea)) bij evaluatie hetzelfde effekt hebben op sW, sR en sT.

4.18 Werk ((Y fn f. fn x. if x eqn 0 then 1 else x * f(x-1)) 2) uit (onder eval4 of onder kode), waarbij Y staat voor (fn f. (fn x. f (x x)) (fn x. f (x x))). Doe dit natuurlijk eerst aan de hand van de reduktieregels.

4.19 Schets de evaluaties (en reduktie) van ((twice (twice sqr)) 3) en ((twice twice) sqr 3), waarbij sqr = (fn x. x*x) en twice = (fn f x. f (f x)).

4.20! Geef een expressie die bij evaluatie volgens eval3 aan sR achtereenvolgens, maar niet noodzakelijk direkt op elkaar volgend, de volgende inhoud geeft. Geef ook de dynamische links aan.



4.21! Verzin zelf wat van die situaties zoals in 4.20 en bedenk er (de) juiste expressies bij. (Zie ook Oefening 4.22).

4.22! Ga na dat in sR de static en dynamic links altijd "naar beneden" wijzen!

4.23! Is er verschil in de mate waarin sR, sW en sT aangroeien gedurende de evaluatie van (sum 1000) en (sum´ 0 1000), waarbij

```
sum rec= fn n. if n eqn 0 then 0 else n+sum(n-1)
sum´rec= fn s n. if n eqn 0 then s else sum´(s+n)(n-1)
```

(In beginsel is het mogelijk dat de evaluatie van (sum´ 0 1000) gebeurt zonder aangroei van sR en sT en sW. We hopen er nog aan toe te komen om dit in een nog te schrijven paragraaf te behandelen.)

4.24! Veralgemeen de eval´s zodat ook expressies zoals (fn (x,y...z). eb) en (df (x,y,..z) = (ex,ey,..ez). eb) zijn toegestaan.

4.25! Teken de stapel sR die op den duur te voorschijn komt bij de niet-terminerende evaluatie van

a. df g rec = (fn f. g (fn x. f x)). g g

b. df g rec = (fn f. g (rec f. fn x. f x)). g g

c. df g rec = (fn f. g f). g g

d. df g rec = (fn f. g g). g g
e. df g rec = (fn f. f g). g g
f. df g rec = (fn f. df h = (fn x. x). g f). g g
g. verzin er zelf een

4.26! Geef de stapel sR direkt bij aanvang van de applikatie (f a) in

df a=3. df f=(fn x. df a=7. a+x). df b=4. f a + f b

4.27! Geef de stapel sR direkt bij aanvang van de applikatie (f x) in

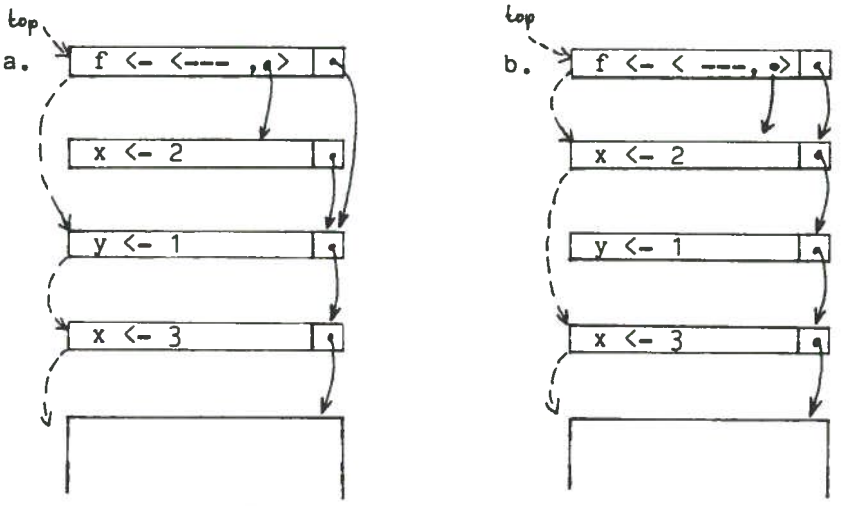
(fn x. df f = (fn y. y+20). f x) 10

4.28! Geef de stapel sR wanneer --- ter evaluatie wordt genomen:

a. df x=0. df y=1. df f=(df x=2. (fnz. x)). ---
b. df x=0. df x= (df y=1. 2). df f = (fn z. x). ---
c. df x = (df x = (df y=1. 2). 0). df f =(fn z. x)). ---
d. df x=0. df y = (df x=2. 1). df f =(fn z. x). ---
e. df x=0. df f = (df y=1. df x=2. (fn z. x). ---
f. df x=0. df f = (df x = (df y=1. 2). (fn z. x)). ---

En wat is het resultaat van (df z=7. f x) wanneer die geëvalueerd wordt met de door u gegeven stapel sR?

4.29! Geef expressies die tijdens de evaluatie ooit eens de volgende vorm aan sR geven. (Kies (fn z. x+y+z) op de plaats van --- .)



En wat is het resultaat van (df x=5. f x) bij die vorm van sR?

4.30! Geef sR wanneer eb ter evaluatie wordt genomen.

a. fn x. (fn y. fn x. eb 1 2) 3
b. fn x. ((fn y. fn x. eb) 1 2) 3
c. (fn x. (fn y. fn x. eb) 1 2) 3
d. (fn x. (fn y. (fn x. eb) 1) 2) 3

4.31 Breidt eval (versie 0 t/m 4) uit voor call-by-name; zie Hoofdstuk 2, Par. 2.2. Wenk, behandel een name-argument als een parameterloze procedure.

4.32 Geef een formele grammatika voor een (zo groot mogelijke) subtaal van HOF waarin funkties niet in het resultaat van een funktie of blok kunnen optreden. Wenk: onderscheid de funktie-variabelen als een aparte syntaktische kategorie en laat funktie-expressies uitsluitend in een definiërende expressie toe.

4.33! Geef het resultaat van kode(fac) waarbij fac staat voor:
(rec f. fn x. if x eqn 0 then 1 else x*f(x-1)). Geef de kodes voor de diverse subexpressie duidelijk aan.

4.34! Geef sW en SR vóór aanvang, direct bij het begin en direct na afloop van de hoofd-applicatie in eval4 ((rec f. fn y. eb) ea) en in eval4 (df f rec= (fn y. eb). f ea).

HOOFDSTUK 5

KORREKTHEID
(eerste koncept)

In Hoofdstuk 1 hebben we een formele definitie gegeven van de syntaxis en (operationele) semantiek van HOF. Maar daarmee zijn we er nog niet. Minstens zo belangrijk als de formele semantiek is een bruikbaar formalisme voor het doen van korrektheidsuitspraken en de bewijzen ervan. Tevens is het begrip "semantische gelijkwaardigheid" al vele malen genoemd, maar nergens precies gedefinieerd. In dit hoofdstuk zullen we dat begrip precies definiëren (of de verscheidene alternatieven noemen), en een stel afleidingsregels en axioma's geven waarmee semantische gelijkwaardigheden kunnen worden bewezen. Daarmee hebben we dan een middel om veel korrektheidsuitspraken te bewijzen.

Een voorbeeld moge dit verduidelijken. Laten we de semantische gelijkwaardigheid van twee expressie e1 en e2 noteren met: e1=e2. Stel voorts dat we een overduidelijk maar uiterst inefficiente expressie e1 hebben voor het voortbrengen van een lijst van oplossingen voor het acht-koninginnen probleem. Als e2 ook zo'n lijst voortbrengt, op veel efficiëntere wijze maar verder nogal ondoorzichtig, dan kunnen we de korrektheid van e2 formuleren als e2=e1. De axioma's en afleidingsregels voor = stellen ons -hopelijk- in staat die uitspraak ook formeel te bewijzen. En wellicht is er ook een stapsgewijze transformatie mogelijk van e1 naar e2, waarbij iedere stap een expressie door een gelijkwaardige vervangt volgens het axiomastelsel. (Er worden zelfs pogingen gedaan zo'n transformatieproces gedeeltelijk of helemaal te automatiseren).

Welnu dan, wat is de relatie = die we wensen te hebben? Allereerst zal die aan de eigenschappen van een "gelijkheid" moeten voldoen, te weten

-reflexiviteit : e = e
-symmetrie : e = e' => e' = e
-transitiviteit : e = e', e' = e" => e = e"
-kongruentie : e = e' => ---e--- = ---e'---
-substitutiviteit: e = e' => (P(e) <=> P(e'))

In de laatste regel staat P voor een Predikaatlogische formule uit een nader te

bepalen klasse, zoals bijvoorbeeld: P(e) = (A m. E n. (e m)=n). Bovenstaande eigenschappen stellen ons in staat de gebruikelijke handelingen te verrichten die je van een begrip zoals "semantische gelijkwaardigheid" zou verwachten.

Daarnaast zijn er nog een aantal specifieke eigenschappen die je van = zou willen eisen. Bijvoorbeeld,

a. (1+2) = 3, ((fn x. x+x) 2) = 4
b. (fn x. x+(1+2)) = (fn x. x+3)
c. (1+2) = (4-1)
d. (rec x. 1:1:x) = 1:(rec x. 1:1:x)
e. (fn x. x+x) = (fn x. x*2)

De geldigheid van a is niet moeilijk te verkrijgen; de ene expressie reduceert tot de ander. De definitie van = moet dus zo luiden dat als de ene expressie tot de ander reduceert, zij in ieder geval gelijkwaardig zijn. Zoiets is ook het geval bij b. Weliswaar reduceert de een niet tot de ander volgens de kompositionele strategie, maar er zijn zulke redukties mogelijk. Geval c laat zien dat niet zo zeer de een tot de ander moet reduceren, maar dat beide slechts een gemeenschappelijk reduktieresultaat hoeven te hebben (konfluentie!). Helaas is zelfs dat nog te zuinig: de expressies van d hebben geen gemeenschappelijk reductieresultaat, maar willen we toch als gelijkwaardig beschouwen. (Reduktietussenresultaten van de een hebben een even aantal enen, terwijl de reduktietussenresultaten van de ander een oneven aantal enen hebben!) Beide expressies in d stellen een oneindige lijst van enen voor. "In de limiet" zijn ze wel konfluent. Tenslotte geval e. Dat zijn geheel verschillende expressies. Maar als we ze alleen toepassen op argumenten van de vorm m (dus bijv. niet true of nil), dan zijn zij wat hun netto effekt betreft toch ononderscheidbaar. Waarschijnijk gaat typering dus een rol spelen bij de (deze!) definitie van =.

Stel nu dat we een bevredigende definitie voor = hebben gevonden. Dan kunnen we aan de hand daarvan allerlei uitspraken van de vorm e=e´ bewijzen. Maar die bewijzen vereisen wellicht diepe kennis van de reduktierelatie, en vereisen misschien indukties naar de lengte van de redukties, terwijl we liever indukties naar de opbouw van expressies willen hebben. Het zijn immers de expressies die we schrijven en waarover we redeneren, en niet de daardoor opgeroepen berekeningen. Vergelijk dit met de bekende regel voor de iteratie:

$$\frac{\{B \text{ and } P\}\ S\ \{P\}}{\{P\}\ \text{while } B \text{ do } S \text{ od } \{P \text{ and not } B\}}$$

De premisse vereist slechts dat we S eenmaal inspekteren, terwijl de konklusie de invariantie van P bevestigt over een mogelijk vele malen herhaalde exekutie van S. Redeneringen aan de hand van de tekst om uitspraken te doen over de opgeroepen berekeningen zijn kruciaal in de konstruktie van korrekte programmatuur. Dus we willen ook voor $=$ induktieregels hebben (met name voor rec-expressies) die de romp eb slechts een of twee keer inspekteren, om dan toch tot een konklusie voor (rec x. eb) te leiden. Aldus zouden we een "praktisch bruikbaar" axioma-stelsel hebben voor $=$. (Het valt overigens nog te bezien of alle waarheden van de vorm $e=e'$ volgens dat axiomastelsel bewezen kunnen worden. Dit is vast niet het geval).

* * *

In afwachting van meer kennis van zaken stel ik verdere uitwerking van bovenstaande uit.

Literatuur bij Hoofdstuk 5

Voor bewijsmethoden van rekursieve funkties zie (Manna et al 1972), (Turner 1982) geeft bewijzen van stellingen over (on)eindige lijsten.

HOOFDSTUK 6

OVER DE UITDRUKKINGSKRACHT VAN HOGERE ORDE FUNKTIES

Par. 6.1 Inleiding

In dit hoofdstuk willen we aantonen dat hogere orde funkties een grote uitdrukkingskracht hebben. Die uitdrukkingskracht is zo groot dat we in beginsel alleen de zuivere lambda-expressies in HOF hadden hoeven opnemen, omdat de datatypen der getallen, waarheidswaarden, lijsten en groepen, en ook rekursie, toch met behulp van hogere orde funkties (de zuivere lambda-expressies dus) uitgedrukt kunnen worden. Dat zullen we dan ook doen.

Het is opmerkelijk dat een HOF-programmeur niet hoeft te weten of bovengenoemde datatypen en rekursieoperator als afzonderlijke taalelementen zijn gedefinieerd of met hogere orde funkties zijn uitgedrukt. Bij onze keuze van de syntaxis blijkt het verschil in schrijfwijzen alleen uit de verplichting om (plus e1 e2) te schrijven in plaats van (e1 + e2); dus konsequent een prefix-notatie met een voorgedefinieerde funktie die de rol overneemt van de pre-, in-, post-, of distfix-operator. Maar herinner je dat we geen aandacht willen besteden aan dergelijke aspekten van de syntaxis; zouden we dat wel doen, en bijvoorbeeld "distfix identifiers" algemeen opnemen in de taal, dan zou zelfs dit syntaktische verschil er niet zijn geweest. (Distfix identifiers zijn standaard in de taal B, zie (Geurts 1982)). Wat betreft de prestatie (= kostenaspekten zoals tijdsduur van de evaluatie en hoeveelheid benodigde opslagruimte) kan er wel verschil optreden. (Ik vermoed echter dat dat verschil tot een vaste faktor beperkt kan blijven, en met name geen verandering in de grootte-orde van die kostot gevolg hoeft te hebben! We komen hierop nog terug).

Het grote voordeel van een kleine taal met een universele of grote uitdrukkingskracht is dit. De taal is klein, dus gemakkelijk te definiëren, te implementeren, te onderzoeken op eigenschappen etcetera. Voor speciale toepassingsgebieden, zoals rekenkunde of lijstverwerking of patroonherkenning of tekstverwerking of boekhouding enzovoorts, zijn de daarbij horende karakteristieke bewerkingen uitdrukbaar (vanwege de universele uitdrukkingskracht). Dat kost misschien wel enige moeite, maar het hoeft maar eenmalig te gebeuren. Op willekeurige machine waarop de taal verwerkt kan worden, is dan alle toepassingsprogrammatuur te gebruiken. Wanneer het toepassingsgebied dat nodig maakt, kunnen altijd enige van de speciale bewerkingen direkt in apparatuur gerealiseerd worden.

Juist door de eenvoud van de taal is de uitwisseling tussen programmatuur en apparatuur goed te doen. De prestaties kunnen daardoor verbeterd worden in vergelijking met de evaluatie van de definiërende expressies van die bewerking. Maar of dit nu wel of niet gebeurt, de toepassingsprogrammatuur blijft overdraagbaar; hij is op alle machines verwerkbaar. Dit is veel prettiger dan het ontwikkelen van speciale toepassingsgerichte talen en daarbij horende machines (=implementaties). Er is dan geen overdraagbaarheid, er bestaat het gevaar dat sommige speciale bewerkingen toevallig niet in de taal waren opgenomen, en er is dan het nadeel dat iedereen verscheidene talen moet leren.

De hogere orde funkties hebben inderdaad een universele uitdrukkingskracht in de volgende zin: ieder effektief berekenbare afbeelding kan ermee worden aangegeven. We zullen dat aantonen door te laten zien dat een willekeurig Turingmachineprogramma met behulp van zuivere lambda-expressies kan worden nagebootst. Het is overigens ook niet moeilijk om de zogenaamde partieel rekursieve afbeeldingen met lambda-expressies uit te drukken, zeker niet wanneer we eenmaal het datatype der getallen met hogere orde funkties hebben gerepresenteerd.

Tussen haakjes, net als lambda-expressies hebben ook Turingmachineprogramma's en Registermachineprogramma's en Combinator-expressies een universele uitdrukkingskracht, en zijn het ook "kleine" talen. Toch hebben de lambda-expressies een groot voordeel boven die andere: abstraktie (in de informele zin: het afzien van sommige aspekten) kan syntaktisch vorm gegeven worden door de onbeperkte mogelijkheid van abstraktie (vorm (fn x. eb) uit eb met betrekking tot x) en naamgeving (in ((fn x. eb) e) en (df x=e. eb) wordt e benoemd met x). In die andere formalismen kan informele abstraktie alleen maar buiten de taal om vorm worden gegeven. Daarenboven is bij lambda-expressies een goede typering gemakkelijk mogelijk, met alle voordelen van dien, terwijl dat bij die andere formalismen niet zo is. Ter illustratie zullen we veel lambda-expressies volgens de SVP-typering typeren.

Par.6.2 Algemeen

Zij e een willekeurige HOF expressie, of een informele beschrijving van een of ander begrip. Dan duiden we met [e] de representatie van e in zuivere lambda-expressies aan.

De korrektheid van de omzetting van willekeurige HOF expressies naar zuivere lambda-expressies luidt formeel als volgt.

a. De semantische gelijkwaardigheid blijft behouden onder de representatie; ofwel
   e $\underline{=}$ e' => [e] $\underline{=}$ [e']
b. Er worden geen ongewenste semantische gelijkwaardigheden geïntroduceerd door de representatie; ofwel
- [m] $\underline{\neq}$ [n] voor m≠n
- e $\underline{\neq}$ e' => [<e,e2>] $\underline{\neq}$ [<e', e2>] en [<e1, e>]$\underline{\neq}$[<e2, e'>]
- [true] $\underline{\neq}$ [false]
- [nil] $\underline{\neq}$ [eh: et] en
  e $\underline{\neq}$ e' => [e: et] $\underline{\neq}$ [e': et], [eh: e] $\underline{\neq}$ [eh: e']

Het is dus wel toegestaan dat [0] en [true] gelijk(waardig) zijn, ook al zijn 0 en true niet semantisch gelijkwaardig. Hetzelfde geldt voor 0 en (0-1)! We zullen inderdaad zo'n representatie kiezen. Middels typering is desgewenst "het gebruik van [true] als een getalrepresentatie" uit te sluiten. Ook dat zullen we in de SVPtypering realiseren.

Een voldoende voorwaarde opdat eis a. hierboven vervuld is, is dat iedere reduktiestap door een reeks reduktiestappen op de representatie gesimuleerd wordt:

We zullen de korrektheid van de representatie echter nergens bewijzen.

* * *

We geven nu de representatie in zuivere lambda-expressies van de df- en de rec-expressie. Vanwege de zorgvuldig nagestreefde overeenkomst tussen Definitie en Parameterisatie, is de omzetting van (df x=ea. eb) triviaal.
[df x=ea. eb] = ((fn x. eb) ea)
Een representatie voor rekursie is niet zo gemakkeijk te vinden. Hoewel er best een motivatie voor de nu volgende keuze te bedenken is, zullen we die kortheidshalve niet geven. De traditionele representatie luidt

[rec x. eb] = (W W) met W = (fn w. (fn x. eb) (w w))
waarbij w niet vrij is in (fn x. eb)

Het is nu gemakkelijk na te gaan dat deze funktie goed is.

```
[rec x. eb] = (W W)
            = (fn w. (fn x. eb) (w w)) W
            =>(fn x. eb) (W W)
            = (fn x. eb) [rec x. eb]
            =>eb[x/[rec x. eb]]
```

Voorts kunnen we (W W) nog iets "eenvoudiger" schrijven als (Y (fn x. eb)) met Y = (fn f. (fn x. f(x x)) (fn x. f(x x))). Traditioneel wordt Y wel de rekursie-operator genoemd.

Par. 6.3 De datatypen funktioneel gerepresenteerd

We gaan nu de overige expressievormen van HOF in zuivere lambda-expressies omzetten. We zullen dat modulair opzetten, zodat alternatieve omzettingen slechts zeer lokaal wijzigingen aanbrengen en de rest ongemoeid laten. Daartoe zullen we samenhangende operaties en konstanten als een geheel uitdrukken. Zo'n samenhangend geheel wordt wel module genoemd, of specifieker: datatype. (Om precies te zijn, onder een datatype verstaan we een stel verzamelingen met bewerkingen erop en konstanten eruit. Zijn we niet geïnteresseerd in de verzamelingen zelf, maar alleen in de onderlinge relaties tussen de bewerkingen en konstanten, dan spreken we van abstrakte datatypen. We zullen in dit hoofdstuk van geen enkel (abstrakt) datatype de onderlinge relaties tussen de bewerkingen en konstanten formeel (met wetten) vastleggen; steeds is op grond van de naamgeving duidelijk welke relaties bedoeld zijn).

Allereerst herschrijven we expressies als volgt: (e+e') wordt (plus e e'). 0 wordt zero, 1 wordt one, ..., (if e then e' else e") wordt (if e e' e"), true wordt true, enzovoorts. Met andere woorden, de (pre-, in-, post- en distfix-) operatorsymbolen worden vervangen door geschikte funktie-identifiers, en net zo voor de konstanten. Dan rest ons nog om de aldus verkregen expressie met de volgende definities te omringen

```
df zero=[0], one=[1], two=[two],..., plus=[fn x y. x+y],
   true=[true], false=[false], if=[fn x y z. if x then y else z],...
```

Afgezien van de prefix funktieapplikatie in plaats van de operatorsymbolen, hoeft een HOF programmeur dus niet eens te weten dat alles met funkties is gerepresenteerd.

De modulariteit van de vertaling in lambda-expressies wordt nog groter door de omringende definities als volgt te schrijven.

```
df natADT  = <[0], [1], [2],..., [fn x y. x+y],...>
,  boolADT = <[true], [false],[fn x y z. if x then y else z],...>
,  zero    = natADT.1
,  one     = natADT.2,...
,  true    = boolADT.1,...
```

Weliswaar hebben we nu groepering gebruikt, maar ook die kan geëlimineerd worden, zoals we zullen zien. Willen we nu een alternatieve omzetting geven voor getallen en de bewerking daarop, dan hoeven we alleen maar de definitie van natADT te wijzigen. Al het andere kan intakt blijven, indien daar nergens van de herroepen representatie gebruik gemaakt is. Middels de SVPtypering kunnen we "verboden gebruik" van die representatie inderdaad syntaktisch onmogelijk maken. (Maar helaas, als we gebruik maken van de SVPtypering is de groepering niet meer in lambda-expressie om te zetten. Dit is nog een gebrek aan die typering. Er wordt gewerkt aan verbeteringen).

Het zal blijken dat we steeds een waarde van een abstrakte datatype zullen representeren door een (of meer) "control structure(s)" waarin die waarde op karakteristieke manier gebruikt wordt.

* * *

Waarheidswaarden: boolADT

Het karakteristieke gebruik ban een waarheidswaarde vindt plaats in het if-gedeelte van een if-expressie: daar wordt zo'n waarde gebruikt om een keuze te sturen. Dus we kiezen voor willekeurige waarheidswaarde b

```
[b] = "if b then e1 else e2" als fct van e1 en e2
    = fn x y. "if b then x else y"
```

Dus

```
[true]  = fn x y. x
[false] = fn x y. y
```

Nu is [if x then y else z] = (x y z), dus we vinden

```
if = [fn x y z. if x then y else z]
   = fn x y z. x y z
   = fn x. x
and = [fn x y. if x then y else false]
   = fn x y. x y [false]

   = fn x y. x y [false]
```

```
or = fn x y. x [true] y
```

Samengevat in een groep krijgen we het volgende ADT:

```
boolADT = <{true} fn x y. x, {false} fn x y. x, {if} ..., ...>
```

Voordat we overgaan tot een volgend datatype, zullen we nog de zojuist verkregen expressies volgens de SVPtypering typeren. Helaas brengt dat wel enige aanpassingen met zich mee, maar gelukkig zijn die aanpassingen slechts toevoegingen! Dus boolADT wordt uitgebreid tot boolADT´.

Allereerst merken we op dat we —onafhankelijk van de alreeds verkregen expressies!!— het volgende type voor boolADT´ zouden wensen:

```
< repr: tp {het type der waarheidswaarderepresentaties, genaamd: repr}
, repr {type voor true}
, repr {type voor false}
, (z:tp -> repr -> z -> z -> z) {type voor if}   *)
.
.
.
>
```

*) Notatie. De pijl in een type-expressie is rechts-associatief, zodat (fn x:tx y:ty. eb) = (fn x:tx. (fn y:ty. eb)) het type (x:tx -> y:ty -> type van eb) = (x:tx -> (y:ty -> type van eb)) heeft. Wanneer in (x:tx -> t) de x niet vrij voorkomt in t, schrijven we (tx -> t). De epsilon in de syllabus over Typering schrijven we nu als een:.

Vergelijken we dit met boolADT dan vallen er twee dingen op. Ten eerste heeft de eerste komponent geen overeenkomstige in boolADT. Die komponent in boolADT´ is nodig om in het type de relatie tussen de typen der komponenten goed te kunnen beschrijven; met name blijkt nu dat de tweede parameter van if een waarheidswaarde moet zijn. Ten tweede zien we dat if nu drie in plaats van twee parameters heeft. De extra eerste parameter is een type, en die wordt gebruikt om af te dwingen dat de then- en else-parameter en het resultaat alle eenzelfde (maar overigens vrij te kiezen) type hebben.

We willen nu de definitie

```
df boolADT´ : <repr:tp, repr, repr, (z:tp->...) ...>
            = ---------
```

voltooien, met op ------ een expressie die uit boolADT ontstaat door slechts toevoegingen te doen. Gezien het feit dat de waarheidswaarderepresentaties het eigenlijke werk van if doen, kiezen we (z:tp->z->z->z) als de eerste komponent van boolADT'. Bijgevolg moeten ook de representaties van true en false enigszins worden aangepast. Dat is op zich niet moeilijk. De uitgewerkte, SVPgetypeerde definitie van boolADT' luidt als volgt.

```
df boolADT' : < repr: tp
              , repr
              , repr
              , (z:tp->repr->z->z->z)
              .
              .
              .
              >
            = < (z:tp->z->z->z)
              , fn z:tp x:z y:z. x
              , fn z:tp x:z y:z. y
              , fn z:tp b:(z:tp->z->z->z). b z
              .
              .
              .
              >
```

Merk op dat we voor de "if-komponent" ook (fn z:tp b:(z:tp->z->z->z) x:z y:z. b z x y) hadden kunnen schrijven. (Want een goedgetypeerde (fn x. e x) is semantisch equivalent met e, zie Hoofdstuk 5).

Tenslotte merken we op dat er nog veel meer definiërende expressies voor boolADT' mogelijk zijn, die ook ieder voldoen aan het gegeven type. Die mogen best door elkaar vervangen worden, omdat de typeringsregels het onmogelijk maken om van de toevallige representatiekeuzen gebruik te maken.

Natuurlijke getallen

Een karakteristiek gebruik van natuurlijke getallen is "om te tellen", met name om herhalingen te besturen. In gebiedende talen gebeurt dat in

```
for i to n do ...
```

In funktionele talen wordt dit de n-voudig herhaalde toepassing van een funktie.

Daarom kiezen we als representatie van willekeurige n

```
    [n] = fn f x. f(f...x))) {n maal een f}
dus [0] = fn f x. x
    [1] = fn f x. f x
    [2] = fn f x. f(f x)
```

Toevallig blijkt [0]=[false] (zie boolADT). Wanneer we de SVPtypering toevoegen kan daar toch geen gebruik van gemaakt worden!

Het is niet moeilijk de successor te representeren.

```
succ = fn n. "de n+1-voudige herhaling van f op x" als funktie van f en x
     = fn n. fn f x. f (n-voudige herhaling van f op x)
     = fn n. fn f x. f (n f x)
     = fn x y z. y (x y z)
```

De funktionele representatie van de predecessor is niet zo eenvoudig. We gebruiken hier gewoon groepering in de representatie, maar zullen verderop zien dat dit gebruik in zuivere lambda-expressies (zelfs: goed SVP-typeerbare!) is om te zetten. Het idee is als volgt. Om de predecessor van n te berekenen passen we de volgende funktie f n-voudig herhaald toe op startwaarde <[0], [doet niet ter zake]>.

```
f (<[m], --->) = <[m+1], [m]>
```

Aldus is f^n (<[0], --->) = <[n], [n-1]> voor n>0: in de linkerkomponent (re)-konstrueert f het objekt [n], en daarbij ijlt de rechterkomponent steeds één slag na. Er rest daarna nog om van <[n], [n-1]> de rechterkomponent te nemen.

```
pred = fn n. (n f <[0], --->).2
       met f = fn "<x, y>". <succ x, x>
             = fn z. <succ (z.1), z.1>
```

De expressie op de streepjes is bepalend voor het resultaat van (pred [0]). In HOF zou de reduktie van (0-1) stokken; zonder extra voorzieningen zijn er in de zuivere lambda-expressies echter geen stokkende redukties. Wel kunnen we er een expressie schrijven die geen eindigende reduktie heeft.

De test op gelijkheid met nul wordt als volgt gerepresenteerd.

```
eqzero = fn n. "pas f n-voudig toe op [true]"
                  met f (...) = [false]
       = fn n. n (fn x. [false]) [true]
```

De overige operaties op natuurlijke getallen laten zich middels rekursie nu ook vrij eenvoudig in zuivere lambda-expressies uitdrukken. We kunnen zelfs in heel veel gevallen de rekursie-expressie en zijn funktionele representatie (die niet goed SVP-typeerbaar is!) vermijden, door steeds van "primitieve rekursie" gebruik te maken. Primitieve rekursie van f (met k+2 veranderlijken) en g (met k veranderlijken) is als volgt gedefinieerd. We noteren met xk een lijst van k variabelen, k≥0.

```
primrec f g = fn n xk. f (..(f (g xk)....) n xk  {n maal een f}
            = fn n xk. { g xk                          voor n=0
                       { f (primrec f g (n-1) xk) voor n>0
```

De zuivere lambda-expressie-representatie voor primrec zullen we kortheidshalve hier niet uitwerken.

Wanneer we de belangrijkste bewerkingen op natuurlijke getallen groeperen krijgen we

```
natADT = < [0], [1], [2],...
         , succ, pred, eqzero, ...primrec, ... >
```

Net zoals bij de typering van boolADT, die noodzaakte tot de aanpassing van boolADT tot boolADT', kunnen we nu ook natADT aanpassen en typeren. Zonder verdere uitleg geven we het resultaat.

```
natADT': <repr: tp {type der getalrepresentaties, genaamd: repr}
         , repr, repr, repr... {type van 0,1,2...}
         , (repr->repr), (repr->repr) {type van succ en pred}
         , (repr->bool) {type van eqzero}, ...
         , ----- {type van primrec}
         >
       = < (z:tp->(z->z)->z->z)
         , {[0]'=} fn z:tp f:(z->z) x:z. x
         , {[1]'=} fn z:tp f:(z->z) x:z. f x
         , ...
         , {succ'} fn n: (z:tp->(z->z)->z->z).
                        fn z:tp f:(z->z) x:z. f (n z f x)
         , {pred'} ...
```

```
, {eqzero'} (fn n: (z:tp->(z->z)->z->z).
                n bool (fn x:(bool->bool). false) true)
.
.
.
>
```

We zijn er bij de komponenten van eqzero vanuit gegaan dat bool, false en true als volgt gedefinieerd zijn

```
df bool: tp == boolADT.1
; false: bool = boolADT.3
; true: bool = boolADT.2
```

Wellicht ten overvloede merken we op dat de representatiekeuze voor natuurlijke getallen (namelijk: een getal wordt gerepresenteerd als een funktie van type (z:tp->(z->z)->z->z)) volledig is afgeschermd buiten de definiërende expressie voor natADT'. Immers, volgens het type van natADT' is slechts het volgende bekend.

```
{nul=}    natADT'.2    heeft type natADT'.1
{een=}    natADT'.3    heeft type natADT'.1
 ...
{succ=}   natADT'.xx   heeft type (natADT'.1 -> natADT'.1)
{pred}    natADT'.yy   heeft type (natADT'.1 -> natADT'.1)
(eqzero}  natADT'.zz   heeft type (natADT'.1 -> bool)
 ...
```

En het type natADT'.1 is niet van de vorm (t->t') zodat een getal niet als een funktie gebruikt kan worden. Met name is

```
natADT'.2  bool (fn x: bool. x) true
```

fout getypeerd (hoewel het semantisch, volgens de reduktieregels, perfekt in orde is!). Aan het type van natADT' is niet te zien hoe de definiërende expressie luidt. De volgende expresssie kan bijvoorbeeld ook als type-korrekte definiërende expressie voor natADT' genomen worden:

```
< nat
, {[0']=} 0
, {[1']=} 1
...
, {succ'=} fn x: nat. x+1
, {pred'=} fn x: nat. x-1
, {eqzero'=} fn x: nat. x eqn 0
...
>
```

gesteld dat nat een type is en 0:nat, 1:nat en (e1+e2):nat indien e1,e2:nat, enzovoorts. (We hebben echter deze expressie niet gekozen omdat we de rekenkundige bewerkingen en konstanten juist in zuivere lambda-expressies wilden uitdrukken en de 0,1,+,-, etc. wilden wegwerken).

Groepering: mk-grpADT

Zonder veel problemen kunnen we ook groepering representeren met hogere orde funkties, zuivere lambda-expressies. Maar helaas, als de komponenten verschillende typen hebben, is de representatie met lambda-expressies niet SVPtypeerbaar.

Herinner je dat we alle operatorsymbolen als (prefix) funkties hebben geschreven, dus (grp e1 e2) voor <e1, e2>, (fst eg) voor eg.1 en (snd eg) voor eg.2. Het karakteristieke gebruik van een groep is het selekteren van een der komponenten. Dit motiveert enigszins de volgende representatiekeuze.

```
grp = "fn s. pas de selektiefunktie s toe op e1 en e2"
       als funktie van e1 en e2.
    = fn x y. fn s. s x y
    = fn x y z. z x y
fst = fn g. g (fn x y. x)
snd = fn g. g (fn x y. y)
```

Groepering werd bij natADT gebruikt in de formulering van de predecessor. Die kan nu dus weggewerkt worden. De twee komponenten waren beide expressies voor getallen, dus met eenzelfde type. Voor dergelijke groepen kan bovenstaande representatie aangepast worden tot een goed-SVPgetypeerde. Zonder verdere uitleg geven we nu de formulering daarvan. Merk op dat mk-grpADT een "geparameteriseerd (abstrakt) datatype" is, ofwel een funktie die als resultaat een datatype oplevert.

```
mk-grpADT : (z:tp -> <repr:tp {type van groepwaarden, genaamd: repr}
                     , (z->z->repr) {type van grp}
                     , (repr->z)    {type van fst}
                     , (repr->z)    {type van snd}
                     >)
     = fn z:tp. < {repr=} ((z->z->z)->z)
                , {grp =} fn x:z y:z. fn s:(z->z->z). s x y
                , {fst =} fn g:((z->z->z)->z). g (fn x:z y:z. x)
                , {snd =} fn g: ((z->z->z)->z). g (fn x:z y:z.y)
                >
```

Voor ieder type t kunnen we nu een aparte grp, fst, snd definiëren.

```
     df grpADTt : < repr:tp,(t->t->repr),(repr->t),(repr->t)>
                = (mk-grpADT t)
      ; grpreprt : tp == grpADTt.1
      ; grpt : (t->t->grpreprt) = grpADTt.2
      ; fstt : (grpreprt -> t) = grpADTt.3
      ; sndt : (grpreprt-> t) = grpADTt.4
```

Lijsten: mk-lijst ADT

Als laatste datatype dat we met zuivere lambdaexpressies representeren nemen we lijsten. Een karakteristieke bewerking op lijsten is de lijstiteratie: een gegeven funktie akkumuleren over de lijstelementen, te beginnen met een gegeven startwaarde a. Dus we kiezen voor lijsten ℓ:

```
     [ℓ] = fn f a. "de lijstiteratie van f en a over ℓ"
         = fn f a. "f(x1, f(x2, ..., f(xN,a)...))
                   met x1: x2: ...: xN : nil = ℓ
```

Dus

```
     [nil] = fn f a. a
```

en bijvoorbeeld

```
[e1: nil] = fn f a. f e1 a

[e1: e2: e3: nil] = fn f a. fe1 (f e2 (f e3 a))
```

De operatie op-kop-van (:) is net als de successor bij natADT eenvoudig uit te

drukken.

```
op-kop-van
  = fn e ℓ. "fn f a. itereer f met a over e:ℓ"
  = fn e ℓ. "fn f a. f(e, iteratie van f met a over ℓ)"
  = fn e ℓ. fn f a. f e (ℓ f a)
```

Inderdaad, er geldt nu dat (op-kop-van e1 [nil]) = [e1: nil], want

```
     op-kop-van e1 [nil]
       = (fn e ℓ. fn f a. f e (ℓ f a)) e1 [nil]
(app)  => fn f a. f e1 ([nil] f a )
       = fn f a. f e1 ((fn f a. a) f a)
(2*app)=> fn f a . f e1 a
       = [e1: nil]
```

en in het algemeen, (op-kop-van e1 [e2: ...: nil]) = [e1: e2: ... : nil].

Ook het nemen van de kop van een lijst gaat eenvoudig.

```
hd = fn ℓ. "itereer (fn k s. k) met een of andere a over ℓ"
   = fn ℓ. ℓ(fn k s. k) (---)
```

De expressie op de streepjes doet eigenlijk niet terzake, maar bepaalt bij deze representatie wel wat het resultaat van [hd nil] is. Desgewenst kunnen we de stokkende reduktie van (hd nil) modelleren met een niet eindigende reduktie. Reken zelf maar na dat (hd [e1: nil]) = e1 en dat (hd [e1: e2: ...: nil]) = e1

Voor het nemen van de staart passen we dezelfde techniek toe als bij de predecessor van natADT. Door de funktie f(x, <ℓ,--->)=<x:ℓ,ℓ > over de lijst te itereren met startwaarde <nil, --->, wordt in de linkerkomponent de lijst zelf gerekonstrueerd en in de rechterkomponent de staart ervan. Dus

```
tl = fn ℓ. ("itereer f met <[nil], ---> over ℓ").2
          met f x <y, ---> = <[x:y], y>
   = fn ℓ. (ℓ f <[nil], --->).2
          met f = fn x g. <op-kop-van x (g.1), g.1>
```

Desgewenst kunnen de groepbewerkingen weer geëlimineerd worden, en kan de expressie op de streepjes zo gekozen worden dat [tl nil] geen resultaat heeft.

We geven nu weer de aanpassing van bovenstaande representaties zo dat ze goed SVPgetypeerd zijn. Let wel, alle lijstelementen zullen eenzelfde type moe-

ten hebben. We nemen hd en tl niet op in het datatype, omdat zij toch in de andere zijn uit te drukken, zoals we hierboven zagen. Dat geeft een gebruiker tevens de gelegenheid zelf te kiezen wat het resultaat van [hd nil] en [tl nil] zal zijn. Net als bij mk-grpADT is mk-lijstADT een funktie die een lijstADT oplevert; ofwel mk-lijstADT is een geparameteriseerd datatype.

```
mk-lijstADT
: (z:tp -> < repr:tp {type der lijstrepresentaties, genaamd: repr}
           , repr {type van nil}
           , (z -> repr -> repr) {type van op-kop-van}
           , (repr -> t:tp ->(z->t->t)->t->t) {type lijstiteratie}
           >)
= fn z:tp. < {repr=}(t:tp ->(z->t->t)->t->t)
           , {nil=} fn t:tp f:(z->t->t) a:t. a
           , {op-kop-van} fn x:z ℓ:(t:tp->(z->t->t)->t->t)
                fn t:tp f:(z->t->t) a:t. f x (ℓ t f a)
           , {litar=} fn ℓ:(t:tp->(z->t->t)->t->t). ℓ          *)
           >
```

De naam litar staat voor: lijstiteratie associërend naar rechts. Het is niet toevallig dat het resultaattype van litar gelijk is aan de keuze voor repr: we hebben lijsten als lijstiteraties willen representeren. (In plaats van regel (*) kan ook: fn ℓ: (t':tp->(z->t'->t')->t'->t') t:tp f:(z->t->t) a:t.ℓ t f a). Let op de type parameter t van "litar".

Nabeschouwing: controlstructure versus data structure

De volgende opmerkingen geven mijn persoonlijk inzicht weer, of misschien beter het gebrek aan inzicht, dat ik nu d.d. mei 1983 ten aanzien van dit onderwerp heb. Ik houd me aanbevolen voor kommentaar, kritiek en terechtwijzingen.

