INFORMATIE Vol 27 (1985) Nr 10 pp 862-873.

FUNCTIONEEL PROGRAMMEREN IN EEN VOGELVLUCHT

door drs. M. M. Fokkinga

We geven een grote verscheidenheid van eenvoudige maar niet-triviale functionele programma's waarmee de lezer zich een beeld kan vormen van het karakter en de stijl van functioneel programmeren. We richten ons tot de oningewijde: bijzondere en taal-specifieke notatie wordt tot het alleruiterste minimum beperkt.

1 INLEIDING

Functionele programmeertalen onderscheiden zich van conventionele (imperatieve, procedurele) programmeertalen in wezen doordat ze geen assignment en assigneerbare variabelen hebben. Ze zijn in snelle opmars omdat ze (naast nadelen ook) een aantal onmiskenbare voordelen hebben boven imperatieve talen. We willen hier géén vergelijking maken tussen beide klassen van talen en evenmin de voordelen van functionele talen opsommen en toelichten. Integendeel, onze bedoeling is het karakter en de stijl van functioneel programmeren duidelijk te maken aan de hand van voorbeelden; de lezer oordele dan zelf maar. We richten ons daarbij tot een oningewijde, die nog nooit iets van functionele programmeertalen gezien hoeft te hebben maar wel enigszins bedreven is in een of andere conventionele programmeertaal.
Omdat we ons tot de oningewijde richten, beperken we ons in de notatie tot het alleruiterste minimum. We maken bijvoorbeeld geen gebruik van characters en strings en waar mogelijk kiezen we de meest vanzelfsprekende schrijfwijze, zodat we soms afwijken van wat in functionele talen gebruikelijk is. Een ingewijde zal er echter totaal geen moeilijkheden mee hebben om de getoonde programma's in talen zoals SASL, KRC en TWENTEL (zie literatuurbeschrijving) om te zetten.
Soms maken we wezenlijk gebruik van 'lazy evaluation'. Het effect hiervan zal bij de voorbeelden nog toegelicht en duidelijk worden. Hier volstaan we met de volgende globale karakterisering. Door lazy evaluation worden argumenten, lijstelementen ja zelfs alle subexpressies niet bij voorbaat al tot hun waarde geëvalueerd, maar pas dan wanneer tijdens de berekening de noodzaak daartoe blijkt. Bovendien wordt door deling (sharing) bij de opslag van waarden/expressies bereikt dat zo'n argument etc. hoogstens éénmaal wordt geëvalueerd, ook al zou hij in de functieromp vele malen gebruikt worden. Niet alle functionele talen worden volgens lazy evaluation geëvalueerd; de recente echter wel.

We verwachten niet dat een oningewijde aan de hand van de voorbeelden zelf in staat zal zijn functioneel te programmeren. Daarvoor is waarschijnlijk een uitgebreidere uitleg over syntaxis en semantiek nodig. Een diepergaande kennis van de evaluatie is nodig om de doelmatigheid (efficiëntie) van de programma's te schatten. De lezer vertrouwe erop dat de voorbeeldprogramma's qua tijdsduur in grootte-orde even efficiënt zijn als de voor de hand liggende imperatieve programma's; het geheugengebruik is ten gevolge van lazy evaluation vaak groter dan in imperatieve talen en bovendien is zogenaamde garbage collection haast onmisbaar in functionele talen.
De rest van dit verhaal bestaat uit de volgende hoofdstukjes.

2. Functies
3. Lijsten
4. Verzamelingsnotatie
5. Standaard voorbeelden (permutaties, quicksort, ...)
6. Modulair programmeren
7. Backtracking (8-koninginnenprobleem)
8. Communicerende processen
9. Invoer en uitvoer (ook interactieve programma's)
10. Pointerstructuren (bomen en grafen)
11. Gegevensbanken
12. Hogere orde functies (taalherkenning)
13. Zelfsturende evaluatie (attributenevaluatie)
14. Tekeningen (Hilbertkrommen)
15. Typering
16. Oefeningen

In hfdst. 2-4 wordt de notatie ingevoerd; in hfdst. 5-14 komen voorbeelden, methoden en aspecten aan bod; in hfdst. 15 laten we zien dat onze programma's sterk getypeerd zijn, ook al lijkt dat misschien niet zo op het eerste gezicht; en in hfdst. 16 suggereren we per hoofdstukje een paar oefeningen. We besluiten met een korte beschrijving van mogelijke vervolgliteratuur.
Ik dank Mirjam Gerritsen, Pierre Jansen en Theo van der Genugten voor stimulerend commentaar.

2 FUNCTIES

Functies vormen een belangrijk bestanddeel van functionele programmeertalen. Als voorbeeld definiëren we hier een functie fib (n) die het n-de Fibonaccigetal oplevert. (De Fibonaccigetallen zijn 1,1,2,3,5,8,13,21,..; ieder getal is de som van de twee voorgaande.)

```
fib (1) = 1
fib (2) = 1
fib (n) = fib (n−2) + fib (n−1)
```

of in één clausule

```
fib (n) = if n=1 then 1 else
          if n=2 then 1 else fib (n−2) + fib (n−1)
```

Deze definitie is erg inefficiënt: door fib (4) worden de berekeningen voor fib (3) en fib (2) opgeroepen, maar door fib (3) wordt de berekening voor fib (2) nogmaals gedaan! Een efficiënte definitie luidt als volgt

```
fib (n) = f (1, 1, 1)
where f (i, a, b) = if i=n then a else f (i+1, b, a+b)
```

De hulpfunctie f (i, a, b) wordt steeds zo aangeroepen dat a het i-de en b het (i+1)-de Fibonaccigetal is. De berekening van fib (n) vergt nu een tijdsduur evenredig met n.

3 LIJSTEN

Lijsten vormen een heel algemene datastructuur waarmee arrays, records, pointers en streams weergegeven kunnen worden. Lijsten mogen zonder uitzondering als argument of resultaat van functies optreden en ook als element van lijsten. We gebruiken in dit verhaal de volgende notatie

```
[2, 4, 1, 3, 4]      lijst met vijf elementen
2: [4, 1, 3, 4]      = [2, 4, 1, 3, 4]
                     = 2 op kop van [4, 1, 3, 4]
[2, 4] ++ [1, 3, 4]  = [2, 4, 1, 3, 4]
                     = [2, 4] gevolgd door [1, 3, 4]
[2..7]               = [2, 3, 4, 5, 6, 7]
[]                   de lege lijst
```

Hier volgen een paar definities van functies met lijsten als argument en/of resultaat.
De kop (head), staart (tail) en lijstsubscriptie definiëren we als volgt:

```
hd (x: X) = x
tl (x: X) = X
sub (x: X, n) = if n=1 then x else sub (X, n−1)
```

Merk op dat geen van deze functies gedefinieerd is voor de lege lijst: uit de vorm van de parameter blijkt dat het argument een niet-lege lijst moet zijn waarvan de kop met x en de staart met X wordt benoemd. (Vaak is er voor deze functies een speciale notatie.) Dus hd ([2, 4, 1, 3, 4]) = 2, tl ([2, 4, 1, 3, 4]) = [4, 1, 3, 4] en sub ([2, 4, 1, 3, 4], 3) = 1. De berekening van sub (X, i) vergt een tijdsduur evenredig met de grootte van i; subscriptie op arrays gaat i.h.a. in constante tijdsduur.
Een functie lth voor de lengte van een lijst wordt gedefinieerd als volgt.

```
lth ([]) = 0
lth (x: X) = 1 + lth (X)
```

of in één clausule:

```
lth (X) = if X=[] then 0 else 1 + lth (tl (X))
```

Dus lth ([2, 4, 1, 3, 4]) = 5.
Een functie prod voor het produkt van lijstelementen wordt gedefinieerd door

```
prod ([]) = 1
prod (x: X) = x * prod (X)
```

Dus prod ([1..n]) levert n! en kan gebruikt worden als de definitie voor de faculteitsfunctie.

