Een les in functioneel programmeren

Maarten Fokkinga, 2 oktober 1985

Werkexemplaar

We vertellen het een en ander over de methodiek van functioneel programmeren, aan de hand van het probleem om bij gegeven tekst een index te vormen.

* * *

1. Het voorbeeldprobleem

We definieren een tekst als een eindige lijst van characters met nl (new line) ter representatie van een regelovergang en sp ter representatie van een spatie. Een woord in een tekst is een niet-lege maximale opeenvolging (sub-lijst) van characters die van sp en nl verschillen. Een index van een tekst is een (niet-geordende) lijst van $[w, [n_1, \ldots, n_K]]$-paren, waarbij w varieert over de verschillende woorden in de tekst en $n_1, \ldots, n_K$ de nummers zijn van de regels waarin woord w voorkomt in de tekst; de lijst $[n_1, \ldots, n_K]$ moet oplopend gesorteerd en zonder duplicaten zijn.

Gevraagd wordt een functie index te construeren zo dat (index t) = een index van de tekst t.

Alvorens over te gaan tot het bespreken van allerlei programmeertechnieken, geven we eerst onze oplossing voor dit probleem. Hierbij, en in de komende paragrafen, gebruiken we de volgende naamgeving.

t :	tekst	c :	character
r :	regel,	R :	lijst van regels
w :	woord,	W :	lijst van woorden
n :	nummer,	nW :	lijst van [w, n]-paren
			= lijst van genummerde woorden

Allereerst definieren we een stel hulpfuncties:

```
--Regels t = de lijst van regels van t
Regels []     = []
Regels (nl: t) = []: Regels t
Regels (c : t) = Add c (Regels t)

-- Add x Y : voegt x toe op kop van eerste elt van Y
Add x (y:Y) = (x:y): Y
Add x []    = (x:[]): []

--Woorden r = de lijst van woorden van r
Woorden []     = []
Woorden (sp: r) = Woorden r
Woorden (c : r) = Add c (Woorden r)
```

```
-- Nummer n X :  nummert de elten in X met n, n+1, ...
Nummer n []     = []
Nummer n (x:X)  = [x, n]: Nummer (n+1) X

-- nW t = de lijst van genummerde woorden in tekst t
nW t = { [w, n] | [r, n] ← Nummer 1 (Regels t);
                  w ← Woorden r }
```

Er zijn nu verscheidene mogelijkheden om de index van een tekst t te vormen uit (nW t). Een mogelijkheid is om eerst (nW t) te sorteren (met een efficient, logaritmisch algoritme zoals quicksort) en dan "in te dikken", dwt. nummers van gelijke woorden samen te nemen en duplicaten weg te laten. Dus:

```
index t = DikIn (sort hl (nW t))
where   -- hl [w, n] [w', n'] = [w, n] is lexicogr. kl. dan [w', n']
        hl [c:w, n] [c':w', n'] = c<c' or c=c' and hl [w,n] [w',n']
        hl [[], n] [[], n'] = n<n'
        hl [w, n] [w', n'] = true

DikIn [] = []
DikIn ([w,n]:[]) = [w, [n]]: []
DikIn ([w,n]: nW) = let [w', N']:index = DikIn nW
                    in  w=w' & n = hd N' → [w', N']: index ;
                        w=w' → [w', n: N']: index ;
                        [w, [n]]: [w', N']: index
```


Een andere methode om uit (nW t) de index te vormen is als volgt.

```
index t = nuh-Index (nW t)

nuhIndex [] = []
nuhIndex ([w,n]: nW) = voegin [w,n] (nuhIndex nW)

voegin [w,n] [] = [w, [n]]
voegin [w,n] ([w', N']: index) =
        w = w' & n = hd N' → [w', N']: index ;
        w = w' {& n < hd N'} → [w', n: N']: index ;
        {w ≠ w' →} [w', N']: voegin [w,n] index
```

We hebben hierbij gebruik gemaakt van het feit dat (nW t) naar stijgende nummers gesorteerd is zo dat ook bij iedere [w; N'] in een index de N' stijgend (zelfs oplopend, dus zonder duplicaten) gesorteerd is.

We besluiten hiermee de probleemstelling en onze oplossing ervoor. We zullen nu aan de hand hiervan alternatieve oplossingen en alternatieve methoden bespreken en de voor- en nadelen toelichten.