Bij programmeertalen wordt gewoonlijk een onderscheid gemaakt tussen zg. control structures (besturingsstrukturen) en data structures (gegevensstrukturen). Onder data structures vallen bijvoorbeeld de getallen, waarheidswaarden, groepen en lijsten van data structures. Onder de control structures worden bijvoorbeeld sequentiële samenstelling (sequentie, de puntkomma), if then else, iteratie (rekursie) en "het procedure mechanisme" begrepen. Voor onze taal HOF zouden abstraktie en applikatie zeker tot de control structures gerekend moeten worden.

Wat we nu in deze paragraaf hebben gezien is dat data structures middels control structures geïmplementeerd kunnen worden. Maar hiertegen zou het volgende bezwaar ingebracht kunnen worden. Weliswaar levert de funktionele konkretisatie een semantisch equivalent (in een nog nader te definiëren zin) programma op, maar het is de vraag of zo'n programma ook equivalent is ten aanzien van (een nog nader te definiëren) kostenaspekt. En let wel, voor programmatuur in het algemeen is zowel korrektheid (semantiek) alsook prestatie (kosten) van belang. Wiskundig korrekte oplossingen zijn vaak eenvoudig te bedenken; bij de informatika zit het probleem vooral in het beheersen van de grootte-orde van de kosten: korrekte programma's die bij een invoer "ter grootte n" een evaluatieduur ter grootte van 2^n vereisen, terwijl een duur ter grootte van n ook haalbaar is (bij gelijkblijvende grootte van de benodigde opslagruimte), zijn praktisch gesproken waardeloos.

Een middel om de prestatie te beheersen is het uitwisselen van tijdsduur van de evaluatie tegen de grootte van de benodigde opslagruimte. En het is nu juist de vraag of zo'n uitwisseling nog mogelijk is wanneer we louter en alleen zuivere lambda-expressies mogen of willen gebruiken.

Een algemeen antwoord op deze vraag ken ik d.d. mei 1983 nog niet. Maar wel is het volgende gebleken; tot mijn grote verrassing. Een geschikt gekozen funktionele representatie van lijsten gedraagt zich in het implementatiemodel van de graafreduktie (hoofdstuk 3) qua grootte-orde van tijdsduur en opslagruimte precies zo als de direkte voor de hand liggende implementatie van de operaties :, hd, tl, nil en eqnil. In Oefeningen 6.16 en 6.17 wordt aangetoond dat ook onbegrensde en begrensde arithmetiek met de gebruikelijke kosten in zuivere lambda-expressies geïmplementeerd kunnen worden.

We zijn geneigd hieruit de volgende konklusie te trekken. Er is geen fundamenteel onderscheid tussen data structures en control structures; die beide begrippen blijken bij nadere beschouwing niet goed gedefinieerd. In hoeverre er in een bijzondere kontekst wel een zinvol onderscheid gemaakt kan worden, valt nog te bezien.

Oefeningen

6.1! Ga na dat in de ongetypeerde versies ([succ] [false]) een zinvolle expressie is, equivalent met [1], maar dat in de getypeerde versies de ermee overeenkomende expressie niet goed SVP-getypeerd is.

6.2 Geef een representatie voor primreck (bij vaste k). Wenk: maak gebruik van de representatie van getallen.

6.3 Geeft het SVPtype dat je voor primreck zou willen hebben (voor vaste k). Verifieer dat de in 6.2 gegeven representatie dat type heeft.

6.4 Definieer optelling, vermenigvuldiging, gehele deling, machtsverheffing enzovoorts met behulp van primitieve rekursie (en kompositie van funkties en al eerder gedefinieerde (rekenkundige) funkties).

6.5! Werk de volgende alternatieve representatie van de successor uit. succ n = (fn f x. "n-voudige herhaling van f op fx"). Heeft deze het type zoals dat in de definitie van natADT'geponeerd is?

6.6! Herschrijf de representatie voor de predecessor zo dat alleen de letters (variabelen) x, y, z gebruikt worden en voorts alleen lambdaexpressies en geen groepen.

6.7 Geef een representatie voor groepen met k komponenten (voor vaste k). Geef ook de getypeerde aanpassing ervan.

6.8 (voor de liefhebbers) Geef een funktie mk-grpADT" die allereerst een k als argument verwacht en dan het datatype voor groepen met k komponenten aflevert. Geef ook de SVPgetypeerde aanpassing ervan. Wenk: zie aanhangsel A van de syllabus over Typering.

6.9 (alleen voor de liefhebbers) Geef een funktie primrec" die allereerst een k als argument verwacht, en dan ongeveer net zo werkt als primreck; in plaats van argumentlijsten komen er enkelvoudige argumenten, namelijk groepen met k komponenten. Geef ook de SVP-getypeerde aanpassing ervan. Wenk: zie Oefening 6.8.

6.10! Geef SVPgetypeerde definities van (i) append, (ii) map, (iii) eqnil, (iv) hd, (v) tl enzovoorts. Deze definities moeten van de SVP-getypeerde definitie van mk-lijstADT gebruik maken; ze moeten niet in mk-lijstADT worden opgenomen maar erbuiten staan. Wenk: druk append, map etc. direkt in litar uit, eerst ongetypeerd en voeg daarna type parameters en argumenten toe en typen bij alle

identifier-introdukties.

6.11 (voor de liefhebbers) Geef representaties in zuivere lambdaexpressies voor bomen, enumeraties (zoals in Pascal), onderscheiden vereniging, enzovoorts enzovoorts. Geef ook de SVPgetypeerde aanpassingen ervan.

6.12! Geef een representatie voor lijsten met zuivere lambdaexpressies, maar zo dat de naar links associërende lijstiteratie in feite ter representatie van lijsten wordt gekozen.

6.13! Geef definiërende expressies voor boolADT', natADT', mklijstADT enzovoort, waarin niet zuivere lambdaexpressies worden gebruikt maar de al aanwezige HOF expressies zoals 0,1,2,..(e+e), .., true, false, (if e then e0 else e1), nil, (eh:et), enzovoorts. Dit moet dus heel makkelijk zijn; waar het nu om gaat is dat ook deze expressies aan de al gegeven typen van boolADT' etc. voldoen. Wenk: geef eerst de typering voor deze HOF expressievormen.

6.14! Representeer een getal n als een lijst ter lengte n (van overigens niet ter zake doende elementen). Geef de bijbehorende representaties voor successor en predecessor enzovoorts. (Als je nu lijsten met lambda-expressies representeert, heb je dat indirekt ook voor getallen gedaan.) Geef ook de SVPgetypeerde aanpassingen. Voldoen de aldus gevonden representaties aan het al voor natADT' geponeerde type?

6.15! Bij de representatie van groepen hebben we grp, fst en snd zo gekozen dat (grp e1 e2) de representatie is van <e1, e2>, (fst eg) van eg.1 en (snd eg) van eg.2. Kies nu een representatie zo dat (grp' e1 e2) de representatie is van <e1,e2>, (eg fst') van eg.1 en (eg snd') van eg.2.

6.16 (alleen voor de liefhebbers) Werk de volgende implementatie voor natuurlijke getallen uit, zodat de tijdsduur voor de optelling van m met n evenredig is met logm + logn en voor de successor van n evenredig met log n. Kies

[n] = fn f g x. (..(f....(g... x)..)..)

binaire schrijfwijze voor n met f voor 0 en g voor 1 en het minst signifikante cijfer links.

Dus ([n] (fn x. 2*x+0) (fn x. 2*x+1) 0) => n. Definieer nu funkties db, dbl, even, half zo dat

db [n] => [2n]

dbl [n] => [2n+1]
half [n] => [n/2] voor even n, [(n-1)/2] voor oneven n
even [n] => true voor even n, false voor oneven n

(Pas voor 'half' dezelfde techniek toe als voor predecessor en voor tl.) Definieer voorts

succ = fn n. if even n then dbl (half n) else (db n)
eqzero = fn n. n (fn x. true) (fn x. false) true

Voor de predecessor kun je als volgt te werk gaan. Zij n=m*2**k, met m oneven (n>0); dus k is het aantal nullen waarmee de binaire representatie van n eindigt. Dan is n-1 = f^k(m div 2) met f x = 1 + 2 * x.
Dus gegeven [n](=[m*2^k]), bouw dan < [m div 2], fn f x. f^k x> op, en vorm hieruit [n-1].

Oefening 6.17 Geef een funktionele representatie voor natuurlijke getallen 0..2**k (voor vaste k) en de bewerkingen daarop, zo dat de optelling van m en n een tijd (en ruimte) kost onafhankelijk van m en n (maar uiteraard wel afhankelijk van k). Wenk: gebruik in eerste instantie k-voudig geneste groepen of groepen met k komponenten ter binaire representatie van de getallen uit 0..2**k.

Par. 6.4 Turingmachines funktioneel gerepresenteerd

In deze paragraaf geven we een rechtstreekse omzetting van Turingmachine-programma's naar zuivere lambda-expressies. Dit doen we voornamelijk om nog eens een niet-triviaal gebruik van funkties te laten zien. Als bijprodukt verkrijgen we ook een middel om de onbeslisbaarheid van allerlei taaleigenschappen aan te tonen, niet alleen van HOF maar ook van Algol 68 en Pascal, en nogmaals een bewijs dat alle berekenbare funkties in lambda-expressies gerepresenteerd kunnen worden. Het opmerkelijke aan de omzetting die wij hier presenteren is dat het resulterende programma, na wat syntaktische aanpassingen, een korrekt Algol 68 programma is (ook Pascal, als Pascal rekursieve typen voor funkties had gekend), waarbij iedere procedure-romp louter de aanroep is van een andere procedure met geschikte procedures als argumenten. Wie niet geïnteresseerd is in de details kan nu verder lezen bij de konklusie.

* * *

Voor Turingmachines en TM-programma's gaan we uit van de definities van (Verbeek 1983). Ter herinnering sommen we hier nog wat begrippen op. De geheugens bestaan uit paren <|sa|, sat> met s,t:Z* en a:Z; Z is het bandalfabet en z is het blanko symbool. We laten ook b variëren over Z, b:Z. De operatie- en testnamen staan, tesamen met hun interpretatie, in onderstaande tabel.

operatie- en testnaam	interpretatie: argument	interpretatie: resultaat
schrijf b	<|sa|, sat>	<|sb|, sbt>
links	<|a|, at>	<|z|, zat>
	<|sab|, sabt>	<|sa|, sabt>
rechts	<|s|, sat>	<|sa|, sat>
	<|sa|, sa>	<|saz|, saz>
a?	<|sb|, sbt>	waar voor b=a
		onwaar voor b≠a

Ter vereenvoudiging hebben we de operatienaam wis, en de testnamen le? en re? weggelaten. Door toevoegen van markeringssymbolen helemaal links en rechts op de band (en door aanpassing van de uitvoerkodering) zijn zij gemakkelijk met de andere operaties en testen te simuleren; zie voorts Oefening 6.20.

We zullen zo straks, onafhankelijk van het te simuleren TM-programma, voor

iedere operatienaam oper en testnaam test en ieder geheugen g een funktie foper, ftest en een representatie voor g kiezen. Een willekeurig TM-programma P zetten we dan als volgt om in een lambda-expressie fP:

```
P = {START: NAAR L0,              fP = (dfrec
     ----                              ----
    Lx: DOE oper NAAR L',                , fLx = foper fL'
     ----                                 ----
    Ly: ALS test NAAR L' ANDERS L",      , fLy = ftest fL' fL"
     ----                                 ----
    Lz: STOP}                            , fLz = fSTOP
                                         . fL0
                                         )
```

De funkties fL zijn wederzijds rekursief gedefinieerd. (Dat kan op zich ook in zuivere lambda-expressies worden uitgedrukt). Zowel fP alsook iedere fL kan nog op een geheugenrepresentatie worden toegepast. Willekeurige geheugens , <|sa|, sat> zullen we representeren door een drietal funkties [s]', [a] en [t]" en we laten fs, fa en ft dergelijke funkties aanduiden. Als we zo'n geheugenrepresentatie expliciet als parameter aangeven, hebben we in fP nergens van funkties als resultaat gebruik gemaakt (vergelijk Algol 68 en Pascal):

```
fP = fn (fs, fa, ft).
       (dfrec
        ----
     , fLy = fn (fs, fa, ft). ftest (fL', fL", fs, fa, ft)
        ----
     . fL0 (fs, fa, ft)
       )
```

Door foper, ftest en de geheugenrepresentatie geschikt te kiezen, zullen we ervoor zorgen dat de lambda-expressie fP het TM-programma P in de volgende zin representeert:

```
<L, <|sa|, sat>> ⊢ < L', <|s'a'|, s'a't'>>
impliceert
fL ([s]', [a], [t]") => fL' ([s']', [a'], [t']")
```

Dus door een aanroep van fL wordt eerst een stap van de berekening gesimuleerd, en vervolgens nog (door de aanroep fL') de gehele verdere voortzetting van de berekening tot en met een eventuele eindkonfiguratie <STOP, <|s"a"|, s"a"t">>, die overeenkomt met fSTOP ([s"]', [a"], [t"]"). De funktie fSTOP is nog vrij te

kiezen; kiezen we bijvoorbeeld fSTOP = (fn (x, y, z). (fn x. x)), dan termineert fP precies wanneer P termineert (maar wel met ongerelateerde resultaten).

In volgorde van opklimmende moeilijkheid zullen we nu fschrijfb, dan fa? en [a], en tenslotte flinks en [s]' (en analoog frechts en [t]") definiëren. De definitie van fschrijfb is het gemakkelijkst, omdat die operatie onafhankelijk is van de bandinhoud (alleen de positie van de lees/schrijfkop speelt een rol):

fschrijfb = fn (fL, fs, fa, ft). fL (fs, [b], ft)

Redelijk eenvoudig is ook de definitie van fa? en [a] (voor willekeurige a:Z). We representeren een symbool a door een keuzefunktie die precies het goede alternatief kiest van de alternatieven die hem door een fb? worden geboden. Neem een of andere volgorde in het alfabet, zeg Z = <a1, a2, ..., aN>. Dan

[ai] = fn (x1,..., xN). xi
[fai?] = fn fL' fL" fs fa ft. fa (fL", ..., fL") fs fa ft
maar fL' op de i-de plaats

We kunnen dit ook uitdrukken zonder funkties als resultaat te gebruiken:

[ai] = fn (x1,..., xN, fs, fa, ft). xi (fs, fa, ft)
[fai?] = fn (fL', fL", fs, fa, ft). fa (fl", ..., fL", fs, fa, ft)
maar fL' op de i-de plaats

Tenslotte dan de definitie van flinks en [s]' (en analoog frechts en [t]"); Pas op dat U niet in de vicieuze cirkels verstrikt raakt! Zoals we al vaker gedaan hebben, representeren we ook nu een object, in casu: het linkerbanddeel s, door zijn karakteristieke "control structure", en dat is: de funktie die het eigenlijke werk van flinks doet, d.w.z. de simulatie vanaf het moment dat de lees-/schrijfkop op dat banddeel komt. We parameteriseren die simulatie met de toekomstige simulatie die begint met het verplaatsen van de lees/schrijfkop naar zijn rechterbanddeel. Dus, met deze keuze kunnen we flinks alvast definiëren:

flinks = fn (fL, fs, fa, ft). fs (G, fL)
met G = fn (F, fL). fL (F, fa, ft)

Nogmaals, de funktie G is de representatie voor het rechterbanddeel zoals dat zal zijn wanneer de lees/schrijfkop daarop komt vanuit de simulatie op het lin-

kerdeel fs. Dus G is gedefinieerd in termen van fa en ft (parameters van flinks), en is op zijn beurt weer afhankelijk van zo'n representatie F voor het linkerbanddeel (zoals dat zal zijn wanneer de lees/schrijfkop t.z.t. daar weer naar toe gaat). Om fs (en F en [s]') en ft (en G en [t]") te typeren, hebben we rekursieve typen nodig! Bijvoorbeeld, definieer

```
type eindresult = "type van fSTOP  ([s]', [a], [t]")"
  , symbool     = (label, ...., label, links, symbool, rechts -> eindresult)
  , label       = (links, symbool, rechts -> eindresult)
  , links       = (rechts, label -> eindresult)
  , rechts      = (links, label -> eindresult)
```

Dan geldt

```
iedere [a]: symbool
iedere fs, F, [s]': links
iedere ft, G, [t]": rechts
```

en voorts

```
foper: (label, links, symbool, rechts -> eindresult)
ftest: (label, label, links, symbool, rechts -> eindresult)
fL, fP: label.
```

Ook al simuleert een aanroep van [s]' de volledige (on)eindige berekening, we kunnen hem toch met induktie naar de lengte van s definiëren en dus volledig uitschrijven, omdat alle informatie over zijn rechterbanddeel als parameter wordt verwacht. Inderdaad,

```
[leeg]' rec= fn (G, fL). fL ([leeg]', [z], G)
[sa]'      = fn (G, fL). fL ([s]', [a], G)
```

We hoeven [sa]' en [at]" niet zelf uit te schrijven, als we als begingeheugen <1, z> nemen. Dan immers wordt fP toegepast op ([leeg]', [z], [leeg]"), en worden de representaties voor niet-triviale geheugens vanzelf gekreëerd. Met name [leeg]' en [leeg]" kreëren nieuwe bandcellen met als inhoud het blanko symbool z.

Hiermee is de funktionele representatie van willekeurig TM-programma voltooid. We merken op dat de lambda-expressie fP zonder moeite ook in Algol 68 genoteerd kan worden en, behoudens de rekursieve typedefinities, ook in Pascal. Zonodig kun je de lambda-expressie-als-argument, zoals G in flinks, als een lokale funktie definiëren. Een andere observatie is dat iedere funktie een andere

-met geschikte argumenten - aanroept totdat uiteindelijk ooit (of nooit) fSTOP wordt aangeroepen. Pas wanneer de aanroep van fSTOP voltooid is, is willekeurige al eerder aangeroepen funktie ook voltooid. Dus in het implementatiemodel van Hoofdstuk 4 is op het moment van aanroep van fSTOP de gehele berekening (ja, alle stappen ervan) nog in gekodeerde vorm in sR en sT aanwezig.

* * *

Konklusie

De funktionele representatie van TM-programma's is een handig hulpmiddel voor het bewijzen of weerleggen van allerlei taaleigenschappen. Allereerst deze; de terminatie van TM-programma's is onbeslisbaar, en dus is ook de terminatie van HOF-programma's onbeslisbaar op grond van bovenstaande konstruktie van fP uit P. (De terminatie van TM-programma's is gereduceerd tot de terminatie van HOF-expressies). Ten tweede, het is nu ook duidelijk dat de bereikbaarheid van willekeurige deelexpressie of funktieromp onbeslisbaar is; waarbij we een deel bereikbaar noemen als het ooit gedurende de evaluatie ter reduktie wordt genomen. Immers, zou de bereikbaarheid beslisbaar zijn dan zou ook de bereikbaarheid van fSTOP binnen fP beslisbaar zijn en daarmee ook de terminatie van P, voor willekeurig TM-programma P. Door een geschikte keuze van de romp van fSTOP zien we nu ook onmiddellijk in dat het onbeslisbaar is of ooit een expressie zoals 2+true ter evaluatie wordt genomen (type-fout, stokken van de berekening). Enzovoorts, enzovoorts.

Bovenstaande onbeslisbaarheidseigenschappen kunnen we nog aanzienlijk versterken. Beschouw bijvoorbeeld de bereikbaarheidseigenschap. In plaats van de feitelijke bereikbaarheid kunnen we ook een formele ofwel schematische bereikbaarheid beschouwen. Ruwweg gezegd zien we dan af van de feitelijke uitkomst van expressies voor getallen en waarheidswaarden, en verklaren we beide takken van een if-expressie formeel bereikbaar. Dus in

if 0 eqn 0 then 1 else (2+true)

is de else-tak wel formeel maar niet feitelijk bereikbaar. We kunnen net zo definiëren dat die if-expressie formeel stokt (hoewel niet feitelijk), en een formele dynamische typefout heeft (maar geen feitelijke dynamische typefout).

Het leuke aan fP is dat de feitelijke en formele bereikbaarheid van fSTOP samenvallen: in fP komen helemaal geen expressies voor getallen of waarheidswaarden of if-expressies voor! Via de onbeslisbaarheid van de formele bereik-

baarheid van fSTOP kan ook de onbeslisbaarheid van veel andere "formele eigenschappen" aangetoond worden.

De moraal luidt dus als volgt. Wellicht wekt het geen verbazing dat een compiler sommige eigenschappen niet in het algemeen kan verifiëren; immers, een TM-programma kan al op eenvoudige wijze met behulp van lijsten en getallen gesimuleerd worden. Maar misschien had je nog verwacht dat "formele versies" van die eigenschappen wel door een compiler geverifieerd zouden kunnen worden. Dat blijkt dus veelal niet het geval te zijn. Dit geldt niet alleen voor HOF, maar ook voor Algol 68 en voor die versies van Pascal waarbij van funktionele parameters het type niet of niet volledig aangegeven hoeft worden (zoals in Algol 60).

Belangrijke onderwerpen uit Hoofdstuk 6

lambda-expressies: de essentie van hogere programmeertalen
datatypen worden gerepresenteerd als hogere orde funkties
SVP-typering ter voorkoming van onbedoeld gebruik
Turingmachineprogramma's gerepresenteerd als funkties (zonder funkties als resultaten)
onbeslisbaarheid van
    bereikbaarheid, terminatie, dynamisch type fout, ... en de formele versies daarvan

Literatuur bij Hoofdstuk 6.

(Reynolds 1975) en (Meertens 1978) geven praktische voorbeelden van procedurele (in plaats van funktionele) representaties van data. (Burge 1975) geeft een aantal praktische en overtuigende voorbeelden van het gebruik van hogere orde funkties. (Desain 1983) geeft een informele methode om induktief gedefinieerde objekten (zoals natuurlijke getallen en bomen) zo met funkties te representeren dat de rekursie-expressie (en zijn niet-typeerbare funktionele simulatie) niet meer nodig is. Uiteraard komt dit onderwerp ook uitgebreid aan de orde in de Lambda Calculus, (Barendregt 1981, 1984). De uitdrukkingskracht van de 2de orde getypeerde lambda-calculus wordt behandeld in (Fortune et al 1983).

Alternatieve funktionele simulaties van Turingmachines en aanverwante mechanismen komen aan bod in (Langmaack 1973, 1973b, 1974). (Matchey & Young 1978) geven een funktionele simulatie in Pascal van Turingmachines met een eindige band en tonen zo de Komplexitietsklasse van een aantal Pascal-taaleigenschappen aan.

Oefeningen

6.18 Geef de reductie van fP voor de volgende TM-programma's P, met initieel geheugen <1, z>.

P: START NAAR L; L: DOE links NAAR L; L': STOP
P: START NAAR L0; ... L(i-1): DOE links NAAR Li (0<i<n)...; Ln: STOP

Schets ook de evaluatie volgens het implementatiemodel van Hoofdstuk 4.

6.19! Geef de definities van frechts en [t]". Bewijs voor alle mogelijke gevallen van de instruktie die gelabeld is met L, dat een stap vanuit <L, <|sa|, sat>> korrekt gesimuleerd wordt.

6.20 Pas de omzetting van P naar fP aan zo dat ook de operatie wis en de tests le? en re? in TM-programma's zijn toegelaten. Ter herinnering volgt hier hun interpretatie.

	argument	resultaat
wis	<|a|, abt>	<|b|, bt>
	<|sab|, sab>	<|sa|, sa>
	<|sab|, sabct>	<|sab|, sabct>
	<|a|, a>	<|z|, z>
le?	<|a|, at>	waar
	<|sab|, sabt>	onwaar
re?	<|sa|, sa>	waar
	<|sa|, sabt>	onwaar

Wenk. Representeer een geheugen <|sa|, sat> als een vijftal [s]', [s]?', [a], [t]?", [t]", waarbij [s]', [a] en [t]" zijn als voorheen en [s]?' "het eigenlijke werk van le?" doet (en analoog voor [t]?").

6.21 Schrijf fP uit in korrekt Algol 68. Doe het ook in Pascal, waarbij U als het type van een funktie als parameter alleen het woord function hoeft te vermelden.

6.22 Geef een funktionele representatie van M-programma's, waarbij M staat voor de stapelmachine of registermachine of Postmachine enz.

6.23! Toon aan dat in Algol 68 de formele bereikbaarheid van het gebruik van een niet geïnitialiseerde variabele onbeslisbaar is. Evenzo voor de formele bereikbaarheid van een dereferencing van nil.

HOOFDSTUK 7

ASSIGNMENT, DE PROGRAMMEERTAAL ASS

In dit hoofdstuk wijden we enige woorden aan het begrip assignment en assigneerbare variabele, en definiëren we de taal ASS als een uitbreiding van HOF met assignment. We gaan ervan uit dat de lezer bedreven is in een taal zoals Algol 68 of Pascal, zodat we in Par. 7.1 en 7.2 de semantiek van assignment bekend kunnen veronderstellen. We laten zien dat assignment en assigneerbare variabelen een abstraktie vormen van het machinemodel dat aan de taal ten grondslag ligt, en dat er verscheidene abstrakties mogelijk zijn.

Met de introduktie van assignment en assigneerbare variabelen gaan we van de beschrijvende (= funktionele, applikatieve) naar de gebiedende (= imperatieve, procedurele) talen. Dit gaat gepaard met een verlies van eenvoud van de taal, met name in de semantiek en pragmatiek. Dit, en de motivatie voor deze overgang, zullen we kort toelichten.

In volgende hoofdstukken komen nog andere aspekten aan bod die vooral bij gebiedende talen een rol spelen.

Par. 7.1 Waarom assignment -- of waarom niet?

Historisch gezien waren de gebiedende programmeertalen er eerder dan de beschrijvende. De Von Neumann machinearchitektuur heeft de assignment als basisprincipe, en de eerste "programmeertalen" waren louter "machinebesturingstalen". Intussen is genoegzaam bekend dat een machine(besturings)taal niet de meest ideale vorm is om algoritmen vast te leggen. Inderdaad, voor het uitdrukken van algoritmen voldoet een beschrijvende taal, zoals HOF, uitstekend en is enige kennis van het machinemodel helemaal niet nodig. (Feitelijk kan alle Konstruktieve Wiskunde worden bedreven in een getypeerde variant van de zuivere lambda-expressies; op die typering is al gewezen in Deel 4.3 van de syllabus over Typering.)

De reden dat assignment desondanks toch wordt opgenomen in de taal, is dat daardoor eventuele machinemogelijkheden beter benut kunnen worden dan met een beschrijvende taal kan, en dat het benutten van de mogelijkheden een essentiële verbetering geeft in de doelmatigheid van de evaluatie. De assignment laat een doelmatig gebruik van de opslagruimte van de machine toe. De plaatselijke bijstelling (selektieve updating) zoals in a[i]:=expr buit bovendien de machinemo-

gelijkheid van indicering uit: deze operatie kan in konstante tijdsduur, dus onafhankelijk van i, geschieden. Het is overigens wel de vraag of men zich bij het ontwerp van programmmeertalen op de Von Neumann architektuur moet richten, of dat juist het ontwerp van machinearchitekturen moet aansluiten bij de gewenste taal! Deze laatste aanpak wordt op het ogenblik wereldwijd onderzocht.

Een andere reden om machinemogelijkheden in de taal op te nemen is dat middels een programma delen van die machine, of randapparatuur, moeten worden aangestuurd. Maar ook dan hoeven die mogelijkheden niet in de hele taal door te schemeren; de nodige algoritmen kunnen in principe best zonder imperatieve kenmerken geformuleerd worden. Zie bijvoorbeeld (Henderson 1982 a, b) voor een beschrijvende formulering van een beheerssysteem en een "plot-programma". We laten deze reden verder buiten beschouwing.

Door de aanwezigheid van assignment verliest een overigens beschrijvende programmeertaal een stuk potentiële eenvoud. We zullen dat hieronder in (a)..(c) toelichten.

(a) Allereerst merken we op dat de evaluatie van een expressie niet alleen een waarde oplevert, maar in het algemeen ook een effekt heeft op de opgeslagen gegevens. Op zich is daar niets op tegen. Maar uiteraard hebben expressies die een waarde-plus-effekt aanduiden een ingewikkelder semantiek; met gevolgen voor een formele definitie en begrijpelijkheid enzovoorts. Bij onbeperkt gebruik van assigneerbare variabelen zijn sommige effekten zeer moeilijk uit de tekst te achterhalen; deze effekten worden wel defekten, of kwaadaardige neveneffekten genoemd.

Programmeertalen waarbij kwaadaardige neveneffekten onmogelijk zijn, zijn nog onbevredigend voor grootschalige programmatuur. Het voornaamste bezwaar geldt de beperkingen die aan argumenten en globalen van procedures en funkties worden gesteld. De goedaardige neveneffekten worden daardoor ook onmogelijk gemaakt. Euclid (Lampson 1977, Guttag 1978) is zo'n taal. Deze is daadwerkelijk gebruikt voor systeemprogrammatuur. Dus de hoop is gerechtvaardigd dat neveneffekten nog wel eens op bevredigende manier getemd kunnen worden.

(b) Door de aanwezigheid van de assignment wordt de manier om over de semantiek of korrektheid van expressies te redeneren een stuk ingewikkelder, en met name gaat die manier afwijken van wat al eeuwenlang in de wiskunde wordt gedaan. In de wiskunde worden expressies zelf getransformeerd, waarbij steeds gelijke (d.w.z. semantisch gelijkwaardige) expressies door elkaar vervangen worden. Dit staat bekend als het axioma van: de substitutiviteit van de gelijkheid, (zie

Hoofdstuk 5). Bij imperatieve talen gaat het zo: allereerst wordt er een beschrijvende assertie taal ontworpen, en de redenering vindt dan plaats door de assertie-expressies te transformeren. Bijvoorbeeld:

> opdat na x:=expr de assertie x+y=z geldt, moet er tevoren de assertie (x+y=z)[x/expr] gelden, (i.e. expr+y=z).

Deze vorm van redeneren faalt bovendien als expr neveneffekten heeft, of wanneer y en z dezelfde variabele als x aanduiden (ze heten dan aliassen). Het is mogelijk om de assertie-taal aan deze verschijnselen aan te passen, maar het werken ermee wordt dan wel een stuk bemoeilijkt zo niet praktisch ondoenlijk. (Voorbeelden hiervan zijn de zg. Intensional Logic (Janssen 1983), en de axiomatisering van de zg. interferentie van programmadelen (Reynolds 1981)). Anderzijds kun je ook de programmeertaal aanpassen zodat bovenstaande redeneervorm wel klopt; dit komt neer op het uitbannen van verborgen neveneffekten (ook de goedaardige) en aliassen.

(c) Door de aanwezigheid van assignment is de volgorde van evaluatie van subexpressies van groot belang. De enige zinvolle keus lijkt de kompositionele strategie te zijn (eventueel met de by-name of by-need strategie voor argumenten), dus: van links naar rechts en onderdelen voor de omvattende gehelen. Maar een abstrakte of wiskundige oplossing bestaat vaak uit de definiëring van een stel hulpgrootheden, in termen waarvan het eindantwoord kan worden uitgedrukt. Als die hulpgrootheden onafhankelijk zijn van elkaar, hoeven de berekeningen ook niet in de tijd geordend te worden. Toch moet de programmeur in een gebiedende taal (met de kompositionele strategie) een totale ordening uitdrukken en dus geweldig overspecificeren; met alle nadelen vandien.

(In een beschrijvende taal doet de volgorde van evalueren er nauwelijks toe wat de semantiek betreft. Niet alleen wordt gedwongen overspecifikatie voorkomen, maar is ook parallelle evaluatie van subexpressies mogelijk. Deze mate van parallelliteit is ongetwijfeld veel groter dan ooit in een gebiedende taal middels uitdrukkelijke programmering mogelijk is; (denk aan de kollaterale evaluatie van Algol 68 of echte parallelle processen met synchronisatie ertussen). Het is dit parallellisme dat de mogelijkheden van VLSI en dataflow-machines misschien ten volle kan benutten).

Hiermee besluiten we de opsomming van de voor- en nadelen van assignment. Het dilemma waarvoor een taalontwerper staat is enerzijds voldoende machine-aspekten te laten doorschemeren in de taal om de machinemogelijkheden te laten benutten, en anderzijds van zoveel mogelijk machineaspecten te abstraheren om zodoende een goede taal voor het uitdrukken van algoritmen over te houden. Bij

bijna alle hedendaagse programmeertalen heeft het eerste de doorslag gegeven.

Par. 7.2 Het machinemodel en abstrakties ervan

We schetsen heel informeel dat deel van het machinemodel dat met assignment en assigneerbare variabelen te maken heeft. Vervolgens bespreken we verscheidene aspekten van assigneerbare variabelen waarin talen van elkaar kunnen verschillen.

Een machine heeft een opslagruimte (geheugen) waarin waarden, gekodeerd als bitrijen, kunnen worden opgeslagen. De opslagruimte is verdeeld in cellen; iedere cel heeft een uniek adres en kan precies een bitrij van een vaste gegeven lengte bevatten. Voor "grote" waarden zijn dus verscheidene cellen nodig. Een waarde kan in een cel geschreven worden; daarbij gaat de oude inhoud van de cel verloren, overschrijven is dus een betere term. De inhoud van een cel kan ook weer uitgelezen worden.

Het begrip assigneerbare variabele is een abstraktie van cel: een assigneerbare variabele kan een waarde bevatten waarbij in meer of mindere mate wordt afgezien van de beperking op de lengte (van de bitrij-kodering). Assignment (:=) is een abstraktie van het schrijven van een waarde in een cel: middels een assignment kunnen bijvoorbeeld verscheidene waarden worden geschreven in evenzovele assigneerbare variabelen, denk aan x,y:=3,4 en aan a:=b voor arrays a en b. De overeenkomstige abstraktie van het uitlezen van de inhoud van een cel zullen we eveneens uitlezen noemen en met een postfix vraagteken noteren; in Algol 68 kringen wordt deze operatie wel dereferencen genoemd.

Merk op dat bovenstaand begrip assigneerbare variabele niets te maken heeft met het begrip variabele zoals wij dat tot nu toe gebruikt hebben. Het één is een abstraktie voor een cel, het ander is louter een hulpmiddel om de syntaktische bindingsrelatie lineair (zonder pijlen) te kunnen weergeven. Deze begrippen moeten niet met elkaar verward worden! Traditioneel wordt bij programmeertalen het begrip variabele anders gebruikt: een variabele is dan per definitie een expressie die een assigneerbare variabele aanduidt. (Wij hebben ons begrip van variabele overgenomen uit de wiskunde en logika.)

In het vervolg van deze paragraaf zullen we "assigneerbare variabele" afkorten tot "cel"; deze afkorting is minder misleidend dan de afkorting tot "variabele".

We zullen nu verscheidene aspekten van cellen beknopt bespreken. Die aspek-

ten zijn: cellen versus waarden, de levensduur van cellen, aanduidbare cellen en de aanduiding van cellen.

Cellen versus waarden

Tot nu toe hebben we in HOF nauwelijks enig onderscheid aangebracht in de verzameling W van waarden. Alleen bij het LIFO-implementatiemodel zagen we dat funktie-bevattende waarden niet als funktieresultaat zouden moeten optreden om een echt LIFO beheer over de opslagruimte mogelijk te maken. Overigens zijn er geen klassen van waarden onderscheiden. Bij de meeste programmeertalen, vooral de gebiedende, is het wel gebruikelijk een klassifikatie aan te brengen. Met name de klassen van waarden die als volgt kunnen optreden:

- als rechterlid in een definitie
- als argument in een applikatie
- als resultaat van een funktie
- als komponent van een groep of lijst
- als bestemming van een assignment
- als bron van een assignment

(De laatste klasse wordt wel de "storable" waarden genoemd, wij zullen het de schrijfbare waarden noemen). Ten aanzien van cellen wordt het karakter van een programmeertaal voor een groot deel door de volgende gevallen bepaald.

1. Cellen kunnen niet als waarden optreden, behalve als bestemming van een assignment.
2. Sommige cellen kunnen wel in sommige van bovenstaande waardeklassen optreden.
3. Alle cellen kunnen in al die waardeklassen optreden, i.e. cellen zijn gewone waarden.

Indien cellen en adressen niet geïdentificeerd worden (daarover meer in Par. 3), doen deze gevallen zich ook voor ten aanzien van adressen.

In geval 1 kunnen cellen als bestemming van een assignment voorkomen en als operand van de uitlees-operatie, maar niet als argument van een funktie of procedure. Zulks is bijna alleen het geval in onderwijskundige minitalen, bijv. (Dijkstra 1976). In geval 2 kan er enigszins met cellen "gerekend" worden. In Pascal kunnen cellen als argument optreden (de var-parameters), maar niet of nauwelijks als rechterlid van een definitie. Bekijken we ook de pointers, dan zijn de mogelijkheden om indirekt met cellen te rekenen al groter. Alle in Algol 68 aanduid-

bare cellen kunnen precies als de overige waarden gebruikt worden: de mode ref M heeft geen speciale status. Toch zijn er nog cellen die helemaal niet met een Pascal of Algol 68 tekst aangeduid kunnen worden, laat staan als waarde gebruikt kunnen worden; bijvoorbeeld de cel met adres 4711 of zoiets. Geval 3 doet zich dan ook alleen maar voor bij machine-kode en assembleertalen.

Levensduur van cellen

De opslagruimte is niet onuitputtelijk. Daarom moet er tijdens een berekening (evaluatie of exekutie van een programma) op de een of andere manier een beheer gevoerd worden over die opslagruimte. Bij programmering in machine-kode is dat geheel de taak van de programmeur. In "hogere" programmeertalen wordt die taak gedeeltelijk door de taalontwerper overgenomen: middels de expressievormen wordt een abstraktie van de opslagruimte voorgeschoteld. Daarin zijn er "in gebruik zijnde" cellen en "niet in gebruik zijnde", en kunnen cellen "in gebruik genomen" dan wel "weer vrij gegeven" worden. In termen van deze attributen wordt het beheer over de opslagruimte dan min of meer automatisch gevoerd. Het (tijds)interval van de programmaexekutie tussen de in gebruikname en de daaropvolgende vrijgave van een cel heet de levensduur of extent (engels voor uitgebreidheid, uitgestrektheid, omvang) van (dat gebruik van) die cel. Het begrip assigneerbare variabele duidt nu zo'n tijdelijk "levende" cel aan; in het vervolg weer afgekort tot cel.

Ten aanzien van dit beheersaspekt kunnen we cellen als volgt klassificeren.

1. statische of permanente levensduur: de ingebruikname en vrijgave vinden plaats voor en na afloop van de programma-exekutie.
2. semistatisch: de vrijgave vindt plaats bij voltooiing van de evaluatie van een speciale expressievorm (meestal block, procedure- of funktieromp) gedurende welke, of bij aanvang waarvan, de in gebruikname heeft plaatsgevonden. In Algol 68 brengt loc dergelijke cellen voort, en in Pascal doen de var-deklaraties en value-parameters dat.
3. semidynamisch: de levensduur strekt zich uit vanaf de geprogrammeerde ingebruikname tot en met de voltooiing van de programma-exekutie. In Algol 68 brengt heap dergelijke cellen voort, in Pascal doet een aanroep van new dat.
4. dynamisch: zowel de ingebruikname alsook de vrijgave worden expliciet en naar believen door de programmeur geprogrammeerd.

Voor de programmeur is er weinig verschil tussen de statische cellen en de semistatische van het buitenste blok van een programma; het verschil zit meer in de

implementatie van de taal. In Fortran worden alle cellen statisch genomen; dat kan vooral omdat de taal geen rekursie en rekursieve aktiveringen kent (van de bij 2 genoemde expressievormen). Wanneer de evaluatie van expressies volgens het LIFO-regime plaatsvindt (de laatst aangevangen evaluatie van een expressie wordt het eerst volledig voltooid), vindt ook de ingebruikname en vrijgave van semistatische cellen volgens het LIFO-regime plaats, en kan dus de opslagruimte als een stapel bespeeld worden. Dat is een erg efficiënt beheer. Dit is bijvoorbeeld niet mogelijk bij parallelle evaluaties en bij coroutines: de laatst aangevangen evaluatie is dan niet het eerst voltooid.

Opmerking In Pascal wordt de term "static variable" gebruikt voor wat wij semistatische cel noemen. In die kontekst, en elders, worden de begrippen "static extent" en "statische scope" nogal eens verward. De extent betreft de levensduur, een tijdsinterval gedurende de exekutie, van een assigneerbare variabele; terwijl de scope een tekstgedeelte aanduidt.

Aanduidbare cellen

De taalontwerper laat een bepaald machinebeeld of implementatiemodel in de programmeertaal in meer of mindere mate doorschemeren. Met name geldt dit ten aanzien van de vraag welke cellen van de opslagruimte in een programmatekst aanduidbaar en dus assigneerbaar zijn. Wij geven hiervan de volgende twee voorbeelden.