Een functie from zo dat from (n) = [n, n+1, n+2, ...]:

```
from (n) = n: from (n+1)
```

Dit voorbeeld laat zien dat dank zij lazy evaluation lijsten ook oneindig mogen zijn. De expressie hd (from (1)) evalueert met weinig berekeningsstappen tot 1, terwijl de daarbij voortgebrachte aanroep from (2) in het geheel niet geëvalueerd wordt. Wanneer we echter de waarde van from (1) opvragen en naar het uitvoermedium (beeldscherm) sturen, dan verschijnen er achtereenvolgens de getallen 1, 2, 3, 4, ... zonder ophouden.

Ter illustratie van het werken met (oneindige) lijsten volgen hier nog enige definities van de lijst fibL van alle Fibonaccigetallen.
a. De meest voor de hand liggende definitie luidt als volgt:

```
fibL = fibLfrom (1)
where fibLfrom (n) = fib (n): fibLfrom (n+1)
      fib (1) = 1
      fib (2) = 1
      fib (n) = fib (n−2) + fib (n−1)
```

Deze definitie is erg inefficiënt omdat voor alle n de inefficiënte berekening van fib (n) afzonderlijk wordt gedaan. Maar het is eenvoudig om de definitie van fib door een efficiëntere te vervangen.

b. Een andere manier om bovenstaande definitie efficiënter te maken luidt als volgt: vervang in de definitie van fib iedere fib(i) door sub(fibL, i). De Fibonaccigetallen die voor de recursie nodig zijn, worden dan in de lijst fibL zelf opgezocht. Dit geeft

```
fibL = fibLfrom (l)
where fibLfrom (n) = fib (n): fibLfrom (n+1)
      fib (1) = 1
      fib (2) = 1
      fib (n) = sub (fibL, n−2) + sub (fibL, n−1)
```

Deze techniek is algemeen toepasbaar bij willekeurig recursieve functies.
c. We kunnen de berekening voor de Fibonaccigetallen ook in de functie fibLfrom stoppen, en krijgen dan

```
fibL = fibLfrom (1, 1, 1)
where fibLfrom (n, a, b) = a: fibLfrom (n+1,
                                        b, a+b)
```

Het voortbrengen van een volgend element in de lijst fibL kost nu slechts twee optellingen en één functieaanroep. (Bij de vorige methode moet per Fibonaccigetal de subscriptie op fibL tweemaal gedaan worden en de kosten van een subscriptie zijn evenredig met de grootte van de index.)

d. De laatste en leukste manier, even efficiënt als de vorige, is gebaseerd op de observatie dat de staart van fibL de componentsgewijze optelling is van fibL zelf en 0: fibL.

```
fibL=     [1, 1, 2, 3, 5, 8, ...]
0: fibL = [0, 1, 1, 2, 3, 5, 8, ...] +
          [1, 2, 3, 5, 8, 13, ...]
          de staart van fibL
```

Dus we vinden als definitie:

```
fibL = 1: plus (fibL, 0: fibL)
where plus (x:X, y:Y) = x+y: plus (X, Y)
```

4 VERZAMELINGSNOTATIE

Een zeer handige notatie om uit lijsten nieuwe te vormen is de zgn. verzamelingsnotatie. Het is niet al te moeilijk om de verzamelingsnotatie om te zetten in de tot nu behandelde notaties voor functies en lijsten, maar de leesbaarheid en overzichtelijkheid gaat dan voor een deel verloren. Hier volgen een paar voorbeelden ter uitleg.

```
{3*x | x ← from (1)}
= de lijst van alle positieve drievouden

{ x | x ← [1..n]; n mod x = 0}
= de lijst van alle delers van n

{[x,y] | x ← X; y ← Y}
= de lijst van alle tweetallen [x, y ] met
x in X en y in Y.
```

Algemener geformuleerd, de waarde van

```
{--x--y-- | x←X; conditie-x; y←Yx; conditie-x-y; ...}
```

is de lijst L van waarden --x--y-- waarbij x van kop naar staart varieert over de elementen van X die voldoen aan conditie-x en, bij iedere x, y varieert over de elementen van Yx die voldoen aan conditie-x-y.... (Wanneer X leeg is, is L dat ook, evenals wanneer alle Yx leeg zijn.) De uitdrukkingskracht van deze notatie moge blijken uit de komende paragrafen.

5 STANDAARD VOORBEELDEN

We geven nu een paar bekende algoritmen geformuleerd als functionele programma's. Het zal blijken dat de algoritmen heel beknopt en aansluitend bij de informele beschrijvingen genoteerd kunnen worden. Met name komen er geen grootheden en identifiers voor die volkomen vreemd zijn aan de probleemstelling en louter nodig zijn voor manipulaties van de representaties; indices voor arrays zijn daarvan een typisch voorbeeld. (Juist de probleem-vreemde grootheden zijn een bron van programmeerfouten; het gebeurt maar al te vaak dat een index één plaats verkeerd wijst.)

a. De lijst van alle permutaties van [1..n] is

```
perm ([1..n])
where perm ([]) = [ [] ]
      perm (X) = {x: p | x ← X; p ← perm
                         ({y | y ← X; y ≠ x})}
```

In woorden: de lijst van permutaties van de lege lijst bevat alleen de lege lijst zelf en de permutaties van een niet-lege lijst X bestaan uit alle x:p waarbij x een willekeurig element van X is en p een willekeurige permutatie van X-zonder-x. Dus perm ([1,2,3]) = [[1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], [3,2,1]].

b. De bekende Quicksort algoritme, geformuleerd als een functie sort:

```
sort ([])    = []
sort (x: X)= sort ({y | y←X; y≤x}) ++
             [x] ++
             sort ({y | y←X; y>x})
```

In woorden: de gesorteerde versie van de lege lijst is de lege lijst zelf en de gesorteerde versie van een lijst x:X bestaat uit: eerst de kleintjes van X, onderling gesorteerd, gevolgd door x zelf, gevolgd door de groten van X, onderling gesorteerd. Het eerste element x wordt hier als criterum voor 'klein' en 'groot' genomen: y is klein als y≤x en groot anders. Dus sort ([2,4,1,3,4]) = [1,2,3,4,4].

c. Een mogelijke formulering van de Linear Search algoritme luidt als volgt.

```
linsearch (cond, X) = hd' ({x | x←X; cond(x)})
where hd' ([]) = []
      hd' (y:Y) = [y]
```

Als er geen element van X is dat aan cond voldoet wordt de lege lijst opgeleverd en anders wordt [y] opgeleverd waarbij y het eerste element is van X dat aan cond voldoet.

d. De driehoek van Pascal. Deze ziet er als volgt uit.

```
                1
              1   1
            1   2   1
          1   3   3   1
        1   4   6   4   1
      . . .           . . .
```

De eerste rij bestaat uit het getal 1 en verder is ieder getal de som van zijn linker en rechter bovenbuur (dit geldt zelfs voor het eerste en laatste getal van een rij, als we de rij erboven met nullen aan weerszijden uitgebreid denken). We definiëren de driehoek als een lijst van lijsten:

```
driehoek = [1]: {plus ([0]++rij, rij++[0]) | rij ←
driehoek}
where plus (x:X, y:Y) = x+y: plus (X,Y)
      plus ([], []) = []
```

Ter verklaring: als 'rij' de 'driehoek' doorloopt, is hij steeds gelijk aan de laatst voortgebrachte rij en is plus (..., ...) dus de eropvolgende rij. Door [0]++rij en rij++[0] elementsgewijze op te tellen, worden in feite steeds twee buren opgeteld: plus ([0,1,2,1], [1,2,1,0]) = [1,3,3,1].

e. De lijst van alle priemgetallen, bepaald volgens de methode van de zeef van Eratosthenes, definiëren we als volgt:

```
priemL = zeef (from (2))
where
zeef (x:X) = x: zeef ({y | y←X; y mod x ≠ 0})
```

Ter verklaring: zij x:X een lijst waaruit alle veelvouden van priemgetallen kleiner dan x al weggezeefd zijn, dan wordt hieruit de lijst van alle priemgetallen $\geq$x verkregen door x te nemen en de staart X van de veelvouden van x te ontdoen en opnieuw aan ditzelfde zeefproces te onderwerpen. Als we de waarde van priemL naar het uitvoermedium sturen, verschijnen er de getallen 2, 3, 5, 7, 11, 13, ... zonder ophouden.

