Aanvullende toets Grammatica's en ontleden 29 maart 2005

1. In deze opgave moet je het antwoord ook bewijzen. Een simpel 'ja' of 'nee' is niet genoeg!

   (a) Is $\{a^n b^n c^k \mid n, k \in \mathbb{N}\}$ een contextvrije taal? (Bewijs het antwoord!)

   (b) Is $\{a^n b^n c^n \mid n \in \mathbb{N}\}$ een contextvrije taal? (Bewijs!)

   (c) Als L en M contextvrije talen zijn, is $L \cap M$ dan ook een contextvrije taal? (Bewijs!)

   (d) Als L en M reguliere talen zijn, is $L \cap M$ dan ook een reguliere taal? (Bewijs!)

   Hint: je kunt hierbij soms de "pumping lemma's" gebruiken:

   - Als

voor alle	n	$: n \in \mathbb{N}$	:
er bestaan	u, z, y	$: uzy \in L$ en $\lvert z\rvert \geq n$	:
voor alle	v, w, x	$: z = vwx$ en $\lvert w\rvert > 0$	:
er bestaat een	$i \in \mathbb{N}$	$: uvw^i xy \notin L$	

     dan is L geen reguliere taal

   - Als

voor alle	c, d	$: c, d \in \mathbb{N}$	:
er bestaat een	z	$: z \in L$ en $\lvert z\rvert \geq c$	:
voor alle	u, v, w, x, y	$: z = uvwxy$ en $\lvert vx\rvert > 0$ en $\lvert vwx\rvert \leq d$	:
er bestaat een	$i \in \mathbb{N}$	$: uv^i wx^i y \notin L$	

     dan is L geen contextvrije taal

2. (a) Gegeven zijn de basis-parsers `symbol`, `epsilon`, `succeed` en `fail`, en de parser-combinators `<*>`, `<|>` en `<$>`. Schrijf met behulp hiervan een nieuwe parser-combinator `optioneel` met het type

   ```
   optioneel :: Parser a b -> Parser a (MayBe b)
   ```

   met behulp waarvan optionele constructies in een taal kunnen worden ontleed. Hierbij is `MayBe` het standaard Haskell type

   ```
   data MayBe a  =  Yes a | No
   ```

   (b) Wat zijn de meest algemene types van `f`, `p`, `q` en `s`, zodat

   ```
   (f <$> p <*> q) s
   ```

   kan worden uitgerekend? In welke volgorde wordt deze expressie geëvalueerd, en wat zijn de types van de twee tussenresultaten en het eindresultaat die daarbij onstaan?

   (c) We bekijken de taal van expressies die een type aanduiden in een Haskell-achtige taal, die echter simpeler is dan het echte Haskell. In deze taal zijn er de volgende soort types:
   - twee primitieve types, namelijk `Int` en `Bool`
   - lijst-types, zoals `[Int]` en `[[Bool]]`
   - het nul-tupel `()`, tweetupels zoals `(Int,[Bool])` en meertupels
   - functietypes zoals `Int->Bool`
   - en voor groepering mag elk type altijd tussen haakjes gezet worden

   Er zijn in deze taal (anders dan in de echte Haskell) dus *geen* type-variabelen, algebraische **data**-typen, classes enz. Merk ook op dat er geen één-tupels bestaan.

   We gaan een parser schrijven voor expressies in deze taal. Laat je niet in verwarring brengen door het feit dat de parser in Haskell geschreven is, en de taal ook Haskell-achtig is: je kunt voor elke taal parsers schrijven, dus ook voor dit taaltje.

   Definieer eerst een datatype `TypeExpr`, die ontleedbomen van type-expressies in dit taaltje beschrijft.

   Definieer daarna de eigenlijke parser

   ```
   parseTypeExpr :: Parser Char TypeExpr
   ```

   Je mag daarbij alle parser-combinators uit de library gebruiken. Zorg ervoor dat de operator `->` uit het taaltje net zo associeert als in de "echte" Haskell.

1

3. Gegeven is het type

```
data Expr = Con Int
          | Var Char
          | Plus Expr Expr
          | Maal Expr Expr
          | Bind Char Expr Expr
```

De laatste constructie is bedoeld voor het representeren van expressies zoals "let $x = 2 + 3$ in $2 \times x$".

(a) Definieer het bijbehorende type `ExprAlgebra`, en geef het *type* van de functie `foldExpr` (je hoeft de functie zelf niet te definiëren).

(b) De functie

```
check :: Expr -> Bool
```

controleert of een expressie uitgerekend kan worden. Dat kan als elke variabele die er in voorkomt gebonden is. Schrijf deze functie in de vorm

```
check = g . foldExpr a
   where g ...
         a ...
```

(c) Schrijf de functie

```
eval :: Expr -> Int
```

die de expressie uitrekent. De functie mag er zonder controle van uitgaan dat de expressie uitgerekend kan worden. Ook deze functie moet de vorm hebben van een aanroep van `foldExpr` met eventueel een nabewerking.

4. (a) Bij LL-ontleden spelen zogeheten *lookahead sets* een belangrijke rol. Waar worden die voor gebruikt?

(b) Hoe kun je de lookahead-set van een productieregel bepalen als van elke nonterminal de *empty*, *firsts* en *follow* bekend zijn?

(c) Aan welke eigenschap moeten de lookahead-sets voldoen wil er sprake zijn van $LL(1)$-ontleden? Wat is daarvan het voordeel?

(d) Bij $LL(1)$-ontleden wordt gebruik gemaakt van een *stack* en een *inputstring*. Beschrijf in woorden wat de ontleder doet als

- er een terminal symbool bovenop de stack staat
- er een nonterminal symbool bovenop de stack staat

(e) Ook de LR-ontleder gebruikt een stack en een inputstring. Bij elke stap is er de keus uit een *shift* of een *reduce*. Wat houdt een *shift* in? En wat houdt een *reduce* in? (Je hoeft niet uit te leggen hoe de keuze wordt gemaakt).

(f) Op welk moment stopt de LL-ontleder? En op welk moment stopt de LR-ontleder?