

Hoofdstuk 1

Antwoorden

1.1 Haskell Brooks Curry Born: 12 Sept 1900 , Died: 1 Sept 1982. Curry's main work was in mathematical logic with particular interest in the theory of formal systems and processes. He formulated a logical calculus using inferential rules. His works include Combinatory Logic (1958) (with Robert Feys) and Foundations of Mathematical Logic (1963).

1.2

```
=>    gereserveerd symbool (staat in de lijst)
3a    niets (naam moet met letter beginnen)
a3a   naam
::    gereserveerd symbool (staat in delijst)
:=    operator [constructor]
:e    niets (opdracht is niet reserved)
X_1   naam [constructor]
<=>   operator
a'a   naam
_X    niets (naam moet met letter beginnen)
***   operator
'a'   niets (naam moet met letter beginnen)
A     naam [constructor]
in    gereserveerd woord (staat in lijst)
:--< operator [constructor]
```

1.3 4000.02, 80.0, 200000.0

1.4 `x=3` is een definitie van een constante (mag bijvoorbeeld achter `where` staan). `x==3` is een boolese expressie (met de waarde `True` of `False`).

1.5 De twee functies zijn:

```
aantal0pl a b c | d>0 = 2
                 | d==0 = 1
                 | d<0 = 0
                 where d = b*b-4.0*a*c
aantal0pl' a b c = 1 + signum (b*b-4.0*a*c)
```

1.6 Je kunt nu stukken programma inactief maken door er commentaar van te maken, ook als dat stuk programma zelf commentaar bevat.

1.7 Vraag het type aan de interpreter met bijvoorbeeld `:type tail`. Specificeer de types zelf door:

```
tail  :: [a] -> [a]
sqrt  :: Float -> Float
pi    :: Float
exp   :: Float -> Float
(^)   :: Num a => a -> Int -> a
(/=)  :: Eq a => a -> a -> Bool
aantal0pl :: Float -> Float -> Float -> Int
```

1.8 Respectievelijk `False` en `True`. In `C` is de eerste expressie `false`, wat in `C` gecodeerd wordt als `0`. De tweede expressie betekent in `C` echter 'x wordt door 3 gedeeld', met de waarde `2` en het neveneffect dat `x` deze waarde krijgt. (In `C` wordt de ongelijkheid genoteerd door `!=`).

1.9 'Syntax error' betekent 'vormfout'. Bij een syntax error staan de symbolen van een formule niet in de goede volgorde. Bij een type error is er wel sprake van een expressie, maar kloppen de types van de parameters niet met de functie of operator die gebruikt wordt.

1.10 $3 :: \text{Int}$ en $\text{even} :: \text{Int} \rightarrow \text{Bool}$, dus $\text{even } 3 :: \text{Bool}$. Je moet controleren of de parameter ‘past’ bij de functie; zo ja, dan is het resultaat van het type zoals gespecificeerd in de functie. Bij polymorfe functies gaat dat ongeveer hetzelfde: $\text{head} :: [\text{a}] \rightarrow \text{a}$ en $[1,2,3] :: [\text{Int}]$; het past dus, want $[\text{Int}]$ is een speciaal geval van $[\text{a}]$. Maar dan moet wel a gelijk zijn aan Int . Het type van $\text{head } [1,2,3]$ is dus niet a , maar Int .

1.11 De expressies hebben de volgende types:

```

until even      :: (Int    -> Int    ) -> Int    -> Int
until or        :: ([Bool] -> [Bool]) -> [Bool] -> [Bool]
foldr (&&) True ::          [Bool]    -> Bool
foldr (&&)      :: Bool      -> [Bool]  -> Bool
foldr until    :: (a -> a) -> [a -> Bool] -> a -> a
map sqrt       :: [Float   ] -> [ Float   ]
map filter     :: [a -> Bool] -> [ [a] -> [a] ]
map map        :: [a -> b  ] -> [ [a] -> [b] ]

```

1.12

1.13

- bij de aanroep $\text{fac } (-3)$ wordt eerst $\text{fac } (-4)$ berekend, waarvoor $\text{fac } (-5)$ nodig is. De ‘basiswaarde’ 0 wordt op deze manier nooit bereikt, en er ontstaat een oneindig lang durende berekening.
- De ordening volgens welke de parameter bij de recursieve aanroep ‘eenvoudiger’ is, moet naar onder begrensd zijn, en het basisgeval moet deze ondergrens behandelen.

