Techniques:

1: Literate Programming
2: Parsing
3: Random Formula Generation

Jan van Eijck

Specification and Testing, Week 3, 2012

Literate Programming

Literate programming document: homepages.cwi.nl/~jve/courses/12/testing2012/lectures/LP.pdf

Bash (linux) script for converting from tex to lhs. Save the following lines in a file called `lhs` and make the file executable.

```
# produce literate haskell code from latex source code
#
cat < $1'.tex' | sed -f /ufs/jve/bin/lhsf.sed > $1'.lhs'
echo "Literate Haskell code written to file $1.lhs"
```

Modify the path as appropriate.

The script calls an auxiliary file that contains the following lines:

```
s/\%\#/\#/
s/\\bc\\begin[{]verbatim[}]/\\begin\{code\}/g
s/\\end[{]verbatim[}]\\ec/\\end\{code\}/g
```

Save these lines in a file called **`lhsf.sed`** and let the path in **`lhs`** point to it.

Module Declaration

```
module Techniques where

import Data.List
import Data.Char
import System.Random
import Week2
```

Lexical Scanning

Tokens:

```
data Token
      = TokenNeg
      | TokenCnj
      | TokenDsj
      | TokenImpl
      | TokenEquiv
      | TokenInt Int
      | TokenOP
      | TokenCP
 deriving (Show,Eq)
```

The lexer converts a string to a list of tokens.

```
lexer :: String -> [Token]
lexer [] = []
lexer (c:cs) | isSpace c = lexer cs
             | isDigit c = lexNum (c:cs)
lexer ('(':cs) = TokenOP : lexer cs
lexer (')':cs) = TokenCP : lexer cs
lexer ('*':cs) = TokenCnj : lexer cs
lexer ('+':cs) = TokenDsj : lexer cs
lexer ('-':cs) = TokenNeg : lexer cs
lexer ('=':'=':'>':cs) = TokenImpl : lexer cs
lexer ('<':'=':'>':cs) = TokenEquiv : lexer cs
lexer (x:_) = error ("unknown token: " ++ [x])
```

Read an integer and convert it into a structured token of the form `TokenInt i`.

```
lexNum cs = TokenInt (read num) : lexer rest
     where (num,rest) = span isDigit cs
```

```
*Lexer> lexer "*(2 3 -4 +("
[TokenCnj,TokenOP,TokenInt 2,TokenInt 3,TokenNeg,
 TokenInt 4,TokenDsj,TokenOP]
```

Parsing

A parser for token type `a` that constructs a datatype `b` has the following type:

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

The parser constructs a list of tuples `(b,[a])` from an initial segment of a token string `[a]`. The remainder list in the second element of the result is the list of tokens that were not used in the construction of the datatype.

If the output list is empty, the parse has not succeeded. If the output list more than one element, the token list was ambiguous.

Success

The parser that succeeds immediately, while consuming no input:

```
succeed :: b -> Parser a b
succeed x xs = [(x,xs)]
```

```
parseForm :: Parser Token Form
parseForm (TokenInt x: tokens) = [(Prop x,tokens)]
parseForm (TokenNeg : tokens) =
  [ (Neg f, rest) | (f,rest) <- parseForm tokens ]
parseForm (TokenCnj : TokenOP : tokens) =
  [ (Cnj fs, rest) | (fs,rest) <- parseForms tokens ]
parseForm (TokenDsj : TokenOP : tokens) =
  [ (Dsj fs, rest) | (fs,rest) <- parseForms tokens ]
parseForm (TokenOP : tokens) =
    [ (Impl f1 f2, rest) | (f1,ys) <- parseForm tokens,
                           (f2,rest) <- parseImpl ys ]
     ++
    [ (Equiv f1 f2, rest) | (f1,ys) <- parseForm tokens,
                            (f2,rest) <- parseEquiv ys ]
parseForm tokens = []
```

Parsing a list of formulas

Success if a closing parenthesis is encountered.

```
parseForms :: Parser Token [Form]
parseForms (TokenCP : tokens) = succeed [] tokens
parseForms tokens =
   [(f:fs, rest) | (f,ys) <- parseForm tokens,
                   (fs,rest) <- parseForms ys ]
```

Parsing implications and equivalences uses separate functions, for these constructions have infix operators.

```
parseImpl :: Parser Token Form
parseImpl (TokenImpl : tokens) =
  [ (f,ys) | (f,y:ys) <- parseForm tokens,
              y == TokenCP ]
parseImpl tokens = []

parseEquiv :: Parser Token Form
parseEquiv (TokenEquiv : tokens) =
  [ (f,ys) | (f,y:ys) <- parseForm tokens,
              y == TokenCP ]
parseEquiv tokens = []
```

The Parse Function

```
parse :: String -> [Form]
parse s = [ f | (f,_) <- parseForm (lexer s) ]
```

```
*Parser> parse "*(1 +(2 -3))"
[*(1 +(2 -3))]
*Parser> parse "*(1 +(2 -3)"
[]
*Parser> parse "*(1 +(2 -3))))"
[*(1 +(2 -3))]
*Parser> parseForm (lexer "*(1 +(2 -3))))")
[(*(1 +(2 -3)),[TokenCP,TokenCP])]
```

Random Formula Generation

Getting a random integer:

```
getRandomInt :: Int -> IO Int
getRandomInt n = getStdRandom (randomR (0,n))
```

Note the output type `IO Int`.

```
*Techniques> :t getStdRandom
getStdRandom :: (StdGen -> (a, StdGen)) -> IO a
*Techniques> :t getRandomInt
getRandomInt :: Int -> IO Int
*Techniques> getRandomInt 5
5
*Techniques> getRandomInt 5
4
*Techniques> getRandomInt 5
5
*Techniques> getRandomInt 5
4
*Techniques> getRandomInt 5
0
```

```
getRandomF :: IO Form
getRandomF = do d <- getRandomInt 4
                getRandomForm d

getRandomForm :: Int -> IO Form
getRandomForm 0 = do m <- getRandomInt 20
                     return (Prop (m+1))

getRandomForm d = do n <- getRandomInt 3
                     case n of
                       0 -> do m <- getRandomInt 20
                               return (Prop (m+1))
                       1 -> do f <- getRandomForm (d-1)
                               return (Neg f)
                       2 -> do m  <- getRandomInt 5
                               fs <- getRandomForms (d-1) m
                               return (Cnj fs)
                       3 -> do m  <- getRandomInt 5
                               fs <- getRandomForms (d-1) m
                               return (Dsj fs)
```

```
getRandomFs :: Int ->  IO [Form]
getRandomFs n = do d <- getRandomInt 3
                   getRandomForms d n

getRandomForms :: Int -> Int -> IO [Form]
getRandomForms _ 0 = return []
getRandomForms d n = do
                     f <- getRandomForm d
                     fs <- getRandomForms d (n-1)
                     return (f:fs)
```

A Test Function for Formulas

```
test :: Int -> (Form -> Bool) -> [Form] -> IO ()
test n _ [] = print (show n ++ " tests passed")
test n p (f:fs) =
  if p f
  then do print ("pass on:" ++ show f)
          test n p fs
  else error ("failed test on:" ++ show f)
```

Running Tests

```
testForms :: Int -> (Form -> Bool) -> IO ()
testForms n prop = do
  fs <- getRandomFs n
  test n prop fs
```

Testing the parser with random formulas:

```
testParser :: IO ()
testParser = testForms 100
   (\ f -> let [g] = parse (show f) in
               show f == show g)
```