Funtional and Logic Programming
Exercise Set 4

Tom Schrijvers Steven Keuchel
{tom.schrijvers,steven.keuchel}@ugent.be

Deadline: Tuesday, October 23rd, 2012, 11:59pm

Parsing

1. Write a function

$$satisfy :: (a \to Bool) \to Parser\ a \to Parser\ a$$

which creates a new parser that only succeeds if the parsed value satisfies the predicate. Implement *lower* and *char* in terms of *satisfy*.

2. Implement a parser

$$token :: String \to Parser\ String$$

that recognizes a complete string, e.g. a keyword.

3. Write a function

$$mapParser :: (a \to b) \to Parser\ a \to Parser\ b$$

that given a function and a parser produces a new parser for the same language with a transformed result value. Implement *digit* in terms of *mapParser*.

4. Implement

$$parens :: Parser\ a \to Parser\ a$$

that recognizes an expression that is enclosed in parentheses. For example

```
Prelude> runP (parens number) "(12)"
Just 12
```

5. Implement

$$tuple :: Parser\ a \to Parser\ b \to Parser\ (a, b)$$

that recognizes tuple expressions. For example

```
Prelude> runP (tuple number number) "(12,42)"
Just (12,42)
```

6.

(a) Implement the combinator

$$many :: Parser\ a \rightarrow Parser\ [a]$$

that recognizes zero or more elements of the given parser and the combinator

$$some :: Parser\ a \rightarrow Parser\ [a]$$

that recognizes one or more elements.

(b) Writer a parser

$$list :: Parser\ a \rightarrow Parser\ [a]$$

that recognizes list expressions. For example

Prelude> *runP* (*list number*) "[1,2,3]"
Just [1, 2, 3]
Prelude> *runP* (*list number*) "[]"
Just [ ]

Note: You don't need to recognize list expressions of the form $1 : 2 : 3 : [\,]$.

7. Writer a parser for the grammar

$$E ::= N \mid E + E$$

where the non-terminal N denotes number literals and produce a value of the following datatype:

data *Exp* = *N Int* | *Exp* :+*Exp*
  deriving *Show*

8. Writer a parser for the grammar

$$E ::= L \mid E + E \mid E * E \mid (E)$$

where the non-terminal L denotes number literals and produce a value of the following datatype:

data *Expr* = *L Int* | *Expr* :+: *Expr* | *Expr* :∗: *Expr*
  deriving *Show*

Make sure that the precedence of multiplication over addition is handled correctly, e.g. "1+2*3" yields the same parse result as "1+(2*3)".

Lazy evaluation

9. In the course you have seen the *unfoldr* function that builds (possibly infinite) lists.

(a) Implement a similar function

$$unfoldStream :: (s \rightarrow (a, s)) \rightarrow s \rightarrow [a]$$

that always produces infinite lists.

(b) Implement the following functions in terms of *unfoldStream*:

- $repeat\ a = a : a : a : ...$
- $nats = 0 : 1 : 2 : ...$
- $iterate\ f\ \ a = a : f\ \ a : f\ \ (f\ \ a) : ...$
- $primes = 2 : 3 : 5 : ...$
  Use the sieve of Eratosthenes.

10.

(a) Implement the function

$$partialSums :: Num\ a \Rightarrow [a] \rightarrow [a]$$

that creates for a sequence of numbers the sequence of non-empty partial sums. For example

$Prelude>$ $partialSums\ [1\,..]$
$[1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66, 78, 91, 105, 120, ...$

Use the same style that you have seen for the list of Fibonacci numbers:

$fibs :: [Int]$
$fibs = 0 : 1 : zipWith\ (+)\ fibs\ (tail\ fibs)$

(b) Dropping every second element from the positive natural numbers gives us the list of uneven numbers

$$1, 3, 5, 7, 9, 11, 13, 15, ...$$

It is well known, that the partial sums of the uneven numbers are the squared numbers

$$1, 4, 9, 16, 25, 36, 49, 64, ...$$

In a similar fashion we can obtain the cubes. Start with the positive natural numbers and drop every **third** element:

$$1, 2, 4, 5, 7, 8, 10, 11, 13, 14, 16, ...$$

then take partial sums:

$$1, 3, 7, 12, 19, 27, 37, 48, 61, 75, 91, ...$$

drop every **second** element:

$$1, 7, 19, 37, 61, 91, ...$$

and form the partial sums again:

$$1, 8, 27, 64, 125, 216, ...$$

The above procedure of alternating between dropping elements and forming the partial sums can be used to obtain the streams of powers of n for any $n \geqslant 1$. This is known as Moessner's theorem. Implement this procedure in a function

$$moessner :: Int \rightarrow [Int]$$

that calculates the stream of powers of n.