1.14 In een lijst is ook de volgorde van de elementen van belang, en het aantal maal dat een element voorkomt.

1.15

1.16 Een lijst van lijsten, waarbij steeds drie keer hetzelfde element gekopieerd is:

```
[ [0,0,0], [1,1,1], [3,3,3], [6,6,6], [10,10,10] ]
```

2.1 $x+h$ is in zijn geheel parameter van f . Als de haakjes er niet staan, zou f alleen op x worden toegepast, omdat functie-toepassing de hoogste prioriteit heeft. Bij $f x$ hoeven er daarom geen haakjes te staan. Omdat $/$ een hogere prioriteit heeft dan $-$, moeten om de linker parameter van $/$ ook haakjes staan.

2.2

- Het eerste paar haakjes is overbodig, omdat functie-applicatie naar links associeert. Het tweede paar is echter wel nodig, omdat anders de eerste plus vier parameters zou krijgen.
- Beide haakjesparen zijn overbodig. Het eerste omdat haakjes om losse parameters niet nodig zijn (wel moet er in dit geval een spatie tussen sqrt en 3.0 , omdat er anders de naam $\text{sqrt}3$ zou staan). Het tweede haakjespaar is overbodig omdat functie-applicatie een hogere prioriteit heeft dan optelling (dat kon u niet weten, maar wel uitproberen).
- De haakjes om de getallen zijn overbodig. De haakjes om de operator niet, omdat de operator hier in prefix-notatie (als een soort functie) wordt gebruikt.
- Haakjes zijn overbodig, want vermenigvuldigen heet uit zichzelf al een hogere prioriteit dan optellen.
- Haakjes zijn nodig, want optellen heeft een lagere prioriteit dan vermenigvuldigen.
- Het tweede paar haakjes is overbodig, omdat \rightarrow naar rechts associeert. Het eerste paar is wel nodig, omdat het hier een functie met een functie-parameter betreft, en niet een functie met drie parameters.

2.3 Door rechts-associatie is a^{b^c} hetzelfde als $(a^b)^c$: eerst b^c berekenen, en dan a tot die macht verheffen. Zou machtsverheffen links associëren, dan is a^{b^c} hetzelfde als $(a^b)^c$, maar dat kun je ook al zonder haakjes schrijven als $a^{b \cdot c}$. Associatie naar rechts bespaart dus haakjes.

2.4 De operator $.$ is associatief, want

```

((f.g).h) x
= (f.g) (h x)
= f (g (h x))
= f ((g.h) x)
= (f.(g.h)) x

```

2.5 Omdat er dan haakjes nodig zijn in expressies als $0 < x \ \&\& \ x < 10$.

2.6 waardeIn0, plus, diff.

2.7 Omdat na naar rechts associeert, mogen er om het rechter pijltje en zijn parameters strafeloos extra haakjes geplaatst worden. Je krijgt dan:

```
na :: (b->c) -> ((a->b) -> (a->c))
```

Hieruit blijkt dat `na` een functie is met één parameter van type `b->c` en een resultaat van type `(a->b)->(a->c)`. In de volgende definitie heeft de functie `na` inderdaad maar één parameter:

```
na y = h
      where h f x = y (f x)
```

2.8

```
f rente eind start
  | eind <= start = 0
  | otherwise     = 1 + f rente eind ((1+rente)*start)
```

2.9 Voor de functie f met $f x = x^3 - a$ geldt $f' x = 3x^2$. De formule voor verbeter `b` in paragraaf ?? is in dit geval als volgt te vereenvoudigen: blz. ??

