Expr.hs

1: {- Author: Richard Eisenberg
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: 
13: -- These imports you know how to deal with.
14: import Data.Char ( isSpace, isDigit )
15: import Text.Read ( readMaybe )
16: 
17: -- These are more advanced, used only in 'main'.
18: import Control.Exception ( SomeException(..), evaluate, catch )
19: import Control.Monad     ( when )
20: import System.Exit       ( exitSuccess )
21: 
22: -- The AST type for parsed expression trees
23: data Expr
24:   = Plus Expr Expr
25:   | Minus Expr Expr
26:   | Times Expr Expr
27:   | Divide Expr Expr
28:   | Num Integer
29:   deriving (Eq, Show)
30: 
31: -- Possible tokens
32: data Token
33:   = PlusT
34:   | MinusT
35:   | TimesT
36:   | DivideT
37:   | NumT Integer
38:   deriving (Eq, Show)
39: 
40: -- Read an input string into a list of tokens.
41: lexTokens :: String -> [Token]
42: lexTokens input = lexNoPrefix (findToken input)
43: 
44: -- Drop any non-lexed prefix of the input. This language
45: -- is so simple that we can just use dropWhile.
46: findToken :: String -> String
47: findToken = dropWhile isSpace
48: 
49: -- Lex an input string, assuming that the first thing
50: -- in the string (if anything) is a token (as opposed to
51: -- whitespace).
52: lexNoPrefix :: String -> [Token]
53: lexNoPrefix []     = []
54: lexNoPrefix (c:cs) = token : lexTokens rest
55:   where
56:     (token, rest) = lex1 c cs
57: 
58: -- Given the first character and the rest of the input string,
59: -- lex one token, returning the remainder of the input string.
60: lex1 :: Char -> String -> (Token, String)
61: 
62: -- lex a number
63: lex1 c cs
64:   | isDigit c
65:   , (more_digs, rest) <- span isDigit cs
66:   , Just n <- readMaybe (c:more_digs)
67:   = (NumT n, rest)
68: 
69: -- lex the operators
70: lex1 '+'  cs = (PlusT, cs)
71: lex1 '-'  cs = (MinusT, cs)
72: lex1 '*'  cs = (TimesT, cs)
73: lex1 '/'  cs = (DivideT, cs)
74: lex1 '^'  cs = (NumT, cs)
75: lex1 ' '  cs = (NumT, 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 (NumT n : rest)
82:   = (Num n, rest)
83: parse1 (op : rest1)
84:   | (arg1, rest2) <- parse1 rest1
85:   , (arg2, rest3) <- parse1 rest2
86:   = (mkOp op arg1 arg2, rest3)
87: parse1 _ = error "Unexpected end of tokens"
88:
89: -- Build an operator expression, given the head token.
90: mkOp :: Token -> Expr -> Expr -> Expr
91: mkOp PlusT   arg1 arg2 = Plus arg1 arg2
92: mkOp MinusT  arg1 arg2 = Minus arg1 arg2
93: mkOp TimesT  arg1 arg2 = Times arg1 arg2
94: mkOp DivideT arg1 arg2 = Divide arg1 arg2
95: mkOp (NumT n) _   _    = error ("Unexpected number in mkOp: " ++ show n)
96:
97: -- Parse a list of tokens into an expression. Errors if there are
98: -- too few or too many tokens.
99: parse :: [Token] -> Expr
100: parse tokens
101:   | (expr, []) <- parse1 tokens
102:   = expr
103:   | otherwise
104:   = error "Too many tokens"
105:
106: -- Evaluate an expression to a number.
107: eval :: Expr -> Integer
108: eval (Plus e1 e2)   = eval e1 + eval e2
109: eval (Minus e1 e2)  = eval e1 - eval e2
110: eval (Times e1 e2)  = eval e1 * eval e2
111: eval (Divide e1 e2) = eval e1 'div' eval e2
112: eval (Num n)        = n
113:
114: -- Evaluate a string into a number.
115: evalString :: String -> Integer
116: evalString str = eval (parse (lexTokens str))
117:
118: -- A read-eval-print loop (REPL)
119: -- (You are not expected to understand this.)
120: main :: IO ()
121: main = do
122:
123:   -- primary user interaction commands
124:   putStrLn ""    
125:   putStrLn "Enter a prefix expression:"
126:   expr_string <- getLine
127:   when (expr_string == "quit")
128:     exitSuccess
129:
130:   -- catch (do value <- evaluate (evalString expr_string)
131:     print value)
132:   exitSuccess
133:
134:   -- This code runs evalString in a way that, if evalString calls 'error',
135:   -- the program will not immediately abort. The Haskell features used here
136:   -- are beyond the scope of CS245. The curious may enjoy looking these
137:   -- functions up online.
138:   catch (do value <- evaluate (evalString expr_string)
139:                print value)
140:     (\ (SomeException e) -> print e)
141:
142:   -- And do it again.