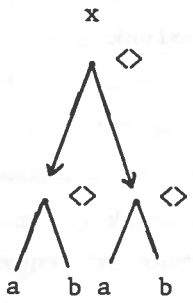
1. Bij het omgevingenmodel, zie Par. 4.2, ligt het voor de hand om precies of alleen die cellen aanduidbaar te laten zijn, die voor de opslag van bindingen gebruikt worden. We zouden HOF dus zo kunnen wijzigen en uitbreiden dat de introduktie van x in (df x=ea. eb) en ((fn x.eb) ea) semantisch de kreatie van een semistatische cel is die geïnitialiseerd wordt met de waarde van ea. Met andere woorden, de cel waarin de binding x<-ea wordt opgeslagen volgens het omgevingenmodel, is assigneerbaar (en in de tekst aanduidbaar met x). Ter illustratie, de evaluatie van

(df y=3; f=(fn x. y:=x+y; x). z:=(f y); y)

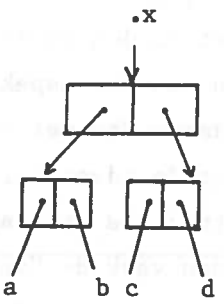
resulteert dan in 6 (de eindwaarde van de cel y), terwijl de globale cel z de waarde 3 gaat bevatten. Dus alle variabelen (identifiers) duiden een assigneerbare variabele (cel) aan, en omgekeerd alle assigneerbare variabelen zijn met een variabele aanduidbaar.

2. Beschouw nu de graafrepresentatie van expressies (en waarden!) zoals gebruikt in het Graafmodel, Hoofdstuk 3. Samengestelde waarden worden daar opge-

slagen door een samengestelde cel te nemen en daarin verwijzingen naar de opslag van de samenstellende delen op te nemen. De taalontwerper kan ervoor kiezen (of niet!) om ook dat soort cellen aanduidbaar en assigneerbaar te laten zijn. Dit is bijvoorbeeld het geval in bijna alle LISP-varianten middels RPLACA en RPLACD. Wij geven hieronder een voorbeeld dat zowel in LISP alsook in een nieuwe ad-hoc uitbreiding van HOF geformuleerd is. Stel dat x een waarde <<a, b>, <c, d>> aanduidt of in LISP: (CONS (CONS a b) (CONS c d)):

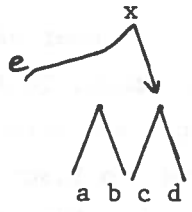


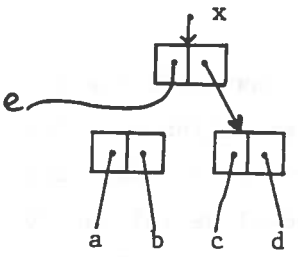
De graafrepresentatie van x,



en uitgetekend in cellen.

Iedere cel(helft) is nu assigneerbaar; b.v. in LISP door (RPLACA x e), wat wij in de gewijzigde HOF met x.1 :=e (en in een alternatieve LISP notatie met (CAR x):=e) zouden kunnen noteren. Dit moet dan resulteren in:





Graafrepresentatie van x, en uitgetekend in cellen, na x.1:=e

Dus stel dat normaliter e.1 tot (de verwijzing naar) e1 evalueert, dan zal e.1 links van := niet daartoe evalueren, maar slechts tot de cel(helft) die (de verwijzing naar) e1 bevat. Net zo voor .2, CAR en CDR.

Door middel van RPLACA en RPLACD in LISP en deze ad-hoc wijziging van HOF is het gemakkelijk om waarden gedeeld (shared, overlappend) te representeren en dus de opslagruimte efficiënt te benutten. Maar bedenk wel dat dergelijke deling in veel gevallen al automatisch tot stand komt in de Graafimplementatie van HOF. Bijvoorbeeld, de lus in de representatie die ontstaat door na een definitie x = ⟨1,0⟩ de assignment x.2 :=x uit te voeren, komt bij de Graafimplementatie van HOF direkt tot stand bij x rec= ⟨1,x⟩. Dit voorbeeldje illustreert tevens het grootste gevaar van de uitbreiding: in HOF zijn linker- en rechterlid van een definitie ten alle tijden voor elkaar substitueerbaar, terwijl dat na de ad-hoc wijziging niet meer het geval is. Dit verschijnsel doet zich ook in geval 1

voor, maar een korrektheidsredenering is nu toch wel heel moeilijk met predikaten of asserties te formuleren.

Aanduiding van cellen

De cellen van de opslagruimte van een machine worden door een nummer, ook wel adres genoemd, geïdentificeerd. In machinekode moeten die nummers ofwel adressen gebruikt worden om cellen aan te duiden. Gelukkig wordt er in "hogere" programmeertalen van dit aspekt afgezien, geabstraheerd. In assembleertalen kan men al een cel aanduiden met een relatief adres, de afstand tot een gegeven cel. Maar meestal doet de identiteit (het adres) van een cel helemaal niet terzake voor een algoritme; als er maar nieuwe cellen in gebruik genomen kunnen worden. Daarvoor wordt dan vaak de "variabele-deklaratie" door de programmeertaal geboden; er wordt daardoor een nieuwe cel aan een identifier gebonden (loc en heap in Algol 68; en de var-deklaratie en value-parameter in Pascal). De Heel Eenvoudige Taal HET (Meertens 1974) zet de abstraktie tot het uiterste voort. Daarin duidt iedere waarde een cel aan; d.w.z. "onder" willekeurige waarde kan een waarde worden weggeschreven, en "uit" willekeurige waarde kan de inhoud worden uitgelezen.

Los van bovenstaande kan het in een programmeertaal in meer of mindere mate doorschemeren dat cellen met adressen aangeduid kunnen worden. In Algol 68 worden de begrippen cel en adres met elkaar geïdentificeerd: een verwijzing (reference) speelt zowel de rol van cel als ook die van adres. In Pascal is er een strikte scheiding tussen die begrippen: beide zijn als apart "datatype" in de taal aanwezig. We lichten dit in de volgende paragraaf nog uitvoerig toe.

Tenslotte merken we nog op dat als cellen met variabelen kunnen worden aangeduid, er meestal ook ingewikkelder expressievormen zijn die cellen aanduiden. Bijvoorbeeld, array-indicering en record-selektie en pointervolging en, in Algol 68, ook de funktie-aanroep, enzovoorts.

* * *

Hiermee besluiten we onze uiteenzetting over aspekten van assigneerbare variabelen. Zoals we gezien hebben in Par.7.1 hebben beschrijvende programmeertalen een aantal voordelen boven gebiedende, maar zijn gebiedende aspekten zoals assignment soms nodig omwille van de efficiëntie. Een taalontwerper biedt middels de expressievormen en hun semantiek een abstraktie van de feitelijke machine aan, en heeft daarbij veel vrijheidsgraden waar verantwoorde keuzen gedaan moeten worden.

(Daarnaast speelt ook de syntaktische vormgeving een rol. Bijvoorbeeld, de dereferingscoercie-regels van Algol 68 worden soms minder prettig ervaren dan de overeenkomstige regels in Pascal; zie ook de volgende paragraaf. Maar in deze syllabus gaan we nauwelijks in op de syntaxis.)

Par. 7.3 Algol 68 references versus Pascal pointers

In deze paragraaf geven we "onze Algol 68 opvatting" en "de Pascal opvatting" van het begrip assigneerbare variabele, en formaliseren we de gebruikelijke plaatjes waarmee asserties grafisch worden weergegeven. Het komt erop neer dat in onze Algol 68 opvatting cellen en hun adressen worden geïdentificeerd (en references worden genoemd), terwijl ze in Pascal worden onderscheiden als verschillende datatypen (nl. variabelen en pointers). De resulterende eenvoud en zwakte van Pascal worden toegelicht.

Dit verhaal dient als uitwerking van sommige ideeen van de vorige paragraaf, maar kan ook afzonderlijk gelezen worden.

* * *

In het hier navolgende heeft de linkerkolom steeds betrekking op Algol 68 en de rechter op Pascal.

ALGOL 68	PASCAL
We gaan uit van de opvatting dat de opslagruimte van een machine is verdeeld in cellen, en dat bij iedere cel een uniek adres hoort.	
In Algol 68 worden cellen met hun adressen geïdentificeerd: zo'n adres, dat dus tevens dienst doet als cel, noem ik een reference of verwijzing (officieel:name)	In Pascal worden cellen en adressen onderscheiden. Een cel wordt variabele genoemd, zijn adres pointer of wijzer.

We laten in het vervolg r variëren over references, v over variabelen, en a over pointers. We gebruiken in de syntaxis x, xx, y, yy, p en q als identifiers die references en gewone of pointer-variabelen aanduiden.

In onderstaande tabel noteren we symbolisch de semantische bewerkingen van de datatypen. Een toelichting volgt na de tabel.

voor REFERENCES r, r′	voor VARIABELEN v, v′
R1 create (levert een nieuwe r)	V1 create (levert een nieuwe v)
R2 r:= ...	V2 v:= ...
R3 r? (levert inhoud van r)	V3 v? (levert inhoud van v)
R4 r=r′(gelijkheidstest)	V4 pointer-to(v) (levert een a)
R5 ...:=r (r is schrijfbaar)	
R6 nil (een speciale r)	
	voor POINTERS a, a′
	P1 a=a′(gelijkheidstest)
	P2 a↑ (levert een v)
	P3 ...:=a (a is schrijfbaar)
	P4 nil (een speciale a)

Toelichting

R1, V1: We zien af van het aspekt van de levensduur ofwel extent. Dus deze bewerking dekt de kreatie die door de loc, heap, var-deklaratie en binnen new() plaatsvindt.

R2, V2: In overeenstemming met het informele machine model is de bestemming van een assignment in Pascal een variabele (=cel) en geen pointer (=adres). Vanwege de identifikatie tussen cellen en adressen is de bestemming in Algol 68 een reference.

R3, V3: Het postfix vraagteken is onze notatie voor de uitlees-operatie. Die wordt in Algol 68 jargon dereferencing genoemd; in Pascal is daar geen naam voor, maar daar wordt wel gesproken over "the current value of a variable". In beide talen kan de uitlees-operatie niet expliciet genoteerd worden, maar wij zullen hem in deze paragraaf overal uitdrukkelijk opschrijven.

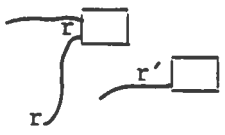
V4: De bewerking pointer-to kan in Pascal niet vrijelijk door de programmeur gebruikt worden, maar vindt slechts en alleen impliciet plaats binnen new:

```
procedure new (var p: ↑ T);
begin  create a new v;
       p := pointer-to (v)
end
```

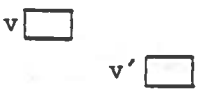
Vanwege de identifikatie van cellen en adressen is de pointer-to operatie in Algol 68 niet nodig; desgewenst kun je hem simuleren met de identiteitsfunktie!
R5, P3: Intuïtief valt er wat voor te zeggen dat cellen niet schrijfbaar zijn, en adressen wel. De Pascal opvatting stemt hiermee overeen. Ten gevolge van de identifikatie van cellen met adressen kan in Algol 68 deze onderscheiding niet aangebracht worden.
R4, P1: Let erop dat de gelijkheidstest tussen references syntaktisch niet met = maar met is wordt genoteerd. In Pascal is er geen gelijkheidstest op cellen; dus aliassing (twee namen voor eenzelfde variabele) kan in Pascal niet getest worden, tenzij je pointers naar die cellen ter beschikking hebt.
P2: Pascal suggereert geen andere terminologie voor a↑ dan: the variable referenced by a. Ik spreek a↑ uit als: a gevolgd. Let erop dat dit een andere operatie is dan de uitlees-operatie! Vanwege de identifikatie tussen cellen en adressen is deze operatie in Algol 68 niet nodig; het zou de identiteit zijn. Voorts geldt dat pointer-to en ↑ elkaars inversen zijn.
R6, P4: nil is een voorgedefinieerde r of a waarop operaties R2, R3 resp. P2 niet toegepast mogen worden; voor het overige onderscheidt nil zich niet van andere r of a.

We geven nu enige voorbeelden. De asserties, ofwel beweringen over de toestand, geven we grafisch weer. Het zijn bij uitstek de gelijkheid van variabelen, pointers en references en de uitlees-funktie die zich grafisch makkelijker laten weergeven dan in formules; andere beweringen over de toestand kunnen vaak beter in formules worden uitgedrukt. (In Algol 68 kun je alles in formules uitdrukken; in Pascal niet tenzij je aan de assertietaal een predikaat toevoegt dat aangeeft of twee expressies aliassen zijn, i.e. verschillende uitdrukkingen voor eenzelfde variabele.)

Een reference r tekenen we als een lijn die eindigt bij een hokje dat uniek is voor die reference. Precieser: het beginpunt van zo'n lijn is "een tekening" van de reference. Dus bij verscheidene tekeningen van r zijn er verscheidene (beginpunten van) lijnen, maar slechts één hokje.

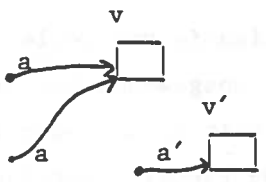


Een variabele tekenen we als een hokje; verschillende hokjes voor verschillende variabelen.



Een pointer tekenen we als een pijl; de punt van de pijl vlak bij de variabele waarvan hij het adres is. Preciezer: het beginpunt van zo'n pijl is "een tekening" van de pointer.

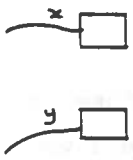
(We geven verderop nog twee varianten hierop).



De hokjes stellen de cellen voor, de erbij eindigende lijnen hun adressen. In Algol 68 worden die met elkaar geïdentificeerd: een reference is een lijn-met-het-hokje erbij. De inhoud van een cel tekenen we in het hokje; die inhoud wordt door de uitlees-operatie opgeleverd.

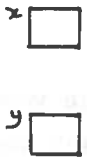
In de voorbeelden hieronder zetten we soms een of meer programma-expressies bij de daardoor aangeduide semantische objekten. Afwisselend volgen nu programmateksten en toestandsbeweringen.

. `ref int x = loc..., y = loc....`
`{ofwel: int x,y;}`



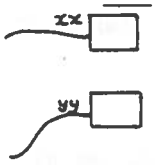
dus x≠y

`var x, y: int`



dus x en y zijn geen aliassen

. `ref ref int xx = loc..., yy= loc...`
`{ofwel: ref int xx, yy;}`

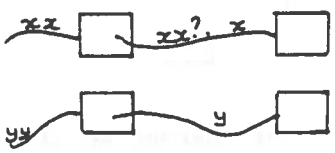


dus xx≠yy

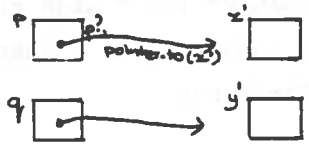
`var p, q: ↑int`

dus p,q geen aliassen

. `xx:=x; yy:=y`

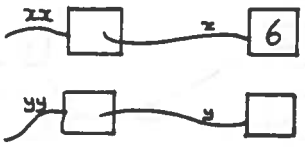


`new (p){i.e. var' x': int;`
`p:= pointer-to(x')}; new (q)`

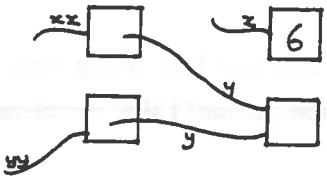


- xx? := 6

(xx eenmaal uitlezen)



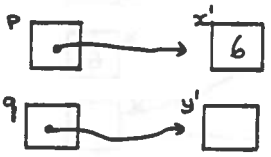
- xx := yy?



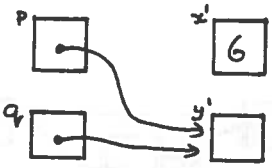
dus x ≠ xx? = yy? = y

p?↑ := 6

(p eerst uitlezen, dan de pointer volgen!)



p := q?



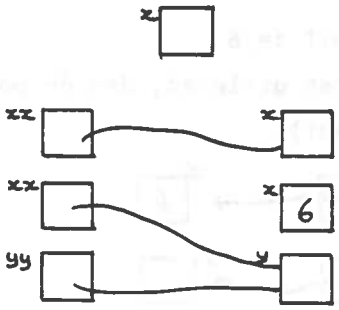
dus p?↑, q?↑ en y' zijn aliassen en pointer-to(x') ≠ p? = q? = pointer-to(y')

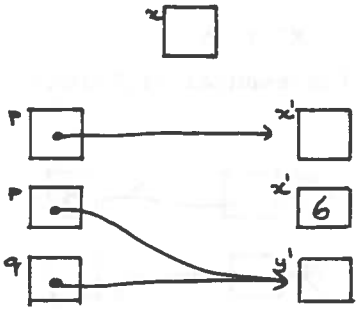
Merk op dat er in xx?:=6 een operatie minder staat dan in p?↑:=6. Dit komt doordat in Algol 68 de references zowel de rol van cel (bestemming van assignment) spelen, alsook de rol van adres (schrijfbaar in een cel).

Onze Algol 68 plaatjes zijn anders dan, maar wel isomorf met, de officiële Algol 68 plaatjes (Lindsey & Van der Meulen 1971, Van der Meulen & Kuhling 1974). Het voordeel van onze plaatjes is dat ze minder tekenwerk vergen en dat ze bij de volgende konventie direkt aansluiten bij de Pascal- en traditionele plaatjes:

> Wanneer het beginpunt van een reference niet terzake doet, met name wanneer dat beginpunt niet in een hokje getekend is, tekenen we de lijn héél heeeeeeel kort, zo dat je hem niet meer ziet.

We krijgen dan voor bovenbehandeld voorbeeld:





Desgewenst kunnen we ook de ref-nivo's in een plaatje weergeven, bijvoorbeeld

In Pascal nauwelijks nodig omdat een ↑↑T type nauwelijks voorkomt.

De reden dat een ↑↑T type nauwelijks in Pascal voorkomt, is dat de operatie pointer-to uitsluitend impliciet binnen new wordt toegepast, hij is met name niet toepasbaar op anderszins gekreeerde variabelen of op komponenten van variabelen. Dit is een paternalistische (welbewuste?) keuze van de taalontwerper geweest; semantisch gezien is er geen reden om het gebruik van pointer-to zo te beperken. Maar die beperking heeft wel de volgende twee voordelen: zogenaamde bungelende pointers (dangling references) zijn "niet zo makkelijk" mogelijk, en plaatjes van samengestelde variabelen zijn eenvoudiger. (Een toelichting komt zodadelijk). De nadelen zijn dat aliassen van komponenten of van gehele variabelen niet zo makkelijk mogelijk zijn en aldus de programmering van sommige algoritmen bemoeilijkt wordt en niet zo efficient kan. We geven nu drie voorbeelden ter illustratie van deze voor- en nadelen.

Allereerst blijken sommige fouten in Pascal moeilijker te programmeren zijn: bungelende pointers. Bij een vrijelijk gebruik van pointer-to is het volgende mogelijk.

```
ref ref int xx = loc ref int;
   ....
( ref int x = loc int := 0;
   xx := x
);
... xx? ...
```

```
var p:↑int;
   ....
( var x: int := 0;
   p := pointer-to (x)
);
...p?↑ ...
```

Na afloop van het binnenblok heeft xx als inhoud een niet bestaande reference, en p een pointer naar een niet bestaande variabele; d.w.z. de betreffende cel is alweer vrijgegeven voor andere doeleinden. Die inhoud van xx of p wordt een bungelende (dangling) reference genoemd. (Foutmeldingen hierop zijn wel mogelijk maar relatief "duur".) Doordat pointer-to niet gebruikt mag worden is bovenstaande in Pascal niet mogelijk. Het syntaktisch korrekte Algol68 programma is semantisch foutief ofwel ongedefinieerd.

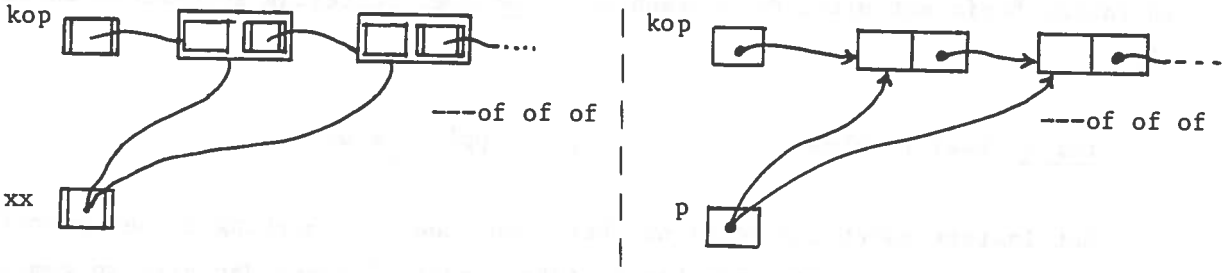
Als tweede voorbeeld beschouwen we het probleem om een gegeven lijst achteraan met een gegeven element uit te breiden.

```
mode m = ref struct (..., m next);      | type t = ↑ record...; next: t end;
m elem = ...                            | elem: t ...
ref m kop = loc m := ...                | var kop: t := ...
```

De standaard manier in Pascal en de overeenkomstige vormgeving in Algol 68 luidt als volgt.

```
if kop? = nil                           | if kop? = nil
then kop := elem                        | then kop := elem
else (ref m xx = loc m := kop?;         | else (var p: t := kop?;
      while (next of xx?)? ≠ nil        |       while p?↑.next? ≠ nil
      do xx := (next of xx?)? od;       |       do p := p?↑.next? od;
      next of xx? := elem               |       p?↑.next := elem
     )                                  |      )
```

De grafische weergave van een gedeelte van de invariant van de herhaling ziet er als volgt uit.



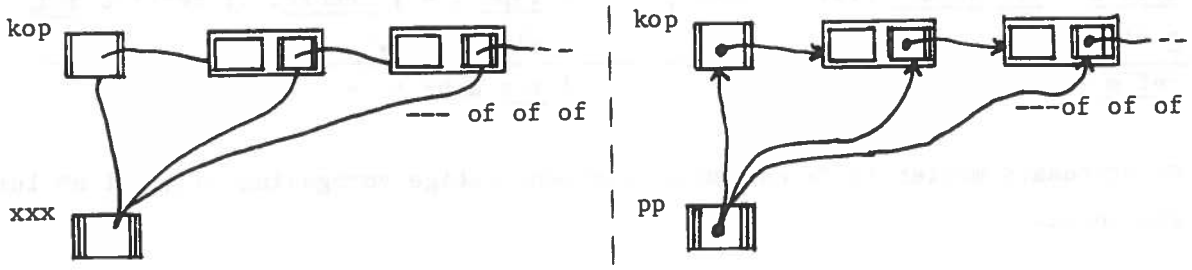
Het Pascal-plaatje is iets eenvoudiger: de komponenten van variabelen hoeven niet als afzonderlijke hokjes getekend te worden, omdat er vanwege de beperking op pointer-to geen pointers naar hen kunnen voorkomen. In Algol 68-plaatjes bestaan i.h.a. zowel lijnen naar een geheel hokje, alsook lijnen naar de deelhokjes; die moeten goed onderscheiden worden.

Wanneer we pointer-to ook zelf en bovendien ook op komponenten mogen toepassen, dan kunnen we een ietwat efficiënter programma schrijven dat tevens korter en eenvoudiger is. In Algol 68 kan dat zonder meer al.

```
ref ref m xxx = loc ref m := kop;
while xxx?? ≠ nil
do xxx := next of xxx??
od;
xxx? := elem
```

```
| var pp: ↑t := pointer-to (kop);
| while pp?↑? ≠ nil
| do pp := pointer-to (p?↑?↑.next)
| od;
| pp?↑ := elem
```

Een invariant van de herhaling ziet er nu als volgt uit.



Merk op dat de onderscheiding tussen de gehele variabele en zijn komponenten nu wel belangrijk is in het Pascal-plaatje. Bedenk voorts dat de uitlees-operatie meestal niet aangegeven hoeft te worden. Dus feitelijk wordt de toekenning na do geschreven als

```
xxx := next of xxx          |   pp := pointer-to (pp↑↑.next)
```

Het uitlezen in xxx? := ... moet in Algol 68 door een cast worden afgedwongen; in Pascal hoeft dat niet; de toekenning na od moet feitelijk geschreven worden als:

```
ref m (xxx) := elem         |   pp↑ := elem
```

Het laatste voorbeeld volgt nu; het toont hoe de beperking op het gebruik van pointer-to het programmeren bemoeilijkt, omdat aliassen dan niet zo gemakkelijk gemaakt kunnen worden. Stel dat m en t een overeenkomende mode en type zijn, en dat

```
(next of a[i][i+j])[k]      |   a[i][i+j].next[k]
```

een m-reference resp. een t-variabele aanduidt. Omwille van de efficiëntie kan het wenselijk zijn zo'n uitdrukking eenmaal uit te rekenen en dan de uitgereken-

de waarde (reference resp. variabele of pointer ernaar) steeds te gebruiken. Met een vrijelijk gebruik van pointer-to kan dat als volgt.

```
( ....                                  |  ( ....
ref m xx := (next of a[i][i+j])[k]      |  var p: t := pointer-to (a[i][i+j].
                                        |  next[k]);
{xx? = (next of a[i][i+j])[k]}          |  {p?↑ en a[i][i+j].next[k] zijn alias-
                                        |  sen}
... xx?? ...                            |  ... p?↑? ...
... xx? := ...xx??...                   |  ... p?↑ := ... p?↑? ...
)                                       |  )
```

Zonder de pointer-to kunnen we ook wel aliassen maken, zij het dan dat we daarvoor het procedure-mechanisme moeten gebruiken, i.h.a.

```
                                        |
(....                                   |  ( ....
ref m x = (next of a[i][i+j])[k];       |  p (a[i][i+j].next[k])
... x? ...                              |  )
... x := ... x? ...                     |
)                                       |
                                        |
                                        |  met
                                        |  procedure p (var x: t);
                                        |  begin ... x? ...
                                        |        ... x := ... x? ...
                                        |  end
                                        |
```

De with-statement in Pascal maakt ook wel aliassen, maar alleen voor komponenten van een record-variabele en bovendien is daarbij de naamgeving niet vrij te kiezen.

Tot besluit

Soms vindt men wel eens dat "pointeraritmetiek" in Pascal eenvoudiger is dan in Algol 68. Dat komt mijns inziens door de volgende redenen.

1. Semantisch. Het onderscheid tussen cellen en hun adressen is intuïtief wel aantrekkelijk, zeker voor de beginnende programmeur. Dus de bestemming van een

assignment is een cel (en geen adres), en cellen zijn niet schrijfbaar maar hun adressen wel. Toch is ook de identifikatie van cellen en adressen niet zo gek, te meer daar zelfs de beginnende programmeur weet dat voor het doorgeven van een array als var-parameter slechts zijn adres gekopieerd wordt: cellen worden door hun adres gerepresenteerd.

2. Syntaktisch. In Pascal wordt van een variabele altijd z'n waarde genomen, behalve wanneer hij voorkomt op een var-positie (zoals links van := , achter with en als var-argument). Dus behalve op var-posities wordt de uitlees-operatie altijd precies eenmaal automatisch tussengevoegd. In Algol 68 gebeurt dit nul, één of meermalen. En erger nog, het aantal uitlees-coercies wordt in Algol 68 door het type (= de mode) van de kontekst bepaald, dus door informatie ver weg, terwijl dat in Pascal door de vorm van de onmiddellijke kontekst wordt bepaald (behalve voor argumenten).

(Nota bene. Selektie en subscriptie moeten gezien worden als een funktie met een var-argument en een var-resultaat. Dit is zo omdat bijvoorbeeld selektie op een record-variabele een deel-variabele oplevert. Het analoge geldt ook in Algol 68).

3. Ingewikkeldheid verleggend. We hebben gezien dat het verbod op het expliciete gebruik van pointer-to de plaatjes eenvoudiger maakt en dangling pointers soms voorkomt. Daartegenover staat het bezwaar dat aliassing in het algemeen niet te testen is (dat zou wel nuttig zijn bij wissel-procedures), en sommige algoritmen moeilijker te programmeren zijn. Merk overigens op dat de programmeur de pointer-to operatie wel moet kennen, ook al mag hij hem niet gebruiken, omdat die kennis nodig is om pointers en vooral het effekt van procedure new te begrijpen. Zie bijvoorbeeld de definitie van new in het Pascal rapport.

Bij deze redenen komt bovendien nog dat "de Algol 68 opvatting" zoals die uit de formele terminologie van het Algol 68 rapport blijkt tegen-intuïtief is, d.w.z. niet strookt met ons machinebeeld. Want in die opvatting is een reference (officieel een 'name') geen verwijzing naar een cel, maar

. a value which can be "made to refer to" some other value...

dus: een verwijzing naar een waarde. Bijgevolg verwacht de assignment een verwijzing naar een waarde als bestemmings-operand. Dit vind ik tegen-intuïtief, hoewel het formeel natuurlijk allemaal wel klopt. Deze opvatting wordt in veel Algol 68 leerboeken uitgedragen. Maar gelukkig wordt onze opvatting ondersteund door het (niet tot de definitie behorende) kommentaar in het Algol 68 rapport dat een reference

... may be thought of as the address of the storage cell
in the computer used to contain the value...

(Revised Report 2.1.3.2.a).

Par. 7.4 De programmeertaal ASS

We geven nu verscheidene "uitbreidingen" van HOF door assignment en Pascal- en Algol 68-achtige assigneerbare variabelen toe te voegen, en noemen die taal dan ASS. (Heel toepasselijk geeft mijn Engels-Nederlandse woordenboek als betekenis voor ass onder andere: struikelblok, moeilijkheid). De semantiek wordt formeel met reduktieregels en een reduktiestrategie gedefinieerd.

Aan de hand van ASS kan het een en ander van voorgaande paragrafen nader gepreciseerd worden. Ook zullen we in volgende hoofdstukken ASS gebruiken om andere kenmerken van gebiedende talen te bespreken. We doen echter een zorgvuldige poging om de syntaxis en semantiek zoveel mogelijk bij de gesuggereerde konstrukties uit Pascal respektievelijk Algol 68 te laten aansluiten, zodat voor het begrijpen van de volgende hoofdstukken deze paragraaf, behoudens de volgende alinea, zonder veel bezwaar kan worden overgeslagen.

ASS is een zogenaamde expressie-gebrienteerde taal. Dat wil zeggen, iedere expressie levert een waarde op, maar heeft daarnaast mogelijk ook nog een neveneffekt: de verandering van inhoud van assigneerbare variabelen. Hierdoor kunnen we volstaan met één klasse van uitdrukkingen, de expressies, net zoals in Algol 68 (de units). In Pascal worden twee klassen onderscheiden: de statements en de expressies. De statements leveren geen waarde op maar hebben alleen een neveneffekt; de expressies leveren een waarde op en zouden geen neveneffekt mogen hebben. Maar een funktieaanroep is in Pascal een expressie, en in een funktieromp is in Pascal ieder gewenst neveneffekt uit te drukken. Dus die onderscheiding in statements en expressies is daar niet zo zinvol. Doordat wij in ASS geen onderscheid maken is de taaldefinitie klein; een zinvol en terecht onderscheid is — onder behoud van algemeenheid-- niet makkelijk te organiseren. (Bovendien lijkt het ons beter om een calculus op te zetten om de afwezigheid van interferentie, i.e. wisselwerking, tussen programmafragmenten te bewijzen dan om een afzonderlijke syntaktische kategorie van neveneffekt-vrije expressies in te voeren. Zie (Reynolds 1981).) Desgewenst kunnen we de expressies die een zekere waarde empty opleveren ook "de statements" noemen.

Uitgangspunten

We nemen een nieuwe konstante (dus ook expressie en zelfs expressie in resultaatvorm) in ASS op, te weten empty, waarvoor er geen enkele reduktieregel

is. Deze kontante bevat dus weinig informatie, zoals het symbool al suggereert. We kunnen hem vooral gebruiken als de waarde van een expressie die er eigenlijk alleen maar staat voor z'n neveneffekt. (In Algol 68 is empty de enige waarde van de mode void, terwijl skip een willekeurige waarde aanduidt van ieder gewenste mode.)

Omdat ASS een expressie-geörienteerde taal gaat worden moeten we ook voor sequentie, assignment enzovoorts een waarde definiëren. Voor de sequentie (e1; e2) kiezen we de waarde van e2 als waarde van het geheel, net zoals in Algol 68. Voor de assignment aan een Pascal-achtige var-variabele is de waarde van de gehele assignment empty. Voor de assignment aan een Algol 68-achtige loc-variabele is de waarde van het geheel de bestemmingsvariabele; soms wordt hiervan echt gebruik gemaakt, zoals in ref int x = (loc int := 0), vaak wordt die waarde direkt uitgelezen zoals in x:=(y:=0) (voor ref int x, y), en meestal wordt het resultaat van een assignment ge-void tot de waarde empty, zoals in (x:=0;...). In de Algol 68-achtige semantiek schemert het onderliggende LIFO-model sterk door: de waarde van een assignment en sequentie staat vlak voor voltooiing van de evaluatie ervan juist op de top van de waardenstapel sW. Deze semantiek stelt de programmeur in staat om althans dit aspekt van de machine meer uit te buiten dan in Pascal mogelijk is.

In overeenstemming met Pascal en Algol 68 zijn ass. variabelen niet automatisch met een vaste waarde geïnitialiseerd; de initiële waarde is willekeurig. We zullen in Hoofdstuk 9 zien dat geïnitialiseerde ass. variabelen-kreaties zelf te definiëren zijn in ASS. De var- en loc-variabelen zijn semistatisch: na voltooiing van de evaluatie van hun introducerend blok houden zij op te bestaan. We nemen in ASS ook de semidynamische (en dynamische) ass. variabelen van Pascal en Algol 68 op. Het voornaamste verschil tussen de var- en de loc- (of heap-) variabelen in ASS is dat de uitlees operatie ? op var-variabelen automatisch (en wel precies één keer) geschiedt terwijl die bij de loc-variabelen in ASS expliciet geschreven moet worden (en wel nul, een of meermalen herhaald). Bijgevolg zijn var-variabelen niet (in var-, loc- etc. variabelen) schrijfbaar, maar loc variabelen wel; alweer net zo als in Pascal en Algol 68.

In termen van het LIFO model geformuleerd, kan een semistatische ass. variabele x net zo geïmplementeerd worden als (df x = ea. eb). Het enige verschil is dat de cel die gebruikt wordt voor de opslag van de binding x<- ea ten gevolge van assignments aan x soms wordt overschreven. Omdat we de vorm (df x=ea. eb) ook in ASS opnemen met de oorspronkelijke semantiek (om een puur naamgevingsmechanisme te hebben), zijn dus niet alle cellen die gebruikt worden voor de opslag van bindingen als assigneerbare cellen aanduidbaar.

We schrijven voor ASS de kompositionele strategie voor, dus ruwweg gezegd: van links naar rechts evalueren en onderdelen voor het geheel. Herinner je dat de kompositionele strategie voor HOF als volgt luidt.

> Kies --als te kontraheren redex-- de meest linker minimale non-resultaat vorm die niet in een resultaatvorm ligt en ook niet in een then- of else-tak noch in het bereik van een bindingsexpressie.

Op deze formulering moeten ten gevolge van de nieuwe expressievormen met neveneffekten een paar kleine wijzigingen gemaakt worden. (Namelijk, (i) redexkern i.p.v. redex, (ii) non-varvorm speelt een soortgelijke rol als nonresultaatvorm, (iii) een deel rechts van een puntkomma speelt net zo'n rol als een then- of else-tak, (iv) sommige nieuw te introduceren bindingsexpressies gaan een uitzondering vormen.)

De syntaktische uitbreiding van HOF tot ASS luidt als volgt.

```
e ::= --- | empty  | (e; e) | (e:=e)
      | (dfvar x. e) | (fnvar x. e)
      | (dfloc x. e) | (e?)   | (e alias e)
      | (dfloc' x. e)
      | (dfheap x. e)| (free e)
      | (new e) | (e↑)
```

Hopelijk zijn de gesuggereerde Pascal- en Algolkonstrukties duidelijk. In (ed:=es) noemen we het deel ed de bestemming (destination) en es de bron (source).

Met deze syntaktische uitbreiding zijn we ook in staat om standaard konstrukties uit gebiedende talen uit te drukken. Bijvoorbeeld, keuze kunnen we uitdrukken omdat de if-expressie al in HOF zit. En iteratie is met rekursie te simuleren; (while eb do es od) kunnen we in ASS schrijven als

```
(rec x. if eb then es; x else empty)
```

Ook zogenaamde exits en vele andere vormen van repetitie kunnen we gemakkelijk uitdrukken. Ter illustratie hiervan geven we een uitgebreid voorbeeld.

Gegeven zijn twee var-variabelen input en output. Beide bevatten een lijst. Gevraagd wordt om achtereenvolgens het 1ste, 3de, 6de, 10de, 15de, ... element van input op kop van output te zetten. Een mogelijk ASS programma daarvoor luidt

als volgt. Bedenk dat na de i-de overheveling van een element van input naar output, er precies i elementen van input moeten worden overgeslagen.

```
dfvar i. i:=0;
  rec verwerkInput.
      {er zijn net i elementen overgeheveld}
      dfvar j. j:=i;
          rec skipJElementen.
          if eqnil input then empty {klaar}
      else if j eqn 0 then output := hd input: output;
                           input := tl input;
                           i:=i+1;
                           verwerkInput
      else {if j>0    then}(input := tl input;
                            j:=j-1;
                            skipJElementen)
```

We hebben weer gebruikelijke voorrangregels gevolgd om minder haakjes te hoeven schrijven: postfix operatoren binden het sterkst, daarna de prefix operatoren, en dan de infix operatoren (waarvan de ; het zwakst en de := het een-na-zwakst bindt). De romp van bindingsexpressies moet zo groot mogelijk gekozen worden.

In dit voorbeeldprogramma zijn twee geneste iteraties te onderscheiden, gelabeld verwerkInput en skipJElementen. Na de eerste then worden beide beëindigd, na de tweede then wordt de binnenste beëindigd en de buitenste doorgestart.

Tenslotte vermelden we dat we in de formele semantiek ervoor kiezen om de cellen van de opslagruimte verspreid in de expressies te representeren. (Er zijn nog tal van alternatieven; onze keuze heeft het voordeel dat er nauwelijks nieuwe begrippen geïntroduceerd hoeven worden.) De expressievormen hiervoor zijn als volgt.

```
e ::= ---
    | (var x ctn e. e)
    | (loc x ctn e. e)
    | (loc' a ctn e. e)      | a
    | (heap x ctn e. e)
    | (heapvar x ctn e. e) | (ptr-to e)
```

waarbij a varieert over A, een verzameling van dingen die we adressen noemen; deze nemen we ook op in de konstanten en kunnen dus in resultaatvormen voorkomen. Spreek ctn uit als 'containing'. Van deze nieuwe expressievormen zijn de

var-, loc-, heap- en heapvar-expressies bindingsexpressies. De loc'-expressies zijn geen bindingsexpressies; a is immers een konstante. Om het Algol 68 programma

```
mode r = ref r ; r x; x:=x
```

ook in ASS mogelijk te maken definiëren we dat het bereik van de binding van (loc x ctn ex. eb) zowel ex alsook eb omvat, en net zo bij var, heap, heapvar. Natuurlijk zijn ook de dfvar-, dfloc-, dfloc'- en dfheap-expressies bindingsexpressies, en noemen we het deel na de punt weer de romp van die expressies.

Hiermee besluiten we de uitgangspunten. We gaan nu van de verscheidene groepen van expressies de semantiek formeel definiëren.

Sequentie

De expressievorm (e1; e2) staat voor sequentie. De waarde van het geheel is die van e2; dus e1 staat er alleen maar voor z'n neveneffekt. De puntkomma gooit de waarde van e1 weg. De reduktieregel luidt

(;) (e1; e2) -> e2

bij de kompositionele strategie moet de te kontraheren subexpressie niet in een deel e2 van (e1; e2) gekozen worden (net zo als die niet in een then- of else-tak gekozen wordt).

Nu is de punt-komma louter "syntactisch suiker", want (e1; e2) is onder de kompositionele strategie gelijkwaardig met ((fn x y. y) e1 e2), dus met (((fn x. (fn y. y)) e1) e2). Voorts willen we dat er een extra uitzondering komt op de regel "onderdelen eerst reduceren alvorens het geheel te kontraheren", namelijk:

Het zou voor ASS overigens geen verschil uitmaken in het eindresultaat van redukties als e2 in (e1; e2) wel eerst tot resultaatvorm gereduceerd zou worden. In dat geval komt de volgorde van reduktiestappen op e1 en e2 precies overeen met die in (fn (x, y). y)(e1, e2).