$$\begin{aligned} & b - \frac{f b}{f' b} \\ = & b - \frac{b^3 - a}{3b^2} \\ = & b - \frac{b^3}{3b^2} + \frac{a}{3b^2} \\ = & \frac{1}{3}(2b + a/b^2) \end{aligned}$$

De functie derdemachtswortel luidt dan:

```
derdemachtswortel x = until goedGenoeg verbeter 1.0
  where verbeter y   = (2.0*b + a/(b*b)) / 3.0
        goedGenoeg y = y*y*y ~ x
```

2.10 Door gebruik te maken van de lambda-notatie hoeft de functie `f` niet apart een naam te krijgen:

```
inverse g a = nulpunt (\x -> g x - a) 0.0
```

2.11 Omdat voor de vereenvoudiging van verbeter `b` het functievoorschrift van de te inverteren functie nodig is. Als de te inverteren functie een parameter is, is dat functievoorschrift niet bekend; het enige wat er opzit is de functie in (veel) punten uit te proberen; dit gebeurt in `nulpunt`.

2.12

3.1 Schrijf `[1,2]` als `1:(2:[])` en pas driemaal de definitie van `++` toe:

```
[1, 2] ++ []
= (1:(2:[])) ++ []
= 1 : ((2:[]) ++ [])
= 1 : (2 : ([] ++ []))
= 1 : (2 : [])
= [1, 2]
```

3.2 `concat = foldr (++) []`

3.3 True zijn de expressies 4, 5 en 6.

3.4 De functie `box` zet zijn parameter als singleton in een lijst als hij aan `p` voldoet, en levert anders de lege lijst op:

```
box x | p x      = [x]
      | otherwise = []
```

3.5 `repeat = iterate id where id x = x`

3.6 ...

3.7 ...

3.8 `qEq (x,y) (p,q) = x*q==p*y`

3.9 Het resultaat van $(a+bi) * (c+di)$ kan berekend worden door de haakjes uit te werken: $ac + adi + bci + bdi^2$. Vanwege $i^2 = -1$ is dit gelijk aan $(ac-bd) + (ad+bc)i$. Voor de uitkomst van $1/(a+bi)$ lossen we x en y op uit $(a+bi) * (x+yi) = (1+0i)$. Dit geeft twee vergelijkingen met twee onbekenden: $ax - by = 1$ en $ay + bx = 0$. De tweede vergelijking geeft $y = (-b/a)x$. Invullen in de eerste vergelijking geeft $ax + \frac{b^2}{a}x = 1$ dus $x = \frac{a}{a^2+b^2}$ en $y = \frac{-b}{a^2+b^2}$.

```
type Complex = (Float,Float)
cPlus, cMin, cMaal, cDeel :: Complex -> Complex -> Complex
cPlus (a,b) (c,d) = (a+c, b+d)
cMin (a,b) (c,d) = (a-c, b-d)
cMaal (a,b) (c,d) = (a*c-b*d, a*d+b*c)
cDeel (a,b) (c,d) = cMaal (a,b) (c/k, -d/k)
  where k = c*c+d*d
```

3.10 De waarde van `stringInt ['1','2','3']` is $((0 * '1') * '2') * '3'$, waarbij de operator `*` de operatie ‘vermenigvuldigd de linker waarde met 10 en tel de `digitValue` van de volgende character erbij op’ is. Je begint aan de linkerkant, en kunt dus `foldl` gebruiken:

```
stringInt = foldl f 0
  where f n c = 10*n + digitValue c
```

In dit geval kan niet `foldr` gebruikt worden, omdat de operator niet associatief is. Dit is trouwens ook eens een voorbeeld waarbij het type van de twee parameters van de operator niet hetzelfde is. Dat kan, kijk maar naar het type van `foldl`.

