

Hoofdstuk 1

Lisp voor Haskell-kenners

1.1 Expressies

1.1.1 Functie-aanroep

In Lisp kun je, net als in Haskell, een functie aanroepen door de naam van de functie en de parameters naast elkaar te schrijven. In Lisp moet echter het geheel nog tussen haakjes geschreven worden.

Het resultaat van een functie-aanroep kan gebruikt worden als parameter van een andere functie. In Haskell moet de deel-aanroep dan tussen haakjes gezet worden om aan te geven dat het één parameter betreft; in Lisp staat de deel-aanroep tussen haakjes omdat elke aanroep tussen haakjes moet staan.

Al met al lijken Lisp-expressies sterk op Haskell-expressies. Het enige verschil is dat er in Lisp ook om de buitenste aanroep in een expressie haakjes moeten staan.

Haskell	Lisp
<code>sqrt 2.0</code>	<code>(sqrt 2.0)</code>
<code>sqrt (exp 1.0)</code>	<code>(sqrt (exp 1.0))</code>

1.1.2 Operatoren

In Haskell zijn er functies en operatoren. De naam van een functie begint met een letter, de naam van een operator bestaat uit symbolen. Functie-namen worden *voor* de parameters geschreven, operatoren er *tussen* (operatoren hebben altijd twee parameters).

In Lisp worden functies en operatoren altijd *voor* de parameter geschreven. Afgezien van de schrijfwijze (letters, resp. symbolen) is er dus geen verschil tussen functies en operatoren.

Prioriteit en associatievolgorde zijn eigenschappen van infix-operatoren. Dit is in Lisp dus geen issue; de berekeningsvolgorde wordt altijd expliciet aangegeven.

Functies kunnen in Lisp een variabel aantal parameters hebben. De functie `+` is daarvan een voorbeeld.

Haskell	Lisp
<code>f 1 2</code>	<code>(f 1 2)</code>
<code>1 + 2</code>	<code>(+ 1 2)</code>
<code>x <= y</code>	<code>(<= x y)</code>
<code>x+y*z</code>	<code>(+ x (* y z))</code>
<code>x*(y+z)</code>	<code>(* x (+ y z))</code>
<code>v+w+x+y+z</code>	<code>(+ v w x y z)</code>

1.1.3 Lijsten

In Haskell kun je lijsten opschrijven door de elementen, gescheiden door komma's, tussen vierkante haken te zetten. In Lisp wordt voor lijsten dezelfde notatie gebruikt als voor functie-aanroep: de elementen staan naast elkaar, gescheiden door spaties, en het geheel staat tussen ronde haken.

Om te voorkomen dat het eerste element van de lijst wordt opgevat als functie, die moet worden uitgerekend, wordt het geheel voorafgegaan door het symbool ' (spreek uit: quote).

De lege lijst wordt in Lisp aanguid met de naam `nil`, of desgewenst met een leeg, rond haakjespaar. Net als Haskell kent Lisp strings, die mogen worden gebruikt als lijst van characters.

Haskell	Lisp
<code>[1,2,3]</code>	<code>'(1 2 3)</code>
<code>[]</code>	<code>nil</code>
<code>[]</code>	<code>()</code>
<code>length [1,2,3]</code>	<code>(length '(1 2 3))</code>
<code>length "aap"</code>	<code>(length "aap")</code>

1.2 Functies op lijsten

Zowel in Haskell als in Lisp kan een lijst ook worden opgebouwd met de op-kop-van functie. In Haskell heet die `:`, in Lisp is dat `cons`. Lijsten kunnen geconcateneerd worden met `++` in Haskell, en met `append` in Lisp.

Haskell	Lisp
<code>1:2:3:4:[]</code>	<code>(cons 1 (cons 2 (cons 3 (cons 4 ())))))</code>
<code>[1,2,3] ++ [4,5,6]</code>	<code>(append '(1 2 3) '(4 5 6))</code>

1.3 Functiedefinitie

1.3.1 Simpele definitie

Ingrediënten van een functiedefinitie zijn de naam van de functie, declaratie van parameters, en een body (expressie waarin de parameters gebruikt mogen worden).

In Haskell is er een speciale syntax voor functiedefinities, te herkennen aan het `=` symbool in het midden. Links daarvan staan de naam en namen voor de parameters, rechts staat de body.

In Lisp heeft een functiedefinitie dezelfde vorm als een functie-aanroep: ronde haakjes met daartussen een aantal onderdelen, gescheiden door spaties. De onderdelen zijn achtereenvolgens: het speciale symbool `defun`, de naam van de functie, een lijst met parameternamen, en de body van de functie.