De var-variabelen: Pascal-achtige semistatische variabelen

De reduktieregels zullen geen verbazing wekken. Zij luiden als volgt.

(dfvar) (dfvar x. eb) -> (var x ctn ex. eb)

waarbij ex een willekeurige expressie in resultaatvorm is.

(var:=) (var x ctn ex. --- x:=ex' ---) -> (var x ctn ex'. --- empty ---)

(auto?) (var x ctn ex. --- x ---) -> (var x ctn ex. --- ex ---)

(var) (var x ctn ex. eb) -> eb

(var app) ((fnvar x. eb) ea) -> eb[x/ea]

Let wel, deze regels zijn omwille van de eenvoud in de presentatie enigszins slordig geformuleerd. Bijvoorbeeld in regels (var:=) en (auto?) moet het aangegeven voorkomen van x door het aangegeven deel var x gebonden worden. En in de regel (auto?) is natuurlijk substitutie van ex voor x bedoeld, dus eventueel gepaard gaande met het nemen van een alfabetische variant van de romp. Voorts mag in regel (var:=) en (var) de bindingsrelatie op de delen ex' en eb natuurlijk niet door de kontraktie veranderen; dit levert echter hier geen beperking als zowel de kompositionele strategie wordt gevolgd en met name regel (auto?) steeds wordt toegepast, als ook funkties niet in een resultaat van een dfvar-expressie voorkomen.

We geven nu een schets van de formele semantiekdefinitie. Allereerst merken we op dat de bestemming van een assignment niet tot een resultaatvorm gereduceerd moet worden, maar tot een var-vorm: een var-vorm is een voorkomen van een x die in de kontekst door een var x gebonden wordt. Ten tweede noemen we een deel van een expressie een var-positie als het de bestemmingsoperand van een assignmentexpressie is, of het argumentdeel van een applikatie met een funktiedeel dat tot (fnvar x. eb) reduceerbaar is. Ten derde herdefiniëren we het begrip strategie enigszins. Noem in regels (:=) en (auto?) de in de romp aangegeven delen de kern van de redex en noem in de overige regels de redex zelf tevens zijn kern. Een strategie behelst dan de keuze van een subexpressie die als kern van een te kontraheren redex genomen moet worden, (eventueel ook met de keuze van een toe te passen reduktieregel). De kompositionele strategie definiëren we nu als volgt:

> Kies als redexkern de meest linker minimale subexpressie die niet binnen een resultaatvorm ligt en die, staande op een var-positie, nog niet in var-vorm is of, staande op een andere positie, nog niet in resultaatvorm is. Bovendien moet die subexpressie niet in een then- of else-tak of rechts van een puntkomma gekozen worden, en evenmin in het bereik van een bindingsexpressie die verschilt van (var x ctn ...).

Anders dan normaal moet het bereik van de bindingsexpressie (var x ctn ex. eb) wel eerst tot resultaatvorm gereduceerd worden, alvorens regels (var) wordt toegepast. Dat kan omdat de vrije voorkomens van x in eb (en ex) niet tot het

stokken van de reduktie leiden: regels (var:=) of (auto?) zijn erop van toepassing. Voorkomens van vrije variabelen leiden in het algemeen tot het stokken van de reduktie, omdat ze geen resultaatvorm zijn.

We geven nu een voorbeeld van een reduktie. De toegepaste reduktieregel wordt steeds aangegeven.

```
                dfvar x. ((fnvar y. x:=0; y:=y+1; x) x)
(dfvar)  =>     var x ctn 4711. ((fnvar y. x:=0; y:=y+1; x) x)
(app)    =>     var x ctn 4711. x:=0; x:=x+1; x
(var:=)  =>     var x ctn 0. empty; x:=x+1; x
(;)      =>     var x ctn 0. x:=x+1; x
(auto?)  =>     var x ctn 0. x:=0+1; x
(+)      =>     var x ctn 0. x:=1; x
(var:=)  =>     var x ctn 1. empty; x
(;)      =>     var x ctn 1. x
(auto?)  =>     var x ctn 1. 1
(var)    =>     1
```

Merk op dat door fnvar aliassen worden gemaakt, terwijl df dat niet doet: (var x ctn 117. df y=x. ---) => (var x ctn 117. df y=117. ---). Het Beginsel van de Overeenkomst tussen Definitie en Parameterisatie zegt dat de parameter-argumentbinding dezelfde is als de binding bij definities. Om dit Beginsel van Overeenkomst te handhaven moeten we ook een met fnvar overeenkomende definitievorm aan ASS toevoegen. Dat kan als volgt.

> Breid de expressievormen uit met (dfvar x=ex. eb), te noemen een (dfvar=)-expressie, met de reduktieregel:
> (dfvar=) (dfvar x=ex. eb) -> eb[x/ea].
> Tevens breiden we de definitie van var-posities uit: het deel ea van (dfvar x=ea. eb) is ook een var-positie.

(De definiërende tekst van de kompositionele strategie hoeft niet veranderd te worden). Nu geldt inderdaad dat (dfvar x=ea. eb) en ((fnvar x. eb) ea) onderling uitgewisseld mogen worden zonder het resultaat te beinvloeden. De vorm (dfvar x=ea. eb) definieert x als een alias van de ass. variabele aangeduid door ea; deze definitievorm is in Pascal niet aanwezig, terwijl de overeenkomende parametervorm er wel is. Een beter symbool dan fnvar en dfvar zou zijn fnalias en dfalias; verwarring met het symbool dfvar van de variabele kreatie is dan uitgesloten.

Merk op dat df-expressies, dfvar=-expressies, applikaties en var-applikaties een var-vorm opleveren wanneer zij op een var-positie staan. Bijvoorbeeld, in de scope van var x ctn 0 geldt

```
(df y=ey. x) := ---      => x := ---
(dfvar y=ey. x) := ---   => x := ---
```

Maar bij een var-expressie is dat anders:

```
(var y ctn ey. x) := ---      => (var y ctn ey. 0) := ---
                              => 0 := ---
                              => stokt want 0 is geen var-vorm
```

((Wanneer we definiëren dat de romp van een var-expressie op een var-positie ook een var-positie is, dan hadden we gekregen

```
(var y ctn ey. x) := --- => x := ---
```

net zo als bij de andere bindingsexpressies. Maar ons inziens is deze veralgemening niet Pascal-achtig, omdat bij de regel (var x ctn ex. eb) -> eb dan een test moet plaatsvinden of eb de x niet vrij bevat (in dit geval zelfs of eb niet x is). Zo'n test hoeft in ASS niet gedaan te worden als eb geen funktie-expressie bevat.))

Loc-variabelen: Algol 68-achtige semistatische variabelen

Anders dan bij de Pascal-achtige variabelen moet de uitlees operatie nu wel expliciet opgeschreven worden. Wanneer dat achterwege gelaten wordt, wordt niet de waarde die bevat is in de cel genomen maar de cel zelf. Dat is het grote verschil; cellen zijn nu schrijfbaar. Voorts is het begrip var-positie niet meer nodig, maar wordt de klasse van expressies in resultaatvorm wel iets uitgebreid.

De reduktieregels liggen weer voor de hand.

```
(dfloc) (dfloc x. eb) -> (loc x ctn ex. eb)
        waarbij ex een willekeurige expressie in resultaatvorm is
(loc:=) (loc x ctn ex. --- x:=ex' ---) -> (loc x ctn ex'. --- x ---)
( ? )   (loc x ctn ex. --- x? ---) -> (loc x ctn ex. --- ex ---)
(loc)   (loc x ctn ex. eb) -> eb
```

met wederom de beperking dat in regels (loc:=) en (loc) de bindende voorkomens van de vrije voorkomens in ex' en eb niet door de kontraktie mogen veranderen.

In tegenstelling tot bij de Pascal-achtige variabelen geeft dit nu wel essentiele beperkingen. Met name mag x niet vrij in eb voorkomen in regel (loc), en in de scope van (dfloc x. dfloc y. ---) zal x:=y stokken (maar y:=x niet).

Om de semantiek formeel te definiëren breiden we eerst de klasse van expressies in resultaatvorm uit: in ~~het~~ de ~~bereik~~ scope van loc x is x ook een expressie in resultaatvorm. (Daarmee is die klasse dus kontekst-afhankelijk geworden.) De formulering van de kompositionele strategie behoeft nauwelijks enige wijzing: de expressie (loc x ctn ex. eb) wordt net zo behandeld als (var x ctn ex. eb), dus eerst eb tot resultaatvorm reduceren alvorens regel (loc) toe te passen.

We geven nu enige voorbeelden van redukties

```
                dfloc x. (fn y. y:=y?+1) (x:=0)
(dfloc)   =>    loc x ctn 4711. (fn y. y:=y?+1) (x:=0)
(loc:=)   =>    loc x ctn 0.    (fn y. y:=y?+1) (x)
(app)     =>    loc x ctn 0.     x:=x?+1
(?)       =>    loc x ctn 0.     x:=0+1
(+)       =>    loc x ctn 0.     x:=1
(loc:=)   =>    loc x ctn 1.     x
          =>    en hier stokt de reduktie; had er nog een uitleesoperatie op de
                romp van dfloc of van de funktieexpressie gestaan, dan hadden we
                gehad
                loc x ctn 1. x?
(?)       =>    loc x ctn 1. 1
(loc)     =>    1

                dfloc x. (dfloc y. y:=x; y?:=0)?
(dfloc)   =>    loc x ctn 4711. (dfloc y. y:=x; y?:=0)?
(dfloc)   =>    loc x ctn 4711. (loc y ctn 117. y:=x; y?:=0)?
(loc:=)   =>    loc x ctn 4711. (loc y ctn x. y; y?:=0)?
(;)       =>    loc x ctn 4711. (loc y ctn x. y?:=0)?
(?)       =>    loc x ctn 4711. (loc y ctn x. x:=0)?
(loc:=)   =>    loc x ctn 0 .   (loc y ctn x. x)?
(loc)     =>    loc x ctn 0 .   x?
(?)       =>    loc x ctn 0 .   0
(loc)     =>    0

                dfloc x. (df y=x. y?:=0)?   (vergelijk vorig voorbeeld.)
(dfloc)   =>    loc x ctn 4711. (df y=x. y?:=0)?
(df)      =>    loc x ctn 4711. (x?:=0)?
(?)       =>    loc x ctn 4711. (4711:=0)?
```

(?) => loc x ctn 4711. (4711:=0)?
=> stokt, want 4711 is geen variabele die door een var of loc wordt gebonden.

Merk op dat de df-definitie nu zowel gebruikt kan worden om aliassen te maken, alsook om de waarden van variabelen te benoemen. Voorts is het opmerkelijk dat er vanwege de beperking bij regel (loc:=) slechts beperkt met cellen "gerekend" kan worden. Deze beperking heeft het voordeel dat dangling references worden voorkomen; we komen hierop nog terug. In andere woorden geformuleerd luidt de beperking dat een cel c niet geschreven mag worden in een cel die c overleeft, en evenmin opgeleverd mag worden als onderdeel van "een resultaat dat c overleeft". Deze beperkingen zijn precies dezelfde als in Algol 68.

Om het "rekenen" met cellen ietwat te verruimen, in ruil voor de mogelijkheid van dangling references, geven we nu een variant van de loc-cellen. Omdat we ze met adressen weergeven, dat zijn konstanten net zo als 0, 1, 2, ..., kunnen ze wel buiten het bereik van hun introducerende expressie geexporteerd worden. Hier is de definitie.

Zij A een verzameling dingen die we adressen noemen; we laten a variëren over A. De konstanten K breiden we uit met adressen,

k ::= ... | a

zodat adressen ook expressies in resultaatvorm zijn. De formulering van de kompositionele strategie laten we ongewijzigd. Als reduktieregels definieren we:

(dfloc') (dfloc' x. eb) -> (loc' a ctn ea. df x=a. eb)
waarbij ea willekeurige expressie in resultaatvorm is, en a een adres dat niet in een deel loc' a voorkomt in de redex of zijn kontekst.
(loc':=) (loc' a ctn ea. --- a:=ea' ---) -> (loc' a ctn ea'. --- a ---)
(?) (loc' a ctn ea. --- a? ---) -> (loc' a ctn ea. --- ea ---)
(loc') (loc' a ctn ea. eb) -> eb

Hierbij gelden natuurlijk weer de gebruikelijke bepalingen ten aanzien van de bindingsrelaties. Maar let wel, loc' is geen binder; adressen zijn gewoon konstanten. Dus in de scope van (dfloc' x. dfloc' y. ...) leidt x:=y niet noodzakelijk tot het stokken van de reduktie. Eveneens mag bij regel (loc') de a wel in eb voorkomen, hoewel je dat als programmeur maar beter kunt vermijden omdat die a dan een dangling reference is. Een dangling reference, bungelende verwijzing, is de aanduiding (in een nog te evalueren expressie of in een al opgeleverd resultaat) van een cel die alweer voor andere doeleinden is vrijgegeven (door de

programmeur zelf of door het beheersalgoritme over de opslagruimte). We geven nu twee voorbeelden van bungelende verwijzingen.

```
                df z = (dfloc' x. x). dfloc' y. --- z --- y ---
(dfloc')    => df z = (loc' ax ctn 4711. df x=ax. x). dfloc' y. --- z --- y ---
(df)        => df z = (loc' ax ctn 4711. ax). dfloc' y. --- z --- y ---
(loc')      => df z = ax. dfloc' y. --- z --- y ---
(df)        => dfloc' y. --- ax --- y ---
(dfloc')    => loc' a ctn 117. df y=a. --- ax --- y ---
                (en a is nu misschien wel en misschien niet gelijk aan ax; ax zelf
              komt nergens meer in een deel loc' ax voor)
(df)        => loc' a ctn 117. --- ax --- a ---
```

Na de tweede kontraktie, waarbij de loc' ax verdwijnt maar de ax zelf niet, is ax een bungelende verwijzing geworden. Het resultaat van de reduktie is niet meer te voorspellen omdat ax gelijk kan zijn aan a, maar er ook van kan verschillen.

```
                dfloc' x. (dfloc' y. x:=y); x?:=0
(dfloc')    =>  loc' ax ctn 4711. df x=ax. (dfloc' y. x:=y); x?:=0
(df)        =>  loc' ax ctn 4711. (dfloc' y. ax:=y); ax?:=0
(dfloc')    =>  loc' ax ctn 4711. (loc' ay ctn 117. df y=ay. ax:=y); ax?:=0
(df)        =>  loc' ax ctn 4711. (loc' ay ctn 117. ax:=ay); ax?:=0
(loc':=)    =>  loc' ax ctn ay. (loc' ay ctn 117. ax); ax?:=0
(loc')      =>  loc' ax ctn ay. ax; ax?:=0
(;)         =>  loc' ax ctn ay. ax?:=0
(?)         =>  loc' ax ctn ay. ay:=0
            =>  stokken van de reduktie omdat geen regels (:=) toepasbaar zijn.
```

Na de kontraktie halverwege waarbij loc' ay verdwijnt, is de ay die nog bevat is in ax een bungelende verwijzing geworden. In dit geval leidt het gebruik van de bungelende verwijzing tot het stokken van de reduktie.

Bungelende verwijzingen worden onmogelijk gemaakt met de loc- en de var-konstrukties. De loc-konstrukties vergen wel run-time testen; bij de var-konstrukties zijn die testen niet nodig als bovendien funkties niet als onderdeel van het resultaat van een dfvar expressie optreden (hetgeen met typering eenvoudig is af te dwingen). Een andere manier om bungelende verwijzingen te voorkomen is om de cellen niet eerder vrij te geven dan nadat er geen verwijzingen meer naar bestaan, (net zo als in het omgevingenmodel bij funkties als resultaten de opslagruimte voor de bindingen niet meer volgens het LIFO principe beheerd kan

worden). Deze laatste oplossing wordt gerealiseerd door de heap-konstrukties (mits free niet wordt gebruikt). Die gaan we nu bespreken.

Heap-variabelen: Algol68-achtige semidynamische variabelen

In tegenstelling tot de semistatische loc- en var-variabelen blijven de heapvariabelen in principe bestaan zelfs nadat het blok waarin zij voortgebracht waren is beëindigd. (In een implementatie mag hun ruimte natuurlijk best wel worden vrijgegeven zodra die niet meer bereikbaar is; dit beheer heet garbage detection en collection). Wij modelleren een semidynamische cel als een soort loc- of var-cel op het buitenste nivo van het programma. We spreken daartoe af dat een programma altijd bestaat uit een expressie e als romp van een meervoudige heap-konstruktie:

(heap (x1, x2, ... xN) ctn (e1, e2, ... eN). e)

We gebruiken een meervoudige heap-konstruktie in plaats van een geneste, opdat de heap-variabelen elkaar kunnen bevatten: het bereik van de heap-binding omvat zowel de romp alsook de delen e1, ..., eN. Bij aanvang van de reduktie is N=0, dus luidt een programma (heap () ctn (). e).

Hier zijn de reduktieregels:

(dfheap) (heap (x1, .., xN) ctn (e1, .., eN). --- (dfheap x. e) ---)
-> (heap (x1, .., xN, x) ctn (e1, .., eN, ex). --- e ---)
met ex een willekeurige expressie in resultaatvorm
(heap:=) (heap (-- x --) ctn (-- ex --). --- x:=ex' ---)
-> (heap (-- x --) ctn (-- ex'--). --- x ---)
(heap?) (heap (-- x --) ctn (-- ex --). --- x? ---)
-> (heap (-- x --) ctn (-- ex --). --- ex ---)
(heap) (heap (x1, .., xN) ctn (e1, .., eN). eb) -> eb

Met natuurlijk weer de gebruikelijke regels ten aanzien van de bindingsrelaties. Dus een loc-variabele kan niet in een heap-variabele worden geschreven, en het eindresultaat van het programma mag geen heap-variabele meer bevatten. Merk op dat regel (dfheap) gepaard gaat met het nemen van een alfabetische variant, opdat de bindingsrelatie de bedoelde blijft. Merk ook op dat regels (heap:=), (heap?) en (heap) alleen in het symbool heap verschillen van de betreffende loc-regels.

Desgewenst kunnen we ook heap' konstrukties invoeren, net zo als de loc'-konstrukties. Er zijn dan weer konstanten (adressen) ter identifikatie in plaats

van identifiers zoals x. In dat geval kunnen we voor free een voor de hand liggende reduktieregel geven zodat (free a) bijna nooit stokt: de a mag best in de resterende expressie voorkomen ook al verdwijnt a als een heap-variabele. Uitwerking hiervan wordt aan de lezer overgelaten.

new: Pascal-achtige (semi)dynamische variabelen

Net zoals loc is gewijzigd in heap bij de overgang van semistatische naar semidynamische variabelen, zo kunnen we ook var wijzigen tot heapvar. Het verschil tussen heap en heapvar zit alleen in de automatische uitlezing van variabelen, met als gevolg dat heapvar-variabelen niet in andere geschreven kunnen worden ("zij zijn al uitgelezen voordat ze weggeschreven worden"). Om het wegschrijven ervan toch mogelijk te maken wordt de pointer-to operatie ptr-to geïntroduceerd. We definiëren: in (ptr-to e) is e ook een var-positie, en in het bereik van heapvar (-- x --) is (ptr-to x) ook een resultaatvorm ("de pointer naar x"). De volgoperatie ↑ zal van een pointer weer de variabele maken.

De reduktieregels voor (dfheapvar), (heapvar:=), (heapvar?) en (heapvar) liggen voor de hand, en geven wij kortheidshalve niet. De andere regels luiden als volgt.

(new) (new e) -> (dfheapvar x. e := ptr-to x)
(↑) ((ptr-to e) ↑) -> e

In Pascal worden alleen new en ↑ ter beschikking van de programmeur gesteld; dfheapvar- en ptr-to-expressies mogen niet door de programmeur zelf geschreven worden (evenmin als de heapvar-expressie om het hele programma heen).

* * *

Hiermee is de formele definitie van ASS voltooid.
In de volgende hoofdstukken kunnen we ons bezinnen op samengestelde variabelen (arrays, records) en andere aspecten van gebiedende talen.

Belangrijke onderwerpen uit Hoofdstuk 7

voor- en nadelen van assignment:
  benutting van machinemogelijkheden
  ingewikkelder (formele) semantiek
  ingewikkelder korrektheidsredeneringen
  overspecifikatie
assigneerbare variabele, assignment, uitlezing
klassifikatie van waarden naar hun toegestane gebruik
levensduur (=extent), gezichtsveld (=scope)
statische, semistatische, semidynamische, dynamische levensduur
Algol 68: references, cellen met adressen geïdentificeerd
Pascal: variabelen en pointers, geen identifikatie
dangling references, bungelende verwijzingen
alias (=verschillende namen voor eenzelfde cel)
automatische uitlezing:
  in Pascal 0 of 1 keer, in Algol 68 n keer ($n \geq 0$)
expressie-geörienteerdheid
statement
syntaxis en reduktieregels voor ASS:
  var- en loc en heap-variabelen
  empty en sequentie

Literatuur bij Hoofdstuk 7

Zie hiervoor de algemene leerboeken gegeven in de Inleiding van deze syllabus. (Reynolds 1970) gaat bij GEDANKEN uit van dynamische variabelen; bovendien worden labels als willekeurige waarden toegelaten. De essentie van Algol-achtige talen wordt in principes en formules vastgelegd in (Reynolds 1981). De semantiekbeschrijving voor ASS komt naar ons weten niet elders voor.

Oefeningen

7.1 Geef minstens twee zinvolle voorbeelden in Algol 68 of Pascal waarbij sprake is van goed-aardige welbedoelde neveneffekten.

7.2 In Par. 7.2. staan waarden geklassificeerd al naar gelang de kontekst waarin ze aangeduid kunnen worden. Geef zo'n klassifikatie voor Pascal, met voorbeelden (en "tegenvoorbeelden") voor iedere klasse.

7.3 Kan in Algol 68 iedere door loc voortgebrachte cel "in gebruik genomen" worden bij aanvang van (in tegenstelling tot: gedurende) het kleinst-omvattende blok waarin die loc voorkomt? (Zie de definitie van "semistatisch" in Par. 7.2.) Wenk: beschouw

```
(mode list = ref elt,
      elt = struct (int i, list t);
 list x:=nil;
 to n do x := loc elt := (0, x) od;
 ---)
```

7.4 In Par.7.2., Deel "Aanduidbare cellen", wordt onder geval 1 een mogelijke wijziging/uitbreiding van HOF gesuggereerd. Geef daarvan een formele definitie. Geef voor- en nadelen van zo'n taal ten opzichte van ASS, of ASS met alleen de var-konstrukties.

7.5! Ga na dat U in plaats van nil ook gewone (semidynamische of statische) references resp. pointers kunt gebruiken (door U zelf 'nix' genoemd). Wat is het bezwaar hiervan?

7.6 Formuleer de volledige invariant van het programma uit Par.7.3. waarin een lijst achteraan met een element wordt uitgebreid. Geef een formule in Algol 68 die het gebruik van plaatjes overbodig maakt.

7.7! Geef een variant van ASS waarin variabelen automatisch by default geïnitialiseerd worden. Geef ook een variant waarbij het gebruik van niet-geïnitialiseerde variabelen tot het stokken van de reduktie leidt. Geef voor- en nadelen van deze varianten ten opzicht van ASS.

7.8 Waarin wijkt de evaluatiestrategie van Algol 68 of Pascal af van die van ASS?

7.9 Wat is in Algol 68 de waarde van een herhalingskonstruktie? Kunt U alterna-

tieven bedenken?

7.10! Probeer het ASS programma uit Par.7.4 dat elementen van de input-lijst overhevelt naar de output-lijst zo getrouw mogelijk in Algol 68 of Pascal te vertalen.

7.11! Geef een ASS een "rekursieve statement" (geen rekursieve procedure of funktie) die in var t het aantal benodigde zetten telt voor het probleem van de Torens van Hanoi, bij initiële hoogte n. (Het gaat om het gebruik van rekursie, dus het programma t:=(2-tot-de-macht-n)-min-1 wordt afgekeurd).

7.12! Hoe wordt de semistatische levensduur van de var- en loc-variabelen in ASS gemodelleerd?

7.13 Is sequentie in ASS associatief, d.w.z. is in ASS (e0; (e1; e2)) "gelijkwaardig" met ((e0; e1); e2)?

7.14! Definieer een variant van ASS waarbij empty het resultaat van (e0; e1) is (zonder dat e1 wordt geëvalueerd alvorens de waarde van e0 is "weggegooid").

7.15 Geef een formele formulering van de regels (var:=), (auto?) en (var). Wenk: gebruik substitutie, ook om de vorm van de redex aan te geven.

7.16 Geef een zinvol voorbeeld waarbij een funktie als resultaat van een dfvar- en dus var-expressie wordt opgeleverd. Wenk: denk aan een random-generator die in een prive-variabele zijn laatst opgeleverde randomwaarde onthoudt. Leidt dit tot een stokkende reduktie?

7.17! Waarom leidt in ASS de expressie (dfloc x. x:=0);1 tot het stokken van de reduktie, terwijl in Algol 68 bij (loc int x; x:=0);1 als resultaat 1 wordt opgeleverd? Wenk: de coercie derefencing speelt hierbij geen rol! Geef de coercie in het Algol 68 programma nauwkeurig aan.

7.18! Programmeer diverse algoritmen in ASS; vermijd arrays met berekende grenzen. Wenk: denk aan fac, fib, gcd, rem, kleinste oplossing in gehele x, y, u, v van x**3 + y**3 = u**3 + v**3 met {x,y} ≠ {u,v}, sommeren etc. van een reeks f(0), f(1), ..., f(n) bij gegeven funktie f, enzovoorts enzovoorts, en "lijstalgoritmen".

7.19! Bewijs dat na een (dfloc')-kontraktie onmiddellijk een (df)-kontraktie volgt.

7.20 Doe Oefening 1.29 voor ASS.

7.21! Ga na dat het resultaat van een expressie nauwelijks beïnvloed wordt wanneer er een willekeurige (dfvar)-kontraktie op wordt gedaan. Geef voorbeelden waaruit blijkt dat dit niet geldt voor willekeurige (auto?) en (var:=)-kontrakties.

7.22 Formuleer een stel geschikte voorwaarden opdat een niet door de strategie gekozen (varapp)-kontraktie het resultaat van een expressie niet (of nauwelijks) beïnvloedt. Wenk: denk aan neveneffektvrije argumenten en argumenten die al in resultaatvorm staan.

7.23 Is het volgende Algol 68 programma syntaktisch legaal en zo ja, wat is officieel het resultaat en wat zal bij veel implementaties het resultaat zijn?

```
((int x:=7; int z:=8; 9); int y; print(y))
```

7.24 Neem aan dat (while e do e' od) een expressie vorm is. Becommentarieer de volgende reduktieregel.

(while) (while eb do e od) -> (e; while eb do e od) als eb = true
-> empty als eb = false

Geef een (beter) alternatief.

7.25 Definieer een Ass-funktie while met twee parameters zo dat een aanroep ervan de while-statement simuleert. (Wenk: evalueer de parameters call-by-name; zie Hoofdstuk 2.) Geef voorbeelden voor het gebruik ervan, met name voor de volgende statement: while i ≠ 10 do i:=i+1; f:=f*i od.

HOOFDSTUK 8

KORREKTHEID BIJ ASS

Hoewel van uiterst groot belang ontbreekt de uitwerking hiervan voorlopig. Het ligt in de bedoeling een systeem te ontwikkelen voor het bewijzen van korrektheidsuitspraken zoals {P} stmnt {Q}. Zolang er zo'n systeem nog niet is, zit er niets anders op om bij een formeel korrektheidsbewijs de reduktieregels maar zelf met de hand toe te passen en naar eigen inventiviteit een induktie-argumentatie op te zetten. Ook het begrip "semantische equivalentie" behoeft nadere studie; een belangrijke vraag hierbij is onder welke voorwaarden een niet door de strategie voorgeschreven kontraktie de semantiek van de expressie respekteert, (vergelijk Oefening7.21 en 7.22).

Literatuur bij Hoofdstuk 8

Korrektheid van imperatieve programma's is voor talen zonder funkties/procedures of zonder hogere orde funkties/procedures al haast gemeengoed. Zie bijvoorbeeld (Gries 1981) voor een eenvoudige taal (neveneffektvrije expressies, geen aliassen) en (Reynolds 1981a) voor een taal met aliassen en expressies met neveneffekten.

Boehm, H.-J., Side Effects and Aliasing Can Have Simple Axiomatic Descriptions. ACM TOPLAS Vol 7 Nr 4 (1985) pp 637-655
geeft een formalisme aan dat voor ASS zeer geschikt lijkt.

HOOFDSTUK 9

MODULEN: REGELING VAN ZICHTBAARHEID EN LEVENSDUUR

In bijna alle moderne programmeertalen zijn konstrukties aanwezig waarmee de zichtbaarheid (scope) van definities en de levensduur (extent) van ass. variabelen op bevredigender manier geregeld kan worden dan zonder die konstrukties mogelijk lijkt. In Ada zijn dat de packages, in Modula de modules, in Simula de classes, in Modulair Pascal (aan de T.H.T. en de R.U.G. ontwikkeld) de programeenheden, enzovoorts. Wij zullen in dit hoofdstuk de taal ASS uitbreiden met twee nieuwe expressievormen (en hun reduktieregel) die tesamen de essentie weergeven van bovengenoemde konstrukties. Tevens blijkt dat die expressievormen niets anders zijn dan een speciale notatie voor wat ook met funktionele argumenten kan worden uitgedrukt.

Par. 9.1 De expressievormen en hun motivatie

Aan de hand van een doorlopend voorbeeld motiveren en introduceren we de nieuwe expressievormen, die we voortaan de module-expressies noemen.

Het voorbeeld is de programmering van een voortbrenger van toevalsgetallen (random number generator). Het algoritme daarvoor luidt als volgt. Bij vaste m en d moeten achtereenvolgens de elementen van de reeks a1, a2, a3, ... worden opgeleverd als toevalsgetallen, waarbij ai = (a(i-1) * m) mod d voor i > 0 en a0 = een zo mogelijk door de gebruiker nog vrij te kiezen startwaarde a (the seed). Een programma hiervoor in ASS luidt als volgt.

```
(dfvar x. x := a;
 df trek = (fn (). x := x*m mod d; x)
 . prog
)
```

Hierbij is prog het "gebruikersprogramma" dat de funktie trek gebruikt om toevalsgetallen opgeleverd te krijgen.

Bovenstaande programmering heeft het bezwaar dat de ass. variabele x ook in prog zichtbaar is. Preciezer gezegd, het bereik van de binding dfvar x omvat ook prog, terwijl alleen de assignment x := a en de definiërende expressie van trek in dat bereik hoeven te liggen. Als dit programma in een groter geheel ingebed

gedacht moet worden, dan blijkt dat de naam x niet zo maar willekeurig gekozen kan worden: hij moet verschillen van de vrije variabelen van prog. Een dergelijke beperking is ongewenst. Bovendien moet in een formeel korrektheidsbewijs aangetoond worden dat de waarde van x niet verandert tussen willekeurig twee opeenvolgende aanroepen van trek. Ook dit is een onnodige belasting. Wat we wensen is een zodanige programmering, desnoods na een geschikte taaluitbreiding, dat alleen de initialisatie van x en de funktiedefinitie van trek in het bereik of gezichtsveld van x vallen. We bespreken nu 5 oplospogingen voor dit probleem.

Poging 1

De eenvoudigste oplossing lijkt: het deel prog te omringen met een definieerend voorkomen van een overigens niet terzake doende x. Bijvoorbeeld

```
(dfvar x. x := a;
 df trek = (fn (). x := x*m mod d; x).
 df x = dummy. prog
)
```

Weliswaar ligt prog nog steeds in het bereik van dfvar x maar gelukkig niet meer in het gezichtsveld ervan: de vrije voorkomens van x in prog worden niet door dfvar x gebonden. Een formeel korrektheidsbewijs is dus al veel eenvoudiger. Helaas worden de vrije x'en in prog wel gebonden: inbedding in een groter geheel waarbij o.a. de betekenis van x wordt vastgelegd, gaat dus nog steeds niet. Deze poging tot oplossing faalt.

Poging 2

We maken gebruik van hogere orde funkties: we introduceren x lokaal in de definitie van trek:

```
df trek = (dfvar x. x := a;
                    (fn (). x := x*m mod d; x))
. prog
```

Maar dit gaat mis: de evaluatie van de definiërende expressie van trek zal stokken, omdat in het op te leveren resultaat de x nog vrij voorkomt. We kunnen dit voorkomen m.b.v. de loc' variabelen, als we trek definiëren door

```
trek = (dfloc' x. x := a; (fn (). x := x?*m mod d; x?))
```

(Herinner je dat we bij loc-, loc'- en heap-variabelen de uitleesoperatie ? expliciet opschrijven.)

De evaluatie van de definiërende expressie van trek stokt nu niet meer, maar wat veel erger is, de x in de opgeleverde funktie is een dangling reference. Immers de levensduur (reservering van opslagruimte) van var- en loc-variabelen eindigt bij voltooiing van de evaluatie van het betreffende blok. Er zou dus een funktie opgeleverd worden die bij aanroep schrijft en leest in x, terwijl de opslagruimte die voor x was gereserveerd al voor andere doeleinden is vrijgegeven. Dit kan verholpen worden door dfvar x of dfloc' x te wijzigen in dfheap x :

```
trek = (dfheap x. x := a;
                  (fn (). x := x?*m mod d; x?))
```

Dit geeft weer een korrekt programma. (Volgens de C-strategie wordt eerst de definiërende expressie tot resultaatvorm gereduceerd alvorens er door de (df)-regel iets voor trek wordt gesubstitueerd. Bij de reductie tot resultaatvorm van die definiërende expressie verdwijnt de dfheap --in ruil voor een heap x helemaal om het programma heen-- en blijft alleen (fn (). --) over. Die funktie-expressie wordt dus overal voor trek gesubstitueerd.) Ondanks de korrektheid zijn er toch bezwaren. De reservering van de opslagruimte voor x blijft nu in principe langer geldig dan nodig is. We zouden graag willen dat x een semistatische variabele is die precies gelijktijdig met trek bestaat.

Poging 3

We geven een ad-hoc uitbreiding van de taal. We gaan definities als afzonderlijke syntaktische kategorie onderscheiden en er meer struktuur in aanbrengen. De grammatika voor de taal wordt:

```
e ::= x | (fn x. e) | (df d. e) | ...                    expressies
d ::= x=e | var x | loc x | heap x
    | (d, d) | (d; d) | (d in d)                         definities
```

Een definitie x=e, var x, loc x, heap x noemen we weer een binding van x. De vormen (d, d), (d; d) en (d in d) heten achtereenvolgens simultane, sequentiële en prive-definities; deze konstrukties zijn ontleend aan (Tennent 1981). Een andere notatie voor (d in d') is (private d within d'). Met behulp van deze definitiestrukturering kan de zichtbaarheid (de bindingsrelatie) beter in de hand gehouden worden; dat is te danken aan de volgende definitie.

Definitie

Het bereik van het voorkomen van een definitie d in een tekst (definitie of expressie) t is een deel van t dat als volgt is bepaald.

- bij (df d. e) is het bereik van d de romp e;
- bij d = (d1, d2) zijn de bereiken van d1 en d2 gelijk aan het bereik van d; (een vereiste is dat "de geïntroduceerde variabelen van d1 en d2" van elkaar verschillen);
- bij d = (d1; d2) is het bereik van d2 dat van d, en het bereik van d1 is d2 plus dat van d;
- bij d = (d1 in d2) is het bereik van d2 dat van d, en is het bereik van d1 alleen d2 (dus zijn inderdaad de definities d1 privé voor d2).

(Zoals gebruikelijk heten voorkomens van x die niet in het bereik van een binding van x liggen vrij, en bindt een binding van x de vrije voorkomens van x in zijn bereik.)

De semantiek zullen we niet formeel definiëren; informeel ligt die wel voor de hand. Belangrijk is het om te bedenken dat zowel (d1; d2) en (d1 in d2) zich qua levensduur van de variabelen in d1 net zo gedragen als (d1, d2). Dus de toevalsgetallenvoortbrenger kunnen we nu programmeren als:

```
(df var x := a in trek = (fn (). x := x*m mod d; x)
 . prog
 )
```

of met de andere notatie

```
(df private
      var x := a
   within
      trek = (fn(). x := x*m mod d; x)
 . prog
 )
```

In prog is dus alleen trek zichtbaar, en x is toch een semistatische variabele met een levensduur die precies de bedoelde is. Merk overigens op dat we wel enig bedrog hebben gepleegd: in plaats van de definitievorm var x hebben we var x := a gebruik. In het algemeen kunnen niet alle initialisaties met zo'n eenvoudige assignment geprogrammeerd worden, denk bijvoorbeeld aan rewrite (f) voor een file-var f. Dus we moeten in de ad-hoc uitbreidig ook initialisatie-statements opnemen. Dat kan als volgt.

```
d ::= ... | (d init e)
```

zodat de definitie in het programma nu luidt:

```
    (var x init x := a) in trek = (fn (). x := x*m mod d; x)
of  var x in (trek = (fn (). x := x*m mod d; x) init x := a)
```

Deze ad-hoc uitbreiding suggereert onmiddellijk om ook deklaraties van finalisatie-statements te voorzien. Bijvoorbeeld, neem als nieuwe deklaratievorm (d final e) met als semantiek dat e wordt geëvalueerd net voordat de evaluatie van de kleinst omvattende df-expressie is voltooid. Een lokale file-variabele zouden we dan als volgt kunnen deklareren:

```
var f: file of t
    init rewrite (f)
    final close (f)
```

Precies bij beëindiging van de levensduur van f wordt de expressie close (f) geëvalueerd; de gebruiker van f hoeft de file niet zelf te sluiten. Behalve dat deze taaluitbreiding, hoe nuttig ook, een nogal ad-hoc karakter heeft, is er bovendien nog steeds het bezwaar dat de startwaarde a in de definitie van de toevalsgetallenvoortbrenger vast wordt gelegd en niet naar willekeur in prog gekozen kan worden. Dus ook deze poging slaagt niet naar bevrediging.

Poging 4

We zien nu af van ad-hoc taaluitbreidingen en pakken de regeling van zichtbaarheid fundamenteler aan. Met hogere orde funkties is een bevredigende oplossing eenvoudig; en let wel, we gebruiken alleen funktionele argumenten en geen funktionele resultaten zodat het LIFO-beheer over de opslagruimte mogelijk blijft. Hieronder wordt het gebruikersprogramma prog als een parameter p doorgegeven aan een funktie die trek aan p meegeeft.

```
(fn p. (dfvar x. df trek = (fn (). x := x*m mod d; x)
        . x := a; p trek
       )
)(fn trek. prog)
```

Merk op dat in feite prog nu ook buiten het bereik van de definitie df trek = ... valt. Het is zelfs zo dat x en trek willekeurig anders genoemd kunnen worden, zonder dat het programma (fn trek. prog) mee-veranderd hoeft te worden; dat komt de modulariteit ten goede: uit (fn trek. prog) blijkt dat het een lokale

keuze is om betreffende funktie de naam trek te geven.

Bovenstaande programmering kunnen we nog iets veralgemenen. We geven daartoe de funktie (fn p. ---) de naam tgv (voor toevalsgetallenvoortbrenger):

```
df tgv = (fn p. (dfvar x. df trek = (---)
                 . x := a; p trek
                ))
```

In het bereik hiervan kunnen we tgv zo nodig verscheidene malen aanroepen:

```
    tgv (fn trek. prog)
en: tgv (fn trek1. tgv (fn trek2. prog'))
```

In de tweede regel zijn er in prog' twee onafhankelijk van elkaar werkende funkties trek1 en trek2. We kunnen tgv ook nog parameteriseren met de startwaarde a:

```
(*)     (df tgv = (fn a p. (dfvar x. df trek = (---)
                            . x := a; p trek
                           ))
          . tgv a1 (fn trek1. tgv a2 (fn trek2. prog'))
        )
```

De twee funkties trek1 en trek2 binnen prog' leveren bij achtereenvolgende aanroepen elk een reeks toevalsgetallen af, onafhankelijk van elkaar, die beginnen bij startwaarde a1 resp. a2.

In beginsel is er dus geen taaluitbreiding nodig om de zichtbaarheid (en de levensduur!) naar bevrediging te regelen. Funktionele argumenten zijn voldoende. Daarentegen vraagt deze programmeertechniek (soms Binnenste-buiten-programmering ofwel Reversed programming genoemd) wel om enige gewenning! We zouden de romp van bovenstaande oplossing (*) wellicht liever schrijven als

```
(df' trek1 = (tgv a1). (df' trek2 = (tgv a2). prog'))
```

of zoiets, hoewel het verschil met (*) uiterst klein is! Een mogelijkheid daarvoor bespreken we in de nu volgende poging.