2. Modulair programmeren

We hebben het probleem opgesplitst in een aantal subproblemen en daarvoor afzonderlijke oplossingen uitgeprogrammeerd: Regels splitst een tekst in regels, Woorden splitst een regel in woorden, Nummer nummert een rij van objecten, etcetera. Deze funkties zijn op zich zinvol en hebben ook buiten de kontekst van het behandelde probleem waarschijnlijk wel toepassingen. Hun effect is op zich elegant en vertoont geen intrinsieke samenhang: ze zijn ieder in isolatie gemakkelijk te begrijpen. Het is daarom dat we ze met recht modulen kunnen noemen. Modulen zijn algemeen trepasbare bouwstenen. Onze probleemoplossing kenmerkt zich door zijn modulaire opzet. Om dit te verduidelijken geven we hieronder een niet-modulaire oplossing. We combineren de taken van Regels, Woorden en Nummer in één functie f en definieren daarmee de functie nW.

```
nW t = f t 1 []
where -- bij een aanroep (f t n w) staat t voor de
      -- nog te verwerken tekst, n voor het nummer
      -- van de huidige regel (die dus vooraan staat
      -- in t) en w voor het alreeds gevormde woord
      -- uit de voorgaande tekst.
```



```
f [] n [] = []
f [] n w = [w,$n$]: []
f (nl:t) n [] = f t (n+1) []
f (nl:t) n w = [w,n]: f t (n+1) []
f (sp:t) n [] = f t n []
f (sp:t) n w = [w,n]: f t n []
f (c:t) n w = f t n (w ++ [c])
```

~~Dit programma~~ Deze definitie van nW is weliswaar korter maar ook minder eenvoudig --vinden wij--. Dat komt door het gebrek aan modulariteit: verschillende taken zoals die van Regels, Woorden en Nummer, zijn nu in f samengesmolten. Het nadeel daarvan blijkt ook al uit het feit dat "alreeds gedeeltelijk gevormde woorden" (de derde parameter van f) leeg kunnen zijn en dat daarmee steeds rekening gehouden moet worden.

Oefening. Maak de bovenstaande definitie van nW ietwat modulairder, als volgt. Definieer een functie f' die net zo werkt als f behalve dat ook lege woorden --samen met hun nummer-- in de resultaatlijst worden opgeleverd. Definieer ~~vervolgens~~ ook een functie g die, gegeven een lijst van genummerde

woorden, de lege woorden eruit filtert. Druk nu nW uit in f' en g. Vind je deze definitie modulairder? Vind je de tekst eenvoudiger? □

3. Coroutines en stream processing

Zoals we zojuist gezien hebben komt de modulaire opzet van programma's de begrijpelijkheid en algemene toepasbaarheid ten goede. En gelukkig hoeft dat niet ten koste te gaan van efficientie, in de zin dat er voor de tussenresultaten (lijsten!) van de modules extra opslagruimte nodig is. Dit komt door de luie evaluatie strategie. We zullen het nu iets uitvoeriger beschrijven.

Beschouw onze oorspronkelijke definities voor Regels, Woorden, en Nummer en nW. Stel eens dat de evaluatie plaats vindt zoals gebruikelijk is bij imperatieve talen, nml. dat eerst de argumenten volledig worden geevalueerd alvorens met die waarden aan de evaluatie van de functieromp wordt begonnen. Dan zouden bij nW éérst alle regels gevormd worden alvorens Nummer daarvan een genummerde lijst maakt en waarna Woorden dan de lijst van alle genummerde

woorden maakt. De afzonderlijke programmering van de splitsing in regels, de splitsing in woorden, en de nummering van regels zou op deze manier veel opslagruimte vergen voor het opbergen van de grote tussenresultaten (de hele lijsten!). Dank zij lazy evaluation, echter, vindt de evaluatie van de afzonderlijk geprogrammeerde modules toch verweven plaats: pas wanneer een element van (nW t) wordt gevraagd, wordt de berekening van de verzamelingsnotatie gestart; die, op zijn beurt, ~~start de berekening~~ vraagt om een element van (Nummer (Regels t)); dus Nummer vraagt om ~~de eerste~~ één regel, zodat (Regels t) zó ver berekend wordt dat één regel wordt opgeleverd; daarmee kan Nummer het gevraagde resultaat leveren, zodat ~~nieuwe~~ Woorden ook een element van z'n resultaatlijst oplevert --en reeds al een heel stuk van z'n resultaatlijst--. Aldus is het gevraagde element van (nW t) geproduceerd, zonder dat bijvoorbeeld (Regels t) helemaal geevalueerd is. In het bijzonder geldt dat er steeds hoogstens ~~slechts~~ één regel als tussenresultaat aanwezig is. ~~en~~ (Na het verwerken van zo'n regel kan de daarvoor gebruikte opslagruimte weer worden vrijgegeven en bijvoorbeeld voor een volgende regel worden gebruikt.)