6 MODULAIR PROGRAMMEREN

Stel dat we het 117-de priemgetal willen weten. We kunnen dan de definitie van priemL geschikt 'ombouwen', maar eenvoudiger en vrijwel even efficiënt is het om alreeds bestaande functies (= 'modules') te combineren:

```
sub (priemL, 117)
```

levert het 117-de priemgetal. Net zo levert:

```
linsearch (groot, priemL)
where groot (p) = p > 117
```

de lijst [y] met y het eerste priemgetal groter dan 117. Let wel, deze programma's zijn vrijwel even efficiënt als 'geschikte aanpassingen van priemL', want de berekeningen voor priemL, linsearch en groot vinden, tengevolge van lazy evaluation, verweven plaats en niet verder dan nodig is om het gevraagde resultaat op te leveren. Lazy evaluation stelt de programmeur in staat om gescheiden taken ook in gescheiden modules te programmeren, terwijl de erdoor opgeroepen berekeningsstappen vermengd gebeuren! In imperatieve talen is dit mogelijk met bijvoorbeeld coroutines.

7 BACKTRACKING

Backtracking ofwel Terugkrabbelen is een probleemoplossingsmethode waarbij systematisch alle zinvolle paden naar een mogelijke oplossing doorlopen worden en steeds, wanneer een kandidaat-oplossing faalt, teruggekrabbeld wordt naar de laatst gepasseerde keuze waar nog een alternatief beschikbaar is. Het acht-koninginnen probleem leent zich voor deze methode. We zullen het hier behandelen.
Een plaatsing P van koninginnen op een schaakbord representeren we als een lijst van de posities der koninginnen, dat wil zeggen een lijst van paren [r, k] waarbij r het rijnummer en k het kolomnummer van een koningin is. De functie die bepaalt of een plaats [r'k] veilig is bij gegeven plaatsing P op kolommen 1 t/m k-1, luidt als volgt.

```
veilig (r, k, []) = true
veilig (r, k, [r', k']: P') =
      r ≠ r' and abs(r-r') ≠ abs(k-k') and
      veilig (r, k, P')
```

We definiëren nu een functie opl die de lijst van alle oplossingen (= plaatsingen) op een bord met 8 rijen en k kolommen oplevert.

```
opl (0) = [ [] ]
opl (k) = {[r, k]: P | P ← opl (k−1); r ← [1..8];
      veilig (r, k, P)}
```

In woorden: voor k=0 is er één oplossing, [], namelijk de plaatsing van nul koninginnen op een 8-bij-0 bord; voor k>0 worden bij iedere plaatsing P van koninginnen op de eerste k−1 kolommen alle acht posities r op de k-de kolom geprobeerd. Dat hier van backtracking gesproken kan worden komt door de manier waarop expressies geëvalueerd worden: lazy evaluation. Wanneer we alleen geïnteresseerd zijn in de eerste de beste oplossing, vragen we eenvoudigweg om de waarde van:

```
hd (opl (8))
```

Van alle oplossingen opl (7) worden er slechts zoveel berekend als nodig is om de eerste van opl (8) te vinden, alweer dank zij lazy evaluation.

8 COMMUNICERENDE PROCESSEN

Communicerende processen zijn soms een uitstekend middel om probleemoplossingen uit te programmeren, zie bijvoorbeeld [Hoare 1978]. Daarenboven zijn bijvoorbeeld ook Operating Systems en simulatie-pakketten vorm te geven als een stelsel communicerende processen. We gaan niet verder in op het nut van communicerende processen, maar laten slechts zien hoe zij in een functionele taal met lazy evaluation gemodelleerd kunnen worden.
Zij p een proces dat steeds bij invoergetal x het kwadraat van x oplevert. We kunnen p modelleren als een functie die op een lijst van invoergetallen werkt en de lijst van kwadraten oplevert:

```
p (x: X) = x*x: p (X)
```

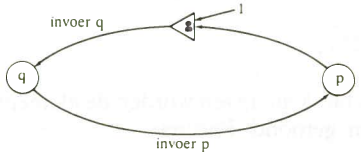
Zij q een proces dat steeds bij invoergetal Y een of ander getal, zeg f(y), als uitvoer oplevert. We kunnen q modelleren als:

```
q (y: Y) = f(y): q (Y)
```

Nu gaan we processen p en q koppelen: een startwaarde 1 op kop van de uitvoer van p nemen we als invoer voor q, en de uitvoer van q nemen we als invoer voor p. Dus invoerp en invoerq worden wederzijds recursief gedefinieerd:

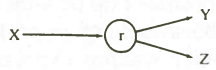
```
invoerq = 1: p (invoerp)
invoerp = q (invoerq)
```

Wanneer we nu de waarde van invoerp of van sub (invoerp, 1000) opvragen, dan wordt de berekening gestart: uit het eerste element van invoerq (nml. 1) berekent q het eerste element van invoerp (nml. f(1)), zodat p het tweede element van invoerq oplevert (nml. f(1) gekwadrateerd), enzovoorts. Een grafische voorstelling van dit stel gekoppelde processen is:



Ingewikkelder koppelingen van processen zijn ook te modelleren. Zij bijvoorbeeld r een proces dat zijn i-de

invoergetal onveranderd de ene dan wel de andere kant opstuurt, al naar gelang het rangnummer i een priemgetal is:



Dit kunnen we modelleren als volgt:

```
r (X) = rhulp (1, priemL, X)
where
rhulp (i, p:P, x:X) =
if i=p then linksOpKop (x, rhulp (i+1, P, X))
       else rechtsOpKop (x, rhulp (i+1, p:P, X))
linksOpKop (x, [Y,Z]) = [x:Y, Z]
rechtsOpKop (x, [Y,Z]) = [Y, x:Z]
```

In woorden luidt de definitie van rhulp (i, p:P, x:X): ingeval i priem is, i=p, wordt x 'alvast' opgeleverd op kop van het linkerlid van wat 'in de toekomst' door rhulp (i+1, P, X) geproduceerd wordt; ingeval i niet priem is, i<p, wordt x op kop van het rechterlid van rhulp (i+1, p:P, X) gezet. In dit laatste geval blijft p behouden om de primaliteit van de volgende rangnummers i+1, i+2, ... te testen. De functie rhulp wordt zó gebruikt dat bij een aanroep rhulp (i, p:P, x:X) de i het rangnummer van x is en p het kleinste priemgetal $\geq$ i. De parameters i en p:P van rhulp vormen als het ware een lokale toestand (de lokale toestandsvariabelen) van proces r.

In plaats van priemL hadden we natuurlijk ook een lijst van toevalsgetallen kunnen nemen, zo dat de invoerlijst van r 'nondeterministisch' over beide uitvoerlijsten van r verdeeld wordt. Buffering van gegevensstromen tussen processen (volgens een of andere bufferingsstrategie zoals LIFO, FIFO etc.) moet gemodelleerd worden door daarvoor aparte processen te nemen.

9 INVOER EN UITVOER

Zoals gebruikelijk bij programmeertalen verschillen ook functionele talen van elkaar in de manier waarop in- en uitvoer geschiedt. Bovendien ligt er het gevaar op de loer dat in- en uitvoervoorzieningen afbreuk doen aan het beginsel van functionele programmeertalen, namelijk dat de evaluatie van expressies geen neveneffecten heeft. We gaan hier niet op details in maar tonen slechts het beginsel volgens welke de in- en uitvoer in SASL, KRC en TWENTEL geregeld zijn.
Om de waarde van een expressie naar het beeldscherm (of ander standaard uitvoermedium) te sturen moet de expressie, gevolgd door een vraagteken, worden ingetikt. Dus na:

```
5+3?
```

verschijnt er 8 op het beeldscherm, en na:

```
sub (priemL, 117)?
```

het 117-de priemgetal. Van lijsten worden de elementen direct achter elkaar getoond. Dus na:

```
priemL?
```

verschijnt er 235711131719232931... . Geneste lijsten worden 'platgeslagen' getoond, zodat

```
[[1, 2, 3], [1, 3, 2], [2, 1, 3], ...]?
```

de afdruk 123132213... geeft. Om een nette uitvoer te krijgen moet men zelf met behulp van characters een lijst opbouwen die wel de gewenste afdruk geeft. Zij bijvoorbeeld sp de notatie voor het spatie-character, dan verschijnt er na:

```
{[p, sp] | p ← priemL}?
```

de afdruk 2 3 5 7 11 13 17 19 .... Regelovergangen zijn te bewerkstelligen door het newline-character op te nemen. Willen we van een lijst ook de haakjes en komma's tonen, dan gaan we als volgt te werk. Zij lb, rb, cm de notatie voor het linkerhaakje (left bracket), rechter haakje (right bracket) en komma (comma).Definieer:

```
show ([]) = [lb, rb]
show (x:X) = [lb, x] ++ {[cm, sp, x] | x ←X} ++ [rb]
```

dan verschijnt na:

```
show (priemL)?
```

de afdruk [2, 3, 5, 7, ...