3.11

```
curry :: ((a,b)->c) -> (a->b->c)
curry f x y = f (x,y)
```

3.12 Net als `zoekOp` gaat `zoekOpBoom` fout als de gezochte waarde niet in de verzameling zit, omdat er geen definitie voor `zoekOpBoom` Blad `z` is gegeven. Desgewenst kan een regel worden toegevoegd, waarin afhankelijk van de toepassing in dit geval een speciale waarde wordt opgeleverd. De te gebruiken speciale waarde zou eventueel zelfs als extra parameter aan `zoekOpBoom` kunnen worden meegegeven.

```
zoekOpBoom :: Ord a => Boom (a,b) -> a -> b
zoekOpBoom (Tak (x,y) li re) z | z==x = y
                               | z<x  = zoekOpBoom li z
                               | z>x  = zoekOpBoom re z
```

3.13 De functie `map` transformeert een functie in een functie tussen *lijsten* van resp. zijn domein en bereik. Aangezien `map` zelf het type $(a \rightarrow b) \rightarrow ([a] \rightarrow [b])$ heeft, staan in het type van `map` vierkante haken om domein en bereik:

```
map map :: [a->b] -> [[a]->[b]]
```

3.14

```
until p f x = hd (dropWhile p (iterate f x))
```

3.15 De operator `<` op lijsten kan gedefinieerd worden met de volgende recursieve definitie:

```
xs < [] = False
[] < ys = True
(x:xs) < (y:ys) = x<y || (x==y && xs<ys)
```

De volgorde waarin de eerste twee regels gegeven staan is van belang. Omdat de regels in volgorde worden geprobeerd, zal bij het vergelijken van twee lege lijsten de eerste regel gebruikt worden, wat inderdaad de bedoeling is.

3.16 Een definitie van `length` met behulp van `foldr` luidt:

```
length = foldr op 0
  where x 'op' n = n+1
```

We beginnen met het voorlopige resultaat 0, en de operator `op` telt hierbij voor elk element in de lijst één op. De parameter `n` stelt het aantal elementen in de rest van de lijst voor, dat al geteld is door de recursieve aanroep in de definitie van `foldr`. De parameters van de operator `op` hebben niet hetzelfde type: de linker parameter is van willekeurig type (het type van de lijstelementen doet er niet toe), de rechter parameter en het resultaat echter zijn van type `Int`. Dus het type van de functie die aan `foldr` wordt meegegeven is $a \rightarrow \text{Int} \rightarrow \text{Int}$. Inderdaad accepteert de functie `foldr` functies met het type $a \rightarrow b \rightarrow b$.

3.17 De functie `qsort` kan als volgt gedefinieerd worden:

```
qsort :: Ord a => [a] -> [a]
qsort [] = []
qsort (x:xs) = qsort (filter (<=x) xs)
              ++ [x] ++
              qsort (filter (>x) xs)
```

Het verschil met `msort` is dat die functie twee willekeurige deel-lijsten neemt, die na het recursieve sorteren met enige moeite samengevoegd moeten worden, terwijl `qsort` de moeite investeert voor het selecteren van elementen in de deel-lijsten, zodat ze na het recursieve sorteren eenvoudigweg aan elkaar geplakt kunnen worden.

3.18 De functie `groepeer` kan als volgt gedefinieerd worden:

```
groepeer :: Int -> [a] -> [[a]]
groepeer n = takeWhile (not.null)
            . map (take n)
            . iterate (drop n)
```

Als parameter van `takeWhile` gebruiken we `not.null` in plaats van `/=[]`, omdat door het vermijden van de operator `/=` het niet nodig is dat de lijstelementen vergelijkbaar zijn. De functie `null` is in de prelude gedefinieerd door

```
null [] = True
null xs = False
```

3.19 De functie `mapBoom` past een gegeven functie toe op alle elementen die in de bladeren staan opgeslagen:

```
mapBoom :: (a->b) -> Boom2 a -> Boom2 b
mapBoom f (Blad2 x) = Blad2 (f x)
mapBoom f (Tak2 p q) = Tak2 (mapBoom f p) (mapBoom f q)
```