Poging 5

We introduceren twee nieuwe expressievormen die in feite louter syntaktisch

suiker zijn: ze maken het mogelijk om de romp van oplossing (*) hierboven ietwat plezieriger te noteren. De nieuwe expressievormen zijn als volgt.

```
e ::= ...
    ¦ (mod --- export e end ---)
    ¦ (inv x = e. e)
```

De notatie (mod --- export ex end ---) staat voor een expressie met "(mod" aan het begin en ")" aan het eind en waarin precies één subexpressie een extra haken-paar export en end heeft; het deel ex noemen we het export-deel; mod is een afkorting van "module" en inv van "invokatie". De inv-expressie (inv x=ex. eb) is een bindingsexpressie net zoals (df x=ex. eb). Deze module-expressies vormen louter syntaktisch suiker omdat we ze qua semantiek zo zullen behandelen dat

```
(mod --- export ea end ---) staat voor (fn f. --- (f ea) ---)
(inv x = em. eb)            staat voor (em (fn x. eb)).
```

Daartoe definiëren we dat een mod-expressie ook een resultaatvorm is, en dat de reduktieregel als volgt luidt.

```
(inv) (inv x = (mod --- export ex end ---). eb) -> --- (df x = ex. eb) ---
```

Met andere woorden, het export-deel uit de module wordt de definiërende expressie voor x, terwijl de rest van de module helemaal om het nieuw gevormde definitie-blok heen komt te staan. Aldus kunnen de bij ex horende initialisatie en finalisatie tesamen met ex in één programma-eenheid geprogrammeerd worden, en bovendien nog onafhankelijk van de plaats waar ex als definiërende expressie wordt gebruikt. Overigens wordt in de reduktieregel natuurlijk substitutie bedoeld in plaats van letterlijke vervanging. Dus zonodig moet van de mod-expressie een alfabetische variant genomen worden zodat de vrije voorkomens van de inv-expressie ook na kontraktie vrij blijven.
Ter illustratie volgen hier een paar voorbeelden van (inv)-kontrakties.

```
    inv x = (mod 1*2*3* export 4 end *5*6). 7+x+8
=>  1*2*3* (df x = 4. 7+x+8) *5*6

    inv x = (mod dfloc y y:=1; export y end; print (y?)). — x --
=>  dfloc y. y:=1; (df x = y. — x --); print (y?)

    inv x = (mod df y=2. x* export x-y-z end *z). v+w+x+y+z
=>  df y'=2. x* (df x = x-y'-z. v+w+x+y+z) *z
```

Met deze taaluitbreiding kunnen we het toevalsgetallenvoorbeeld als volgt programmeren.

```
(df tgv = (fn a. mod (dfvar x. df trek = (---)
                        . x := a; export trek end
                      ))
. inv trek1 = (tgv a1). inv trek2 = (tgv a2). prog'
)
```

Dit is inderdaad precies oplossing (*) van Poging 4 met de gewenste vorm voor de romp. De reduktie hiervan verloopt als volgt. Het rechterlid van de definitie van tgv is in resultaatvorm, dus regel (df) wordt toegepast en levert

```
inv trek1 = ((fn a. mod ...) a1).
inv trek2 = ((fn a. mod ...) a2). prog'
```

Aannemende dat a1 in resultaatvorm is, wordt nu regel (app) toegepast en vervolgens regel (inv) :

```
(dfvar x. df trek = (---)
 . x := a1; df trek1 = trek
            . inv trek2 = ((fn a. mod ...) a2). prog'
)
```

Nu wordt door de (dfvar)-regel een ass. variabele x geschapen, en dan wordt met de reduktie van de romp van deze expressie begonnen. Daarbij worden onder andere de definities van trek en trek1 verwerkt. Gekomen bij (de reduktie van) inv trek2 = ... wordt weer eerst a2 gereduceerd, dan de (app)-regel toegepast en vervolgens de (inv)-regel. Als we afzien van de zojuist beschreven reduktiestappen, levert deze inv-kontraktie

```
(dfvar x. df trek = (---)
. x := a1; df trek1 = trek
           . (dfvar x. df trek = (---)
              . x := a2; df trek2 = trek. prog'
)           )
```

Wanneer x vrij voorkomt in prog' zijn de aangegeven voorkomens van x stelselmatig hernoemd, zo dat de voorkomens in prog' ook nog vrij zijn in deze kontekst. Het moge duidelijk zijn dat zowel de zichtbaarheid alsook de levensduur nu naar bevrediging geregeld zijn. Deze poging is dus geslaagd te noemen.

Naast de alreeds genoemde packages, modules, classes en program-eenheden zijn er vele andere taalkonstrukties voorgesteld om de zichtbaarheid en tevens levensduur verfijnder te regelen dan met de gewone blockstruktuur mogelijk lijkt. Met gebruik van funktionele argumenten zijn dergelijke konstrukties in beginsel overbodig. Desgewenst zouden speciale notaties bedacht kunnen worden om speciale (efficiente) implementaties te vergemakkelijken of de "leesbaarheid" te verhogen. Onze module-vormen zijn daar een voorbeeld van.

Vermeldenswaard is nog de aanpak van (Swierstra 1980) in LAWINE. In die taal is er een expressievorm die ongeveer met de mod-expressie overeenkomt, terwijl er geen onderscheid gemaakt wordt tussen een definitie-blok en module-invokatie. Ondanks het feit dat het onbeslisbaar is of een (definiërende) expressie tot een module-expressie reduceert, is er toch een bijna-LIFO-beheer over de opslagruimte mogelijk: twee stapels zijn voldoende (bij verdere afwezigheid van funktionele resultaten).

Par. 9.2 Toepassing: dynamische arrays

Als voorbeeld van de module-expressies geven we nu een toepassing. Zonder dat er in ASS konstrukties voor dynamische arrays aanwezig zijn, kunnen we die arrays toch programmeren en met behulp van de module-expressies leesbaar noteren. Onder een dynamisch array verstaan we hier een rij semistatische variabelen waarvan het aantal (de lengte van de rij) niet berekenbaar is (uit de tekst, dus statisch ofwel at compile-time), maar pas tijdens de evaluatie (at run-time) bepaald wordt. De elementen van de rij zijn dus assigneerbaar, maar de rij als geheel niet.

De programmering luidt als volgt. Allereerst definiëren we een funktie dyn-array die, gegeven n, een module oplevert met als export een lijst ter lengte n van —net buiten de export gedefinieerde-- semistatische variabelen.

```
dyn-array rec= fn n. if n eqn 0 then (mod export nil end)
                     else (mod dfloc x.
                                 inv tail = dyn-array (n-1)
                                 . export x: tail end
                     )
```

In het gezichtsveld van deze definitie levert een module-invokatie van dyn-array

een lijst variabelen op:

```
(inv a = (dyn-array 8). --- hd a := (hd a)? + 1 ---)
```

In de romp staat a voor een lijst van acht semistatische variabelen die onafhankelijk van elkaar gebruikt kunnen worden en precies de juiste levensduur hebben; een assignment aan a zelf is niet mogelijk. Deze inv-expressie reduceert na een stel reduktiestappen tot

```
loc x8 ctn e8.
 loc x7 ctn e7.
  ...
   ...
    loc x1 ctn e1.
     df tail1 = nil.
      ...
       ...
        df tail7 = x6: tail6.
         df  tail8 = x7: tail7.
          df a = (x8: tail8). --- hd a := (hd a)? + 1 ---
```

Merk op dat we loc-variabelen hebben gebruikt; de var-variabelen zouden in de definities van tail automatisch uitgelezen worden. Om die uitlezing te voorkomen zouden we nog een paar taaluitbreidingen moeten doen.

We kunnen ook meer-dimensionale dynamische arrays op deze manier programmeren, en dus vierkante of zelfs grillig gevormde (driehoekige, elliptische) matrices. Daartoe gebruiken we lijsten van loc-variabelen. Voor een "driehoekige" matrix gaat dat als volgt.

```
df driehoek = fn n. if n eqn 0 then mod export nil end
                                else mod inv a = dyn-arr n.
                                         inv d = driehoek (n-1)
                                         . export a: d end
. (inv dh = (driehoek m). ------)
```

(Ook in Algol 68 zijn dergelijke dynamische arrays mogelijk, zij het om ietwat andere redenen. De door loc gegenereerde variabelen hebben een levensduur die ruwweg gezegd samenvalt met de evaluatie van de kleinst omvattende clause die ook nog een (identity, mode of operation)-declaration bevat. Bijvoorbeeld, als er verder geen deklaraties staan, bestaat het 55-tal gegenereerde variabelen in

for i to 10 do for j to i do ... loc ... od od

ook na afloop van de evaluatie van de for-statement nog.)

Belangrijke onderwerpen uit Hoofdstuk 9

regeling van levensduur en zichtbaarheid gewenst
definities/deklaraties als afzonderlijke syntaktische kategorie:
  privé, simultane en sequentiële definities
  initialisatie- en finalisatie-expressies
Binnenste-buiten-programmering, reversed programming
  de oplossing met funktionele argumenten
module-expressies en reduktieregels

Literatuur bij Hoofdstuk 9

Onze module-expressies zijn een veralgemening van de envelope-constructs van (MacKeag & Welsh 1980). Leerboeken over Ada, Modula 2 en andere moderne programmeertalen bevatten uiteraard ook uiteenzettingen over "modulen".

Oefeningen

9.1! Een stapel met maximale grootte n kan met een array [1..n] of t geïmplementeerd worden. Doe dit met gebruikmaking van prive-definities in Pascal (of in ASS waaraan je voor het gemak Pascal-arrays toevoegt). Denk eraan dat alleen de stapelbewerkingen van buiten af zichtbaar mogen zijn. Geef tevens een omvattend programma schematisch aan, waarbij er twee stapels gebruikt worden.

9.2! Idem als bij de vorige oefening, maar nu met de module-expressies.

9.3! Programmeer een funktie tel = (fn f. (mod ---)) zo dat (inv f = (tel g). prog) hetzelfde effekt heeft als (dfvar n. df f = g. prog; n := "aantal keren dat f is aangeroepen in prog"; n). Veronderstel g niet-recursief.

9.4 Idem als bij Oefening 9.0 en 9.1, maar nu voor lineaire lijsten en queues en bomen, allen geïmplementeerd met behulp van arrays. Enzovoorts, enzovoorts.

9.5! Geef de reduktie van

```
(dfvar s. s := 0;
  (inv x = (mod dfvar i. i := 10; while i>0 do export i end; i := i-1 od)
   . s := s+x
   ); s
)
```

9.6! Is (mod if true then export 0 end else export 1 end) een (legale) expressie?

9.7! Wat is het resultaat van

```
    (dfvar x. x := 0;
     df m = (mod if x eqn 0 then 1 else export 10 end)
     . (inv y = m. 2*y)
     )
```

9.8 Beschouw de definitie-vormen

```
    d ::= (loc x) | (x=e) | (d in d) | (d init e) | (d final e).
```

a. geef een systematische vertaling van definities d naar invokaties inv x = em, (waarbij voor samengestelde d de x en em zijn opgebouwd uit de direkte onderdelen van de vertaling van d) zodat (df d.e) en (inv x = em. e) equivalent zijn. Wenk: vertaal (loc x) in: inv x = (mod dfloc x. export x end).

b. Wat zijn de problemen als (var x) als definitievorm wordt toegelaten?

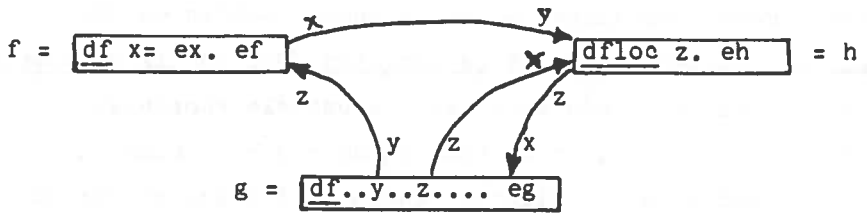
c. Veralgemeen de module-expressies tot (inv (x1, ..., xN) = em. eb) en (mod ---

export (e1, ..., eN) end ---). Geef de vertaling van definitievormen nu ook indien sequentiële en simultane definities zijn toegestaan.

9.9! Bewijs dat de expressie (df dyn-array rec = (---). inv a = (dyn-array 8). ---) inderdaad tot de in Par. 9.2 gegeven expressie reduceert.

9.10 Laat zien dat de envelop-konstruktie een speciaal geval is van de module-konstrukties, (MacKeag & Welsh 1980).

9.11! Beschouw het volgende schema



Ter verklaring: de in f gedefinieerde grootheid x is in h bekend onder de naam y (en wordt in eh gebruikt); net zo voor de andere pijlen.
Geef met behulp van hogere orde funkties en/of module-expressies een stel definities dat de zichtbaarheidsregeling van bovenstaand schema realiseert. Wenk: definieer eerst

f' = (fn z. mod df x=ex. export <ef, x>)

en net zo voor g' en h'; definieer daarna f, g, h, xf, yg, zg, zh simultaan rekursief, onder andere

f, xf = (f' zg)

d.w.z. fxf = (f' zg), f = fxf.1, xf = fxf.2

HOOFDSTUK 10

AFHANDELING VAN UITZONDERINGEN

Afhandeling van uitzonderingen (exception handling), ook wel foutafhandeling (error handling) genoemd, is een onderwerp waarover grote onenigheid is. Er zijn veel voorstanders van speciale taalkonstrukties daarvoor; er zijn ook felle tegenstanders. (Bijna?) alle voorgestelde taalkonstrukties hebben het bezwaar dat ze evenzeer voor "gewone gevallen" gebruikt kunnen worden en dus ten onrechte "uitzonderingen" of "fouten" heten af te handelen. Net zo als de toevoeging van assignment aan een beschrijvende taal naast eventuele voordelen ook zwaarwegende nadelen heeft, zie Par. 7.1, zo moeten ook de voor- en nadelen van een konstruktie voor de afhandeling van uitzonderingen goed tegen elkaar worden afgewogen.

Wij zullen in Par.1..4 een (overtuigend?) betoog houden dat leidt tot een paar nieuwe expressievormen (met hun semantiek), die de essentie weergeven van de meeste taalvoorstellen ten aanzien van dit onderwerp. Een belangrijk bestanddeel van ons betoog is dat we regels aangeven voor de manier waarop die expressievormen gebruikt zouden moeten worden, (ook al wordt dat syntaktisch niet afgedwongen). Vervolgens laten we in Par.5 zien dat die nieuwe expressievormen vaak niet nodig zijn wanneer er geschikte expressievormen voor onderscheiden vereniging (discriminated union) zijn en ook voor lokale beëindigingen (zoals exits uit repetities en returns uit funktierompen).

Zoals gebruikelijk bij gebiedende talen gaan we uit van de kompositionele strategie van evalueren: van links naar rechts en in het algemeen onderdelen eerst. Het staat nog open voor nader onderzoek in hoeverre de nieuwe expressies ook zinvol gebruikt kunnen worden in HOF bij de luie strategie.

Par. 10.1 Fouten in de invoer

Bij de meeste programma's moet de invoer aan bepaalde voorwaarden voldoen (de zgn. prekonditie) opdat de uitvoer de gewenste eigenschappen heeft (de zgn. postassertie), of met andere woorden, de korrektheid van een programma legt ook eisen op aan zijn invoer. Dit geldt ook voor funkties; de invoer wordt gevormd door de argumenten (en de inhoud der globaal gebruikte variabelen), terwijl het resultaat (en ook weer de globale variabelen) de uitvoer vormen. Bijvoorbeeld, de invoermatrix voor een inverteringsfunktie moet niet-singulier zijn, omdat de

inverse anders niet eens bestaat. De schrijver van een programma zal, als het goed is, weliswaar de korrektheid aantonen, maar hij kan onmogelijk bewijzen dat de invoer aan de noodzakelijke voorwaarden voldoet. Dit geldt ook voor de schrijver van een funktie die in een program-o-theek wordt gezet: hij heeft geen mogelijkheid om het gebruik op foutieve invoer te voorkomen. Hij kan hoogstens voorzorgsmaatregelen nemen zodat zoiets niet tot ongewenste resultaten leidt.

Het meest ongewenste resultaat bij foutieve invoer is een resultaat dat ook bij korrekte invoer verkregen had kunnen worden, of dat daar sterk op lijkt. De uitvoer is dan onbetrouwbaar omdat je er niet zeker van kunt zijn of het de uitvoer bij korrekte invoer is of dat er bij de invoer --per ongeluk-- fouten zijn gemaakt. Dit is het geval bij een inverteringsfunktie die een willekeurige matrix oplevert als de invoermatrix singulier is, en bij een fakulteitsfunktie die 1 oplevert bij negatieve argumenten.

Het is daarentegen wenselijk dat de evaluatie bij foutieve invoer leidt tot een van de volgende drie gevallen.

1. Normale beëindiging met (een resultaat dat te interpreteren is als) een foutmelding.
2. Voortijdige beëindiging ofwel een zgn. foutstop, of het stokken van de evaluatie, zo mogelijk met een indikatie van het optreden van de fout.
3. Geen beëindiging, non-terminatie.

Deze gevallen zijn opgesomd in volgorde van afnemende voorkeur. Ook het laatste geval is nog te verkiezen boven een onbetrouwbaar resultaat! Programma's, funkties en in het algemeen programmafragmenten, die dit gewenste gedrag vertonen noemen we robuust ofwel bestand tegen invoerfouten.

Een normale beëindiging met een geschikte foutmelding lijkt in het algemeen moeilijk haalbaar. Want de toetsing op de korrektheid van de invoer moet dan hetzij voorafgaande aan de eigenlijke bewerking gedaan worden, hetzij gedurende en als onderdeel van de bewerking. Beide hebben nadelen. In het laatste geval komt de programmatekst vol met if-then-else konstrukties waarmee de normale voortzetting van de evaluatie wordt voorkomen zodra een invoerfout ontdekt is. Dergelijke programma's zien er weinig appetijtelijk uit. (Dit is een zwak punt in ons betoog; in Par.5 laten we zien dat zoiets nog best aantrekkelijk genoteerd kan worden!) In het eerste geval, de test vooraf, is er dit nadeel: de doelmatigheid van de algoritme (met name de berekeningstijd) dan wel de struktuur van de programmatekst wordt ongunstig beïnvloed. De berekeningsduur kan toenemen omdat de test op korrektheid al vaak besloten ligt in de eigenlijke be-

werking op de invoer. Dit is heel duidelijk het geval bij standaardoperaties zoals de optelling: de som der operanden mag maxint niet overschrijden, maar het is dubbelop omdat voorafgaande aan de optelling te testen omdat het al door de opteloperatie zelf gedaan wordt. Ook bij het inverteren van een matrix blijkt de singulariteit tijdens het gewone inverteringsprokes: doordat een pivot, een deler, (bijna) nul is. Een extra non-singulariteitstest vooraf is dus ondoelmatig. En wanneer je onder behoud van doelmatigheid tokh een test vooraf wil, dan moeten de eventuele tussenresultaten van die test onthouden worden en later gebruikt worden zonder ze opnieuw te berekenen. Dit vergt een strukturele wijziging van het programma. En dat is jammer want kennelijk kan het rekening houden met foutieve invoer niet los van de eigenlijke taak voor korrekte invoer gezien worden. Het zou mooi zijn om eerst een programma te konstrueren voor korrekte invoer en daarna pas voorzieningen aan te brengen voor foutieve invoer; (separation of concerns!).

De konklusie is duidelijk. Er is behoefte aan een expressie waarmee (bij zich openbarende inkorrektheid van de invoer) de normale evaluatie kan worden onderbroken. De meest beheerste vorm van onderbreking is een voortijdige beëindiging, ofwel foutstop. Een expressie daarvoor geven we in de volgende paragraaf. We zouden het stokken van de reduktie als een foutstop kunnen interpreteren (en dus (0-1) of (hd nil) of (<> <>) als die expressie kunnen nemen), maar we zullen het iets algemener houden zodat een voortijdige beeindiging (van een programmadeel) ook weer in een normale voortzetting van de evaluatie (van de rest van het programma) omgezet kan worden. Dat komt in de paragrafen daarna aan bod.

Par. 10.2 Voortijdige beëindiging en robuustheidsvoozieningen

We voegen een nieuwe expressievorm toe aan de taal met een semantiek zo dat evaluatie ervan als voortijdige beëindiging is te interpreteren. Voortijdige beeindiging verschilt van een stokkende reduktie: bij een stokkende reduktie is het resultaat niet gedefinieerd, terwijl na een voortijdige beëindiging er nog wel een resultaat is, namelijk een indikatie dat de reduktie voortijdig geëindigd is.

De expressievorm is als volgt

```
e ::= ... | T
```

We definiëren dat T geen konstante is en op zikhzelf dus geen resultaatvorm, het is dus wel een non-resultaatvorm. (In de tralietheorie worden de symbolen T en T-op-z'n-kop wel als "top" en "bottom" gebruikt. Als de tralie-ordening een in-

formatie-ordening is, geeft top de meeste, dus vaak een teveel of inkonsistente, informatie aan en bottom de minste, dus vaak een te weinig ofwel ongedefinieerde informatie).

De informele interpretatie van T leggen we als volgt in reduktieregels gedeeltelijk vast. We voegen een heel stel reduktieregels toe zo dat als een samenstellend deel van een expressie T is ook de hele expressie tot T kontraheert. (Sommige regels zullen door een strategie misschien nooit gekozen worden). Dit geeft ons bijvoorbeeld:

(T+) (T + e) -> T, (e + T) -> T
(T;) (T; e) -> T, (e; T) -> T
(Tfn) (fn x. T)-> T

In het algemeen, bij een expressie met n samenstellende subexpressies komen er n (T)-regels. Het is duidelijk dat bij een geschikte strategie een eenmaal voortgebrachte T niet meer verdwijnt en dus voortijdige beëindiging goed modelleert. In die zin zullen we nu de strategie aanpassen:

als, zonder deze aanpassing, een subexpressie T wordt gekozen ter kontraktie (en dus de reduktie zou stokken), dan moet nu de kleinst omvattende expressie {tot T} gekontraheerd worden volgens een van de (T)- regels (zonder dat de onderdelen daarvan eerst nog gereduceerd worden).

Nu kan een programmeur met behulp van T een voortijdige beëindiging programmeren op die plaatsen waar invoerfouten eenvoudig te testen zijn. (Wellicht dat de taalontwerper de semantiek ook iets wijzigt, door namelijk sommige regels zoals

(m - n) -> T voor m<n
(<e1, e2> ea) -> T
(hd nil) -> T

toe te voegen. We hebben dan voortijdige beëindiging in plaats van een stokkende reduktie.) Programma's kunnen nu robuust gemaakt worden: is het niet door een normale beëindiging met een foutmelding als resultaat, dan toch wel door voortijdige beëindiging.

Stel dat de voorzieningen voor robuustheid pas getroffen worden nadat een korrekt programma (voor korrekte invoer) is gekonstrueerd. Het is dan prettig als die toevoegingen daarvoor syntaktisch te herkennen zijn. Bij het lezen van de programmatekst kunnen zij dan overgeslagen worden, en als op andere gronden bekend is dat de invoer altijd korrekt zal zijn, dan kunnen die toevoegingen ge-

makkelijk opgespoord en weer verwijderd worden. Men kan robuustheidsvoorzieningen met een extra hakenpaar [en] aangeven; door weglaten van die haken en de ingesloten tekst moet er een legale en korrekte expressie overblijven. Heel vaak komen die haken voor in de konteksten

```
[if e1 then e2 else] e
[if e1 then e2 fi;]  e
```

Expressie e1 is dan een test op invoerkorrektheid en e2 zal menigmaal louter T zijn. Bijvoorbeeld, bij een robuuste fakulteitsfunktie:

```
fac rec= fn n. [if n<0 then T else]
                  if n=0 then 1 else n * fac(n-1)
```

Het is overigens opmerkelijk dat bij guarded commands en guarded expressions robuustheid en korrektheid grotendeels hand in hand gaan. De te volgen methode is: kies de bewakers bij een if-konstruktie zo sterk mogelijk en bij een do-konstruktie zo zwak mogelijk. Vergelijk het volgende paar programma's ter berekening van n! in de scope van dfvar f, i := 1, 0:

```
(a)  do i≠n -> f, i := f*(i+1), i+1 od; f
(b)  do i<n  -> f, i := f*(i+1), i+1 od; f
```

De konditie i≠n is zwakker dan i<n, bijgevolg is versie (a) robuuster dan (b): bij negatieve n levert (b) het onbetrouwbare resultaat 1 op, terwijl (a) niet termineert. (Overigens, om terminatie te bewijzen moet o.a. de invariantie van i<n aangetoond worden, en in konjunktie met deze invariant zijn beide bewakers even sterk: beide programma's zijn (even) korrekt. Uit de prekonditie i<n (nodig voor de invariantie van i<n) en de initiële waarde i=0 volgt dat negatieve waarden voor n tot foutieve invoer gerekend moeten worden!) Iets soortgelijks geldt ook bij

```
(c)  rec x. if i<n -> f, i := f*(i+1), i+1; x
            |  i=n -> f
            fi
(d)  rec x. if i<n -> f, i := f*(i+1), i+1; x
            |  i>=n-> f
            fi
```

Versie (c) is robuust, terwijl (d) dat niet is; de bewaker i=n in (c) is inderdaad sterker dan i>=n in (d). De robuuste (c) is zelfs te verkiezen boven de robuuste (a) omdat (c) bij negatieve n tot het stokken van de berekening leidt (of

voortijdige beeindiging, afhankelijk van de precieze semantiek voor guarded commands), terwijl (a) slechts nonterminatie geeft.

Merk voorts op dat T niet door de programmeur geschreven hoeft te worden indien de semantiek van de if-guarded command T geeft wanneer geen der bewakingskondities vervuld is. De expressie if false -> empty fi, of zelfs if fi, geeft dan T. Ga ook na dat een if-then-else gezien kan worden als een if-guarded command met een heel zwakke laatste bewaker; de if-then-else-expressies lenen zich dus minder voor de konstruktie van robuuste programma's dan de if-guarded commands.

Par. 10.3 Afhandeling van voortijdige beëindiging

Beschouw eens het volgende probleem. Gegeven een rij invoergetallen; gevraagd de rij van de fakulteiten van die getallen. Stel dat de taal ook negatieve getallen kent en dat we een robuuste fakulteitsfunktie hebben die bij een negatief argument voortijdig eindigt. Dan is een korrekt programma f voor dat probleem, dat tevens robuust is, snel geschreven:

```
f rec= fn invoer. if eqnil invoer then nil
                 else fac (hd invoer): f (tl invoer)
```

of konventioneler

```
f = fnvar (invoer, uitvoer).
      while not eqnil invoer
         do uitvoer := uitvoer ++ [fac (hd invoer)];
            invoer := tl invoer
         od
```

Bij een negatief getal op de invoer heeft het voortijdige einde van een aanroep van fac ook het voortijdige einde van f tot gevolg. Omdat f robuust is, mogen we er niet ontevreden mee zijn als oplossing voor het gestelde probleem.

Na deze bevredigende oplossing is het niet onredelijk als alsnog de taakstelling voor het programma iets wordt uitgebreid. De opdrachtgever zal zich bijvoorbeeld realiseren dat hij nog tevredener is naarmate hij bij meer foutieve invoergetallen toch zinvolle antwoorden krijgt in plaats van louter voortijdige beëindiging. Het getal 0, de waarheidswaarde false of de tekst "negatief invoergetal" is bijvoorbeeld een zinvol antwoord bij een foutief invoergetal. Merk op dat een korrekt programma voor deze uitgebreide probleemstelling nog steeds een

korrekt en robuust programma is voor het oorspronkelijke probleem; met name 0 is niet te verwarren met de fakulteit van een natuurlijk getal.

We geven nu een nieuwe expressievorm waarmee bovengewenste aanpassing van het programma eenvoudig te realiseren is: een expressievorm voor de omzetting van een voortijdige beëindiging in een gewone voortzetting van de evaluatie. De syntaxisuitbreiding luidt:

e ::= ... | e[=>e]

(We zullen de vorm e[=>e] in de volgende paragraaf nog veralgemenen tot e[e=>e].) We noemen e[=>e'] een afhandelingsexpressie en e' het afhandelingsdeel. De reduktieregels zijn

(T-afh) T[=>e'] -> e'
(afh) e[=>e'] -> e voor e in resultaatvorm.

(De afkorting afh komt van afhandeling.) Bij alle strategieën moet het afhandelingsdeel e' in e[=>e'] (en in e[e"=>e']) net zo'n rol vervullen als de then- en else-tak van een if-expressie: het wordt niet gereduceerd alvorens het geheel volgens (T-afh) is gekontraheerd.

De gewenste aanpassing van programma f bestaat nu uit de achtervoeging van [=>0] aan de aanroep (fac (hd invoer)). De voortijdige beëindiging wordt dan omgezet in een passende gewone voortzetting van de evaluatie.

Alweer geldt dat weglating van alle tekst tussen en inklusief de haken [en] een programma overlaat dat nog steeds korrekt is ten aanzien van de oorspronkelijke specifikatie. Maar pas op, in het algemeen geldt dit alleen maar wanneer de programmeur de afhandelingsexpressies volgens de beschreven methodologie gebruikt. Er staat hem niets in de weg om ze ook anders te gebruiken. Beschouw bijvoorbeeld de volgende funktie die op een globale var input werkt.

```
read = (fn (). if eqnil input then T
               else df x = hd input. input := tl input; x)
```

Door afhandeling van de door read veroorzaakte voortijdige beëindiging kunnen we een "oneindige herhaling" laten eindigen:

```
dfvar s.
    s := 0;
    (rec herhaal. s := s+read(); herhaal)[=>empty];
    s
```

Het behoeft geen betoog dat deze herhaling "minder duidelijk" maar wel efficiënter is dan

```
(rec herhaal. if eqnil input then empty
              else s := s+read(); herhaal)
```

Het ontwerp van read is dus te bekritiseren; maar wellicht dat de ontwerper van read in de verleiding is gebracht om het zo te doen als boven, door de aanwezigheid van T en de afhandelingsexpressie in de taal.

Par. 10.4 Onderscheiding van voortijdige beëindigingen

We veralgemenen de expressievormen T en e[=>e'] nu zo dat voortijdige beeindigingen kunnen worden onderscheiden. Daardoor is het onder andere mogelijk om de oorzaak van de voortijdige beëindiging enigszins aan te geven en de afhandeling te laten afhangen van de oorzaak. Op zich weer geen onredelijke wens.

We zullen variabelen (identifiers) x ter identifikatie gebruiken. Om lokale naamgeving mogelijk te maken, d.w.z dat alfabetische variaties mogelijk zijn (zonder dat de semantiek wijzigt), moeten identifikaties ook een bindend voorkomen hebben. Daarvoor kiezen we de volgende expressievorm (exc staat voor exception identifikatie):

```
(dfexc x. e)
```

Dit is een bindingsexpressie met binder dfexc en e als bereik van de binding dfexc x. Z'n enige rol is de x lokaal te maken, dus semantisch heeft die expressie nauwelijks enig effekt. De reduktieregel luidt

```
(dfexc)  (dfexc x. eb) -> eb    mits x niet vrij in eb
```

Bij de kompositionele strategie moet eb eerst tot resultaatvorm gereduceerd zijn alvorens deze regel mag worden toegepast. In de scope van dfexc x is x zelf ook een resultaatvorm. (Dit is geheel analoog aan de formele semantiek voor de var- en loc-expressies, zie Hoofdstuk 7).

De expressievormen T en e[=>e'] veralgemenen we nu tot T[e"] en e[e"=>e']. Het is dan de bedoeling dat e" een identifikatie x is of daartoe evalueert. Dus de waarde van e" (een identifikatie x) onderscheidt de voortijdige beëindigingen; een achtervoegsel [x=>e'] handelt alleen de met x gemerkte voortijdige beeindigingen af. De reduktieregel (afh) luidt nu dus:

(afh) e[e"=>e'] -> e voor e in resultaatvorm
(T-afh) T[x][x=>e'] -> e'
(T-afh) T[x][y=>e'] -> T[x] voor y≠x

(Eigenlijk moet in de laatste twee regels nog worden aangegeven dat de redex in de scope van dfexc x en dfexc y valt; vergelijk reduktieregels (:=) en (?)uit Hoofdstuk 7.) Bovendien moeten ook de (T)-regels veralgemeend worden: wanneer een samenstellend deel T[x] is, dan kontraheert het geheel tot T[x]. En tenslotte moet ook de kompositionele strategie passend veralgemeend worden: wanneer een deel T[x] ter reduktie zou worden gekozen, dan moet de kleinst omvattende expressie volgens een (T)-regel of volgens (T-afh) gekontraheerd worden. Het deel e' in e[e"=>e'] speelt net zo'n rol als een then- of else-tak: daarbinnen worden geen kontrakties gedaan. Het deel e" in e[e"=>e'] wordt eveneens niet tot resultaatvorm gekontraheerd indien regel (afh) dat niet eist, i.e. indien e gewoon termineert.

We geven nu twee voorbeelden; één waarin verschillende identifikaties een rol spelen, en één waarin de vorm T[e"] en e[e"=>e] wordt gebruikt zonder dat e" een identifier is.

De probleemstelling voor het eerste voorbeeld is om van een gegeven invoerrij de fakulteiten te sommeren. Wanneer de som maxint zou overschrijden, willen we maxint zelf als resultaat, en wanneer er een negatief getal op de invoer staat willen we een voortijdige beeindiging gemerkt met 'invoerfout'. We gaan uit van de volgende standaard omgeving.

```
dfexc overflow, neg, invoerfout
df plus = {optelling; geeft mogelijk T[overflow]}
df fac  = {fakulteit; geeft mogelijk T[neg]}
dfvar input {bevat een lijst van invoergetallen}
```

We zouden dan het volgende programma kunnen geven

```
(dfexc none.
 df next = (fn (). if eqnil input then T[none]
```

```
                    else df x = hd input. input := tl input; x).

    dfvar s.
        s := 0;
        (rec loop. s := plus (s, fac (next ())); loop)[none=>empty];
        s
    )[overflow=>maxint, neg=>T[invoerfout]]
```

Nu het tweede voorbeeld. We geven allereerst een module voor een stapel. Behalve de gebruikelijke push en pop is er ook een exception identifikatie emptyPop. De module luidt als volgt.

```
df stapel =
  (mod dfexc emptyPop. dfvar s.
       s := nil;
       export <{push} fn x. s := x: s
              ,{pop} fnvar x. if eqnil s then T[emptyPop]
                              else (x := hd s; s := tl s)
              , emptyPop
              >
       end[emptyPop => T[no-handler-present]];
       if not eqnil s then print ("stack not emptied") fi
  )
```

Deze definitie moet in de scope liggen van een (standaard) dfexc no-handler-present. Door middel van de 3de komponent heeft de invoker van deze module kennis van de exception identifikatie die de module ontwerper bedacht heeft. Een invokatie kan er als volgt uit zien.

```
(inv stackADT = stapel.
 --- (stackADT.2 z)[stackADT.3 => z:=0] ---
)
```

Bij de aangegeven aanroep van stackADT.2 (= pop) wordt kennelijk een lege stapel geïnterpreteerd als een niet-lege die een 0 op de top heeft. De exception identifikatie stackADT.3 kan niet alleen in afhandelingen gebruikt worden, maar ook in veroorzakingen T[stackADT.3] van voortijdige beëindigingen door de invoker. (Was dat niet de bedoeling geweest, dan had de module schrijver de export-expressie maar anders moeten kiezen. Zie Oefening 10.3). Wanneer de invoker de emptyPop niet afhandelt, dan gebeurt dat in de module zelf! En wanneer de invoker de stapel niet leeg achterlaat, dan wordt daarvan een melding gegeven die in de

module geprogrammeerd staat!

* * *

Hiermee zijn de expressies (dfexc x. eb), T[e] en e[e"=>e'] volledig gedefinieerd en toegelicht. Dergelijke konstrukties --uiteraard met de nodige verschillen en verdere syntaktische suiker-- staan in de literatuur bekend als exception handling mechanisme. Andere notaties zijn bijvoorbeeld

```
raise x                          voor T[x]
(def exproc x = e'. eb)          voor eb[x=>e']
eb except when x : e' end        voor eb[x=>e']
```

en

```
x                                voor T[x]
(but for x : e' do eb)           voor eb[x=>e']
```

en

```
exit                             voor T
trap eb with e'                  voor eb[=>e']
```

Soms kunnen de e" in T[e"] en eb[e"=>e'] ook van argumenten respektievelijk formele parameters worden voorzien. Het aantal en het type der parameters kan dan bij de binding dfexc x worden vastgelegd.

Een ogenschijnlijk minder krachtig mechanisme dan onze exception handling is het zogenaamde escape mechanisme. Escape funkties zijn funkties die bij aanroep het blok waarin zij lokaal gedeklareerd zijn beëindigen. Zij vormen dus een veralgemening van returns uit procedure-rompen en exits uit herhalingen. Als we even het exception handling mechanisme met parameters uitbreiden, dan worden escape funkties als volgt met exception handling gemodelleerd.

```
(dfesc f = (fn x. eb).  e)
```

komt overeen met

```
(inv f = escape (fn x. eb). e)
waarbij
escape = (fn g. mod dfexc exc.
                        export (fn y. T[exc(y)])
                        end [exc(z) => g(z)])
```

Na lezing van de volgende paragraaf zal duidelijk zijn dat escape funkties een ("beheerster") alternatief bieden voor exception handling. (Een modellering in

termen van de exception handling konstrukties is dan minder op z'n plaats dan een direkte formulering van de semantiek, zie Oefening 10.4)

Par. 10.5 Een alternatief voor exception handling konstrukties

We verwijzen hiervoor naar

M. Fokkinga: Exception handling constructs considered unnecessary. Memorandum INF-84-4. T.H. Twente, April 1984. (zie pag. 197.1 t/m 197.11)

EXCEPTION HANDLING CONSTRUCTS CONSIDERED UNNECESSARY

Maarten M. Fokkinga
Twente University of Technology
Enschede, Netherlands

Abstract In his Ph.D. Thesis "Exception handling: the case against" (Univ. of Oxford, January 1982) Andrew P. Black argues that normal control structures, together with the data type discriminated union, suffice to replace exception handling facilities in a satisfactory way. We want to propagate some of his ideas with an example. Moreover we show that (i) even with the discriminated union approach incremental program construction is still possible, and that (ii) the programs using discriminated unions resemble the programs using exception handling facilities more than Black suggests.

1. Introduction

In his Ph.D. Thesis Black gives a thorough analysis of what usually is meant by "exception" and "exception handling", and he also shows how the examples which are used to motivate exception handling facilities can be programmed by normal constructs. I will not repeat his arguments but for the treatment of one example which clearly shows the most important technique. In this way I hope to propagate some of his ideas.

The second aim of this paper is the following. One of the advantages of exception handling facilities is claimed to be the possibility of incremental program construction. That is, first a program is constructed which is correct for the normal case, i.e. assuming that the input entities satisfy the precondition, and then the program is adapted so that the exceptional inputs are dealt with satisfactoryly too; see (Bron & Fokkinga 1977). This aspect of exception handling, facilitating a separation of concerns, is hardly considered by Black in his Thesis. We show that incremental program construction is also possible with the normal constructs advocated by Black to replace the exception handling constructs.

Our third aim is the following. We propose a suitable syntax for the operations of a discriminated union so that the elimination of exception handling constructs brings about only minor modifications of the program texts;

in our example the texts with and without exception handling constructs resemble each other more than in a similar example treated by Black. To be fair, however, we note that Black already hinted at such syntactic sugar.

******

Black gives various techniques to handle exceptions using normal constructs, the most important of which is the discriminated union. Discriminated unions look like the union construct of Algol 68 and the variant record of Pascal; they are explained in detail in Section 3. Another construct which is sometimes needed, is the procedure as parameter; in our example we can do without it. Finally, often "local termination" is needed, that is, a construct to terminate a textually enclosing program fragment, like statements, blocks, repetitions and procedure bodies. Most programming languages contain some such constructs: exists, returns, leaves, Zahn's events, and so on. It is like hitting a mosquito with a sledge-hammer if one introduces exception handling constructs solely for local terminaton. Indeed, it is generally the purpose of "raising an exception" to terminate a chain of dynamically invoking (rather than textually enclosing) blocks.

It turns out that the main technique to eliminate exception handling constructs boils down to the following. First, the dynamically nested block incarnations are now terminated locally; each one separately from the others. Second, the case analysis (exception analysis and handling) is now written precisely at the place where the exception is detected "by the user" (after suitable notifications by invoked procedures), rather than at the end of the block which is to be terminated. Thus the program becomes now more "structured" in the sense that more of its behaviour (more of its correctness proof) can be deduced by local inspection of its text.