Dat ook invoer vanuit het achtergrondgeheugen op een dergelijke manier gerealiseerd kan worden, zal geen verbazing wekken. We gaan hier niet verder op in.
Interessanter is het dat ook interactieve programma's mogelijk zijn. Beschouw daartoe nogmaals het proces p dat steeds zijn invoergetal gekwadrateerd oplevert:

```
p (x:X) = x*x: p (X)
```

Wanneer we nu het commando

```
p interactive
```

intikken, dan worden de getallen die daarna vanaf de terminal worden ingetikt als invoerlijst voor p genomen en wordt de uitvoer(lijst) van p tevens op het beeldscherm getoond. (Eigenlijk worden de ingetikte characters als invoerlijst voor p genomen, maar het is niet moeilijk een hulpproces te definiëren dat uit een characterlijst een getallijst vormt.) Dus er verschijnt steeds het kwadraat van het ingetikte getal, en sterker: de gebruiker kan aan de hand van het zojuist getoonde kwadraat het nieuw in te tikken getal verzinnen. Vanzelfsprekend kunnen we p ook vragende en verklarende tekst laten produceren en een terminatie-mogelijkheid inbouwen.

10 POINTERSTRUCTUREN

Pointers zijn vaak een hulpmiddel om gestructureerde data te representeren, met name inductieve of recursieve datastructuren. In functionele programmatuur kan dat met lijsten en recursie. Ter illustratie behandelen we hier binaire bomen en grafen.
Binaire bomen representeren we als volgt: [] voor de lege boom en een drietal [L, K, R] voor een niet-lege boom waarbij L en R de representaties voor de linker en rech-

ter subboom zijn en K de representatie voor de informatie van de knoop. De inorder lijst van de knopen wordt dan door de volgende functie opgeleverd.

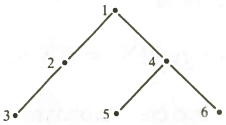
```
inL ([]) = []
inL ([L, K, R]) = inL(L) ++ [K] ++ inL(R)
```

Om een boom op te bouwen gaan we als volgt te werk. Stel dat de informatie van een knoop louter een positief getal is, en dat de pre-order lijst van de knopen, met 0 ter representatie van de lege boom, is gegeven. Dus uit de invoer:

```
[1, 2, 3, 0, 0, 0, 4, 5, 0, 0, 6, 0, 0]
```

moet de volgende boom worden opgebouwd:

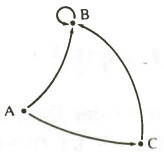
```
[[[[], 3, []], 2, []], 1, [[[], 5, []], 4, [[], 6, []]]]
=
```



De functie boom die we nu gaan definiëren, levert bij gegeven lijst X een tweetal [B, Y] op waarbij B de boom (representatie) is die door een initieel stuk van X bepaald wordt en Y de rest van X is.

```
boom (0:X) = [[], X]
boom (n:X) = [[L, n, R], Z]
             where [L, Y] = boom (X)
                   [R, Z] = boom (Y)
```

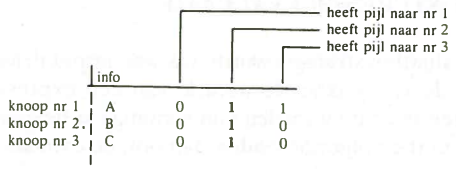
Het wezenlijke is dat boom (X) een tweetal oplevert; de gevraagde boom is daarvan het linkerlid.
Niet-hiërarchische structuren, zoals grafen, kunnen soms ook rechtstreeks gerepresenteerd worden. Lussen in een graaf worden dan recursie in de programmatekst. Bijvoorbeeld, de graaf



kan gerepresenteerd worden door de lijst:

```
[a, b, c]
where a = [A, [b, c]]
      b = [B, [b]]
      c = [C, [b]]
```

De buren van een knoop zijn nu sequentieel toegankelijk. We kunnen zo'n graafrepresentatie ook uit de buurmatrix (lijst van lijsten) opbouwen. Beschouw de volgende tabel,



en stel dat die gerepresenteerd wordt door de lijst:

```
T = [[A, 0, 1, 1], [B, 0, 1, 0]], [C, 0, 1, 0]]
```

De volgende functie bouwt uit zo'n tabel T de gewenste graafrepresentatie.

```
graaf (T) = G
where G = {knoop (K) | K ← T}
      knoop (info: buren) =
          [info, {sub (G, i) | i ← [1..lth(T)];
          sub (buren, i) = l}]
```

Onder lazy evaluation resulteert de aanroep sub (G, i) niet in een kopie van de i-de knoop van G maar in een pointer daarnaar toe. (Overigens lijkt mij met deze representatiekeuze het veranderen van grafen niet efficiënt doenlijk.)

11 GEGEVENSBANKEN

We laten nu het beginsel zien waarmee gegevensbanken functioneel geprogrammeerd kunnen worden. De gebruiker tikt commando's in (toevoegingen, verwijderingen en vragen) en krijgt daarop de bedoelde informatie te zien op het beeldscherm. We kiezen uiteraard een heel eenvoudige gegevensbank en offeren efficiëntie op aan doorzichtigheid.
De gegevensbank bevat paren [m, n] van getallen, bijvoorbeeld te interpreteren als 'm kent n' waarbij m en n persoonscodes zijn. Commando's hebben de volgende vorm en interpretatie.

[1, m, n]	voeg [m, n] toe aan het bestand
[2, m, n]	verwijder [m, n] uit het bestand
[3, m]	geef alle n met [m, n] in het bestand
[4, n]	geef alle m met [m, n] in het bestand

Een gegevensbestand zullen we representeren als een lijst zonder duplicaten en met de meest recente toevoegingen vooraan; de commando's worden dan geïmplementeerd door de volgende functies uit, in, alleN en alleM, terwijl initB een leeg bestand voorstelt.

```
initB = []
uit (m, n, B) = {[m', n'] | [m', n'] ← B; [m', n'] ≠ [m, n]}
in (m, n, B) = [m, n]: uit (m, n, B)
alleN (m, B) = {n | [m', n] ← B; m'=m}
alleM (n, B) = {m | [m, n'] ← B; n'=n}
```

Het is niet moeilijk om initB en de vier functies zo te herschrijven dat een bestand bijvoorbeeld als een geordende zoekboom georganiseerd is.

We zullen nu een functie system definiëren die op een lijst van commando's werkt en als resultaat een lijst van de bedoelde informaties oplevert. (Bij een toevoeging of verwijdering wordt het commando zelf ge-echood ten teken dat de actie voltooid is.) Er rest ons daarna slechts

```
system interactive
```

in te tikken: de invoer vanaf de terminal wordt dan de invoerlijst van system en de uitvoerlijst van system wordt naar het beeldscherm gestuurd; zie het hoofdstukje Invoer en uitvoer. (Eigenlijk moet system nog gecombineerd worden met een functie die de characterlijst van de

terminal omzet in een lijst van commando's, en net zoiets geldt voor de uitvoer naar het beeldscherm.)
Hier is de definitie van system (B staat voor Bestand):

```
system (Commands) = f (initB, Commands)
where
f (B, [1, m, n]: C) = [1, m, n] : f (in (m, n, B), C)
f (B, [2, m, n]: C) = [2, m, n] : f (uit (m, n, B),C)
f (B, [3, m]: C) = alleN (m, B) : f (B, C)
f (B, [4, n]: C) = alleM (n, b) : f (B, C)
```

De B-parameter van f vormt als het ware de lokale toestand(s variabele) voor system; (vergelijk rhulp in het hoofdstukje Communicerende processen).

