Expr.hs

1: {- Author: Richard Eisenberg, edited by <your name here>
2:    File: Expr.hs
3: 
4:    Defines a simple expression language evaluator.
5: -}
6: 
7: module Main where
8: 
9: -- The import statements in this file include import lists, which state
10: -- exactly what is imported. This can be nice documentation, so that
11: -- readers know what comes from where.
12: -- These imports you know how to deal with.
13: import Data.Char ( isSpace, isDigit )
14: import Text.Read ( readMaybe )
15: import Control.Exception ( SomeException(..), evaluate, catch )
16: import Control.Monad     ( when )
17: import System.Exit       ( exitSuccess )
18: 
19: -- The AST type for parsed expression trees
20: data Expr =
22:   deriving (Eq, Show)
23: 
24: -- Possible tokens
25: data Token =
26:   PlusT | MinusT | TimesT | DivideT | NumT Integer
27:   deriving (Eq, Show)
28: 
29: -- Read an input string into a list of tokens.
30: lexTokens :: String -> [Token]
31: lexTokens input = lexNoPrefix (findToken input)
32: 
33: -- Drop any non-lexed prefix of the input. This language
34: -- is so simple that we can just use dropWhile.
35: findToken :: String -> String
36: findToken = dropWhile isSpace
37: 
38: -- Lex an input string, assuming that the first thing
39: -- in the string (if anything) is a token (as opposed to
40: -- whitespace).
41: lexNoPrefix :: String -> [Token]
42: lexNoPrefix input = lexNoPrefix (findToken input)
43: 
44: -- Given the first character and the rest of the input string,
45: -- lex one token, returning the remainder of the input string.
46: lex1 :: Char -> String -> (Token, String)
47: lex1 ('+':cs) = (PlusT, cs)
48: lex1 ('-':cs) = (MinusT, cs)
49: lex1 ('*':cs) = (TimesT, cs)
50: lex1 ('/':cs) = (DivideT, cs)
51: lex1 c = Nothing
52: 
53: -- lex a number
54: lex1 c cs = (NumT c, rest)
55: | isDigit c
56: | (more_digs, rest) <- span isDigit cs
57: | Just n <- readMaybe (c:more_digs)
58: | (NumT n, rest) = lex1 c cs
59: 
60: -- lex the operators
61: lex1 '(':cs = (PlusT, cs)
62: lex1 '(':cs = (MinusT, cs)
63: lex1 '):cs = (TimesT, cs)
64: lex1 '):cs = (DivideT, cs)
Expr.hs

73: lex1 '/\ cs = (DivideT, cs)
74:
75: -- Otherwise, we have a lexical error
76: lex1 c cs = error ("No lex: " ++ (c:cs))
77:
78: -- Parse one expression from a list of tokens, also returning
79: -- the remaining, unparsed tokens.
80: parse1 :: [Token] -> (Expr, [Token])
81: parse1 = error "not implemented yet"
82:
83: -- Parse a list of tokens into an expression. Errors if there are
84: -- too few or too many tokens.
85: parse :: [Token] -> Expr
86: parse = error "not implemented yet"
87:
88: -- Evaluate an expression to a number.
89: eval :: Expr -> Integer
90: eval = error "not implemented yet"
91:
92: -- Evaluate a string into a number.
93: evalString :: String -> Integer
94: evalString str = eval (parse (lexTokens str))
95:
96: -- A read-eval-print loop (REPL)
97: -- (You are not expected to understand this.)
98: main :: IO ()
99: main = do
100:
101: -- primary user interaction commands
102: putStrLn ""
103: putStrLn "Enter a prefix expression:
"
104: expr_string <- getLine
105:
106: -- allow users to quit
107: when (expr_string == "quit")
108: exitSuccess
109:
110: -- This code runs evalString in a way that, if evalString calls 'error',
111: -- the program will not immediately abort. The Haskell features used here
112: -- are beyond the scope of CS245. The curious may enjoy looking these
113: -- functions up online.
114: catch (do value <- evaluate (evalString expr_string)
115:          print value)
116:   (\ (SomeException e) -> print e)
117:
118: -- And do it again.
119: main