Haskell	Lisp
<code>kwadraat x = x*x</code>	<code>(defun kwadraat (x) (* x x))</code>
<code>boven n k = fac n</code>	<code>(defun boven (n k)</code>
<code> / (fac k * fac (n-k))</code>	<code> (/ (fac n)</code>
	<code> (* (fac k) (fac (- n k))))</code>

1.3.2 Definitie met gevalsonderscheid

Haskell heeft een aparte syntax voor een functiedefinitie met gevalsonderscheid: voor het `==`-teken in een functiedefinitie mag een `|` staan gevolgd door een voorwaarde. In een functiedefinitie kunnen meerdere voorwaarden en bijbehorende functie-bodies staan.

In Lisp worden de gevallen *in* de body van de functie uitgesplitst, door aanroep van de speciale functie `cond`. Deze functie heeft een aantal paren voorwaarde–body als parameter.

Haskell	Lisp
<code>signum x x<0 = -1</code>	<code>(defun signum (x)</code>
<code> x==0 = 0</code>	<code> (cond ((< x 0) -1)</code>
<code> x>0 = 1</code>	<code> ((= x 0) 0)</code>
	<code> ((> x 0) 1)))</code>

Haskell	Lisp
<pre>> 1<='a' Error: Type error in application > 2<=3 1<='a' Error: Type error in application</pre>	<pre>> (<= 1 'a) Error: A is not of type FLOAT > (or (<= 2 3) (<= 1 'a)) T</pre>

1.4.2 Types van lijsten

In Haskell moeten alle elementen van een lijst van hetzelfde type zijn. In Lisp hoeft dat niet. Een apart tupel-type voor een vast aantal elementen van verschillend type is in Lisp dus ook niet nodig. In Haskell zijn lijsten van lijsten mogelijk, mits dan ook alle elementen een lijst zijn. In Lisp kunnen sommige elementen van een lijst atomair zijn, en andere elementen een lijst.

Haskell	Lisp
<pre>[1,2,3] (1,"aap") [[1,2,3], [4,5,6]] [5, [4,5,6]] -- typeringsfout</pre>	<pre>'(1 2 3) '(1 "aap") '('(1 2 3) '(4 5 6)) '(5 '(4 5 6))</pre>

1.4.3 Overloading

In Haskell kan een functie gedefinieerd worden op meerdere types. Zo'n functie moet dan een member zijn van een class, waarna de verschillende types instance worden gemaakt van die class. door de compiler wordt de juiste versie van de functie gekozen aan de hand van het type.

In Lisp kun je run-time het type van de parameter opvragen, en op die manier een functie op meerdere types toepasbaar maken. De juiste versie wordt met gevalsonderscheid expliciet gekozen.

Haskell	Lisp
<pre>class Seq a where nummer :: a -> Int instance Seq Int where nummer n = n instance Seq Char where nummer c = ord c instance Seq Bool where nummer False = 0 nummer True = 1</pre>	<pre>(defun nummer (x) (cond ((integerp x) x) ((characterp x) (ord x)) (x 1) (t 0)))</pre>

In Haskell kun je reeds bestaande overloaded operatoren (zoals ==) nog verder overladen, door een nieuw type ook instance te maken van de betreffende klasse (Eq in het geval van ==). In Lisp is dat onmogelijk.

1.4.4 Polymorfie

Haskell is getypeerd volgens het Hindley-Milner typesysteem, waarin bijvoorbeeld onderscheid wordt gemaakt tussen lijsten-van-integer en lijsten-van-lijsten-van-boolean, en tussen functie-van-int-naar-char en functie-van-int-naar-bool. Door middel van type-variabelen is het mogelijk om aan te geven dat een functie bijvoorbeeld het type functie-van-willekeurig-type-naar-datzelfde-type heeft.

In Lisp zijn er types voor getallen (met subtypes voor integer, float, rational etc.), lijsten, en functies. Er kan geen onderscheid gemaakt worden tussen verschillende typen van functies.

1.5 Hogere-ordefuncties

1.5.1 Map / Mapcar

Zowel in Haskell als in Lisp is het mogelijk om een functie mee te geven als parameter aan een andere functie. Een bekend voorbeeld is de functie `map`, die in Lisp bekend staat onder de naam `mapcar`. In Lisp moet expliciet worden aangegeven dat de functie die als parameter wordt meegegeven moet worden opgezocht in de verzameling van alle eerder gegeven functie-definities. Daarom staat er in Lisp `#'f` in plaats van `f` als parameter van `map`.