The remainder of the paper is organized as follows. First, in Section 2 we show the example program using exception handling constructs; we take this program for granted. In Section 3 the exception handling constructs are eliminated; discriminated union is introduced and explained. Some further elaborations are given in Section 4 and we conclude with Section 5.

2 An example with exception handling

We consider the following problem statement. "Construct a procedure sum

which yields as its result the sum of several numbers, the denotations of which are placed on the input separated by one or more spaces. For the conversion of a string (a denotation) to its integer value the procedure s2i (short for: string to int) is to be used".

To be more specific we assume the following context for the definition of procedure sum. There are several exceptions, some of which are used by s2i and +.

```
exc overflow, badformat(char), fatal-error, too-large;
proc s2i: (string --> int) possibly raising badformat, too-large;
operation +: (int, int --> int) possibly raising overflow;
```

Procedure s2i raises the exception badformat(c) if its argument is not a proper denotation; c equals the first invalid character. If on the other hand the number represented by the denotation exceeds maxint, the exception too-large is raised. Similarly, + raises overflow if the result would exceed maxint. Exception fatal-error is a standard one; we assume that a programmer cannot handle it, i.e. if it is raised the complete program is terminated (and handled by the operating system).

See figure 1 for the definition of sum; some explanatory notes follow.

Notes.

1. We do not claim that the program is elegant; we only show it to illustrate some use of exception handling.
2. On the second line a new exception is introduced: none. It is raised in the middle of readint where it turns out that there is no more denotation on the input. In the third line from below the while-statement is possibly terminated due to the raising of none from within readint; the handling of that exception consists of doing skip followed by normal continuation of the execution after the while-statement.
3. In the heading of readint it is not specified that it possibly raises the exceptions too-large and bad-format. Therefore these exceptions are by default reraised at the end of readint as fatal-error. In other words, the end of readint actually reads:

   ```
   end[badformat(c) => raise fatal-error, too-large => raise fatal-error]
   ```

4. Similarly, overflow exceptions from within the body of sum are reraised by default as fatal-error.

```
proc suml: int;
   exc none;
   var s: int;
   proc readint: int possibly raising none;
        var denot: string;
        begin skip spaces:
                  while not eof(input) cand input↑ = space
                  do get(input)
                  od;
              if eof(input) then raise none;
              accumulate denotation:
                  denot := input↑; get(input);
                  while not eof(input) cand input↑ /= space
                  do denot := denot concat input↑;
                     get(input)
                  od;
              return s2i (denot)
        end readint;
   begin s := 0;
         while true do s := s + readint od[none => skip];
         return s
   end
```

figure 1

One should note that during the construction of the program only the normal situation is taken into account. An exceptional input is not at all considered, and the programmer need not even be aware of the fact that + and s2i possibly raise exceptions. Nevertheless the program is robust and reliable: erroneous input does not lead to unreliable results.

Now we want to deal with exceptional inputs too. First the requirements for sum are extended. If too large a denotation is encountered, or the sum of the numbers exceeds maxint, exception overflow is to be raised. And if a denotation in bad format is encountered, a new exception badf(d) is to be raised, where d is the invalid denotation, i.e. a string. In a previous paper (Bron & Fokkinga 1977) we have argued that exception handling will do well for this purpose. Indeed, we may adapt the program by inserting additions only; nothing of the original text needs to be rewritten or changed. The new text is shown in figure 2.

```
exc badf (string);
proc sum2: int possibly raising overflow, badf ;
     exc none;
     var s: int;
     proc readint: int possibly raising none , too-large, badf ;
          var denot: string;
          begin skip spaces:
                ------------
                if eof(input) then raise none;
                accumulate denotation:
                     ------------
                return s2i (denot) [bad format (c) =>
                                    raise badf (denot)]
          end readint;
     begin s := 0;
           while true do s := s + readint od [none => skip];
           return s
     end [too-large => raise overflow]
```

figure 2

Even if sum is to deliver maxint as result rather than raising overflow, it is easy to adapt the program. This, and similar variations, are left to the reader.

3. Exception handling eliminated

We now show how the handling of exceptions can be expressed using normal programming language constructs. For our example we need constructs to terminate a textually enclosing block and repetition, and the discriminated union. We first explain our syntax and the semantics of the discriminated union, and then present the new formulations of sum1 and sum2, called sum1′ and sum2′. It should be obvious from the strong resemblance with the old versions that exactly the same reasoning and methodology can be applied to derive sum1′ and sum2′, or to adapt sum1′ to sum2′, as was done before with sum1 and sum2.

The discriminated union is a data structure with the following operations: injection (postfix ↑i), projection (postfix ↓i), inspection (postfix ?i) and case selection (postfix ?[f1,..., fn]). Their semantics is explained by the following axioms; i and j are constants 1, 2, 3, ...

$$\text{expr}\ \uparrow i\ \downarrow j = \begin{cases} \text{expr} & \text{if } i=j \\ \underline{\text{fatal error}} & \text{if } i\neq j \end{cases}$$

$$\text{expr}\ \uparrow i\ ?j = \begin{cases} \text{true} & \text{if } i=j \\ \text{false} & \text{if } i\neq j \end{cases}$$

$$\text{expr}\ \uparrow i\ ?[f1,\ldots,fn] = fi(\text{expr})$$

The type of a discriminated union is written as t1+...+tn where t1,...,tn are the types of its summands. So e.g. an injection expr↑i is well-typed only if expr has type ti and the whole has type t1+..+ti+..+tn, for some types t1,...,tn.

For the sake of readability we often give "summand identifiers" in the type (like field identifiers in records) and use them instead of 1,2,3,... in the operations; and if possible we also label the cases of a case selection with the summand identifiers. Moreover we define two abbreviations (coercions, syntactic sugar) in order that the use of discriminated unions can compete notationally with exception handling constructs: the operations ↑1 and ↓1 need not be written, but are automatically inserted (by the compiler) if the context so requires. Note that expr↓1↑1 may be written expr, and yields fatal error if expr?1=false.

We are now ready to present the transliteration, eliminating the exception handling constructs and using the discriminated union and exits and returns instead. For simplicity we use labelled blocks and assume that a $\underline{\text{return}}_L$ terminates the textually least enclosing block labelled with L. (A more general approach would use escape procedures instead of returns and exists; an invocation of an escape procedure terminates the textually least enclosing block in which it has been declared.) The program text is given in figure 3; it is to be interpreted in the following context.

```
proc s2i: (string --> int + bf: char + too-large: void)
oper + : (int, int --> int + ovf: void)
```

Void is a type with only one element, denoted by: empty.

```
proc suml': int;
    var s: int;
    proc readint: int + none: void;
         var denot: string;
      A: begin skip spaces:
                  ----------
              if eof(input) then return_A empty ↑ none;
              accumulate denotation:
                   ----------
              return_A s2i(denot)
         end readint;
 B: begin s:=0;
         while true
         do s := s + readint?[i: i, none e: exit]   (*)
         od;
         return_B s
    end
```

(*) Notation. The texts "i: i" and "e: exit" are notations for functions with formal parameter i resp. e and body i resp. exit. Note that the second case is labelled with summand identifier 'none', see the type of readint.

figure 3

Note that s2i(denot) actually means s2i(denot) ↓1↑1, so that fatal error results when denot has a bad format or represents too large an integer. A similar remark applies to (s + readint?[...]): overflow results in fatal error. Note also that the programmer needs only know, and take account of, the first summand of the result types of s2i and +.

Again we now may extend the specification of the procedure, and extend the program text accordingly, so as to take exceptional inputs into account. Procedure sum2' is given in figure 4.

Especially from figure 4 and 2 it is clear that the elimination of exception handling constructs gives rise to longer program texts. One reason is that the flow of control is expressed explicitly; termination of the chain of dynamically invoking blocks is programmed by several local terminations of only textually enclosing blocks. Another reason is to be found in the syntax for case

```
proc sum2': int + ovf: void + badf: string ;
    var s: int;
    proc readint: int + none: void + too-large: void + badf: string ;
        var denot: string;
      A: begin skip spaces:
                 ---------
             if eof(input) then return_A empty ↑ none;
             accumulate denotation:
                 ---------
             return_A s2i(denot)   ?[i: i
                                   , bf c: denot ↑ badf
                                   , too-large e: e ↑ too-large
                                   ]
        end readint;
  B: begin s := 0;
        while true
        do s := (s + readint?[i: i
                              , none e: exit

                              , too-large e: return_B e ↑ ovf
                              , badf s: return_B s ↑ badf

                              ]
                 ) ?[i: i, ovf e: return_B e ↑ ovf]
        od;
        return_B s
    end
```

figure 4

selection; our syntax requires that each case is treated separately and explictly, in contrast to the exception handlers. Some abbreviation is quite well conceivable; after all, seven of the nine cases are merely an identity or an identity followed by a return.

One may also dislike the exists and returns out of subexpressions. Rightly so. It is caused by our wish to take procedure sum1 and sum2 as a starting point. In those programs the exits and returns exist as well, but they are not written explicitly! Our aim was to simulate sum1 and sum2 as precise as

We found it furthermore quite surprising to observe that auxiliary variables like s and denot kept their original types. At first we had expected that some of them should get a discriminated union type. By now it is clear that no such thing will happen due to the elimination of the exception handling constructs. (However, such types may appear of course during programming if the programmer so wishes. Discriminated unions are a normal data structure, like arrays and records).

By the way, in practice the handling of exceptions may be less refined than suggested by procedures sum1 and sum2. For example consider a program or module in which exception overflow, caused from within the standard +-operation, is handled only at the outermost block B. If the programmer now uses

```
operation +: (int, int --> int + ovf: void),
```

then it seems as if he needs to write a case selection at each + and procedure invocation in order to propagate the untimely termination up to the outermost block B. There is however a better way; the programmer should redefine the +-operation (only once) so that (x+y) has the effect of

```
    var result;
A:  begin result := (x+y)?[i: i, ovf e: return_B]
          return_A result
    end .
```

In this way the programs with and without exception handling constructs resemble each other again very much. (An impossibility to redefine operations is a weakness and irregularity of the language design and surely not a motive to introduce exception handling constructs.)

4. Some further elaborations

We first give an exercise for the reader and then discuss alternative ways for writing s := s + readint. Nothing new is explained in this Section.

In our example we have used both get and eof to operate upon the input. These two procedures might be replaced by just one: readchar, resulting in an exception end-of-file if no more character is present. We leave the adaptations of sum1 and sum2, and the transition to sum1' and sum2', as an exercise to the

reader. (Notice that now there is a greater similarity between readchar and readint. However one could as well replace readint by a pair of procedures, int-eof and int-get say, which cooperate via a private look-ahead variable. This is another technique advocated by Black to eliminate exception handling constructs.)

Now we consider the statement s := s + readint. Assume for simplicity that readint does indeed yield an integer as result --without exceptions--. Suppose we had written s +:= readint, and we wanted to eliminate the exception handling of overflow caused by the + operation. The problem is where to put the case-selection ?[i: i, ovf e: return e ↑ovf] without changing anything of the given program text. The solution is simple. Together with the abbreviation s +:= expr for s := s + expr, one should also devise an abbreviation for s := (s + expr)?[...], for example s +:=?[...] expr. Thus incremental programming is still possible.

The reader may now wonder how to deal with plus-and-becomes (s, readint) instead of s := s + readint. There is no problem here too. Indeed, let plus-and-becomes be defined as follows.

```
proc plus-and-becomes: (var x: int, e: int --> void)
                          possibly raising overflow;
begin x := x + e;
      return empty {a void result!}
end
```

According to our elimination scheme this is translated as follows.

```
proc plus-and-becomes: (var x: int, e: int -->
                          {normal:} void + ovf: void);
begin x := (x + e)?[i: i, ovf e: return e ↑ovf];
      return empty {with the coercion: ↑1}
end
```

writing in suml′ respectively in sum2′:

- plus-and-becomes(s, readint) {with the coercion: ↑1}
- plus-and becomes(s, readint) ?[e: e, ovf e: return e ↑ovf]

Again we see that the exceptional termination of the body of plus-and-becomes is

programmed explicitly, rather than implicitly via the raise of an exception from within the dynamically enclosed (i.e. invoked) +-operation.

5. Conclusion

Following (Black 1982) we have shown by means of one nontrivial example how constructs for local termination and discriminated union may replace exception handling constructs. The main characteristic feature is that the dynamically determined jumps of control are replaced by accumulating textually determined control jumps. This slightly increases the text, but may have its benefits with respect to readability and efficiency.

The scheme to replace exception handling constructs appears quite uniformly applicable. There seems however one important exception, namely when many different exceptions are handled by one handler. This is possible in Ada via an others clause and in Modular Pascal (Bron & Dijkstra 198x) via x-any (a pseudo-identifier for any exception). The technique of Section 3 gives rise to a many-fold duplication of text and is therefore unattractive. It is open for further study whether there exist other satisfactory techniques or programming methods which are applicable in such cases.

Black argues that the notion of "exception" is ill-defined and that the programmer should take account of exceptional situations right from the beginning, thus reducing their status to normal situations. We do not want to discuss this position, but only remark that incremental programming (i.e. taking care of exceptional situations only after a correct program for the normal cases has been constructed) is as feasible with exception handling constructs as with their replacements.

References

Black, A.P.: Exception handling, the case against.
Ph.D. Thesis, University of Oxford, 1982, 238 pages.

Bron, C. & Dijkstra, E.J.: On the use of exception handling in Modular Pascal.
State University Groningen, Netherlands, 198x.

Bron, C. & Fokkinga, M.M.: Exchanging robustness of a program for a relaxation of its specification. TW-Memorandum nr 178, September 1977, Twente University of Technology, Netherlands.

Belangrijke onderwerpen uit Hoofdstuk 10

robuust = bestand tegen invoerfouten
voortijdige beëindiging = foutstop ≠ stokkende berekening
robuustheid en guarded commands/expressions
scheiding van korrektheids- en robuustheids-argumentatie en -voorzieningen
T en de reduktieregels
afhandelingsexpressies en de reduktieregels
onderscheiding van voortijdige beëindigingen
bedoeld en "oneigenlijk" gebruik van "exception handling"
escapes, returns, exits
onderscheiden vereniging
exception handling en modulen

Literatuur bij Hoofdstuk 10

Een overzichtsartikel is (Goodenough 1975); zie voorts de leerboeken over Struktuur etc. van Programmeertalen van na 1980. Bewijsregels voor exception handling worden o.a. gegeven in (Fokkinga 1978) en (Bron & Fokkinga 1977). (Black 1982) geeft konventionele alternatieven voor exception handling. Landin's J-operator (Landin 1964, Burge 1975) vormt een heel "krachtige" (=onbeheerste?) veralgemening van de voortijdige beëindiging.

Oefeningen

10.1! Met welk van de gevallen (a)..(d) uit Par. 10.2 komt het volgende overeen?
(e) rec x. if i<n then f, i := f*(i+1), i+1; x else f

10.2! Vergelijk (f) hieronder met (c) van Par. 10.2. De bewakingskonditie i≠n is zwakker dan i<n; is (f) ook robuust of niet?

```
(f) rec x. if i≠n -> f, i := f*(i+1), i+1; x
           | i =n -> f
           fi
```

10.3! Definieer een module stapel zo dat de invoker geen voortijdige beëindigingen T[emptyPop] kan veroorzaken maar wel kan afhandelen; vergelijk Par. 10.4. Geef ook een voorbeeld van het gebruik ervan. Wenk: exporteer in plaats van emptyPop een funktie f zo dat (f e e´) overeenkomt met e[emptyPop=>e´]. Denk eraan dat de parameters van f name-parameters moeten zijn; zie Par. 2.2.

10.4! Definieer de semantiek van escape funkties direkt in termen van reduktieregels, resultaatvormen (en de strategie).

10.5! Hoe kunt u robuuste programma´s schrijven zonder van T gebruik te maken? Wenk: beschouw (rec x. x) in plaats van T.

10.6! Schrijf een robuust programma dat bij invoerrij g1: g2: g3: ... de uitvoerrij g1 div g2: g2 div g3 ... voortbrengt. Welke invoer is foutief? Breidt de taakstelling zo uit dat ook voor foutieve invoer nog een zinvol antwoord opgeleverd wordt. Pas uw programma daaraan aan. Doe dit zowel met als zonder exception handling, of onderscheiden vereniging.

10.7 Doe de oefening van Par. 10.5.4.

10.8! Beschouw HOF uitgebreid met de expressievormen T en e[=>e´] en hun reduktieregel. Geef een expressie die bij de U- of L-strategie 0 oplevert en bij de C-strategie 1. (Dus de volgorde van evalueren is bij afhandelingen kruciaal geworden, net zo als bij assignment.)

10.9! Geef voor HOF of ASS (met exception handling) extra reduktieregels zodat van geen enkele expressie de reduktie nog stokt, (maar in plaats daarvan wel voortijdig eindigt).

10.10 Schrijf een programma voor het volgende probleem. Gegeven is een lijst van haakjes die netjes gepaard voorkomen. Bijvoorbeeld: {()}[()({[]}{})]. (Representeer de haakjes eventueel met getallen of zo.) Gevraagd een kopie van die lijst, waarbij een lijst slechts element voor element gekopieerd kan worden. Maak vervolgens uw programma robuust; onderscheid zoveel mogelijk verschillende soorten invoerfouten. Is het nu gemakkelijk om uw programma aan de volgende specifikatie aan te passen? Bij foutieve invoer moeten "extra" haakjes in de uitvoer worden toegevoegd, of moeten overbodige haakjes van de invoer worden overgeslagen, zodat de uitvoer zeker wel uit netjes gepaarde haakjes bestaat. Is uw aangepaste programma nog korrekt en robuust voor het oorspronkelijke probleem?

10.11! Beschouw de volgende definities.

```
df fac rec= fn n. if n<0 then T else
                    if n=0 then 1 else n * fac(n-1)

,   f rec= fn inv. if eqnil inv then nil
                    else (fac (hd inv))[=>0] : f (tl inv)

,   g rec= fn inv. if eqnil inv then nil
                    else fac (hd inv) : (g (tl inv))[=>0]

,   h rec= fn inv. if eqnil inv then nil
                    else (fac (hd inv) : h (tl inv))[=>0]

,   ℓ= (0: 1: 2: 3: -2: 4: 5: nil)
```

Wat is het resultaat van (f ℓ), (g ℓ), (h ℓ)? Motiveer uw antwoord.
Wordt (fac 4) berekend, (in ieder van die gevallen)?

AANHANGSEL 1

SASL IN EEN NOTEDOP

SASL (St Andrews Static Language) is een beschrijvende programmanotatie met hogere orde funkties, partiële parameterisatie en lazy evaluation. Voorlopig is SASL nog ongetypeerd.

Datatypen

. gehele getallen; operaties +, -, *, div, rem, **(exp.), <, <=, >, >=
. waarheidswaarden; true en false met operaties &, ¦, ~. De rechteroperand van & en ¦ wordt niet geëvalueerd als de linkeroperand de uitkomst al bepaalt.
. characters; namelijk %a, %b, ..., %z, np, nl, sp (voor new page, new line, space)
. lijsten van willekeurige struktuur en inhoud
  () (lege lijst)
  2, (lijst met één element, nl. 2)
  2,3,4 (lijst met drie elementen)
  : (prefixing van een element aan een lijst)(rechts associatief)
  ++ (konkatenatie van twee lijsten)
  ℓ 4 (het 4-de element van de lijst ℓ; lijstindicering)
  2,3,4 = 2:(3,4) = 2: 3: (4,) = 2: 3: 4: ()
. funkties met willekeurig parameterdatatype en resultaatdata-type
  f a (funktie f toegepast op argument a)
  f.g (funktie kompositie van f en g: f na g)(rechts associatief)

Voor alle datatypen zijn de volgende funkties en operaties gedefinieerd: de vergelijkingen = en ~= (verschilt van) (voor lijsten vindt de vergelijking van kop naar staart plaats; op funkties is het resultaat van vergelijking onbepaald), en de domeintesten

number, logical, char, list en function

Expressies zijn opgebouw uit namen en bovengenoemde konstanten en operaties middels de volgende vormen:

```
expr1 -> expr2; expr3     (voor: if e1 then e2 else e3)
expr where defs           (lokale definities)
```

Prioriteiten zijn zoals wiskundig gebruikelijk is; funktietoepassing bindt het sterkst, de : en ++ heel zwak, de komma het zwakst. Haakjes dienen uitsluitend

om de prioriteit te doorbreken; ze mogen altijd!! Tenzij uitdrukkelijk anders vermeld is een operator links associatief, m.n. (f a b c) = (((f a) b) c).

(Meervoudige) definities hebben de vorm

namelist = expr

waarbij namelist is opgebouwd uit namen en konstanten, middels haakjes en prefixing (:). De expr moet een waarde opleveren die past bij de struktuur van namelist. De identifiers in namelist worden zo gedefinieerd dat de vergelijking geldt!

Een funktiedefinitie bestaat uit één of méér clauses van de vorm

f (namelist1) ... (namelistN) = expr

waarbij de haakjes om een namelist niet hoeven als die niet-samengesteld is. De volgorde van de clauses is belangrijk; bij oproep van f wordt de eerstpassende toegepast. N.B. partiële parameterisatie is toegestaan.
De definities en clauses van definities moeten onderling door een ; gescheiden worden.

Lexikale konventies

. Strings zijn louter lijsten van characters; ze kunnen verkort genoteerd worden
met stringquotes '(links) en "(rechts). Nesting is toegestaan. Bijvoorbeeld
'abcdef" = (%a, %b, %c, %d, %e, %f)
'a'bc"def" = (%a, (%b, %c), %d, %e, %f)
. Kommentaar begint bij || en gaat tot aan het einde van de regel.
. Onderstreepte woorden in hoofdletters typen, en niet als namen gebruiken.
. Buitenspelregel: in e'-> e; e" (en in namelist = e) moet geheel e geschreven/getypt worden in het kwadrant ten Z.O. van -> (resp. =).
. Puntkomma op het einde van een regel mag weggelaten worden, (indien ontleding als funktie applikatie vanwege de buitenspelregel onmogelijk is).

De SASL-machine kent onder andere de volgende twee kommando's.

def defs ? (neem defs op in de globale definities)
expr? (druk het resultaat v.d. evaluatie van expr af)

Deze mogen naar willekeur herhaald en afgewisseld worden.

De verzamelingsnotatie

De verzamelingsnotatie is een handig uitdrukkinsmiddel om uit lijsten andere lijsten te konstrueren. De notatie is geen SASL maar kan er wel eenvoudig in worden uitgedrukt, zie Oefening 15. De expressie

{-—x-—y-—z-—; x<-D; Cx; y<-Dx; Cxy; ...; z<-Dxy; Cxyz}

duidt de lijst van waarden ~x~y~z~ aan, waarbij x (van kop naar staart!) varieert over de elementen van de lijst D die voldoen aan de konditie Cx, en (bij iedere x) y varieert over de elementen van Dx die voldoen aan Cxy, en ... (bij iedere y) z varieert over de elementen van Dxy die voldoen aan Cxyz. Een konditie die true is mag weggelaten worden, tesamen met de eraan voorafgaande puntkomma. In plaats van een identifier zoals x in x<-D mag er ook een namelist staan zoals bij definities; de elementen van D moeten dan dezelfde struktuur hebben als de namelist. Hier volgen vier voorbeelden.

```
{x; x<-X; x mod 7 = 0}
      || = de lijst van 7-vouden die in X zitten
heads XX  = {x; (x: X)<-XX}
      || = (x1, ..., xn) voor XX = (x1:X1, ..., xn: Xn)
concat XX = {x; X<-XX; x<-X}
      || = (X1++...++Xn) voor XX = (X1, ...,Xn)
Factabel  = (0,1): {(n+1: n*f); (n,f)<-Factabel}
      || = (0,0!): (1,1!): (2,2!): (3,3!): ...
```

Tenzij uitdrukkelijk anders vermeld mag u de verzamelingsnotatie naar believen gebruiken bij de oefeningen.

De standaard omgeving: oefeningen voor SASL programmering

```
hd      || hd (a1,..an) = a1 (mits n>0)
tl      || tl (a1,..,an) = (a2,..,an) (n>0)
length  || length (a1,..,an) = n
reverse || reverse (a1,..,an) = (an,..,a1)
sum     || sum (a1,..,an) = a1+..+an
product || analoog
and     || analoog
or      || analoog
count   || count a b = (a, a+1, ..., b)
from    || from a = (a, a+1, ... )
map     || map f (a1,..,an) = (f a1, .., f an)
for     || for a b f = (f a, ..., f b)
zip     || zip=ritssluiting
        || zip((a1,...,an),(b1,..,bn))=((a1,b1), ...,(an,bn))
        || zip((a1,..,an), ..., (z1,..,zn))=((a1,..,z1),..., (an, ..,zn))
while   || while f g x = g^i x met i minimaal en zo dat f (g^i x) = false
until   || until f g x = g^i x met i minimaal en zo dat  f (g^i x) = true
|| bij de volgende drie worden verzamelingen gerepresenteerd als lijsten
|| zonder duplikaten
```

```
member ¦¦ parameters: v x
union  ¦¦ parameters: v v'
intersection ¦¦ parameters: v v'
spaces ¦¦ spaces n = lijst van n spaties
width  ¦¦ width x = aantal karakters waarmee x wordt afgedrukt
Ljustify¦¦ een afdruk van (Ljustify n x) is als die van x
        ¦¦ maar dan rechts aangevuld met spaties tot een totale breedte van n
Rjustify, Cjustify ¦¦ analoog; C staat voor Centraal
show   ¦¦ een afdruk van (show x) geeft de (lijst)struktuur van
       ¦¦ x ook aan, d.w.z. de haakjes en komma's.
letter ¦¦ letter x = x is 'n letter
digit  ¦¦ digit x = x is 'n cijfer
abs    ¦¦ abs x = absolute waarde van x
digitval ¦¦ de getal-waarde van een cijfer
code   ¦¦ code x = een getal ter representatie van het character x (bijv. ASCII
       ¦¦ code)
decode ¦¦ de inverse van code
```

N.B. decode en code zijn systeemafhankelijk. Deze hoef je niet zelf te definiëren.

Oefeningen

1! Genereer de lijst van Fibonacci getallen: (1, 1, 2, 3, 5, 8, 13, 21,...), en het eerste fibgetal dat deelbaar is door k, en de eerste die groter dan k is. Doe dit met en zonder de verzamelingsnotatie.

2! Gegeven twee geneste rijen van getallen, bepaal of ze --afgezien van de struktuur-- dezelfde rijen getallen zijn.
Bijvoorbeeld
(1,((2,3), 4), 5) en (((1,2),(3,4)), 5) hebben gelijke rijen,
(1,((2,3),4),5) en (1,((2,3),4), 6) hebben verschillende rijen.

3 Hoe zou opgave 2 zonder lazy evaluation toch efficiënt kunnen, d.w.z. zonder de bomen verder te inspekteren dan nodig is?

4! Schrijf Quicksort in SASL; en merge-sort, bubble-sort, heapsort enzovoorts.

5! Definieer in SASL de funktie "perfekt" die voldoet aan perfekt n = (n = som van alle faktoren van n die van n verschillen)
Bijvoorbeeld (perfekt 6) = true want 6=1+2+3,
(perfekt 8) = false want 8≠1+2+4

6! Genereer alle permutaties op (1, ..., n).

7! Genereer alle deelverzamelingen (als lijsten gerepresenteerd) van (1,...,n).

8 Genereer de lijst van alle getallen $2^i 3^j 5^k$ in opklimmende volgorde;
i, j en k variëren over de natuurlijke getallen.

9! Gegeven een tekst T bestaande uit regels van 125 characters, gerepresenteerd door een lijst van characterlijsten zonder nl. Gevraagd een tekst T' die uit T ontstaat door de regelovergangen door een spatie te vervangen, en door twee opeenvolgende sterretjes * door een | te vervangen, en dan het geheel in regels van 80 characters op te splitsen. Bij deze opsplitsing mogen woorden met lengte <= 81 niet afgebroken worden en moeten de regels rechts aangevuld worden met spaties.

10 We noemen twee objekten structureel equivalent (se) als ze beide de lege lijst () zijn, of beide een niet-lege lijst met se koppen en se staarten, of beide geen lijst zijn (maar bijvoorbeeld een getal). Geef een funktie f zo dat (f obj lst) = een objekt opgebouwd uit de voorste elementen van de oneindige lijst lst en dat structureel equivalent is met obj. Bijvoorbeeld

f (1, (2, 3), 4) (a, b, c, d, e, ...) = (a, (b, c), d)
Wenk: veralgemeen de vraagstelling!

11! Schrijf een funktie die (de getallen van) de knopen van een binaire boom achtereenvolgens, in inorder volgorde ophoogt met 1, 2, 3, 4, ...
Representeer binaire bomen d.m.v geneste lijsten.

12! Schrijf een SASL programma dat de driehoek van Pascal op nette wijze afdrukt.

13! Genereer bij gegeven lijst X alle lijsten(Y, Z) zodat X = Y ++ Z.

14! Herinnner je de funktie map en definieer zonder de verzamelinsnotatie te gebruiken een funktie filter zo dat

filter p X = {x; x<-X; p x}

Elimineer nu de verzamelingsnotatie uit de volgende uitdrukkingen, zonder de funktie set van Oefening 15 te gebruiken.

a. {x*x; x<-X}
b. {x; x<-X; x mod 7 = 0}
c. nats where nats = 0: {n+1; n<-nats}
d. {x*x; x<-X; x mod 7 = 0}
e. {x-y; x<-X; x mod 7 = 0; y<-count 1 x; x mod y = 0}

Doe het nu nogmaals met gebruikmaking van de funktie set uit Oefening 15.

15! Geef een echte SASL funktie set zó dat

set ((D1, C1), ... , (Dn, Cn)) E =
{ E x1 ... xn| x1<-D1; C1 x1; ... ; xn<- Dn x1 ... x(n-1);Cn x1 ... xn}

Geef met gebruikmaking van de verzamelingsnotatie een programma dat de lijst van alle priemgetallen genereert, en vertaal het met behulp van bovenstaande funktie set naar SASL.

16! Definieer de lijstiteratie-associerend-naar-rechts, litar, zó dat

litar f a (x1, ..., xN) = (f x1 ... (f xN a) ...)

Zij g een funktie met 1 veranderlijke, wat is dan

litar (f. g) a (x1, ..., xN)

d.w.z. elimineer litar.
NB. litar f a (x1, x2, x3, x4) = (f x1 (f x2 (f x3 (f x4 a))))

17! Kies geschikte f en a zo dat NB. litr is litar, zie oefening 16.

```
sum = litr f a
product = litr f a
append x y= litr f a x  || append x y = x ++ y
concat = litr f a       || concat (x1,..,xn)=x1++...++xn
map g = litr f a
length = litr f a
sumsquares = litr f a
reverse = litr f a
identity = litr f a     || identity X = X   voor lijsten X.
```

18! Definieer geschikte veralgemeningen van sum, product, concat, map, length en reverse die ook werken voor geneste lijsten.

19! Definieer lital, lijstiteratie-associërend-naar-links, analoog aan litar.

20! Geef een funktie f zo dat (f n) = een lijst van alle rijtjes (x, y, z) met $1 \leq x \leq y \leq z$ en $n = x + y + z$. Doe dit met en zonder de verzamelingsnotatie.

21! Geef een funktie f zo dat (f n) = een lijst van alle rijtjes (x, y, ..., z) met $1 \leq x \leq y \leq \ldots \leq z$ en $n = x + y + \ldots + z$. (Neem f 0 = (),) Doe dit met en zonder de verzamelingsnotatie.

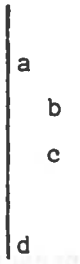
22! Schrijf een programma dat bij gegeven m, k en b de volgende tabel afdrukt.

MACHTENTABEL

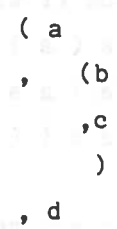
N	N**2	N**3	N**4	...	N**m
1	1	1	1		1
2	4	8	16		.
3	9	27	81		.
...					.

waarbij N loopt tot en met k en de paginabreedte b is. (Veronderstel dat de breedte van regel N=k hoogstens b is)

23! Geef een funktie f zo dat voor geneste lijst X (f X) een afdruk van X is met gelijke indentatie voor elementen van gelijke nestingsdiepte, en grotere indentatie voor elementen van grotere nestingsdiepte. Bijvoorbeeld, een afdruk van (f (a, (b, c), d)) moet figuur 1 zijn.



figuur 1



figuur 2

24! Net als de vorige oefening, maar nu moeten ook de haakjes en komma's worden afgedrukt, zie figuur 2.

25! Gegeven is een tekst, gerepresenteerd als een lijst van characters, met nl voor een regelovergang. Gevraagd uit de tekst een lijst van (w, (r1, ..., rk))-paren te berekenen waarbij w varieert over de woorden van de tekst en (r1, ..., rk) de opklimmend gerangschikte lijst is van alle regelnummers van voorkomens van het woord w. (Een woord is een niet-lege maximale opeenvolging van characters zonder spatie of regelovergang.)

26! Beschouw de volgende representaties voor teksten.

R1: een tekst wordt gerepresenteerd door een lijst van letters, spaties en het character nl voor een regelovergang,

R2: een tekst wordt gerepresenteerd door een lijst van regels, een regel is een lijst van letters en spaties (zonder nl dus),

R3: een tekst wordt gerepresenteerd door een lijst van regels, een regel is een lijst van woorden, een woord is een lijst van letters.

Geef funkties fij (voor i,j: 1..3) die de i-de representatie omzet in de j-de.

27! Geef een funktie die een tekst, gerepresenteerd als een lijst van characters, omzet in de alfabetisch gesorteerde lijst van de woorden van de tekst (zonder duplikaten).

28! Een tekst wordt gerepresenteerd door een lijst van regels, waarbij een regel een lijst van characters zonder nl is. Gevraagd worden funkties f, g en h zo dat een afdruk van (f t), voor een tekst t, er net zo uit ziet als een afdruk van t behalve dat er vooraan iedere regel een van de characters %>, %= of %< is bijgekomen, en wel %> indien in de voorafgaande tekst inklusief die regel het aantal openingshaakjes groter is dan het aantal sluitingshaakjes, en %= resp. %< bij een gelijk of kleiner aantal. Een afdruk van (g t) is net zo als (f t), behalve dat het bijgeplaatste character nu aansluitend achteraan de regel komt. Bij (h t) komt het extra character op positie nr.80 (terwijl de regellengtes ten hoogste 79 mogen zijn).

Voorbeeld.

afdruk van t	afdruk (f t)	afdruk (g t)
a (a)	= a (a)	a (a) =
a) a a	< a) a a	a) a a <
a a ((a	> a a ((a	a a ((a >

29 Beschouw gerichte grafen met gelabelde knopen. Een buurrepresentatie van zo'n graaf is een tweetal (L, B) zó dat

L i = de label van de i-de knoop

B i j = true als knoop j direkt vanuit knoop i bereikbaar is
false anders

voor een of andere nummering van de knopen.

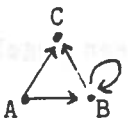
Een direkte representatie van zo'n graaf is een lijst R zó dat

R i = (de label van de i-de knoop, (R j1, ..., R jk))
waarbij j1, ..., jk precies de vanuit i direkt bereikbare knopen zijn

voor een of andere nummering van de knopen.

Voorbeeld:

Graaf G:



Buurrepresentatie van G:

```
((%A, %B, %C),
((false, true, true),
 (false, true, true),
 (false, false, false)))
```

Direkte representatie van G:

```
(x, y, z)
 where x = (%A, (y, z))
       y = (%B, (y, z))
       z = (%C, ())
```

Geef een funktie die een buurrepresentatie van een graaf omzet in een direkte representatie, en een funktie die het omgekeerde doet.

30! Geef bij ieder van de volgende SASLprogramma's (met verzamelingsnotatie) voor ieder variabele-voorkomen zijn bindend voorkomen aan. Geef tevens de door de expressie(s) opgeleverde waarde(n).

a.
```
def x = 1
    Y = {sum {x; x<-count 1 z}; z<-from x; x<-count 1 z}
?
Y 10 ?
```

b.
```
def member (x:Y) x = true
    member Y x = false
?
member (1, 3, 5, 7, 9) 5?
member (1, 3, 5, 7, 9) 4?
member (1, 3, 5, 7, 9) 0?
```

```
c. def x = 1
       y = 2 * x
       u = x + y
           where x = 5 + y
           where y = x * 5
       v = x + y
           where x = 5 + y
                     where y = x * 5
       w = x + y
           where x = 5 + y
                     where y = x * 5
                           x = 7
   ?
   u?
   v?
   w?
```

AANHANGSEL 2

OEFENINGEN EN OPGAVEN OVER TYPERING

Algemeen, stof tot nadenken.

1! Wat zou het nut zijn van de vereiste in Pascal dat de onder- en bovengrens van subscriptbereiken van arrays constanten zijn?

2! Ga na of de volgende teksten goed-getypeerd zijn.

a.
```
var x: 1..10; y: 1..5;
begin x:=y; y:=x; x:=5; y:=x; x:=15 end
```

b.
```
type t = ↑ elt;
var p: t; x: elt;
procedure proc;
     type elt = 1 .. 10;
     var y: elt; q: t;
     begin y:=p↑; q:=p; p↑:=x end
```

c.
```
var i: integer;
begin for i:=1 to 10 do begin write(i); i:=i+1 end
end
```

d.
```
(Algol 68:) int i; for i to 10 do print (i); i:=i+1 end
```

3! Breidt de minitaal uit de syllabus uit met nieuwe expressievormen en/of standaardfunkties, voor bijv. files, strings, komplex, read, write ----. Breidt ook de typeringseisentabel uit op een volgens U zinvolle manier.

Ad Hoofdstuk 3 van de syllabus over Typering.

4! Ga na of de volgende fragmenten goed-getypeerd kunnen worden; geef zo mogelijk van iedere identifier bij z´n introduktie z´n type aan.

a.
```
def linsearch = (f, n, x):
          (def ls = (i): if i>n then false else
                         if f(i)=x then i else ls(i+1)
           in ls (1) )
```

b.
```
def fakulteit = (n):
          (def F = (g, m): if m=0 then 1 else m * g(g, m-1) fi
           in F (F, n) ni )
```

c. def f = (a, x): if a(1)=x and a(2)=x then true else false fi
in if f ((n): n*2, 2) and f ((n): n=e, 2)
then 0 else 1 fi
ni

d. def compose = (f, g): (x): f(g(x))
in ...
def succ = (n): n+1, pred = (n): n-1
in def f = compose (succ, pred)
in ... f(4)... ni
ni ni

5! Typeer m.b.v. rekursieve typen opgave 4b hierboven.

6 Beschouw de volgende stellen notaties van typen.

a. inttree = int + <inttree, int, inttree>
intboom = int + <t, int, inttree>
where t = int + <intboom, int, t>

b. fct = (fct -> fct)
fct' = (fct' -> (fct' -> fct'))
fct" = ((fct" -> fct") -> fct")

Duiden zij eenzelfde type aan? Wenk: konstrueer een oneindig diepe boom (type-expressie) die door zo'n rekursief gedefinieerde type-identifier wordt aangeduid, en ga na of die bomen gelijk zijn.

7 Kunt U een (altijd terminerend) algoritme verzinnen dat de gelijkheid van twee rekursief genoteerde typen beslist? (In een Algol 68 kompiler zit zo'n algoritme ingebouwd).

8 Geldt in uw favoriete programmeertaal dat (coerce-to-real(3) + coerce-to- real (4)) en coerce-to-real (3 + 4) gelijke waarden aanduiden?

9 Geef een zo zuinig mogelijke subtyperelatie op {int, real, bool, compl, void, char, string} zo dat de volgende fragmenten goed-typeerbaar zijn.

a. def apply = (f, x): f(x, y)
in ... apply (succ, 3) + 4 ... round (apply (sqrt, 3.14))..

b. def print = (...):...
in print (3) ...print ("abc")...print("3")...

c. def f = (x): x
in f(3) ... f("abc")...f("3")concat "maal 3 is 9"...

10 Welke coercies kent u in Algol 68 (er zijn er 5), en in Pascal? Vindt er bij de aanroep van p hieronder een coercie plaats?
Pascal: proc p; begin x:=x+1 end; begin ...; p;..end
Algol 68: proc p = void: x:=x+1; ...; p; ... end

11! Kunnen de volgende teksten goed-getypeerd worden? (Gebruik de subtyperelatie en het type acc t)

a. (if x < y then min else max) := (min div 2)/(max mod 2)
b. (if x < y then x else y) := round (sqrt (x div y))

Ad Hoofdstuk 4, 5 van de syllabus over Typering.