De functie `foldboom` past een gegeven operator toe in elke Tak:

```
foldBoom :: (a->a->a) -> Boom2 a -> a
foldBoom op (Blad2 x) = x
foldBoom op (Tak2 p q) = op (foldBoom op p) (foldBoom op q)
```

3.20 De inductieve versie:

```
diepte (Blad2 x) = 0
diepte (Tak2 p q) = max (diepte p) (diepte q)
```

De versie met gebruik van `map` en `fold`:

```
diepte = foldBoom max . mapBoom 0
```

3.21 We schrijven een hulpfunctie, met als extra parameter de diepte (en dus het aantal toe te voegen extra spaties) van de boom.

```
toonBoom = toonBoom' 0
toonBoom' n Blad = replicate n ' ' ++ "Blad\n"
toonBoom' n (Tak x p q) = toonBoom' (n+5) p
                        ++ replicate n ' ' ++ show' x ++ "\n"
                        ++ toonBoom' (n+5) q
```

3.22 Als de boom helemaal scheef is gegroeid, bevat hij het minimale aantal bladeren: $n + 1$. Een helemaal gevulde boom heeft 2^n bladeren.

3.23 We delen de lijst in twee ongeveer gelijke stukken en een los element, en passen de functie recursief toe op de twee helften. Dat levert twee ongeveer even diepe bomen. Die bomen voegen we samen toe één boom, waarbij ook het losse element wordt toegevoegd:

```
maakBoom [] = Blad
maakBoom xs = Tak x (maakBoom as) (maakBoom bs)
  where as = take k xs
        (x:bs) = drop k xs
        k = length xs / 2
```

4.1 Niet alleen worden de eerste zeven en de laatste vier elementen omgewisseld. Door het effect in de recursieve aanroep worden ook die zeven elementen anders gerangschikt:

```
segs [1,2,3,4] = [ [1], [1,2], [1,2,3], [1,2,3,4], [2], [2,3], [2,3,4], [3], [3,4], [4], [] ]
```

4.2 `segs = ([:] . filter (/=[])) . concat . map inits . tails`

4.3 $(n+1)$, $1 + (n^2+n)/2$, 2^n , $n!$, $\binom{n}{k}$.

4.4 Daar waar de functie `subs` kiest voor 'x in het resultaat opnemen' kiest `bins` voor 'een 1 in het resultaat opnemen'. Waar de functie `subs` kiest voor 'x niet in het resultaat opnemen' kiest `bins` voor 'een 0 in het resultaat opnemen':

```
bins 0 = [ [] ]
bins (n+1) = map ('0':) binsn ++ map ('1':) binsn
  where binsn = bins n
```

4.5 Het kan zowel recursief (`gaps`) als met gebruikmaking van andere functies (`gaps'`):

```
gaps [] = [ ]
gaps (x:xs) = xs : map (x:) (gaps xs)
gaps' xs = zipWith (++) (init(inits xs)) (tail(tails xs))
```

4.9 De functie `cp` combineert de elementen van twee lijsten op alle mogelijke manieren met elkaar. De functie `crossprod` combineert de elementen van een *lijst van* lijsten op alle mogelijke manieren met elkaar. We schrijven de functie met inductie. Als er slechts één lijst is, vormen de elementen daarvan in hun eentje de 'combinaties van één element'. Dus:

```
crossprod [xs] = map singleton xs
```

De functie `singleton` kan geschreven worden als `(: [])`. Voor het recursieve geval bekijken we bijvoorbeeld de lijst `[[1,2] , [3,4] , [5,6]]`. Een recursieve aanroep van `crossprod` op de staart levert de lijst `[[3,5], [3,6], [4,5], [4,6]]`. Deze elementen moeten op alle manieren gecombineerd worden met `[1,2]`. Dit geeft de definitie:

```
crossprod (xs:xss) = cpWith (:) xs (crossprod xss)
```