In Lisp heeft de functie `map` een variabel aantal parameters. In plaats van op één lijst kan hij ook op twee lijsten worden toegepast. De functie-parameter moet dan een functie met twee parameters zijn; het effect is hetzelfde als dat van `zipWith` in Haskell. Ook generalisaties naar meer lijsten zijn mogelijk.

Haskell	Lisp
<code>map f [1,2,3]</code>	<code>(mapcar #'f '(1 2 3))</code>
<code>zipWith (+) [1,2,3] [4,5,6]</code>	<code>(mapcar #'(lambda (x y) (+ x y)) '(1 2 3) '(4 5 6))</code>

1.5.2 Foldr / Reduce

De functie overeenkomstig met Haskell's `foldr` heet in Lisp `reduce`. Een neutrale waarde hoeft om mij onduidelijke redenen niet te worden meegegeven; toch kan `reduce` op lege lijsten worden toegepast.

Haskell	Lisp
<code>foldr (+) 0 [1,2,3]</code>	<code>(reduce #'(lambda (x y) (+ x y)) '(1 2 3))</code>

1.5.3 Currying / Lambda-notatie

In Lisp zijn functies niet gecurried, en ze kunnen dus ook niet partieel geparametriseerd worden. Het effect kan wel worden bereikt met behulp van de lambda-notatie. Hiermee worden nieuwe, naamloze functies gemaakt, meestal met het doel om ze aan een andere functies mee te geven. De lambda-notatie bestaat ook in Haskell, maar is minder vaak nodig omdat in plaats daarvan partiële parametrisatie kan worden gebruikt.

Haskell	Lisp
<code>\x -> x*x</code>	<code>(lambda (x) (* x x))</code>
<code>map (\x->x*x) [1,2,3]</code>	<code>(mapcar #'(lambda (x) (* x x)) '(1 2 3))</code>
<code>map (+1) [1,2,3]</code>	<code>(mapcar #'(lambda (x) (+ x 1)) '(1 2 3))</code>
<code>until (>9) (*2) 1</code>	<code>(until #'(lambda (x) (> x 9)) #'(lambda (x) (* x 2)) 1)</code>

1.6 Filosofie

1.6.1 Haskell: referentieel transparant

Het resultaat van een functie-aanroep in Haskell is geheel bepaald door de waarden van de parameters. Het functieresultaat is de enige manier waarop een functie zijn omgeving kan beïnvloeden. Haskell-functies hebben dus geen *neveneffecten*; er zijn geen globale variabelen die veranderd kunnen worden.

Deze eigenschap maakt lazy evaluatie mogelijk: zonder dat het voor het eindresultaat uitmaakt, kan een berekening later, of helemaal niet worden uitgevoerd. Als deze berekeningen neveneffecten gehad zouden hebben, dan kan de uitreken-volgorde het eindresultaat wèl beïnvloeden.

De afwezigheid van neveneffecten maakt het mogelijk om over Haskell-programma's te redeneren als waren het wiskundige formules; zo zijn bijvoorbeeld in alle omstandigheden de aanroepen

`map f (xs++ys)` en `map f xs ++ map f ys` gelijk. Als de functie `f` neveneffecten op globale variabelen zou hebben, kun je niet zonder meer op de geldigheid van deze wet vertrouwen.

In Lisp is het wel mogelijk een waarde toe te kennen aan een globale variabele: door aanroep van de functie `setf` wordt een waarde aan een variabele toegekend.

1.6.2 Lisp: meta-circulair

De syntax van Lisp-programma's (functie-aanroepen zoals `(f 1 2)` lijkt sterk op die van Lisp-data (lijsten zoals `'(1 2 3)`). De Lisp-interpreter is als het ware een functie die een lijst, voorstellende een functie-aanroep, interpreteert. Dit maakt het eenvoudig om Lisp-programma's te schrijven die andere programma's manipuleren. Het is zelfs mogelijk om een programma door een programma te laten schrijven, en dat vervolgens uit te voeren.

Dat laatste effect zie je in Haskell meestal in de vorm van hogere-ordefuncties. Functies kunnen functie opleveren, die pas later op werkelijke data worden toegepast.

1.7 Haskell en Prolog

1.7.1 Lijsten

De notatie van lijsten is enigzins verwarrend als je afwisselend in Haskell en Prolog programmeert. Een opsomming gebeurt op dezelfde manier: komma's tussen de elementen en vierkante haken om het geheel. Singletons en de lege lijst worden op dezelfde manier gerepresenteerd. Maar het patroon waarbij een lijst in een kop en een staart wordt verdeeld is verschillend: in Haskell wordt de operator `:` gebruikt, in Prolog wordt het symbool `|` tussen kop en staart gezet, maar *bovendien* vierkante haken om het geheel. In Haskell moet je die extra vierkante haken vooral *niet* neerzetten: daarmee zou je een lijstnivo toevoegen!