Met behulp van – de niet behandelde – invoer- en uitvoervoorzieningen is het mogelijk het initiële bestand van het achtergrondgeheugen in te voeren en het finale bestand daarin op te bergen (na een terminatiemogelijkheid in system te hebben ingebouwd).

12 HOGERE ORDE FUNCTIES

We noemen een functie van hogere orde als sommige van zijn argumenten of resultaten weer functies zijn. Het komt vrij vaak voor dat functies als argument optreden (bijvoorbeeld de cond-parameter van linsearch), ook in conventionele talen, maar functies als resultaat zijn zeldzamer. In conventionele talen worden zij meestal verboden omdat het beheer over de opslagruimte anders niet meer met een stapelorganisatie te implementeren is. Wij geven hier een voorbeeld waarbij hogere orde functies, met name functies als resultaat, gebruikt worden om een algoritme (zeer beknopt en) op hoog abstractieniveau te noteren: het herkennen van contextvrije talen.

Beschouw de volgende grammatica.

```
<zin>      ::= 0 <eenreeks> 0 <nulrij> 1
<eenreeks> ::= 1 | 1 <eenreeks>
<nulrij>   ::= | 0 <nulrij>
```

In woorden: een <zin> is een 0 gevolgd door een <eenreeks> gevolgd door een 0 gevolgd door een <nulrij> gevolgd door een 1; een <eenreeks> is een 1 of een 1 gevolgd door een <eenreeks>; een <nulrij> is een lege rij of een 0 gevolgd door een <nulrij>. Dus een <eenreeks> is een cijferrij 11...1 met minstens een 1; een <nulrij> is een cijferrij 0...0 met eventueel nul 0'en; een <zin> is een cijferrij van de vorm 011...100...01. De kortste <zin> is 0101 en 01011 is geen <zin>.

We willen nu een functie zin definiëren die test of een cijferrij, als getallijst gerepresenteerd, een <zin> is. Daartoe definiëren we eerst functies een, nul en nix die corresponderen met 1, 0 en de lege rij.

```
een (X) = X=[1]
nul (X) = X=[0]
nix (X) = X=[]
```

Verderop definiëren we functies alt en seq die corresponderen met alternatie (de rechte streep in de grammatica) en sequentie (het achterelkaar-zetten in de grammatica): alt (nul, een) zal een functie zijn die test of een lijst gelijk is aan [0] of [1], en seq ([nul, alt (nul, een), nul]) zal een functie zijn die test of een lijst gelijk is aan [0, 0, 0] of [0, 1, 0]. Daarmee zijn we in staat om direct aan de hand van de grammatica de gevraagde functie zin te definiëren te zamen met hulpfuncties eenreeks en nulrij:

```
zin = seq ([nul, eenreeks, nul, nulrij, een])
eenreeks = alt (een, seq([een, eenreeks]))
nulrij = alt (nix, seq([nul, nulrij]))
```

Dus zin is een functie die test of een lijst een <zin> is: zin ([0, 1, 1, 1, 0, 0, 0, 1]) = true en zin ([0, 0, 1, 0]) = false. Doordat functies net zo als andere data als argument, resultaat en lijstelement mogen optreden, zijn we in staat de vorm van de definities exact te laten aansluiten bij de vorm van de grammatica-regels. De correctheid is dus evident en het abstractieniveau is precies goed.

Er rest ons nog om alt en seq te definiëren. De definitie van alt ligt voor de hand:

```
alt (f, g) = h where h(X) = f(X) or g(X)
```

(Het rechterlid van and en or wordt niet uitgerekend als het linkerlid de uitkomst al bepaalt.) Voor seq definiëren we eerst een hulpfunctie seq2 met: seq2 (f, g) = een functie h zo dat h(X) = true precies wanneer er een [Y,Z] is met Y ++ Z = X én f(Y) and g(Z) = true.

```
seq2 (f, g) = h
where
h(X) = {[Y,Z] | [Y,Z] ← splits (X); f(Y) and g(Z)} ≠ []
splits ([]) = [ [[], []] ]
splits (x:X) = [[], x:X]: {[x:Y,Z] | [Y,Z] ← splits (X) }
```

(Merk op dat de berekening van h(X) feitelijk de Backtrack-methode volgt, zie het hoofdstukje Backtracking.)
De functie seq is louter een veralgemening van seq2: seq ([f, g, ...]) = seq2 (f, seq2 (g, ...)), seq ([f]) = f en zelfs seq ([]) = nix. Dus:

```
seq ([]) = nix
seq (f:F) = seq2 (f, seq (F))
```

Het behoeft geen betoog dat deze methode voor alle grammatica's toepasbaar is. Er zit slechts één addertje onder het gras. De 'evidente correctheid' betreft slechts partiële correctheid. Dat wil zeggen: een fout antwoord wordt nooit opgeleverd maar soms zal een test niet termineren. Met name als we de grammatica en dus de functiedefinitie als volgt veranderen:

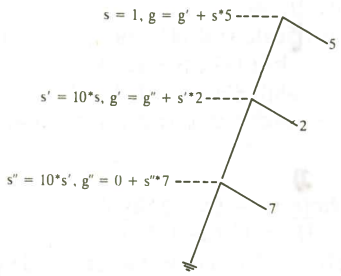
```
<eenreeks> ::= <eenreeks> 1 | 1
eenreeks = alt (seq ([eenreeks, een)], een)
```

Elke aanroep van eenreeks resulteert nu in een aanroep van eenreeks, zo dat nooit terminatie optreedt.

13 ZELF-STURENDE EVALUATIE

De lazy evaluation strategie wordt ook wel output driven genoemd: de vraag naar de waarde van een expressie roept vragen naar de waarden van sommige subexpressies op en in die volgorde vinden dan ook de evaluaties

van expressies plaats. Bijgevolg hoeft de programmeur, met name bij wederzijds recursieve definities, niet expliciet de berekeningsvolgorde aan te geven omdat die vanzelf tot stand komt. We geven hiervan twee voorbeelden; ze zijn gerelateerd aan de zgn. attributen-evaluatie van een ge-attributeerde ontledingsboom.
Beschouw onderstaande ontledingsboom van de cijferrij 7 2 5 ; per knoop is er een g-attribuut gedefinieerd dat de getalwaarde van de onderhangende subboom aangeeft en een (hulp)-s-attribuut voor de schaalfactor.



Een vertaler of programmeur zou conform deze ontleding de cijferrij 7 2 5 kunnen omzetten in een stel wederzijds recursieve definities:

$$
\begin{aligned}
s &= 1\\
g &= g' + s*5\\
s' &= 10*s\\
g' &= g'' + s'*2\\
s'' &= 10*s'\\
g'' &= 0 + s''*7
\end{aligned}
$$

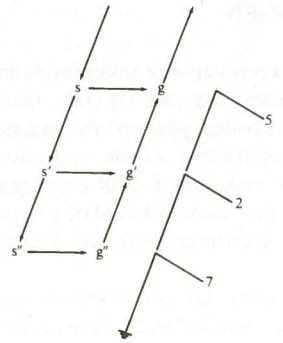
Wanneer we nu om de waarde van g vragen, de getalwaarde van de cijferrij, dan zal de evaluatie inderdaad 725 opleveren. Schematisch gaat dat als volgt.

$$
\begin{aligned}
& g\\
?g:\ &= g' + s*5\\
?g':\ &= (g'' + s'*2) + s*5\\
?g'':\ &= ((0 + s''*7) + s'*2) + s*5\\
?s'':\ &= ((0 + (10*s')*7) + s'*2) + s*5\\
?s':\ &= ((0 + (10*(10*s))*7) + (10*s)*2) + s*5\\
?s:\ &= ((0 + (10*(10*1))*7) + (10*1)*2) + 1*5\\
&= \ldots = 725
\end{aligned}
$$