De definitie van het basisgeval kan ook anders geschreven worden:

```
crossprod [xs] = cpWith (:) xs [[]]
```

Wat het `crossprod` van een lege lijst is, is niet geheel duidelijk. Maar als we definiëren

```
crossprod [] = [[]]
```

dan is de definitie voor een lijst met één element gelijk aan die van een lijst met meer elementen:

```
crossprod [xs] = cpWith (:) xs (crossprod [])
```

Daarmee heeft de definitie de structuur gekregen van `foldr`, en kan dus ook geschreven worden als:

```
crossprod = foldr (cpWith (:)) [[]]
```

De lengte van het `crossprod` van een lijst van lijstjes is het product van de lengtes van de lijstjes:

```
lengthCrossprod xs = product (map length xs)
```

Vandaar ook de naam *cross product*.

4.10 Als we ervoor zorgen dat er nooit staarten met nullen worden opgeslagen, dan is de graad 1 minder dan de lengte van de lijst (behalve voor het 0-polynoom):

```
pGraad [] = 0
pGraad xs = length xs - 1
```

Er is dan wel een vereenvoudig-functie nodig, die staarten met nullen verwijdert:

```
pEenvoud [] = []
pEenvoud (x:xs) | x==0.0 && staart==[] = []
                | otherwise           = x : staart
  where staart = pEenvoud xs
```

Voor het optellen van twee polynomen kunnen de overeenkomstige termen worden opgeteld, waarna het resultaat wordt vereenvoudigd:

```
pPlus xs ys = pEenvoud (zipWith (+) xs ys)
```

Definitie van `pMaal` geschiedt met inductie naar het eerste polynoom. Als dit het 0-polynoom is, is het resultaat ook het 0-polynoom is. Anders moet elke term van het tweede polynoom vermenigvuldigd worden met het eerste element (de constante term) van het eerste polynoom. De rest van het eerste polynoom wordt met het tweede polynoom vermenigvuldigd, waarna alle exponenten één verhoogd worden. Dat laatste kan door een 0 op kop van het resultaat te zetten. De twee resulterende polynomen worden opgeteld met `pPlus`. Het resultaat hoeft niet meer vereenvoudigd te worden, want dat doet `pPlus` al:

```
pMaal [] ys = []
pMaal (x:xs) ys = pPlus (map (x*) ys)
                  (0.0 : pMaal xs ys)
```

5.1 De data-declaratie wordt uitgebreid met vier nieuwe constructoren:

```
data Expr = .....
          | Sin Expr
          | Cos Expr
          | Exp Expr
          | Log Expr
```

De definitie van `afg` wordt daarom ook uitgebreid met vier nieuwe patronen voor deze vier nieuwe expressies:

```
afg (Sin f) dx = Cos f *: afg f dx
afg (Cos f) dx = Con 0.0 -: Sin f *: afg f dx
afg (Exp f) dx = Exp f *: afg f dx
afg (Log f) dx = afg f dx ./: f
```

5.2 De functie `norep` moet recursief worden toegepast op eventuele deelstatements. De `Repeat` kan worden verwijderd door hem te vervangen door `While`. Van het predicaat moet daarbij de `Not` genomen worden, omdat ‘while’ doorgaat, maar ‘repeat’ juist stopt als het predicaat ‘true’ is. Bovendien wordt in de functie rekening gehouden met het feit dat het statement achter ‘repeat’ minstens eenmaal moet worden uitgevoerd.

```

norep :: Stat -> Stat
norep (Assign v e) = Assign v e
norep (If p s1 s2) = If p (norep s1) (norep s2)
norep (While p s) = While p (norep s)
norep (Repeat s p) = Compound [ norep s
                                , While (Not p) (norep s) ]
norep (Compound ss) = Compound (map norep ss)

