Tentamen Functioneel Programmeren—11 november 2011

De nagekeken tentamens zijn in te zien bij de docent, J.H. Jongejan, Bernoulliborg kamer 366.

Opmerkingen:

- Schrijf **netjes** en duidelijk, met zwarte of blauwe pen.
- Zet op het eerste blad alle gegevens als naam, etc., en het totaal aantal ingeleverde bladen, en nummer de ingeleverde bladen.
- Lees de opgaven eerst goed door.
- Houd je programma's kort en helder, mede door verstandig gebruik te maken van standaardfuncties uit het boek (in het bijzonder uit het gedeelte over lijsten) en/of door listcomprehension.
- Motiveer je antwoorden.

1. (10 punten)

   Geef het type en de implementatie van `zipWith`.

2. (15 punten)

   Bewijs met volledige inductie over alle eindige lijsten `xs` dat

   ```
   map f (xs++ys) = (map f xs) ++ (map f ys)
   ```

3. (20 punten)

   Even opfrissen:

   ```
   curry :: ((a,b) -> c) -> (a -> b -> c)
   uncurry :: (a -> b -> c) -> ((a,b) -> c)
   id :: a -> a
   id x = x    -- de identiteitsfunctie
   ```

   a) Laat zien wat het type is van `uncurry id`.
   b) En wat is het type van `curry id`?

4. (20 punten)

Een **deque** is een queue waar je zowel vooraan als achteraan elementen kunt toevoegen en verwijderen. Gebruik de volgende data definitie:

```
data Deque a = Dq [a]
```

We kunnen een abstract data type maken middels een `module Deque`. Geef typedeclaraties en implementeer de volgende functies op een `Deque`:

```
module Deque (
    emptyDq,
    isEmptyDq,
    addFront,
    addEnd,
    remFront,
    remEnd
  ) where ...
```

5. (20 punten)

Gegeven is dat de invoer voldoet aan de grammaticaregels:

```
   E  = '[' Num NumTl ']'
   Num = '0' | '1' | ... | '9'
   NumTl = | Num NumTl
-- 1e alternatief is leeg!
```

a) Geef een data definitie Exp om E's te representeren.
b) Bouw een parser `pE :: Parse Char Exp`, die de input omzet naar een `Exp`. Je mag hierbij gebruik maken van:

```
type Parse a b = [a] -> [(b,[a])]

succeed :: b -> Parse a b
spot    :: (a -> Bool) -> Parse a a
token t  = spot (==t)
alt     :: Parse a b -> Parse a b -> Parse a b
(>*>)   :: Parse a b -> Parse a c -> Parse a (b,c)
build   :: Parse a b -> (b -> c) -> Parse a c
```