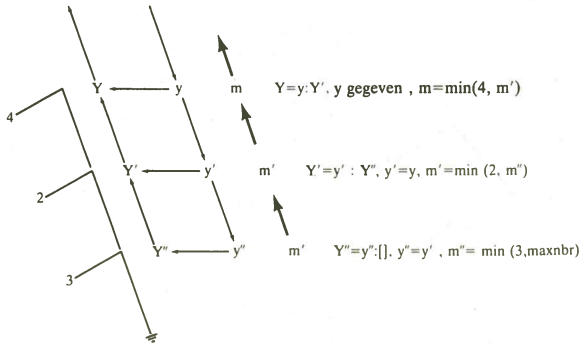
Overigens, de g-waarde van de bovenste knoop wordt ook opgeleverd door:

$$
\begin{aligned}
&\text{eval } ([5, 2, 7], 1)\\
&\underline{\text{where}}\ \text{eval } ([], s) = 0\\
&\qquad \text{eval } (x{:}X, s) = \text{eval } (X, s*10) + s*x
\end{aligned}
$$

De evaluatie hiervan volgt exact hetzelfde patroon als de berekening volgens het stel definities van s, g, s', g', s'' en g''. Merk op dat het s-attribuut tot parameter van eval is gemaakt: een s-attribuut hangt alleen af van de attributen van de vader; het g-attribuut is het resultaat van eval geworden: een g-attribuut hangt juist van de attributen van de zonen af. (Waren er n van dergelijke attributen geweest, dan zou eval een n-tal, een lijst dus, moeten opleveren.) De attribuut-afhankelijkheden zien er schematisch als volgt uit.



We gaan nu over tot het tweede voorbeeld. Beschouw de volgende ge-attributeerde boom.



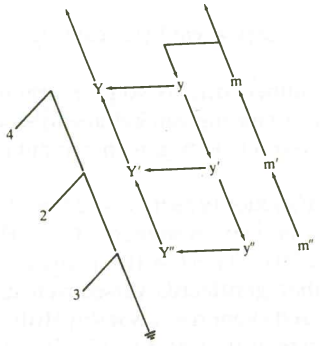
De Y- en m-waarde van dit stel wederzijdse recursieve definities worden opgeleverd door eval ([4, 2, 3], y) wanneer gedefinieerd is:

$$
\begin{aligned}
&\text{eval } ([], y) = [[], \text{maxnbr}]\\
&\text{eval } (x{:}X, y) = [y{:}Y', \min(x, m')]\\
&\qquad \underline{\text{where}}\ [Y', m'] = \text{eval } (X, y)
\end{aligned}
$$

Voor alle y levert de aanroep eval ([4, 2, 3], y) het tweetal [[y, y, y], 2] op; bedenk dat 2 = min (4, min (2, min (3, maxnbr))). Definiëren we nu:

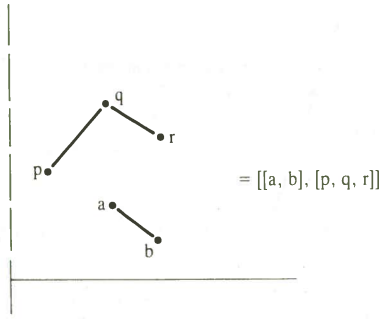
$$[Y, y] = \text{eval } ([4, 2, 3], y)$$

(dus met y recursief gedefinieerd!), dan geldt dus [Y, y] = [[y, y, y], 2], ofwel: [Y, y] = [[2, 2, 2], 2]. De grap is dat de lijst [4, 2, 3] slechts éénmaal door eval wordt doorlopen en toch ieder element vervangen wordt door het minimum van het geheel. (Als het niet waar was, zou ik het niet geloven!) Het bijbehorende plaatje van de attribuut-afhankelijkheden ziet er als volgt uit:



14 TEKENINGEN

Het programmeren van een tekening komt in feite neer op het produceren van een rij (file, lijst) van getallen (coördinaten en codes 'pen op' en 'pen neer') waarmee de tekentafelapparatuur wordt aangestuurd. Tekeningen kunnen dus ook functioneel geprogrammeerd worden. Wij geven hier als voorbeeld de programmering van de zgn. Hilbertkrommen, ook wel Peanokrommen genoemd.
We gaan ervan uit dat een tekening gerepresenteerd wordt als een lijst van gebroken lijnstukken, waarbij een gebroken lijnstuk op zich weer een lijst van (hoek-, begin- of eind-)punten is waarvan de coördinaten ten opzichte van een vaste oorsprong zijn genomen. Bijvoorbeeld



= [[a, b], [p, q, r]]

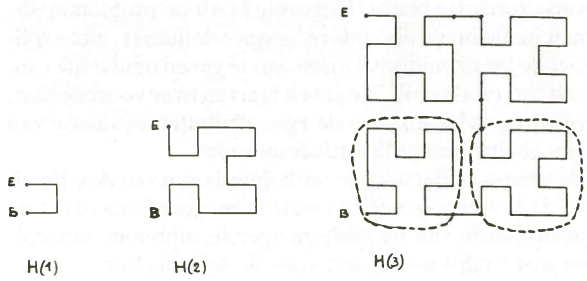
Een tekening heeft dus een begin- en eindpunt; met bovenstaande representatie is a het beginpunt en r het eindpunt.
Het is niet moeilijk de volgende functies te definiëren; kortheidshalve zullen we dat hier niet doen.
rot (t, h) = t tegen de klok in gedraaid over een hoek h
scale (t, fx, fy) = de t′ verkregen uit t door alle x-coördinaten met fx en de y-coördinaten met fy te vermenigvuldigen
shift (t, vx, vy) = de t′ ontstaan uit t door alle x/y-coördinaten met vx/vy te vermeerderen
joint (t1, t2) = t1 samen met t2
cat2 (t1, t2) = de concatenatie van t1 met t2, d.w.z. t1 met t2 middels hun verbindingslijnstuk ge-joined.
pnt (x, y) = de tekening bestaande uit een punt (x, y)

Merk op dat scale (t, −1, +1) de gespiegelde van t ten opzichte van de y-as oplevert. De uitbreiding van cat2 tot cat is standaard.

```
cat ([t]) = t
cat (t: T) = cat2 (t, cat(T))
```

Dus cat ([t1, ..., tn]) = cat2 (t1, cat2 (t2, ..., tn)).

De Hilbertkrommen zijn als volgt opgebouwd; we nemen steeds de oorsprong van het assenstelsel in het beginpunt, en markeren de begin- en eindpunten met B en E.
De kromme H(n) met breedte b = 2**n−1 is louter de concatenatie van vier krommen H′ = H(n−1) met breedte b′ = 2**(n−1) −1 = (b−1) div 2, die ieder op geschikte manier geroteerd, verschoven en geschaald zijn. Dit geldt zelfs voor n=1, waarbij H(0) met breedte 0 uit louter één punt bestaat. Omwille van de efficiëntie maken we de breedte b tot een extra parameter van H.




```
H (0, 0) = pnt (0, 0)
H (n, b) = cat ([scale (rot (H′, pi/4), −1, 1)
                , shift (H′, b′+1, 0)
                , shift (H′, b′+1, b′+1)
                , shift (scale (rot (H′, −pi/4), −1, 1),
                                                  b′, b)
                ])
           where b′ = (b−1) div 2
                 H′ = H (n−1, b′)
```

De aanroep H(n, 2**n−1) produceert nu de gevraagde Hilbertkromme. Bovenstaande aanpak is geïnspireerd op [Cole 1983].

15 TYPERING

Het is alom bekend dat zgn. sterke typering, met compile-time type-controle, veel programmeer- en denkfouten vroegtijdig ontmaskert (en daarnaast ook nog andere voordelen heeft). Als nadelen worden wel genoemd de verlenging van de programmateksten en de onmogelijkheid van algemeen toepasbare functies zoals hd, tl, sub, linsearch en de straks te geven reduce. Wij schetsen hieronder een sterke typering die deze nadelen niet heeft; in het bijzonder is het niet nodig om in de programma's zelf type-informatie op te nemen.