12! Voeg type-parameters en -argumenten toe, en typeer álle identifier- introduk-
ties expliciet, zo dat zoveel mogelijk van de volgende fragmenten goed
getypeerd zijn volgens de tweede orde typering

```
a. def compose = (f, g): (x): f (g(x))
     in ... compose (succ, (x): x*2) ...
        ... compose (sqrt, (x): x/2) ...
        ... verzin zelf nog meer mogelijkheden ...
b. def apply-to-each = (f, pair): <f(pair.1), f(pair.2)>
     ,  reverse = (pair): <pair.2, pair.1>
     in ... apply-to-each (succ, x) ...
        ... apply-to-each ((y): not y, x) ...
        ... apply-to-each (reverse, <<3,4>, <5,6>>) ...
        ... apply-to-each (reverse, <<3,4>, <true, false>>)..
        ... apply-to-each (reverse, <<3, true>, <4, false>>).
```

13! Welke van de fragmenten uit opgave 12 hierboven zijn goed-typeerbaar volgens Milner's type polymorfie (Hoofdstuk 5)?

14! Geef SVP-typen voor "lijsten", "stapels", "queues", "sets", "bomen" etc., en geef goed-getypeerde implementaties ervan. Geef ook steeds een paar voorbeeldjes van een gebruik.

15! Schrijf algemeen bruikbare (SVP getypeerde, 2de orde getypeerde of vlgs. Milner's type polymorfie goed getypeerde) sorteerroutines, binary search routines, permutaties, lijstverwerkingsprocedures enz. enz. Geef voorbeelden van een gebruik ervan.

16 In Pascal kun je als volgt een "polymorfe" sorteerprocedure typeren (volgens een suggestie van C. Bron in ca. 1982):

```
procedure sort (van, tot: integer
                ; procedure wissel (j: integer)
                ; function kleiner-of-gelijk (i,j: integer): boolean
                )
```

Geef de voor- en nadelen t.o.v. de typering die U in opgave 15 heeft bedacht. Kunt U Quicksort als body geven? Geef een voorbeeld van het gebruik van sort.

17! Beschouw het volgende SASL programma.

```
def F 0 x = x,              || de singleton lijst met enig lid x
    F n g = F(n-1) (g 1) ++ F (n-1) (g 0)
?
F 3 g where g x y z = 4 * x + 2 * y + z ?
      || levert (g 1 1 1, g 1 1 0, g 1 0 1, g 1 0 0, g 0 1 1, g 0 1 0,
      ||         g 0 0 1, g 0 0 0)
      || i.e. (7, 6, 5, 4, 3, 2, 1, 0)
```

Kunt U F volgens een van de gegeven typeringen (konventioneel, 2de orde, SVP, Milner) goed typeren?

18! Geldt bij Milner's type polymorfie de Overeenkomst tussen Definitie en Parameterisatie syntaktisch (d.w.z. wat de typering betreft)? En semantisch (d.w.z. wat het resultaat na evaluatie betreft)?

19! Wat is het verschil tussen

```
        f: (t: tp -> (z->t->t) ->t ->t)
en      f': (t    -> (z->t->t) ->t ->t)
```

(De notatie van Hoofdstuk 6 is hier gebruikt voor SVP-typen.) Verzin enige goed-getypeerde en fout-getypeerde aanroepen van f en f'.

20! Geef voor ieder variabele voorkomen zjn bindend voorkomen aan. (De notatie uit Hoofdstuk 6 is hier gebruikt.)

```
(dfx: (x: tp -> < y: tp, (y->(x->x)), ((x->x)->u) >)
    = fnx: tp. <(x->x), (fn z:y y:x. z y), (fn y: (x->x). y)>
  . df x == (x int).
    df f: x.1 = (x.3 succ)
    . (x.2 f 3)
)
```

Geef een alfabetische variant waarbij zinvolle, suggestieve identifiers worden gebruikt. Wat is het resultaat van de expressie?

21.a! Geef een SVP-getypeerde introduktie (i.e. het deel "df x:t" zonder = e") van een automaat x. (Een automaat wordt gespecificeerd door

- een input-alfabet (i.e. inputtype)
- een output-alfabet (i.e. outputtype)
- een toestandsverzameling (i.e. toestandstype)
- een transitiefunktie die bij een inputsymbool en een toestand een nieuwe toestand geeft
- een outputfunktie die bij een toestand een outputsymbool geeft.)

21.b Geef vooor bovenstaande introduktie een SVP getypeerde expressie e zodat de automaat x

(i) een eindige automaat is,

(ii) een stapelautomaat is (toestand = stapelinhoud),

enzovoorts.

21.c! Geef een SVP-getypeerde funktie die op automaten x werkt (zoals geïntroduceerd in 21.a) en de berekeningsfunktie b van x oplevert, i.e. de funktie b zo dat (b inputrij) = "de door de automaat geproduceerde outputrij". (Deze opgave is onafhankelijk van de uitwerking van 21.b)

AANHANGSEL 3

TENTAMENS EN UITWERKINGEN ERVAN

Tentamen Struktuur van Programmeertalen (211050)
Vrijdagmiddag 23 december 1983, 13.30 - 17.00 uur.

Opg. 1 Programmeeropgave in SASL

Onder een tekst verstaan wij een rij van kleine letters en spaties. Een kommentaar in een tekst is een deelrij van de tekst die begint met een onevende voorkomen van "co" en eindigt met het daaropvolgende evende voorkomen van "co". Het signifikante deel van een tekst t is de tekst die uit t ontstaat door alle kommentaar weg te laten. (We beschouwen alleen teksten met een even aantal voorkomens van "co").
Voorbeeld. Het signifikante deel van de tekst:

dit-is-co-echt-co-geen-co-leuk-co-voorbeeld

is de tekst:

dit-is--geen--voorbeeld

(We hebben spaties met - genoteerd.)

We beschouwen twee representaties voor teksten in SASL; in beide wordt de tekst in regels ingedeeld.

(R1) de tekst wordt als een lijst karakters gerepresenteerd, waarbij het karakter nl (newline) de regelovergangen aangeeft.

(R2) de tekst wordt als een lijst van 'regels' gerepresenteerd, waarbij een 'regel' een lijst van karakters is --zonder nl--.

Bijvoorbeeld, de bovenstaande voorbeeldtekst kan als volgt gerepresenteerd worden:

```
rt1 = (%d, %i, %t, sp, %i, %s, sp, nl, %c, %o, sp,
       %e, %c, %h, %t, sp, %c, %o, nl, sp, ....)
rt2 = ((%d, %i, %t, sp), (%i, %s, sp, %c, %o, sp),
      (%e, %c, %h, %t, sp), (%c, %o, sp ...), ....)
```

Let wel, de regelindeling van rt1 verschilt van die van rt2, maar beide representeren eenzelfde tekst.

Gevraagd wordt een SASL-funktie te schrijven die bepaalt of de signifikante delen van twee meegegeven teksten gelijk zijn, waarbij de eerste tekst volgens (R1) wordt gerepresenteerd en de tweede tekst volgens (R2). Bijvoorbeeld, bij

rt1 en rt2 als argumentpaar moet de funktie het resultaat true opleveren.

(U mag gebruik maken van de SASL standaardomgeving, d.w.z. de voorgedefinieerde funkties zoals hd, tl, length, reverse, ...enzovoort. Maar deze zijn nauwelijks nodig.)

Opgave 2

Een queue is een gegevensstruktuur met de volgende 'bewerkingen':

- het toevoegen van een element;
- het opleveren-en-tevens-verwijderen van een (en wel: het eerst toegevoegde en nog niet verwijderde) element;
- de test of een queue leeg is;
- de lege queue.

a. Kies in HOF een geskhikte representatie voor queues en programmeer/definieer de 'bewerkingen'.

b. Geef nogmaals de programmateksten maar nu voorzien van de konventionele typering; en ook met de polymorfe typering (Sektie 5 v.d. syllabus over Typering).

c. Programmeer het geheel als één expressie (of definitie), en voorzie de tekst van de SVP-typering; (Sektie 4.2 v.d. syllabus over Typering). (De expressie of definitie staat dus voor zoiets als het abstrakte datatype der queues.) Geef schetsmatig een voorbeeld waarin dat datatype gebruikt wordt.

d. Programmeer wederom de 'bewerkingen'; nu zonder typering maar met "foutafhandeling" voor het geval dat van de lege queue een element opgeleverd-en-verwijderd zou moeten worden. Geef schetsmatig een of een paar voorbeelden waarin zo'n fout wordt afgehandeld.

e. Geef een module die als resultaat de-'bewerkingen'-met-de-mogelijkheid-tot-foutafhandeling oplevert. De fouten die niet door de gebruiker (invoker) van de module worden afgehandeld, moeten binnen de module worden afgehandeld. Geef schetsmatig een of een paar voorbeelden waarin dit plaats vindt.

Opgave 3

Geef een HOF expressie zo dat bij de implementatie volgens het (traditionele) LIFO-model (Hoofdstuk 4 van de syllabus) ooit eens een element (f <- < fn x.eb, r0>,r1) op sR wordt gezet, met r0 ≠ r1. (Dit is opgave 4.5 van de syllabus.)

Leg kort en bondig uit waarom uw expressie aan de gevraagde voorwaarde voldoet.

Struktuur van Programmeertalen Uitwerking tentamen
d.d. 23 december 1983

Opg. 1

```
def
||(signif t) = het signifikante deel van tekst t.
signif ()    = ()
signif (%c: %o : l) = signif (vanafco l)
                      where vanafco (%c : %o : l) = l
                            vanafco (x : l) = vanafco l
signif (x : l) = x : signif l
||(tekstUitR1 rt1) = tekst t zo dat: rt1 is R1-representatie van t.
tekstUitR1 () = ()
tekstUitR1 (nl : l) = tekstUitR1 l
tekstUitR1 (x : l)  = x : tekstUitR1 l

||(tekstUitR2 rt2) = tekst t zo dat: rt2 is R2-representatie van t.
tekstUitR2 () = ()
tekstUitR2 (l:ll) = l ++ tekstUitR2 ll

gevraagdeFct rt1 rt2 = signif (tekstUitR1 rt1) =
                       signif (tekstUitR2 rt2)
|| Desgewenst kan de infix = -operator hierboven vervangen worden door
|| een (prefix) eq-funktie, gedefinieerd door:
   eq () () = true
   eq (x1:l1) (x2:l2) = x1=x2 & eq l1 l2
   eq x y   = false
```

Opg. 2

We representeren queues als lijsten met het eerst te verwijderen element op kop. (We geven ook de typering aan; zie deel b)

```
df add^(t,Q->Q) rec = fn e^t 1^Q. if eqnil 1
                                  then e:nil else (hd 1): add e (tl 1)
  , delete^(Q-><t,Q>) = fn 1^Q. < hd 1, (tl 1)>
  , is Empty^(Q->bool) = fn 1^Q. eqnil 1
  , empty^Q           = nil
```

deel a. Zie de tekst hierboven.

deel b. Voor de konventionele typering: neem voor t een of ander vast type, bijvoorbeeld int, en voor Q dan (list t), in casu (list int). Voor de polymorfe typering: neem voor t een of andere generische type-variabele, bijvoorbeeld z, en voor Q dan (list t), in casu (list z). Nota bene, als we niet alleen van de definiërende voorkomens het type expliciet aangeven, maar ook van alle sub-expressies, dan krijgen we:

```
add^(t,Q->Q) rec= (fn e^t, 1^Q.(if (eqnil 1^Q)bool .
                                 then (e^t : nil^Q)^Q
                                 else ((hd 1^Q)^t: ((add^(t,Q->) e^t)(Q->Q)(tl 1^Q)^Q)^Q)^Q
                  )(t,Q->Q)
```

enzovoorts

deel c.

```
(df queueADT : (elt : tp ->
                    < repr : tp
                    , (elt, repr -> repr)
                    , (repr -> <elt, repr>)
                    , (repr -> bool)
                    , repr
                    >)
   = fn elt : tp .
                    < (list elt)
                    , rec add:(elt,list elt->list elt).fn e:elt, l:list elt. ...
                    , fn l: (list elt).< hd 1, tl 1>
                    , fn l: (list elt).eqnil 1
                    , nil
                    >
```

```
. ------ ({voorbeeld}
        df intQueue : < repr : tp, (int, repr->repr), repr-><int, repr>, ....>
                    = (queueADT int)

        .   (df q: intQueue.1 = intQueue.5
              . ....(intQueue.3 (intQueue.2 3 q)).1 {= 3}...
)       )   )
```

```
deel d. (dfexc error
         . (df add rec= fn e, l ...
            , delete = fn l. [if eqnil l then T[error] else] <hd l, tl l>
            , isEmpty= ...
            , empty  = ...
            .
                    {voorbeeld}
                    (delete empty)[error => <0, empty>] ....(*)
                    ---
                    ---
                    (delete empty)                      ....(**)
                    ---
                    ---
         )[error => T[noHandlerPresent] ]               ...(***)
      )
```

De "foutieve" aanroep op regel (*) wordt ter plekke afgehandeld: als resultaat van afhandeling wordt <0, empty> opgeleverd. De "fout" in de aanroep op regel (**) leidt onmiddellijk tot beeindiging van (df add .....); en dan wordt de "fout" T[error] omgezet in de "fout" T[noHandlerPresent] waarbij verondersteld is dat noHandlerPresent in een omvattende expressie bijv. in de standaard omgeving, gedefinieerd is (met dfexc).

Nota bene 1 Sommigen hadden delete gedefinieerd als:

```
delete = fn l. <hd l, tl l>[=> <0, nil>{of: "foutmelding"}]
```

Daarbij veronderstellen zij dat de reduktieregels

```
(hd nil) -> T
(tl nil) -> T
```

geldig zijn. Op zich is dit wel korrekt, maar op deze manier heeft de gebruiker niet de mogelijkheid om zelf de fout, veroorzaakt door de foutieve aanroep (delete empty), af te handelen. Bovendien is voor deze organisatie geen [=>...]-expressie nodig, omdat we delete nu ook zouden kunnen definiëren door

```
delete = fn l. if eqnil l then <0, nil> else <hd l, tl l>
```

Nota bene 2. In de veronderstelling dat (hd nil)->T en (tl nil)->T, kunnen we delete ook definiëren als

```
delete = fn l. <hd l, tl l> [=> T[error] ].
```

De voortijdige beëindiging veroorzaakt door (hd nil) wordt dan met een andere identifikatie gepropageerd. Overigens blijft alles hetzefde als in de gegeven uitwerking.

```
deel e.
 (df queueDT =
      (mod (dfexc error
            . (df add rec= fn e, l. ---
               , delete = fn l. [if eqnil l then T[error] else] <hd l, tl l>
               , isEmpty = ...
               , empty   = ...
               , export < error, add, delete, isEmpty, empty > end
               )
               [error => T[noHandlerPresent]]       -----(***)
            )
      )
```

```
. ... (inv queue = queueDT

       . ---

       (queue.3 queue.5)[queue.1 => < 0, queue.5>] ....(*)
         ----
         ----
       (queue.3 queue.5)                          ...(**)
         ----
         ----



       )
 )
```

Het voorbeeld is bijna identiek aan dat van uitwerking d. De fout op regel (*) wordt ter plekke afgehandeld; de foutieve aanroep op regel (**) wordt in de module, op regel (***) afgehandeld (resulterend in beëindiging van de hele expressie).

Opgave 3

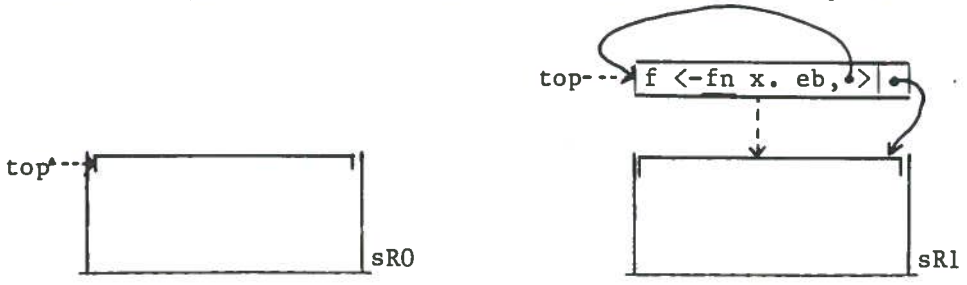
Kennelijk moet de omgeving r0 waarin de abstraktie (fn x. eb) voorkomt, een andere zijn dan de omgeving r1 waarin de closure <fn x. eb, r0> gebonden wordt aan f. De eenvoudigste oplossing hiervoor is dat r0 van r1 verschilt doordat r0 ook nog de binding aan f bevat:

```
r0 = r1 [f <- < fn x. eb, r0 >]
```

Dit doet zich voor bij rekursieve functies. Bijvoorbeeld:

```
(df f rec= (fn x. eb). -------)
```

Zij sR0 de stapel van omgevingen bij het ter evaluatie nemen van deze expressie, dan is sR1 de stapel bij het ter evaluatie nemen van de romp van die expressie.



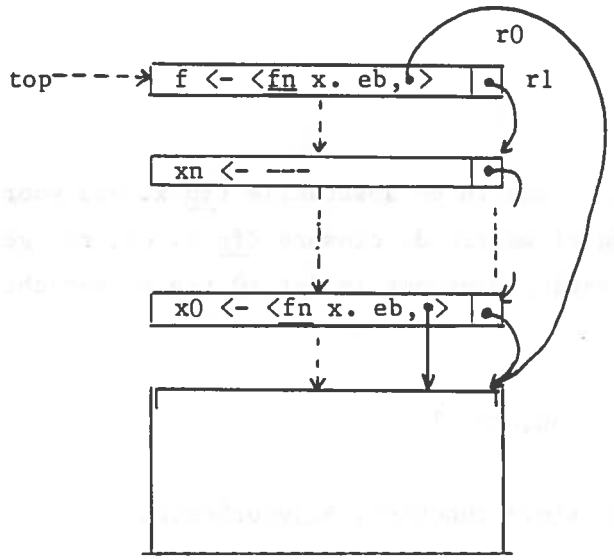
Een andere oplossing is als volgt. We plaatsen de abstraktie op een geheel andere plaats in de tekst dan de plaats waar de binding aan f veroorzaakt wordt.

Bijvoorbeeld, voor n >= 0

```
(df x0 = (fn x. eb)
  . (df x1 = ----
     . (df x2 = ----
        . (df x3 = ----


                (df xn = ----
                  . (df f = x0
                     . ----e-----
)   )  )   )..)   )
```

Wanneer e ter evaluatie wordt genomen ziet de stapel van omgevingen er als volgt uit.



Tentamen Struktuur van Programmeertalen, 10 februari 1984.

1. Programmeren in SASL

Onder een tekst wordt een rij kleine letters en spaties verstaan, die in regels is opgedeeld. Wij representeren een tekst in SASL door een lijst van karakters %a, ..., %z, sp en nl, waarbij nl een regelovergang van de tekst representeert. Voorbeeld:

	tekst	representatie
regel 1	abc-ab	(%a, %b, %c, sp, %a, %b, nl,
		%c, sp, sp, %a, %b, %c, sp,
regel 2	c--abc-abc	%a, %b, %c, sp)

(- staat voor een spatie)

Een woord in een tekst is een deelrij van de tekst die geheel uit kleine letters bestaat (geen spatie bevat) en geheel op een regel staat. De woorden in bovenstaande tekst zijn dus: abc, ab, en c.

Gevraagd wordt een SASL funktie die, gegeven de representatie van een tekst, een lijst (l1, l2, ..., ln) oplevert, waarbij n = aantal regels in de tekst, en li = de lijst van woorden van regel nr i (in de volgorde zoals ze op de regel voorkomen, evt. met herhalingen).
Dus bij bovenstaande voorbeeldtekst moet de volgende lijst worden opgeleverd:

((’abc", ’ab"), (’c", ’abc", ’abc"))

Nota bene, ’abc" is een alternatieve SASL notatie voor (%a, %b, %c).

(U mag gebruik maken van alle voorgedefinieerde SASL funkties.)

2. Typepolymorfie (Hoofdstuk 5 van de syllabus over Typering)

Beschouw de volgende twee expressies (in de HOF notatie).

a) (df reverse = fn u. <u.1, u.0>
; F = fn (f, x, y). <f x, f y>
. F (reverse, <3, 4>, <true, false>)
)

b) (df reverse = fn u. <u.1, u.0>
; resersepair = fn (x, y). <reverse x, reverse y>
. reversepair (<3, 4>, <true, false>)
)

Welke van die expressies zijn volgens de typepolymorfie (Hoofdstuk 5 van de syllabus over Typering) goed typeerbaar? Motiveer (of beter nog: bewijs) uw antwoord.

3. LIFO implementatiemodel

Beschouw de volgende twee (verschillende!) expressies.

a) df f = (fn x. fn y. x + y). (f 2 6)
b) (df f = (fn x. fn y. x + y). f 2) 6

applikatie I — applikatie II

Geef (teken) de stapel sR bij evaluatie volgens eval3 (Hoofdstuk 4 van de syllabus) op de volgende momenten:

- direkt voorafgaande aan
- direkt na het begin van
- direkt na beeindiging van

} applikatie I en II.

En geef ook de door die applikaties opgeleverde waarde. Doe dit voor beide expressies. (Denk erom ook de "top SR" goed aan te geven).

4. Programmeren in HOF met de SVP-typering

Herinner je het spel Kruisje-Nulletje, ook wel Drie-Op-Een-Rij of Tik-Tak-Toe genoemd. Een voorbeeld van een spelsituatie is:

		0
	x	
x	0	

Bordposities worden als volgt genummerd:

3	6	9
2	5	8
1	4	7

Wanneer dit spel in programmatuur wordt nagebootst, kunnen we o.a. de volgende bewerkingen, tests en waarde onderscheiden:

zk(p,b) : zet kruisje op positie p van bord b,
zn(p,b) : zet nulletje op positie p van bord b,
bk(p,b) : positie p van bord b bevat een kruisje,
bn(p,b) : positie p van bord b bevat een nulletje,
lb : het lege bord.

a. Het 6-tal, bestaande uit bovenstaande 5 grootheden plus het type voor de representatie van borden, noemen we: het (abstrakte) datatype van kruisje-nulletje, kortweg knADT. Geef volgens de SVP-typering een type voor knADT (maar houd ook rekening met vraag b en c).

b. Representeer nu een bord door een tweetal <f, g> waarbij f in feite de funktie bk, en g de funktie bn is. (De implementatie van bk en bn is nu bijna triviaal.) Voltooi met deze keuze van de representatie de definitie

knADT : "onder a gegeven type"
= .... ? ....

c. Representeer nu een bord door een 9-tal < b1, b2, b3, b4, b5, b6, b7, b8, b9> waarbij iedere bi een representatie voor Kruisje, Nulletje of "leeg" is. Voltooi ook met deze keuze de definitie

knADT : "onder a gegeven type"
= .... ? ....

d. Programmeer -en typeer, alweer volgens de SVPtypering- in de scope van de onder b en c gegeven definitie de funktie

k3 : op bord b zijn er drie kruisjes op een rij
(horizontaal, vertikaal of diagonaal)

(Denk eraan dat k3 goedgetypeerd moet zijn t.a.v. de typering van knADT. Het programma op zich is nogal "dom".)

5. Exception handling

a. Schrijf in HOF, uitgebreid met "exception handling" faciliteiten (Hoofdstuk 10 van de syllabus), een funktie sub die, gegeven een lijst l en een getal n, het n-de element van de lijst oplevert. Ga ervan uit dat het op te leveren n-de element bestaat.

b. Voeg nu voorzieningen toe zodat er een gemerkte foutstop resulteert indien de meegegeven lijst l geen n-de element heeft. Doe dit zo dat de gebruiker van die funktie sub desgewenst zo'n foutstop nog kan afhandelen. (Geef de plaats van dfexc duidelijk aan.) Geef een eenvoudig voorbeeld ter illustratie.

Uitwerkingen Tentamen Str V Prt d.d. 10 februari 1984.

1. We gebruiken t, r, w, u, x als identifiers voor teksten, regels, woorden, spatie-of-letter, en letters. Methode: splits een tekst op in regels, een regel in woorden, en konstrueer daarmee de gevraagde funktie.
(NB. Een elegantere uitwerking wordt na de uitwerking van opgave 5b gegeven!)

```
regellijst () = ()                          || maakt lijst-van-regels uit een tekst
regellijst (nl:t) = (): regellijst t
regellijst t      = r: regellijst tt where r, tt = regelVan t
regelVan (nl:t)   = (), t  || splitst tekst in (1ste regel, rest van de tekst)
regelVan (u:t)    = u:r, tt where r, tt = regelVan t
regelVan ()       = (), ()

woordlijst ()     = ()   || maakt uit een regel de lijst van woorden
woordlijst (sp:r) = woordlijst r
woordlijst r      = w : woordlijst rr where w, rr = woordVan r
woordVan (sp:r)   = (), r || splitst een regel in (1ste woord, rest van de
                          || regel)
woordVan (x:r)    = x:w, rr where w, rr = woordVan r
woordVan ()       = (), ()

gevraagdeFct t    = map woordlijst (regellijst t)
```

Een andere methode is als volgt. De funktie f vormt de gevraagde woordlijst van t, waarbij wlt de alreeds gevormde woordlijst van voorafgaande regels is, wr de woordlijst van de voorafgaande woorden in de huidige regel en w het huidige te vormen woord. (x als laatste element toevoegen aan een lijst l kan uitgedrukt

worden als l ++ (x,))

```
f wlt wr w ()      = w = () -> wlt ++ (wr,); wlt ++ (wr ++(w,),)
f wlt wr w (nl:t) = w = () -> f (wlt ++ (wr,)) () () t;
                              f (wlt ++ (wr++(w,),)) () () t
f wlt wr w (sp:t) = w = () -> f wlt wr () t;
                              f wlt (wr ++ (w,)) () t
f wlt wr w (x:t)   = f wlt wr (w ++(x,)) t

gevraagdeFct t = f () () () t
```

Dit programma is wel korter, maar heeft twee grote nadelen. Ten eerste volgt het de hierarchische struktuur van teksten niet (tekst-regel-woord), en leent zich daarom minder voor aanpassingen aan gewijzigde vraagstellingen. Ten tweede is tijdens de berekening van (gevraagdeFct t), en met name vlak voor het opleveren van het eindresultaat, de tekst t verspreid over de wt, wr, w en t-parameters van f, geheel opgeslagen. Bij de eerste methode is dit niet het geval: daar kan al met het eerste woord van de eerste regel van t "verder gerekend worden", ook al is van de rest van t de woordlijst nog niet voltooid. (Zie de syllabus, Hoofdstuk 2)

2. $(\underline{df}\ \text{reverse}^a = (\underline{fn}\ u^b.\ \langle u^b.1^c,\ u^b.0^d\rangle^e\ )^f$

$$;\ F^g = (\underline{fn}\ (f^h,\ x^i,\ y^j).\ \langle (f^h\ x^i)^k,\ (f^h\ y^j)^l\ \rangle^m)^n$$

$$.\ F^o\ (\text{reverse}^p,\ \langle \underline{3}^{int},\ \underline{4}^{int}\rangle^{\langle int\ int\rangle},\langle \underline{true}^{bool},\ \underline{false}^{bool}\rangle^{\langle bool,\ bool\rangle}\ )$$

Uit regel 1 volgt vlgs de typeringseisen

b = <..,c>
b = <d,..> dus b = <d,c>
e = <c,d>
f = (b -> e), dus = (<d,c> -> <c,d>)
a = (<d,c> -> <c,d>)

Dus het type van reverse is (<d,c>-><c,d>) waarbij c en d nog generisch instantieerbare type variabelen zijn.
Uit regel 2 volgt net zo:

h = (i -> k)
h = (j -> l) dus i=j en k=l
m = <k,l>
g=n = (h, i, j -> m) dus
= ((i->k), i, i -> <k, k>)

Dus het type van F is ((i->k), i, i -> <k,k>), waarbij i en k nog generisch in-

stantieerbaar zijn.

Volgens regel 3 moet nu gelden

o = (p, <int,int>, <bool,bool> -> ....), dus

((i'->k'), i', i'-><k', k'>) = (p, <int,int>, <bool,bool>-> ....)

Dit kan niet opgelost worden, omdat onder andere

i'= <int,int>, en

i'= <bool,bool>

zou moeten gelden. Dit kan niet, voor geen enkele keuze van i'.

* * *

(df reverse$^{(<d,c>->)}$ = ... net als bij a. ...

; reversepair^{g} =

(fn (x^i, y^j). < (reverse$^{(<d',c'->)}$ x^i)^k

,(reverse$^{(<d",c">->)}$ y^j)^l

>^{m})^n

. reversepair^{o} (<3,4>$^{<int,int>}$, <true, false>$^{<bool,bool>}$)

)

Typering van de eerste regel levert, net als bij a, dat (<d,c> -> <c,d>) het type van reverse is, waarbij c en d generisch instantieerbaar zijn. Op regel 2 zijn dan ook nieuwe keuzen c', d' resp. c", d" voor c en d gemaakt. Er volgt nu dat

i = <d',c'> en k = <c',d'>

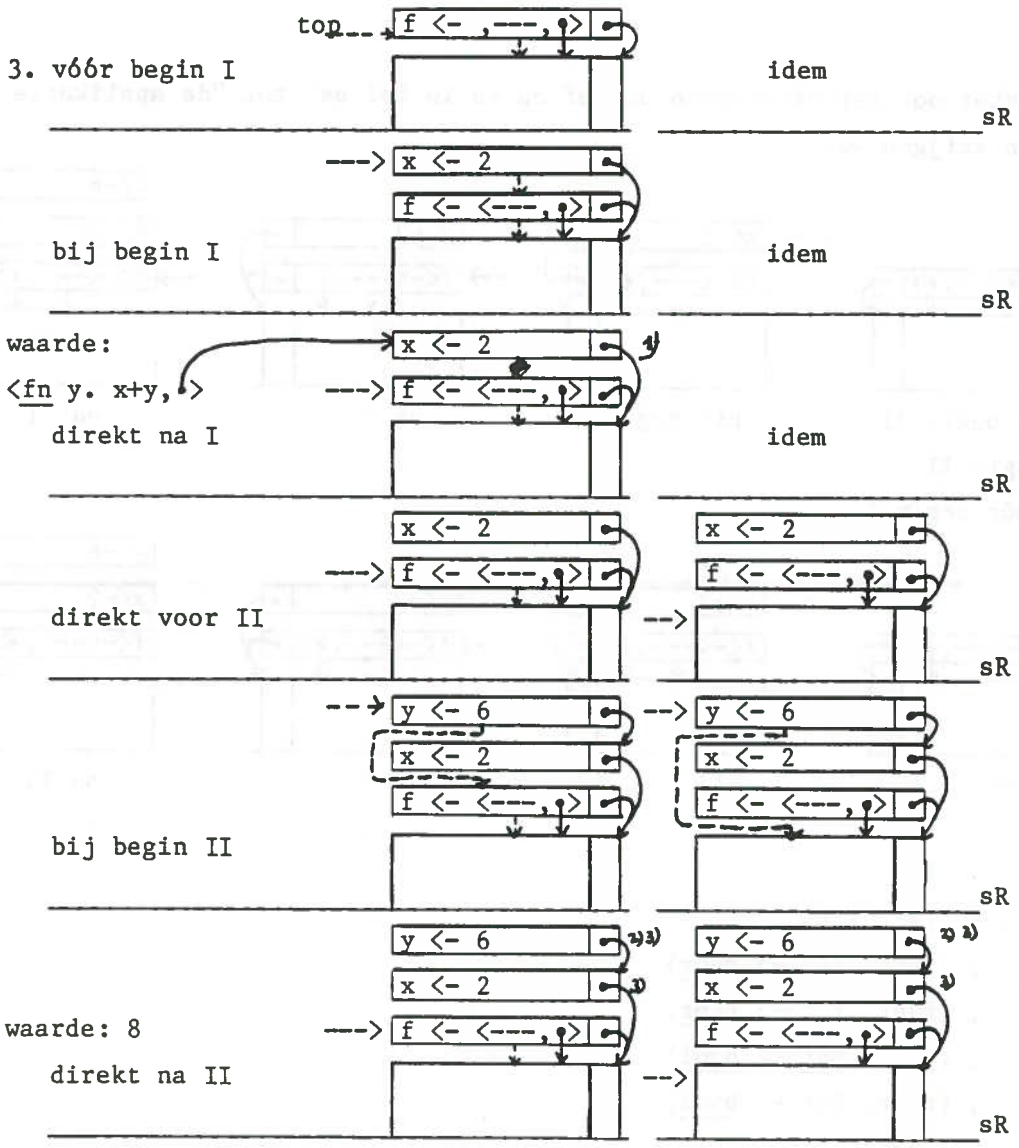
j = <d",c"> en l = <c",d">

m = <k,l> = <<c',d'>, <c",d">>

g = n = (i, j -> m) = k

= (<d', c'>, <d", c"> -> <<c', d'>, <c", d">>)

En dit is het type van reversepair. Hierin zijn c', d', c", d", generisch instantieerbare typevariabelen. Kiezen we c'= d'= int en c" = d"= bool, dan is daarmee regel 3 goedgetypeerd.



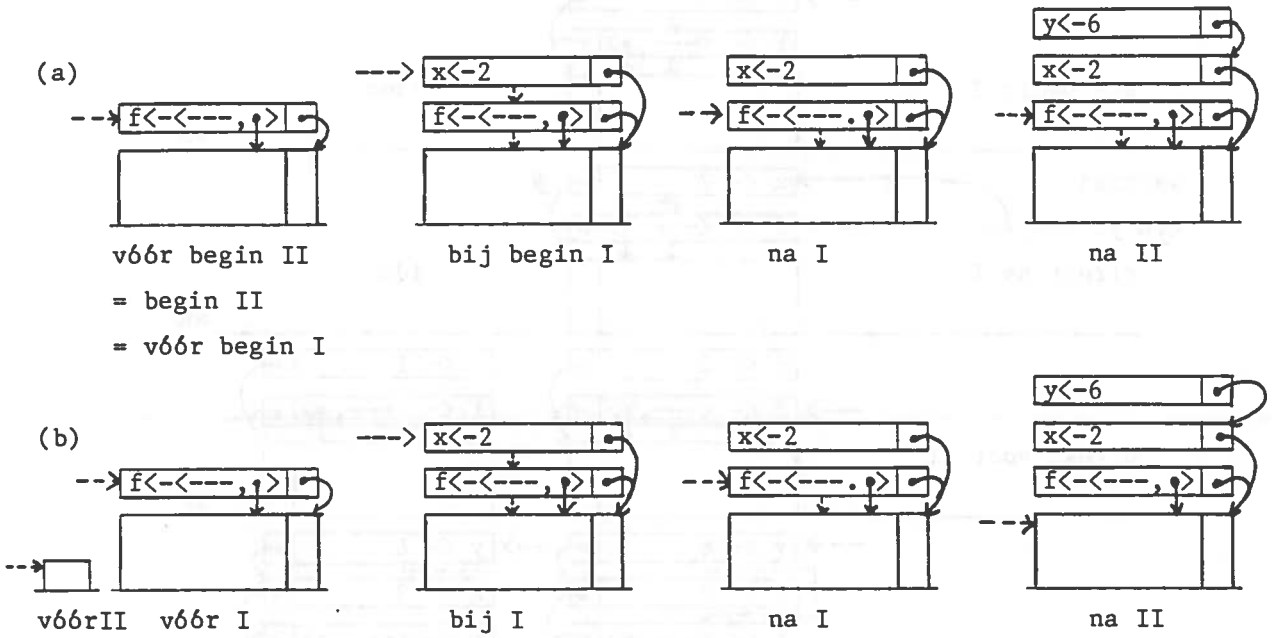
Het verschil tussen de expressies uit zich tijdens de evaluatie door een verschil in de omgevingen, met name de omgeving (top sR) direkt voor, bij, en na afloop van de applikatie II.

De merktekens bij de bindingen betekenen het volgende.

1) : bij een echte LIFO bespeling van de opslagruimte zouden deze cellen nu moeten worden vrijgegeven. Maar omdat ze nog bereikbaar zijn, via de opgeleverde waarde, mag dat nog niet.
2) : bij een echte LIFO bespeling van de opslagruimte zouden deze cellen nu moeten worden vrijgegeven. Omdat ze onbereikbaar zijn, kan dat inderdaad gebeuren.
3) : nu zijn zowiezo deze cellen onbereikbaar, en mogen ze verdwijnen.

N.B. Hierboven ben ik ervan uitgegaan dat het uitrekenen van ef en ea in (ef ea) niet tot "de applicatie" behoort. M.a.w. "de applikatie" behelst louter de (app)regel + verdere evaluatie tot resultaat; zie de afspraak bij eval3.

Als we echter ook het uitrekenen van ef en ea in (ef ea) tot "de applikatie rekenen, dan krijgen we:




```
4a knADT : < (repr : tp
           , (repr, int -> repr)
           , (repr, int -> repr)
           , (repr, int -> bool)
           , (repr, int -> bool)
           , repr
           >

4b       = < < (int -> bool), (int -> bool)>
           , fn fg: < (int -> bool), (int -> bool)>, p:int
             . < fn p':int. if p=p' then true else fg.1 p'
               , fg.2
               >
           , ...analoog...
           , fn fg: < (int -> bool), (int -> bool)>, p:int
             . (fg.1 p)
           , ...analoog...
           , < fn p:int. false, fn p:int. false>
           >
```

```
4c   *)  = < <int, int, int, int, int, int, int, int, int>
          , fn negental: <int,...,int>, p:int
             . if p eqn 1 then < 1, negental.2, ..., negental.9> else
               if p eqn 2 then < negental.1, 1, negental.3, .....> else
               enzovoorts
          , analoog (nu een 2 in de negentallen)
          , fn negental: < int, ..., int>, p:int
             . if p eqn 1 then negental.1 eqn 1 else
               if p eqn 2 then negental.2 eqn 1 else
               enzovoorts
          , analoog (nu steeds de test ...eqn 2)
          , < 0, 0, 0, 0, 0, 0, 0, 0, 0>
          >
```

*) We gebruiken bi=0 als kodering van een leeg veld, bi=1 als kodering van een kruisje, en bi=2 voor een nulletje. Voorts veronderstellen we dat .0 de eerste selector is, en dus .9 de negende selector.

```
4d k3 : (knADT.1 -> bool)
      = fn b : knADT.1
         . df bkb : (int -> bool) = (knADT.4 b)
           . ((bkb 1) and (bkb 2) and (bkb 3))
          or ((bkb 4) and (bkb 5) and )bkb 6))      } vertikaal
          or ((bkb 7) and (bkb 8) and (bkb 9))      }
          or ((bkb 1) and (bkb 5) and (bkb 9))      } diagonaal
          or ((bkb 3) and (bkb 5) and (bkb 7))      }
          or ...                                    }
          or ...                                    } horizontaal
          or ...                                    }
```

```
5a df sub = (rec f. fn n l.
                 if n eqn 0 then (hd l) else f (n-1)(tl l)
            )
```

```
5b (dfexc bestaatNiet
     . (df sub = (rec f. fn n l.
                   [if eqnil l then T[bestaatNiet] else]
                   if n eqn 0 then (hd l) else f(n-1)(tl l)
                  )
           . ¦
             ¦
             ¦
             ¦
   (*)        (sub x (1 : 2 : nil))[bestaatNiet => false] ..
  (**)        (sub x (1 : 2 : nil))[bestaatNiet => 0] ...
           )
      )
```

In het voorbeeld op regel (*) is ervoor gekozen om bij een getal x>=2 als resultaat van subskriptie op (1:2:nil) de waarde false te nemen. Op regel (**) daarentegen wordt 0 genomen als resultaat van subskriptie met een te grote "subskript".

NB. Alternatief voor de uitwerking van opgave 1.

```
Regels () = ()
Regels (nl: t) = (): Regels t
Regels (x: t) = add x (Regels t)

Woorden () = ()
Woorden (sp:t) = Woorden t
Woorden (x:t) = add x (Woorden t)

add x (y:z) = (x:y): z
add x () = (x:()): ()

gevraagdeFct t = {Woorden r ¦ r <- Regels t}
```

Tentamen Struktuur van Programmeertalen (211050) 28 Mei 1984.

(Normering . U mag één opgave overslaan om toch een 9 te behalen. Alle opgaven tellen even zwaar.)

Opgave 1 Programmeren in SASL

Geef een SASL-funktie die bij n als argument ($n \geq 0$) een lijst oplevert die bestaat uit alle drietallen (i, j, k) met $0 \leq i < j < k \leq n$ en i+j+k = n. Bijvoorbeeld voor n=4 moet de lijst de volgende drietallen bevatten (en geen andere):

(0, 0, 4), (0, 1, 3), (0, 2, 2), (1, 1, 2)

(U mag van de standaardfunkties gebruik maken.)