Dit verschijnsel van het coroutine-effect (hier in de

vorm van stream processing) ~~houdt niet zomaar tot stand maar moet~~ treedt niet bij iedere programmatekst op. Het is de taak van de programmeur om z'n ~~teksten zo~~ definities zó te formuleren dat het gewenste effect ook optreedt. Ter verduidelijking geven we nu een variant waarbij ondanks de lazy evaluation geen coroutine-effect optreedt. Beschouw bijvoorbeeld de functie Nummer; deze gaan we iets anders definieren. Het idee is om niet onmiddellijk een gedeelte van het functieresultaat te produceren, maar voortdurend de alreeds beschikbare resultaatgedeelten op te sparen in een extra parameter. ~~Dit wordt wel~~ Zo'n parameter wordt wel accumulatie-parameter genoemd, en deze methode de parameter-accumulatie.

```
Nummer n X = N n X []
where N n [] Out = Out
      N n (x:X) Out = N (n+1) X (Out ++[[x,n]])
```

Wanneer nu voor zekere lijst R het eerste element van (Nummer 1 R) wordt gevraagd, dan wordt volgens de definitie van N eerst geheel R genummerd totdat alles genummerd is en volgens de eerste clausule voor N de hele genummerde lijst wordt ~~op~~ opgeleverd.

Oefening. Geef alternatieve definities voor de functies Regels, Woorden, f, ~~en~~ f' en g zodat ook daar het coroutine-effect volledig verdwijnt. □

Eerlijkheidshalve moeten we nog opmerken dat bij de berekening van de functies sort en voegin toch wel (bijna) de hele woordenlijst als tussenresultaat aanwezig is ook al is er slechts om het eerste element van (index t) gevraagd. Dit komt niet door een gebrekkige programmering maar is inherent aan de probleemstelling: ~~te~~ ieder woord in (index t) kan overal in t zijn voorgekomen en om de regelnummers van al die voorkomens op te sommen moet de hele tekst dus behandeld zijn.

4. Dubbele resultaten versus dubbele herhalingen

Het komt vaak voor dat een lijst gesplitst moet worden in een begin- en een eindstuk. Bijvoorbeeld, (Regels t) zou je ook kunnen verkrijgen door van de lijst t de eerste regel en de rest afzonderlijk te bepalen en dan recursief op de rest weer de functie Regels toe te passen. Deze aanpak leidt tot de volgende definitie.

```
Regels []  = []
Regels t   = (EenRegel t) : Regels (Rest t)
where
EenRegel (nl:t) = ()
EenRegel (c:t)  = c : EenRegel t
EenRegel []     = []

Rest (nl:t) = t
Rest (c:t)  = Rest t
Rest []     = []
```

We zien hier twee funkties, EenRegel en Rest, die ieder afzonderlijk de tekst t op gelijke wijze doorlopen en ieder hun eigen resultaat opleveren. Dit soort dubbele herhalingen is vaak eenvoudig in te ruilen voor een enkele herhaling: combineer de twee funkties tot één, met een paar als resultaat; de gecombineerde funktie hoeft de lijst daarn maar één maal te doorlopen. Letterlijke toepassing van deze werkwijze levert dan de volgende definitie.

```
Regels []= []
Regels t = r: Regels t'  where [r, t'] = EenRegelEnRest t
where
EenRegelEnRest (nl: t) = [ [], t ]
```



```
EenRegelEnRest (c: t) = [c:r, t']
                        where [r, t']= EenRegelEnRest t
EenRegelEnRest []     = [[], []]
```

Deze definitie levert een efficientere berekening dan wanneer EenRegel en Rest apart zijn gedefinieerd: de winst zit enerzijds in het feit dat het doorlopen van t (de test of t leeg is en of het eerste element gelijk nl is) nu maar eenmaal gedaan hoeft worden, en anderzijds in het feit dat de lijst t minder lang bewaard hoeft worden. ≠

Oefening. Herdefinieer de funktie Woorden volgens beide bovenstaande manieren. Geef ook, volgens beide bovenstaande manieren, een funktie Splits zodat (Splits x Y) = [{y | y ← Y; y < x}, {y | y ← Y; y ≥ x}]; gebruik hierbij de verzamelingsnotatie niet!

≠ Eerlijkheidshalve moeten we er wel bij zeggen dat de grootte-orde van de tijdsduur of benodigde opslagruimte in beide gevallen lineair is in de lengte van de lijst t.