Beschouw de volgende definities.

```
mult (x, y) = x*y
tel (x, n) = n+1
reduce (f, a, []) = a
reduce (f, a, x:X) = f (x, reduce (f, a, X))
```

Dus reduce (f, a, [x, y, ..., z]) = f (x, f(y, ... f(z, a))) zodat
reduce (mult, 1, [x, y, ..., z]) = (x* (y* ... (z*1)))
reduce (tel, 0, [x, y, ..., z]) = (1+(1+ ... (1 + 0)))

We hadden de functies prod en lth uit hoofdstukje 3 dus ook kunnen definiëren door:

```
prod (X) = reduce (mult, 1, X)
lth (X) = reduce (tel, 0, X)
```

Gebaseerd op de kennis van de operaties zoals +, *, : en ++ leidt compile-time type-controle tot de conclusie:

mult: (nbr, nbr → nbr)

dat wil zeggen, mult heeft het type 'functie met twee number (= int of real) parameters en een number resultaat'. Voor tel wordt gevonden:

tel: (α, nbr → nbr)

Dit type is eigenlijk een type-schema: α is een nog per gebruik vrij te kiezen type(schema). Dus tel (4, 3), tel (true, 3) en tel ([1,2], 3) zijn alle goedgetypeerd; de keuzen voor α zijn achtereenvolgens nbr, bool en nbr-list. Het typeschema dat voor reduce gevonden wordt luidt:

reduce: ((α, β → β), β, αlist → β)

We zien onder andere dat het resultaattype en het type van de tweede parameter gelijk zijn maar verder nog vrij te kiezen (per gebruik van reduce). Typecontrole van reduce (mult, 1, X) levert dat in dit geval α = β = nbr en dus moet X van type nbr-list zijn; bijgevolg heeft prod als type: (nbr-list → nbr) en dit type zou ook gevonden worden volgens de definitie in hoofdstukje 3. Type-controle van reduce (tel, 0, X) levert dat in dit geval β = nbr (en α is nog steeds vrij te kiezen) en dus moet X van type αlist zijn (voor willekeurige α); bijgevolg heeft lth als type: (αlist → nbr) en dit type zou ook gevonden worden volgens de definitie in hoofdstukje 3.

De type-controle kan nauwelijks versneld worden als de programmeur de typen wel zou hebben opgegeven. Ter documentatie kan bij het intikken van een definitie het type dat bij de controle wordt gevonden ook op het beeldscherm en de papieruitvoer afgedrukt worden. Vrijwel alle programma's die wij gepresenteerd hebben zouden, zonder extra toevoegingen, bij type-controle in orde bevonden worden! Tenslotte vermelden we dat de programmeur naar eigen believen zelf nieuwe typen kan definiëren; bijvoorbeeld type αtree met bijbehorende functies dat daarna net zo behandeld wordt als αlist met bijhorende operaties. Bovenstaande typering wordt formeel beschreven in [Milner 1978] en [Fokkinga 1983] en wordt bijvoorbeeld gebruikt in de taal ML (maar niet in SASL en KRC).

16 OEFENINGEN

Om de ijverige lezer behulpzaam te zijn bij eventuele pogingen om functioneel te leren programmeren, suggereren we per hoofdstukje een paar oefeningen die heel nauw aansluiten bij de gegeven voorbeelden.

2a. Geef een recursieve definitie van de faculteitsfunctie.

2b. Definieer mult recursief zo dat mult (x, y) = y + ... + y + y (x maal); gebruik de vermenigvuldigingsoperator * niet.

2c. Definieer hanoi recursief zo dat hanoi (n) = het aantal zetten van de oplossing voor de Torens van Hanoi bij initiële hoogte n; gebruik geen vermenigvuldiging en machtsverheffing.

3a. Definieer reverse zo dat reverse ([x1, ..., xn]) = [xn, ..., x1].

3b. Definieer append zo dat ([x1, ..., xm], [y1, ..., yn]) = [x1, ..., xm, y1, ..., yn] zonder ++ te gebruiken.

3c. Definieer fromTo (m, n) = [m, n+1, ..., n] zonder [x .. y] te gebruiken.

3d. Definieer de lijst facL van faculteiten analoog aan fibL. (Wenk: de staart van facL is de elementsgewijze vermenigvuldiging van facL zelf en from (1).)

3e. Definieer een lijst randomL van toevalsgetallen: randomL = [r1, r2, r3, ...] met voor i > 1: r(i)=(c+r(i−1)*m) mod d, waarbij r1, c, m en d vrij willekeurig gekozen kunnen worden.

4a. Definieer concat ([X, Y, ..., Z]) = X ++ Y ++ ... ++ Z met behulp van de {}-notatie en zonder ++ te gebruiken.

4b. Herschrijf de drie gegeven voorbeelden zonder van de {}-notatie gebruik te maken.

4c. Definieer de lijst [[1,1], [2,1], [1,2], [3,1], [2,2], [1,3], [4,1], ..., [1,4], [5,1], ..., [1,5], enz.].

5a. Definieer de lijst van alle deelverzamelingen (als lijsten gerepresenteerd) van [1..n].

5b. Definieer insertion sort. Wenk: de gesorteerde versie van x:X bestaat uit de gesorteerde versie van X waarin x op de juiste plaats is tussengevoegd; geef een afzonderlijke functie insert (x, X) die x in een geordende X tussenvoegt.

6a. Definieer het kleinste Fibonaccigetal dat een 117-voud is.

6b. Definieer de lijst van kwadraten van priemgetallen.

6c. Definieer de lijst van priemfactoren van n, in volgorde van afnemende grootte. Wenk: zie ook oefening 3a.

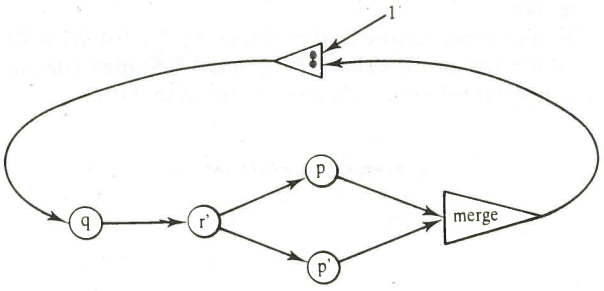
7a. Definieer een lijst van alle drietallen [x, y, z] met $1 \leq x \leq y \leq z$ en x + y + z = n voor gegeven n.

7b. Definieer een lijst van alle [x, y, ..., z] met $1 \leq x \leq y \leq ... \leq z$ en x + y + ... + z = n, voor gegeven n.

7c. Gegeven zijn k 6-tallen X1, ..., Xk van getallen uit 1..k, en een lijst P van functies die 6-tallen permuteren. Gevraagd wordt om te bepalen of er p1, ..., pk uit P zijn (niet noodzakelijk alle verschillend) zo dat in de 'matrix' [p1(Xl), ..., pk(Xk)] op ieder van de eerste vier kolommen ieder getal uit 1..k (precies eenmaal) voorkomt. (Toelichting. Dit is een veralgemening van het Instant Insanity Problem: de 6-tallen zijn kubussen, de getallen 1..k zijn kleurcoderingen, k=4, de eerste vier kolommen zijn de vier aangezichten van een toren van kubussen en de permutaties in P zijn de manieren waarop een kubus geplaatst kan worden. Het probleem is om de kubussen zó tot een toren te stapelen dat ieder aangezicht alle k kleuren toont.)

8a. Definieer een proces r′ dat steeds zijn invoergetal de ene dan wel de andere kant opstuurt al naar gelang dat invoergetal zelf een priemgetal is.

8b. Definieer een proces p′ dat steeds zijn invoergetal verdubbelt. Veronderstel voorts dat merge een proces is dat twee invoerstromen tot één mengt en wel zo dat ieder invoergetal dat aankomt ook gegarandeerd wordt uitgevoerd. Geef nu het programma dat het volgende schema modelleert.



9a. Geef een expressie waarvan de afdruk een nette tabel van alle priemgetallen kleiner dan 1000 is; bijv. 5 kolommen ieder ter breedte van 6 posities. Gebruik (en definieer) de volgende functies,

width (n) = aantal cijfers in de decimale notatie van n
space (n) = [sp, ..., sp] (n spaties)

en gebruik nl als het newline character (dat bij afdruk een overgang op een nieuwe regel tot gevolg heeft).