Haskell	Prolog
<code>[1,2,3]</code>	<code>[1,2,3]</code>
<code>[5]</code>	<code>[5]</code>
<code>[]</code>	<code>[]</code>
<code>(x:xs)</code>	<code>[x xs]</code>
<code>[x:xs]</code>	<code>[[x xs]]</code>

1.7.2 Functies en relaties

Het belangrijkste verschil tussen Haskell en Prolog is dat je in Haskell *functies* specificeert, en in Prolog *relaties*. Bij een functie geef je aan wat bij bepaalde invoer de uitvoer is; bij een relatie geef je aan dat invoer en uitvoer met elkaar een relationeel verband hebben. Daarom hebben Prolog-relaties altijd een parameter meer dan de overeenkomstige Haskell-functies.

Daar waar je in Haskell een aanroep doet van een andere functie, krijgt de Prolog-relatie *voorwaarden*. Je zult merken dat je een extra naam nodig hebt, om het tussenresultaat te kunnen benoemen.

Haskell	Prolog
<code>d x = 3</code>	<code>d(x,3) .</code>
<code>f x = h (g x)</code>	<code>f(x,z) :- g(x,y), h(y,z) .</code>

1.7.3 Patronen

Gelukkig kent Prolog wel patronen, waarmee bijvoorbeeld de gevallen 'lege lijst' en 'niet-lege lijst' onderscheiden kunnen worden. Zo kan een relatie worden gedefinieerd waarmee wordt aangeduid dat twee lijsten aan elkaar geplakt een derde lijst opleveren. In Prolog heet deze relatie `append`; de overeenkomstige functie in Haskell heet `++`.

Haskell	Prolog
<code>[] ++ ys = ys</code>	<code>append([],ys,ys) .</code>
<code>(x:xs) ++ ys = x:(xs++ys)</code>	<code>append([x xs],ys,[x zs]) :- append(xs,ys,zs) .</code>

Merk op dat het resultaat van de recursieve aanroep in de Prolog-versie een aparte naam moet krijgen. Bij wat meer ingewikkelde functies/relaties kan dat tamelijk onoverzichtelijk worden:

Haskell	Prolog
<code>ins x [] = [x]</code>	<code>ins(x,[],[x]) .</code>
<code>ins x (y:ys)</code>	<code>ins(x,[y ys],[x y ys])</code>
<code> x<y = x:y:ys</code>	<code>:- x<y .</code>
<code> x>=y = y:ins x ys</code>	<code>ins(x,[y ys],[y zs])</code>
	<code>:- x>=y, ins(x,ys,zs) .</code>

1.7.4 Richtingloze definities

Een voordeel van de Prolog-definities is dat ze ook omgekeerd gebruikt kunnen worden. De append-relatie kan behalve voor het samenvoegen van twee lijsten ook worden gebruikt om een lijst in alle mogelijke delen te splitsen. In Haskell zou hiervoor een andere functie nodig zijn.

Haskell	Prolog
<pre>> [1,2]++[3,4] [1,2,3,4] splits [] = [([],[])] splits (x:xs) = ([],(x:xs)) : map f (splits xs) where f (a,b) = (x:a,b) > splits [1,2,3] [([], [1,2,3]) ,([1], [2,3]) ,([1,2], [3]) ,([1,2,3], [])]</pre>	<pre>> append([1,2],[3,4],Z). Z = [1,2,3,4] > append(X,Y,[1,2,3]). X=[] Y=[1,2,3] X=[1] Y=[2,3] X=[1,2] Y=[3] X=[1,2,3] Y=[]</pre>

Helaas is dit gebruik van relaties in twee richtingen alleen mogelijk als bij de definitie van relaties geen gebruik is gemaakt van ingebouwde relaties die niet in twee richtingen gebruikt mogen worden, zoals plus.

1.7.5 Constructorfuncties

In Prolog kunnen op de parameterplaatsen van predicaten ook weer functies aangeroepen worden. Dit stelt dan echter geen functie-aanroep voor van een elders gedefinieerde functie – dat dient immers te gebeuren met het eerder beschreven mechanisme.

Functies in Prolog komen overeen met wat in Haskell constructorfuncties genoemd zou worden.

Haskell	Prolog
<pre>data Boom = Tip Int Tak Boom Boom size (Tip x) = 1 size (Tak p q) = size p + size q</pre>	<pre>size(tip(x),1). size(tak(p,q),a+b) :- size(p,a), size(q,b).</pre>