```

5.3 A symbol equal to a satisfies equality to a :

```
symbol a = satisfy (==a)
```

5.4 As $\langle| \rangle$ is a lifted version of $++$, it is more efficiently evaluated right associative.

5.5 Without the `just` transformer, also partial parses are reported in the successes list

```

? nesting "()()()"
[[[],2), ("()",2), ("()()()",1), ("()()()()",0)]
? nesting "()"
[(")",1), ("()()",0)]

```

5.6 The function `just` can be written as a list comprehension:

```

just p xs = [ ([],v)
              | (ys,v) <- p xs
              , null ys
              ]

```

5.7 The operator $\langle * \rangle$ associates to the right, so $a \langle * \rangle b \langle * \rangle c \langle * \rangle d$ really means $a \langle * \rangle (b \langle * \rangle (c \langle * \rangle d))$, which explains the structure of the result.

The parser `epsilon` yields the empty tuple `()` as parse tree. The function `const Nil` is applied to this result, thus effectively discarding the empty tuple and substituting `Nil` for it. Instead of `epsilon <@ const Nil` we can also write `succeed Nil`.

5.8 Without parentheses, we obtain `open * (parens <* (close<*>parens))`, and we would only keep the result of the first recursive use of the `parens` parser.

5.9 The functions `parens` and `nesting` can be written as partial parametrizations of `foldparens`, by supplying the functions to be used for the first and second alternative:

```

parens = foldparens Bin Nil
nesting = foldparens (max.(1+)) 0

```

5.10 We define $\langle : * \rangle$ as an abbreviation of postprocessing $\langle * \rangle$ with the `list` function:

```
p <:*> q = p <*> q <@ list
```

Then we can define

```
many p = p <:*> many p <|> succeed []
```

5.11 The empty alternative is presented last, because the $\langle| \rangle$ combinator uses list concatenation for concatenating lists of successes. This also holds for the recursive calls; thus the ‘greedy’ parsing of all three `a`’s is presented first, then two `a`’s with singleton rest string, then one `a`, and finally the empty result with untouched rest string.

5.12 The `many1` combinator can be defined using the `many` combinator:

```

many1 :: Parser s a -> Parser s [a]
many1 p = p <*> many p <@ list

```

5.13 The functions are defined as follows:

```

sequence :: [Parser s a] -> Parser s [a]
sequence = foldr (<:*>) (succeed [])
choice :: [Parser s a] -> Parser s a
choice = foldr (<|>) fail

```

5.14 The function is defined as follows:

```

token :: Eq [s] => [s] -> Parser s [s]
token = sequence . map symbol

```

5.15 This was `chainl`:

```

chainl :: Parser s a -> Parser s (a->a->a) -> Parser s a
chainl p s = p <*> many (s <*> p)
             <@ uncurry (foldl (flip ap2))

```

To obtain `chainr`, change `foldl` into `foldr`, swap `flip` and `fold`, change `ap2` into `ap1` and reorder the distribution of `many` over the $\langle * \rangle$ operators:

```
chainr :: Parser s a -> Parser s (a->a->a) -> Parser s a
```

```
chainr p s = many (p <*> s) <*> p
             <@ uncurry (flip (foldr ap1))
```

The auxiliary functions used are:

```
ap2 (op,y) = ('op' y)
ap1 (x,op) = (x 'op')
```

5.16 Easiest is to do the case analysis explicitly:

```
integer :: Parser Char Int
integer = option (symbol '-') <*> natural <@ f
  where f ([],n) = n
        f (_ ,n) = -n
```

But nicest is to use the <?@ operator, yielding the identity or negation function in absence or presence of the minus sign, which is finally applied to the natural number:

```
integer :: Parser Char Int
integer = (option (symbol '-') <?@ (id,const negate))
          <*> natural
          <@ ap
  where ap (f,x) = f x
```

5.17 A floating point number is a fixed point number with an optional exponent part:

```
float :: Parser Char Float
float = fixed
      <*>
      (option (symbol 'E' *> integer) <?@ (0,id) )
      <@ f
  where f (m,e) = m * power e
        power e | e<0      = 1.0 / power (-e)
                | otherwise = fromInteger(10^e)
```