9b. Schrijf een interactief programma waarbij de gebruiker een getal in 1..1000 moet raden dat door de computer verzonnen wordt. Ga als volgt te werk. Definieer een proces p dat bij invoerlijst [m, n1, n2, ...] allereerst uit m en de randomlijst randomL (zie oefening 3e) een getal g in 1..1000 berekent en dan een lijst [c1, c2, ..., ck] oplevert met $c_i=0$ als $n_i < g$ en $c_i=1$ als $n_i > g$, en $c_i=2$ als $n_i=g$ (en k= de kleinste i met $n_i=g$).

10a. Bouw de binaire boom op uit een gegeven post-order opsomming van de knoopgetallen (met 0 ter representatie van de lege boom).

10b. Definieer een functie f op binaire bomen zo dat f(b) = de b′ verkregen uit b door de knoopgetallen, in preorder volgorde, te verhogen met 1, 2, 3, ... Wenk: definieer een algemenere functie g zo dat g (b, m) = [b′, n] waarbij b′ verkregen wordt uit b door de knopen in preorder volgorde te verhogen met m, m+1, m+2, ... en n−1 de laatst gedane verhoging is (of beter: n−m is het aantal knopen van de boom).

12a. Definieer alt′ zo dat alt′([f, g, h, ...]) = alt (f, alt (g, alt (h, ...))).

12b. Beschouw naast alternatie en sequentie ook de grammaticale operatoren CHAIN en OPTION; xxx-CHAIN is een afkorting voor: xxx | xxx xxx-CHAIN en xxx-OPTION is een afkorting voor: (lege string) | xxx. Hiermee kunnen we <eenreeks> en <nulrij> dus definiëren door

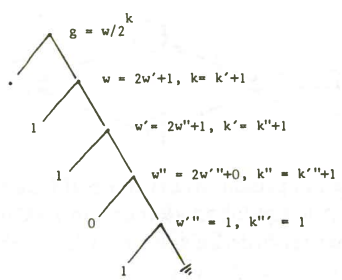
<eenreeks> ::= 1-CHAIN
<nulrij> ::= 0-CHAIN-OPTION

Definieer functies chain en option die corresponderen met CHAIN en OPTION, zo dat

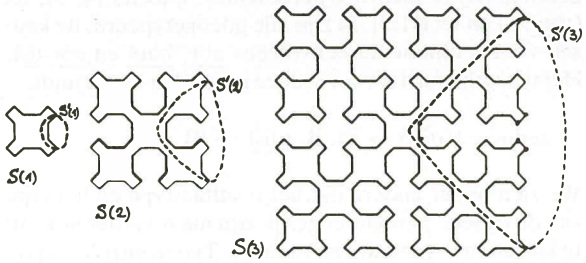
eenreeks = chain (een)
nulrij = option (chain (nul))

13a. Definieer een functie die elk knoopgetal van een binaire boom vervangt door het minimum van alle knoopgetallen, maar de boom slechts éénmaal doorloopt.

13b. Beschouw de volgende geattributeerde ontledingsboom van de rij . 1 1 0 1 die een binaire fractie voorstelt. Het g-attribuut geeft de getalwaarde van de binaire fractie aan.
Definieer een functie eval zo dat eval ([2,1,1,0,1]) = de waarde van g volgens bovenstaand stel definities, (de 2 in de lijst representeert de punt in de tekst .1101).



14a. De Sierpinskikrommen zien er als volgt uit.



Merk op dat S(n) is opgebouwd uit 4 kopieën van S′(n) en dat S′(n) is opgebouwd uit 4 kopieën van S′(n−1). Geef een definitie voor S′(n) (en in termen daarvan een voor S(n)).

15a. Definieer sort′ als:

```
sort′ (leq, []) = []
sort′(leq, x:X) = sort′ (leq, {y | y ←X; leq (y, x)} ++
                         [x] ++
                         sort′ (leq, {y | y ←X; not leq (y, x)}
```

Geef nu het type-schema voor sort′. Geef een aanroep om [4,1,3,2] te sorteren en verifieer de typecorrectheid.

15b. Definieer een functie leq zo dat leq (X,Y) = "X komt alfabetisch (lexicografisch) voor Y"; d.w.z. [x1, ..., xm] komt alf. voor [y1, ..., yn] als er een $k \leq \min(m,n)$ is met x1=y1, ..., xk=yk en (k=m≤n of k<m en k<n en x(k+1) < y(k+1)). Verifieer nu de type-correctheid van

sort′ (leq, [[1,7,1], [], [3,1], [1,7], [1,1,7,7]])

en geef het resultaat, (met de sort′ van 14a.).

17 LITERATUUR

Een ander aanbevolen en inleidend verhaal over functioneel programmeren is [Turner 1982]; daarin wordt KRC als notatie gebruikt. De essentie van lazy evaluation wordt in meer detail, maar nog wel informeel, beschreven in [Fokkinga 1984, Hoofdstuk 3]; een diepgaander behandeling wordt gegeven in [Turner 1979].

Leerboeken over functioneel programmeren zijn [Glaser et al 1984], [Burge 1975] en [Henderson 1980]; de laatste twee gaan niet uit van lazy evaluation maar geven wel taalconstructies waarmee lazy evaluation geprogrammeerd kan worden. [Darlington 1982] bevat een aantal interessante verhalen, en een veel verwezen artikel is [Backus 1978].

Andere en uitgebreidere toepassingen van hogere orde functies worden getoond in [Burge 1975], [Henderson 1980] en [van der Hoeven 1984]; van communicerende processen in [Henderson 1982] (een operating system) en in [Joosten 1985] (discrete simulatie); van geometrische beschrijvingen in [Henderson 198?]; terwijl [Turner 1981] en [Augusteijn 1983] ook een aardige toepassing geeft.

Functionele programmeertalen met lazy evaluation zijn SASL, KRC en TWENTEL (en nog een paar andere); zij worden gedefinieerd in [Turner 1976], [Turner 1982] en [van der Hoeven 1984].

- Augusteijn, A.; *An applicative algorithm for generating a list of paraffines*. Memorandum INF-83-7, T.H. Twente, 1983.
- Backus, J.; Can programming be liberated from the Von Neumann style? A Functional style and its algebra of programs. *Communications ACM 21* (1978) 8, pp 613-642.
- Burge, W. H.; *Recursive Programming Techniques*. Addison Wesley, 1975, 277 pag.

- Cole, A. J.; A note on space filling curves. *Software-Practice and Experience,* **13** (1983) pp 1181-1189.
- Darlington, J., Henderson, P., Turner, D. (eds); *Functional programming and its applications.* Cambridge University Press, Cambridge, U.K., 1982.
- Fokkinga, M. M.; *Over het nut en de mogelijkheden van typering – een overzicht.* Collegesyllabus T.H. Twente, 1983, (herziene druk 1984).
- Fokkinga, M. M.; *Structuur van Programmeertalen.* Collegesyllabus T.H. Twente, 1984, (herziene druk 1985).
- Glaser, H., Hankin, C., Till, D.; *Principles of Functional Programming.* Prentice-Hall, Englewood-Cliffs N.J., 1984.
- Henderson, P.; *Functional programming – application and implementation.* Prentice-Hall, Englewood Cliffs NJ., 1980.
- Henderson, P.; *An operating system.* In [*Darlington* et al, 1982].
- Henderson, P.: Functional geometry. In *ACM Conference on Lisp and Functional programming* (198?).
- Hoare, C. A. R.; Communicating Sequential Processes. *Communications ACM 21* (1978) 8, pp 666-677.
- van der Hoeven, G. F.; *Preliminary Report on the language TWENTEL.* Memorandum INF-84-5, Technische Hogeschool Twente, 1984.
- Joosten, S.; *The applicative description of discrete systems.* Graduate Thesis, Technische Hogeschool Twente, 1985.
- Milner, R.; A theory of type polymorphism in programming. *JCSS,* 17 (1978) pp 348-375.
- Turner, D. A.; *SASL Language Manual.* Saint Andrews University U.K., 1976.
- Turner, D. A.; A new implementation technique for applicative languages. *Software-Practice and Experience,* 9 (1979) 1, pp 31-49.
- Turner, D. A.; *The semantic elegance of applicative languages.* In Proc. 1981 ACM Conf on Functional Languages and Computer Architecture, 1981, pp 85-92.
- Turner, D. A.; *Recursive equations as a programming language.* In [Darlington et al 1982].