Expr.hs

1: {-- Author: Richard Eisenberg
2:    File: Expr.hs
3:    Defines a simple expression language evaluator.
4: -}
5:
6: module Main where
7:
8: -- The import statements in this file include import lists, which state
9: -- exactly what is imported. This can be nice documentation, so that
10: -- readers know what comes from where.
11: -- These imports you know how to deal with.
12: import Data.Char  ( isSpace, isDigit )
13: import Text.Read   ( readMaybe )
14: import Control.Exception ( SomeException(..), evaluate, catch )
15: import Control.Monad     ( when )
16: import System.Exit       ( exitSuccess )
17: -- These are more advanced, used only in 'main'.
18: data Expr      = Plus Expr Expr
19:               | Minus Expr Expr
20:               | Times Expr Expr
21:               | Divide Expr Expr
22:               | Num Integer
23:               deriving (Eq, Show)
24:
25: data Token    = PlusT
26:               | MinusT
27:               | TimesT
28:               | DivideT
29:               | NumT Integer
30:               deriving (Eq, Show)
31: -- Read an input string into a list of tokens.
32: lexTokens     :: String -> [Token]
33: lexTokens     = lexNoPrefix . findToken .
34: -- Drop any non-lexed prefix of the input. This language
35: -- is so simple that we can just use dropWhile.
36: findToken      :: String -> String
37: findToken      = dropWhile isSpace
38: -- Lex an input string, assuming that the first thing
39: -- in the string (if anything) is a token (as opposed to
40: lexNoPrefix    :: String -> [Token]
41: lexNoPrefix    = lex1 .
42: -- Given the first character and the rest of the input string,
43: lex1           :: Char -> String -> (Token, String)
44: lex1           = case c of
45:                   isDigit c
46:                   , (more_digs, rest) <- span isDigit cs
47:                   , Just n <- readMaybe (c:more_digs)
48:                   = (NumT n, rest)
49:                   _   = (token, rest) = lex1 c cs
50: -- Lex a number
51: lex1 c cs      = if isDigit c
52:                   , (more_digs, rest) <- span isDigit cs
53:                   , Just n <- readMaybe (c:more_digs)
54:                   = (NumT n, rest)
55: -- lex the operators
56: lex1 '*' cs    = (PlusT, cs)
57: lex1 '-' cs    = (MinusT, cs)
58: lex1 '*' cs    = (TimesT, cs)
59: lex1 '*' cs    = (TimesT, 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: -- Assume the head token is not a number.
91: mkOp :: Token -> Expr -> Expr -> Expr
92: mkOp PlusT   arg1 arg2 = Plus arg1 arg2
93: mkOp MinusT  arg1 arg2 = Minus arg1 arg2
94: mkOp TimesT  arg1 arg2 = Times arg1 arg2
95: mkOp DivideT arg1 arg2 = Divide arg1 arg2
96: mkOp (NumT n) _   _    = error ("Unexpected number in mkOp: " ++ show n)
97:
98: -- Parse a list of tokens into an expression. Errors if there are
99: -- too few or too many tokens.
100: parse :: [Token] -> Expr
101: parse tokens
102:   | (expr, []) <- parse1 tokens
103:   = expr
104:   | otherwise
105:   = error "Too many tokens"
106:
107: -- Evaluate an expression to a number.
108: eval :: Expr -> Integer
109: eval (Plus e1 e2)   = eval e1 + eval e2
110: eval (Minus e1 e2)  = eval e1 - eval e2
111: eval (Times e1 e2)  = eval e1 * eval e2
112: eval (Divide e1 e2) = eval e1 'div' eval e2
113: eval (Num n)        = n
114: evalString :: String -> Integer
115: evalString str = eval (parse (lexTokens str))
116: -- A read-eval-print loop (REPL)
117: main :: IO ()
118: main = do
119:   putStrLn "Enter a prefix expression:"
120:   expr_string <- getLine
121:   when (expr_string == "quit") exitSuccess
122:   catch (do value <- evaluate (evalString expr_string)
123:             print value)
124:         (
125:             (SomeException e) -> print e)
126: exitSuccess
127: main = do
128:   putStrLn ""