Opgave 2 Over lijsten en SVP-typering

a. Herinner je de lijstiteratie associerend naar rechts, litar:

```
litar (x: y: ... z: nil) f a = (f x (f y ... (f z a) ...))
```

Geef in termen van litar een funktie tail zó dat

```
tail (x: staartlijst) = staartlijst
tail (nil)            = nil
```

b. Men kan lijstexpressies uit HOF weglaten, en ze met behulp van andere HOF-expressies simuleren. Stel dat dat als volgt gebeurt in een SVP-getypeerde variant van HOF:

```
df mk-lijstADT
    : (z: tp ->
       < {type van lijsten met elementtype z} repr: tp
       , {type van de lege lijst} repr
       , {type van de opKopVan operatie} (z, repr -> repr)
       , {type van litar}
          (repr, t: tp, (z, t -> t), t -> t)
       >)
    = (fn z: tp.   <---, ---, ---, --->)
```

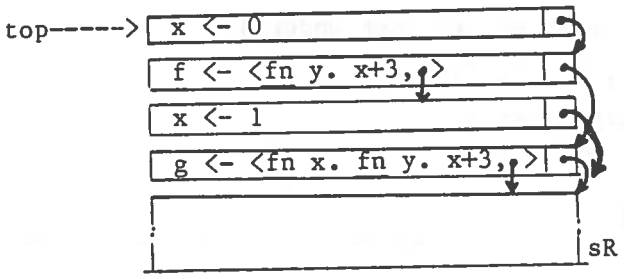
Dus (mk-lijstADT bool).1 is het type van boolean lijsten en (mk-lijstADT bool).4 is het type van de SVP-getypeerde versie van litar voor boolean lijsten.

Gevraagd: geef in termen van mk-lijstADT het type en de definitie van de SVP-

typeerde versie van tail (uit opgave 2a).

Opgave 3 Over het LIFO implementatiemodel

a. Geef een HOFexpressie zo dat gedurende de evaluatie volgens eval3 (Paragraaf 4.5) stapel sR ooit eens de volgende vorm heeft.



De stapelstruktuur van sR (de dynamic links ofwel calling pointers; in de syllabus met gestippelde pijlen getekend) is hierboven niet aangegeven (behalve die van (top sR)). Geef bij uw expressie de stapelstruktuur van bovenstaande sR wel aan.

b. Wat is het resultaat van (eval3 (df x=2. f x)) indien sR initieeel bovenstaande —en door u met dynamic links aangevulde-- vorm heeft?

Opgave 4 Over het Graafimplementatiemodel

a. Beschrijf in hooguit 5 regels tekst het verschil in de L-evaluatie van twee expressies, zeg e1 en e2, die slechts van elkaar verschillen doordat in e1 op zekere plaats de expressie (fn x. ((1+2) +3) +x) voorkomt, terwijl op de overeenkomende plaats in e2 de expressie (fn x. ((x+3) +2) +1) voorkomt. Schetsmatig zien de twee expressies e1 en e2 er dus als volgt uit

e1 = (--- (fn x. ((1+2) +3) +x) ---)
e2 = (--- (fn x. ((x+3) +2) +1) ---)

b. Motiveer (of bewijs) uw bewering van deel a door geschikte expressies e1 en e2 te kiezen en essentiële situaties tijdens de L-evaluatie in graafrepresentaties uit te tekenen (of anderszins heel precies aan te geven).

Opgave 5 Over modules

Beschouw de module expressies (mod ~~ export e end ~~) en (inv x = em. eb) van Hoofdstuk 10. Gevraagd wordt een module-expressie em zó dat in het bereik van inv s = em er een stapel-variabele bestaat

- die geïnitialiseerd is als een lege stapel;
- die veranderd kan worden qua inhoud door de operaties s.1 (= push) en s.2 (=pop);
- die maximaal 100 elementen, zeg integers, kan bevatten (zodat een implementatie met array[1..100] of integer mogelijk is).

Om precies te zijn, s.1 is een funktie (procedure) die een integer als argument verwacht terwijl s.2 een integer variabele verwacht (daaraan wordt het top element van de stapel toegekend).
Schrijf de gevraagde expressie em in HOF uitgebreid met variabelen en var-parameters, of in een geschikte uitbreiding van Pascal of Algol 68. (Het gaat erom dat je laat zien hoe module-expressies "werken".)

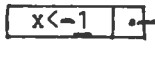
Uitwerking Tentamen Struktuur van Programmeertalen d.d. 28 mei 1984

```
1. def f n = map g (count 0 (n div 3))
             where
             g i = map (h i) (count i (i+ (n-i) div 2))
                   where
                   h i j = (i, j, n-i-j)
```

```
2.a. tl = fn ℓ. (litar ℓ f a).2
     met f = fn x g. <x: g.1, g.1>
         a = <nil, nil>
```

```
  b.tl: (z:tp, (mk-lijstADT z).1 -> (mklijstADT z).1)
      = fn z:tp. ℓ:(mk-lijstADT z).1.
       ((mklijstADT z).4 ℓ (mk-lijstADT z).1, (mklijstADT z).1) f a).1
  met
     f = fn x: z, g:<(mklijstADT z).1 (mklijstADT z).1>
         <(mklijstADT z).3 x (g.1), g.1>
     a = <(mklijstADT z).2, (mklijstADT z).2>
```

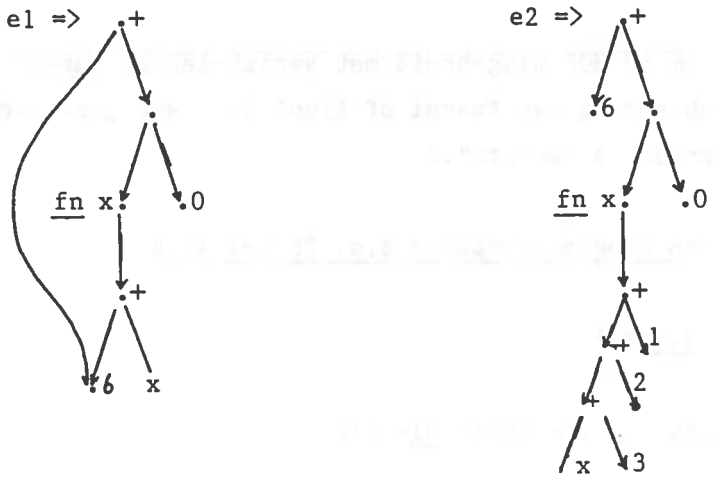
```
3a. (df g = (fn x, fn y. x+3). df f = (g 1). df x = 0. ~~)
```

Dynamic links: top~~> (x<-0) ~~>(f<-<...>)~~>(g<-<...>)~~>
(dus de omgeving  ... is van sR gepopped).

b. Het resultaat is 4, onafhankelijk van de dynamic links.

4a. Het aantal reduktiestappen van e2 is minstens zo groot als dat van e1 omdat in e1 het deel ((1+2)+3) bij de eerste aanroep van die funktieexpressie al tot 6 reduceert en dus door 6 wordt vervangen. Dat kan in e2 niet gebeuren.

b. Beschouw e1 = (df f =(fn x. ((1+2)+3)+x).(f 0)+(f 0)) en e2 = (df f = (fn x. ((x+3)+2)+1).(f 0)+(f 0)). De graaafrepresentatie ná de eerste aanroep van f ziet er dan zó uit.



5. em = (mod var a: array [1..100] of integer; i: integer;
proc push (x: integer);
begin i:= i+1; a[i]:= x end;
proc pop (var x: integer);
begin x:=a[i]; i:=i+1 end;
begin i:=0;
export<push, pop>end
end)

Tentamen Struktuur van Programmeertalen, 5 november 1984.

Iedere opgave telt even zwaar mee in de beoordeling. U mag één opgave overslaan, terwijl u dan toch een 9 kunt behalen.

Opgave 1 Programmeren in SASL

Schrijf een funktie f die bij positief geheel getal g (g≥1) als parameter een lijst oplevert bestaande uit alle partities van g. Onder een partitie van g verstaan we een lijst positieve gehele getallen (allen ≥ 1), opklimmend gerangschikt en met gezamenlijke som g. Bijvoorbeeld,

```
f 1 = {de lijst met als enig element de partitie met enig lid 1 :} (1,),
f 4 = een lijst met als elementen (in een of andere volgorde):
      - 4,        {de partitie met enig lid 4}
      - 1,3       {de partitie met leden 1 en 3}
      - 2,2
      - 1,1,2
      - 1,1,1,1
```

Opgave 2 Typering van ADTs

Beschouw de volgende SVPgetypeerde definitie.

```
df boolADT : < repr: tp
             , {true:} repr
             , {false:} repr
             , {not:} (repr --> repr)
             , {or:} (repr, repr --> repr)
             >
           = < (z:tp --> z --> z --> z)
             , fn z:tp x:z y:z. x
             , fn z:tp x:z y:z. y
             , ---
             , ---
             >
```

a. Neem aan dat boolADT het datatype der waarheidswaarden representeert: de 2de, 3de, 4de en 5de komponent representeren respektievelijk true, false, not en or. Geef nu in het gezichtsveld van boolADT een SVPgetypeerde definitie van een funktie die and representeert. (Let wel, u moet dus een afzonderlijke definitie

geven die van boolADT gebruik maakt en niet de definitie van boolADT met een nieuwe komponent uitbreiden!)

b. Geef een geschikte uitwerking voor de 4de komponent van de definiërende expressie van boolADT. (Dus: een representatie voor not).

Opgave 3 Typering

Beschouw de volgende definities.

```
df linsearch0 =
     fn (f,m,n,x).
          (df g rec= fn i. if i=n+1 then false else
                           if f(i)=x then i else
                            g(i+1)
          . g(m)
          )

df linsearch1 = fn (f,m,n,x). if m = n+1 then m else
                              if f(m) = x then m else
                              linsearch1 (f, m+1, n, x)
```

a. Welke van bovenstaande definities is/zijn goed typeerbaar volgens Milner's typepolymorfie (Hoofdstuk 5 van de syllabus over Typering)?
Motiveer uw antwoord.

b. Geef een SVPgetypeerde variant van linsearch1 zó dat-ie toepasbaar is op argumenten f van type (int --> t) voor vele keuzen van t.

Nota bene. Volledigheidshalve moet de typeringstabel (Par. 3.1 van de syllabus over Typering) uitgebreid worden met de volgende regel.

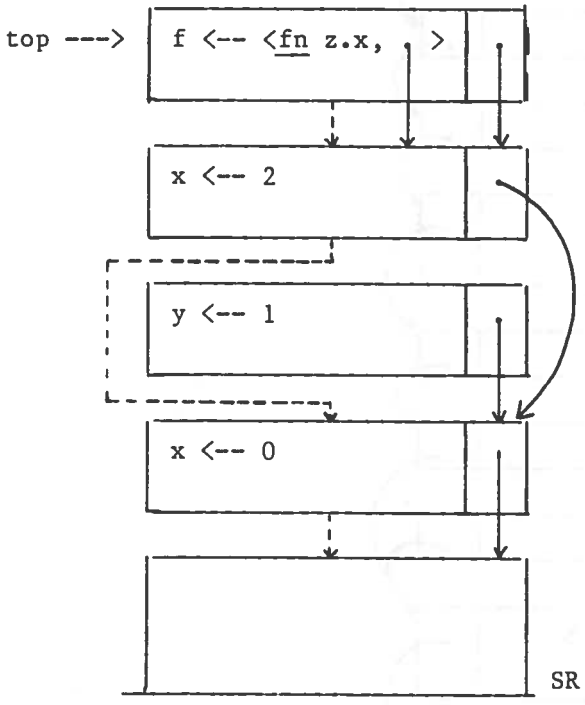
nr	e	eisen
...	$e1^{t1} = e2^{t2}$	t = bool, t1 = t2

Opgave 4 Over modulen

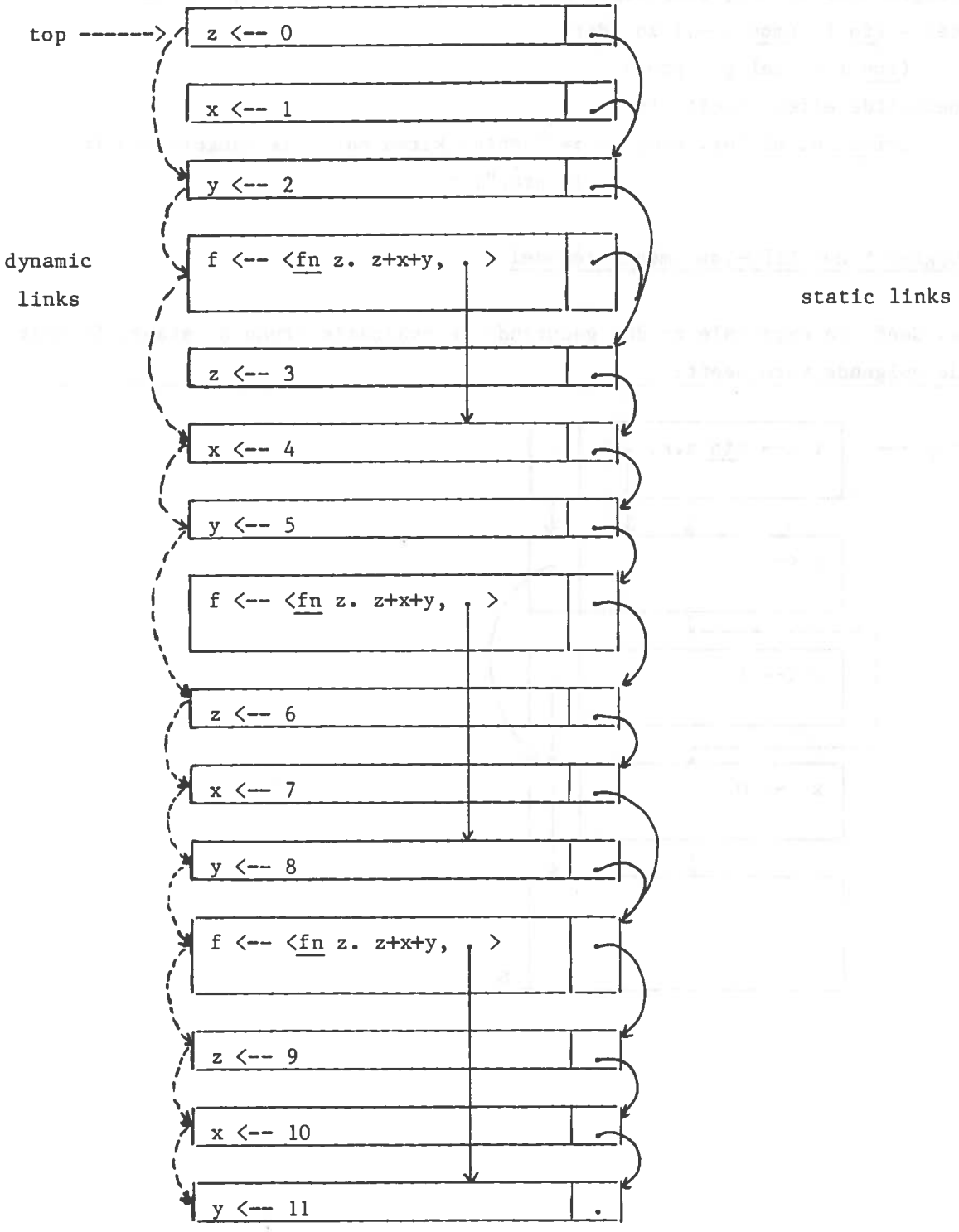
Programmeer in ASS, uitgebreid met de module-konstrukties, een funktie
tel = (fn f. (mod ---)) zo dat
    (inv f = (tel g). prog)
hetzelfde effekt heeft als
    (dfvar n. df f=g. prog; n := "aantal keren dat f is aangeroepen in
                                in prog"; n)

Opgave 5 Het LIFO-implementatiemodel

a. Geef een expressie zo dat gedurende de evaluatie ervan de stapel SR ooit eens de volgende vorm heeft:



b. Wat is het resultaat van eval3 (f z) wanneer SR onderstaande vorm heeft?



Examen/tentamen Structuur van Programmeertalen
Donderdag 20 juni 1985, 13.30 - 17.00 uur

Normering. Alle opgaven tellen even zwaar en per opgave tellen alle onderdelen even zwaar.

Opgave 1. Schrijf een SASL-funktie f zó dat, voor iedere lijst X, f X = een lijst van alle tweetallen (Y,Z) met Y ++ Z = X. (U mag hierbij de zgn. verzamelingsnotatie niet gebruiken.)

Opgave 2 Beschouw de volgende programmatekst, in de notatie van de syllabus over Typering.

```
def compose = (f, g): (x): f(g(x))
in  ...
    def succ = (n): n+1, pred = (n): n-1
    in  def f = compose (succ, pred)
        in  ... f(4) ... ni
ni  ni
```

Geef bij elk identifier-voorkomen in deze tekst het type aan dat volgens de zgn. type-polymorfie (Hoofdstuk 5 van de syllabus over Typering) aan dat identifier-voorkomen wordt toegewezen.

Opgave 3. Een automaat over (globaal gegeven en bekend verondersteld) inputalfabet A en outputalfabet B bestaat uit een tweetal, nml.

- een toestandsverzameling, zeg Q,
- een transitiefunktie, zeg t, die bij een inputsymbool a uit A en een toestand q uit Q een (nieuwe) toestand q' uit Q en een outputsymbool b uit B oplevert, (symbolisch: t(a,q) = <q',b>).

Geef een SVP-type voor dit soort automaten.

Opgave 4. Beschouw het volgende HOF programma.

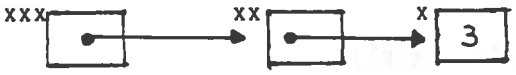
```
df g rec= (fn f.  g (fn x. f x)). g g
```

Dit programma termineert niet en de omgevingenstapel sR (bij de zgn. LIFO-implementatie) zal onbeperkt aangroeien. Teken nauwkeurig de vorm die de stapel sR uiteindelijk krijgt.

Opgave 5. Beschouw het volgende Algol 68 programmafragment.

```
ref int x = loc int,
ref ref int xx = loc ref int,
ref ref ref int xxx = loc ref ref int;
x:=1;A xxx:=xx;B x:=2;C xx:=x;D x:=3;E ...
```

a. Geef de toestand, bij voorkeur in de vorm van een plaatje volgens de conventies van Par. 7.2, op de "tijdstippen" A,B,C,D en E.
b. Wat verandert er indien alle loc's (drie stuks) door heap worden vervangen?
c. Geef de ASS-programmatekst die met bovenstaande Algol 68 tekst overeenkomt (qua semantiek).
d. Geef een Pascal - programmafragment zó dat op zeker moment de toestand als volgt is:



Dat wil zeggen, er zijn drie variabelen (cellen) die in de programmatekst met de identifiers xxx resp. xx en x worden aangeduid, en xxx bevat een pointer naar xx, xx bevat een pointer naar x en x bevat 3. (Wenk: introduceer hulpvariabelen var ppp: ↑↑↑integer, pp: ↑↑integer, p: ↑integer en laat xxx een alias zijn voor ppp↑ etc.)

Opgave 6. a. Beschouw een imperatieve programmeertaal zoals ASS (of Pascal), waaraan de zgn. module-constructies van Hoofdstuk 9 zijn toegevoegd. Stel dat print, space en newline standaardprocedures zijn met het volgende gebruikelijke effect. (Een parameterloze procedure krijgt in ASS bij aanroep de lege argumentlijst ().)

newline(): bewerkstelligt een overgang naar het begin van de volgende regel,
print(x): drukt x af,
space(n): drukt n spaties af.

Gevraagd wordt een expressie expr te geven zó dat het heel makkelijk wordt om in de scope van

(*) inv x = expr. df begin, end, newline = x.0, x.1, x.2

afdrukken met indentatie (= intanding, inspringen) te verzorgen:

er is een "onzichtbare" globale variabele ind, geinitialiseerd op 0,
begin(): heeft effect van ind:=ind+3; newline(); space(ind),
end(): heeft effect van ind:=ind-3; newline(); space(ind),
newline(): heeft effect van newline(); space(ind).

Nota bene, de newline genoemd in de effect-beschrijving is de "oude" standaard-procedure, deze wordt her-gedefinieerd op regel (*).
b. Reduceer een programma dat begint met regel (*) zó ver volgens de Compositionele strategie dat de aangegeven inv verdwijnt.

Uitwerking tentamen/examen Str. van Progr.talen, 20 juni 1985

Opgave 1.

```
def  f () = ()
     f (x:X) = ((), x:X): map g (f X)
               where  g(Y,Z) = x:Y, Z
```

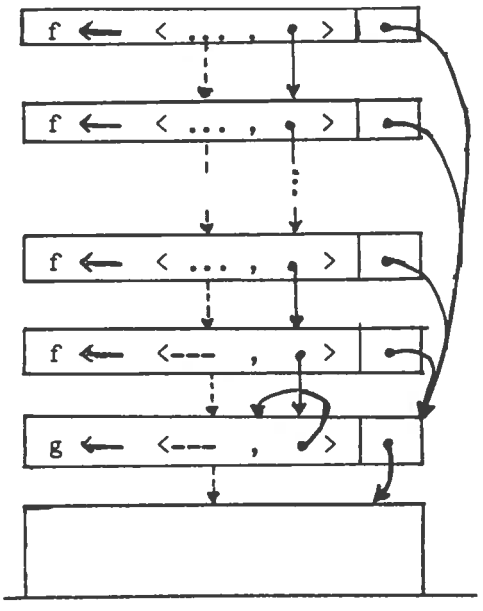
Opgave 2.

```
def compose ((a->b),(c->a)-->(c->b)) =
    (f^(a->c),g^(c->a)): (x^c):f^(a->b)(g^(c->a)(x^c))
in  ...
    def succ^(int->int)= (n^int): n^(int+1), pred^(int->int)=(n^int):n^(int-1)
    in def f^(int->int)= compose^((int->int),(int->int)->(int->int))
                          (succ^(int->int), pred^(int->int))
       in  ... f^(int->int)(4) ... ni
ni  ni
```

Opgave 3.

<Q: tp, (Q, A' --> <Q, B'>)>
waarbij A' en B' globaal gegeven typen zijn voor het inputalfabet resp. outputalfabet.

Opgave 4.

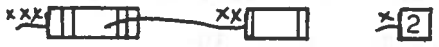


Hierbij staat
... voor (fn x. f x)
--- voor (fn f. g (fn x. f x))

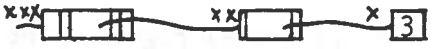
Opgave 5

a. A: 

B: 

C: 

D: 

E: 

b. Met loc zijn de variabelen semi-statisch, met heap zijn ze semi-dynamisch, d.w.z. ze blijven bestaan/gereserveerd tot aan het einde van de programma-executie. Aan de plaatjes verandert er niets.

c. dfloc x. dfloc xx. dfloc xxx. x:=1; xxx:=xx; x:=2; xx:=x; x:=3; ...
of: dfloc x, xx, xxx. x:=1; xxx:=xx; x:=2; xx:=x; x:=3; ...

```
d.  var ppp: ↑↑↑integer; pp: ↑↑integer; p: ↑integer;
    begin new(ppp); new(pp); new(p);
          ppp↑:=pp; pp↑:=p; p↑:=3;              ... (*)
          q (ppp↑, pp↑, p↑)
          ...
          waarbij gedeclareerd is:
          procedure q (var xxx: ↑↑integer;
                       var xx: ↑integer;
                       var x: integer)
          begin ... end
```

Regel (*) kan eventueel ook binnen q geformuleerd worden als: xxx:=pp; xx:=p; x:=3. Binnen q zijn xxx en ppp↑ aliassen, en net zo voor xx en pp↑, x en p . Het kan ook zonder pp en p:

```
        var ppp: ↑↑↑integer;
        begin new(ppp); new(ppp↑); new(ppp↑↑); ppp ↑↑↑= 3;
              q (ppp↑, ppp↑↑, ppp↑↑↑)
              ...
```

Opgave 6.

a. De expressie is bijvoorbeeld

```
(mod dfvar ind. ind:=0;
                export <(fn (). ind:=ind+3; newline(); space(ind))
                       ,(fn (). ind:=ind-3; newline(); space(ind))
                       ,(fn (). newline(); space(ind))
                       >
                end)
```

b. De reductie van de (in de opgave met (*) gemerkte) regel begint met een (inv)-contractie met als resultaat:

```
        dfvar ind'. ind':=0;
           df x = < (fn (). ind':=ind'+3; newline(); space(ind'))
                  , ---
                  , ---
                  > .
              df begin, end, newline = x.0, x.1, x.2. ...
```

waarbij ind' een variabele is die niet vrij voorkomt in de expressie ... .

Technische Hogeschool Twente

ONDERAFDELING INFORMATICA
Kenmerk: INF86/TIF-Fka/065/tth
Datum : 12 juni 1986

Examen Structuur van Programmeertalen
Maandag 16 juni 1986, 13.30 - 17.00 uur

Normering. Alle opgaven tellen even zwaar mee in de beoordeling.

Opgave 1. "Insertion sort" is de sorteermethode waarbij steeds een nog te verwerken element op de juiste plaats wordt ingevoegd tussen de al verwerkte en gesorteerde elementen. Bijvoorbeeld, de achtereenvolgende stadia gedurende de sortering van 4 3 1 7 3 1 volgens deze methode zien er als volgt uit.

Nog te verwerken	al verwerkt en gesorteerd
4 3 1 7 3 1	-
3 1 7 3 1	4
1 7 3 1	3 4
7 3 1	1 3 4
3 1	1 3 4 7
1	1 3 3 4 7
-	1 1 3 3 4 7

a. Geef een functiedefinitie (functioneel programma) voor insertion sort. (U mag een functionele programmeertaal naar keuze gebruiken.)
b. Noteer de zojuist gegeven functiedefinitie in de taal HOF.

Opgave 2. a. Geef een expressie die onder statistische scope stokt maar onder dynamische scope niet.
b. Kan het ook andersom: een expressie die onder dynamische scope stokt maar onder statistische scope niet?

Opgave 3. Geef de inhoud die de omgevingstaal sR heeft (bij evaluatie volgens eval3) direkt bij aanvang van de onderstreepte applicaties:
a. df a=3. df f = (fn x. df a=7. a+x). df b=4. f a + f b
b. (fn x. df f = (fn y. y+20). f x) 10

Opgave 4. a. Geef een module-expressie die op de plaats van ??? moet worden gezet opdat in de romp van de inv-expressie e iedere aanroep exit() de evaluatie van e onmiddellijk beeindigt (met een of ander resultaat dat u zelf mag kiezen).

```
(df exitGenerator = ??? .
    --- (inv exit = exitGenerator. --- exit() ---) ---
)
```

inv-expressie e

b. Beredeneer de correctheid van de door u gegeven module-expressie.

Opgave 5. a. Geef een SVP-type voor het abstracte data type der verzamelingen, die ten hoogte N elementen (alle uit type T) kunnen bevatten, met de volgende operaties:

- insert: het toevoegen van een element aan een verzameling
- delete: het weglaten van een element uit een verzameling
- empty: de lege verzameling
- isEmpty: de test of een verzameling leeg is
- isFull: de test of een verzameling N elementen bevat
- any: levert een (willekeurig) element van een verzameling op

Hierbij zijn T en N als parameters te beschouwen. U moet dus in de definitie

(*) df mkVerzamelingen ADT : ??? = ---

het type ??? geven.

b. Geef in de scope van definitie (*) een SVP-getypeerde definitie van de functie union (voor de vereniging van twee verzamelingen).

Technische Hogeschool Twente

Kenmerk: INF86/TIF-Fka/089/tth
Datum: 11 augustus 1986

Uitwerking Examen Structuur van Programmeertalen 16 juni 1986

Opgave 1a:

In SASL:

```
insort () = ()
insort (x:X) = insert x (insort X)
               where
               insert x () = x:()
               insert x (y:Y) = x ≤ y -> x:y:Y; y: insert x Y
```

Alternatief:

```
insort X = isort X ()
           where
           isort () S = S
           isort (x:X) S = isort X (insert x S)
           insort :- als boven
```

In HOF:

```
df insert rec= (fn x Y. if eqnil Y then x:Y else
                        if x ≤ hd Y then x:Y else
                       {if x > hd Y then} hd Y: insert x (tl Y))
; insort rec= (fn X.    if eqnil X then X else
                        insert (hd X) (insort (tl X)))
```

Opgave 2

Voorbeeld 1:

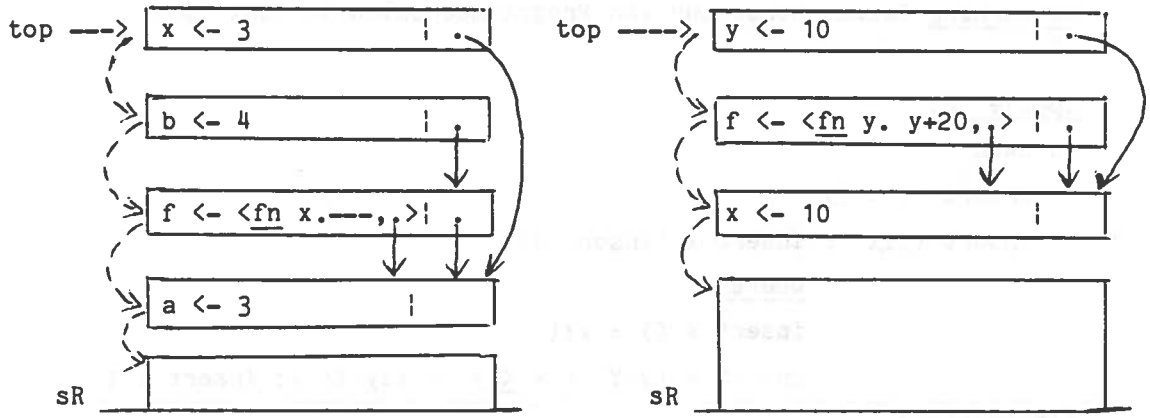
```
df f = (fn x. "stok"). df g = (fn y. f 0). df f = (fn x.x). g 1
```

Voorbeeld 2:

```
df x = <0,1>. df g = (fn y. x+y). df x = 2. g 3
```

Onder statische scope stokken de evaluaties van beide expressies, terwijl ze onder dynamische scope 0 respectievelijk 5 opleveren. Als we de definierende expressies voor f, respectievelijk voor x, omwisselen dan stokt de evaluatie onder dynamische scope, terwijl er onder statische scope weer 0 resp. 5 wordt opgeleverd.

Opgave 3.



Opgave 4.

Kies ??? = (mod dfexc stop. export (fn(). T[stop])end [stop => "waarde"])

De C-reductie verloopt in grote lijnen als volgt.

(df exitGenerator = ??? . ~~~(inv exit = exitGenerator, ---)~~~)
(df)=> ~~~(inv exit = ??? . ---) ~~~
...
(nu passen we de inv-regel vervroegd toe; dat mag omdat ??? al in resultaatvorm staat)
...
(inv) => ~~~(dfexc stop. (df exit = fn().T[stop]. ---)[stop => "waarde"])~~~

We zien nu dat aanroepen van exit() binnen --- onmiddellijk de evaluatie van --- beeindigen en "waarde" als resultaat opleveren. (Het is onmogelijk dat binnen --- de voortijdige beeindiging wordt omgezet omdat oorspronkelijk ??? buiten --- lag en dus de scope van 'stop' zich niet over --- uitstrekte.)

Opgave 5.

```
df mkVerzamelingenADT
   : (elt: tp, int ->
         <repr: tp
         ,{insert:} (elt, repr -> repr)
         ,{delete:} (elt, repr -> repr)
         ,{empty:} repr
         ,{isEmpty:} (repr -> bool)
         ,{isFull:} (repr -> bool)
         ,{any:}    (repr -> elt)
         > )
   = --- .
```

In de derde regel hieronder staat "mVadt" voor 'mkVerzamelingenADT (elt, N)', (vergelijk regel 6).

```
df mkUnion
   : (elt: tp, N: int ->
         ("mVadt".1, "mVadt".1 -> "mVadt".1)
     )
   = (fn elt:tp, N:int.
         df mVadt == mkVerzamelingenADT (elt, N)
         ;  Verz  == mVadt.1
         ;  un:  (Verz, Verz -> Ver)
            rec= (fn v:Verz, w:Verz.
                      if mVadt.5 v then w
                      else df x:elt = (mVadt.7 v)
                                . mVadt.2 (x, un (mVadt.3 (x,v), w))
                  )
         . un
     )
```

De functie mkUnion levert de gewenste union-functie op voor elke soort verzamelingen, bijvoorbeeld:

```
df int100union: --- = mkUnion (int,100)
df real50union: --- = mkUnion (real, 50)
df bool76union: --- = mkUnion (bool,76)
```

TREFWOORDENLIJST

LITERATUURVERWIJZINGEN

Aberson, H., Sussman, G.J.: Structure and Interpretation of Computer Programs. The MIT Press, McGraw Hill Book Co, 1985.

Backus, J.: Can programming be liberated from the Von Neumann style? A functional Style and its algebra of programs. Comm. ACM 21 (1978) 8, p.613-642.

Bakker, J.W. de, Vliet, J.C. van (eds): Algoritmic Languages. North-Holland Publ. Co., Amsterdam, 1981,(413 pages).

Barendregt, H.P.: The Lambda Calculus - its syntax and semantics. Studies in Logic, Vol. 103. North-Holland Publ. Co., Amsterdam, 1981, 615 pages.

Barendregt, H: Introduction to Lambda Calculus. In: Nieuw Archief voor Wiskunde, 2 (1984) 4 p.337-372

Bauer, F.L., Wösnner, H.: Algorithmic Language and Program development. Springer-Verlag, Berlin, 1982. 497 pages.

Bjorner, D.: Programming Languages - Formal Development of Interpreters and Compilers. In: Proc. Int. Computing Symposium ICS 77, North-Holland Publ. Co., Amsterdam, 1977.

Bjorner, D., Jones, C.B.: The Vienna Development method - the Meta language. Springer Verlag, LCNS 61 (1978).

Black, A.P.: Exception Handling - The Case Against. TR - 82-01-02, Univ. of Washington, U.S.A., (1982) 238 pages.

Bron, C. & Fokkinga, M.M.: Exchanigng robustness of a program for a relaxation of its specification. T.W. Memorandum 178, T.H.Twente, 1977.

Burge, W.H.: Recusive Programming Techniques. Addison-Wesley, 1975, 277 pages.

Darlington, J., Henderson, P., Turner, D. (eds.): Functional programming and its applications. Cambridge Univ. Press, U.K., 1982.

Desain, P.: Representatie van induktief gedefinieerde objekten en rekursie in een 2de orde getypeerde lambda-calculus. D-verslag, T.H.T.-T.W., augustus 1983.

Dijkstra, E.W.: A mild variant of combinatory logic. EWD-note 735, May 1980.

Fokkinga, M.M.: Axiomatization of Declarations and The Formal Treatment of an Escape Construct.
In E.J. Neuhold (ed.): Formal Descriptions of Programming Concepts. North-Holland, Amsterdam, (1978) p. 221-235.

Fokkinga, M.M.: Over het nut en de mogelijkheden van typering.
Kollegesyllabus, T.H.T., 1983, herziene druk: febr. 1984, 49 pagina's.

Fortune, S., Leivant, D., O'Donnell, M.: The expressiveness of simple and second order type structures.
J. ACM 30 (1983) 1, p. 151-185.

Ghezzi, C., Jazayeri, M.: Programming Langage Concepts.
John Wiley & Sons, Inc., New York etc., 1982, (327 pages).

Goodenough, J.B.: Exception handling - Issues and proposed notation.
Comm. ACM 18 (1975) 12, p. 683-696.

Gordon, M.J.C.: The denotational description of programming languages - an introduction.
Springer-Verlag, New York, 1979, 160 pages.

Gries, D.: The Science of Programming.
Springer Verlag, Berlin etc. (1981), 366 pages.

Henderson, P.: An Operating System.
In: (Darlington et al 1982).

Henderson, P.: Functional Geometry
ACM Conference on LISP and Functional Programming (198?).

Hoeven, G.F. van der: Preliminary Report on the Language Twentel.
INF-memorandum nr. INF-84-5, Twente Univ. of Techn., Netherlands, 1984, 87 pages.

Horowitz, E.: Programming Languages - A Grand Tour.
Springer-Verlag, Berlin etc. (1983), 664 pages.

Joosten, S.: The applicative description of discrete systems.
Graduate Thesis, T.H.Twente dept. EL, January 1985.

Landin, P.J.: The mechanical evaluation of expressions.
Computer Journal 6 (1964) 4, p.308-320.

Landin, P.J.: A correspondence between Algol 60 and Church'Lambda-Notation.
Comm. ACM 8 (1965) p.89-101, 158-165.

Landin, P.J.: The next 700 programming Languages.
Comm. ACM 9 (1966) p.157-166.

Langmaack, H.: On correct procedure parameter transmission in higher order programming languages.
Acta Informatica 2 (1973) p.110-142.

Langmaack, H.: On procedures as open subroutines I.
Acta Informatica 2 (1973) p.311-333.

Idem Part II,
Acta Informatica 3 (1974) p.227-241.

Ledgard, H.F.: Ten mini-languages - a study of topical issues in programming languages.
Computing Surveys 3 (1971) 3, p.115-146.

Ledgard, H., Marcotty, M.: The programming language landscape.
SRA, 1981.

MacKeag, M. & Welsh, J.: Structured System Programming.
Prentice Hall, Englewood Cliffs (N.J.), 1980, 324 pages.

MacLennan, B.J.: Principles of Programming Languages - design, evaluation and implementation.
C.B.S. College Publishing, Holt Rinehart & Winston, (1983), 544 pages.
(Prettig leesbaar boek, geeft ontwikkeling van programmeertalen, heel redelijke uiteenzettingen over Smalltalk, Lisp, Prolog.

Manna, Z., Ness, S., Vuillemin, J.: Inductive methods for proving properties of programs.
In: Proc. ACM Conference on proving Assertions About Programs, Jan. 1972, p.27-50.

Matchey, M. & Young, P.: An introduction to the general theory of algorithms.
Elsevier North-Holland Inc., Amsterdam, 1978, p.220-227.

Meertens, L.G.L.T.: Procedurele Datastrukturen.
In: Colloquium Datastrukturen, 17 maart 1978, Matematisch Centrum, Amsterdam, 1978, 16 paginas.

Milne, R.E., Strachey, C.: A theory of programming language semantics, part a and b.
Chapman & Hall, London, 1976.

Pagan, F.G.: Formal specifications of programming languages.
Prentice Hall, Englewood Cliffs, N.J., 1981, 245 pages.

Plotkin, G.D.: A structural approach to operational semantics.
DAIMI-FN-19, Aarhus University, DK, 1981.

Pratt, T.W.: Programming Languages: design and implementation.
Prentice-Hall, Englewood Cliffs, N.J., 1975.

Reynolds, J.C.: Gedanken - a simple typeless language based on the principle of completeness and the reference concept.
Comm. ACM 13 (1970) p.308-319.

Reynolds, J.C.: Definitional Interpreters for higher order programming languages.
In: Proc. ACM 25th Nat. Conf., 1972, p.717-740.

Reynolds, J.C.: User-defined data types and procedureal data structures as complementary approaches to data abstraction.

In: New Directions in Algoritmic Languages, 1975, ed. S.A. Schuman, IRIA, France, 1976, p.154-165.

Reynolds, J.C.: The Craft of programming.
Prentice Hall Inc., London, 1981a, 434 pages.

Reynolds, J.C.: The essence of Algol.
In:(de Bakker en van Vliet , Algorithmic Languages, 1981b, p.345-372)

Stoy, J.E.: Denotational Semantics - the Scott-Strachey approach to programming language theory.
The MIT Press, Cambridge (Ma) and London (UK), 1977, 414 pages.
(Vooral Hoofdstuk 4 en 5.)

Swierstra, S.D.: Lawine - an experiment in language and machine design.
Doctoral Thesis, Twente Univ. of Techn., 1981, 243 pages.

Tennent, R.D.: The denotational semantics of programming languages.
Comm. ACM 19 (1976) 8, p.437-453.

Tennent, R.D.: Principles of Programming Languages.
Prentice Hall, 1981, 271 pages.
(Bevat literatuurlijst over programmeertalen in Appendix A).
(Bevat per hoofdstuk bibliografie.)

Turner, D.A.: SASL Language Manual.
St. Andrews University, UK, 1976, 43 pages.

Turner, D.A.: A new implementation technique for applicative languages.
Software-Practice and Experience, 9 (1979) 1, p.31-49.

Turner, D.A.: Recursive equations as a programming language.
In: (Darlington et al 1982a)

Turner, D.A.: Functional Programming and proofs of program correctness.
In: Tools and Notions for Program Construction - an advanced course (ed. D. Néel). Cambridge Univ. Press, Cambridge, U.K., 1982, p.187-210.

Verbeek, L.A.M.: Inleiding Theoretische Informatica. Collegesyllabus T.H. Twente, afd. INF, 1983