module Main where

-- The import statements in this file include import lists, which state
-- exactly what is imported. This can be nice documentation, so that
-- readers know what comes from where.

-- These imports you know how to deal with.
import Data.Char ( isSpace, isDigit )
import Text.Read ( readMaybe )

-- These are more advanced, used only in 'main'.
import Control.Exception ( SomeException(..), evaluate, catch )
import Control.Monad     ( when )
import System.Exit       ( exitSuccess )

-- The AST type for parsed expression trees
data Expr
  = Plus Expr Expr
  | Minus Expr Expr
  | Times Expr Expr
  | Divide Expr Expr
  | Num Integer
  deriving (Eq, Show)

-- Possible tokens
data Token
  = PlusT
  | MinusT
  | TimesT
  | DivideT
  | NumT Integer
  deriving (Eq, Show)

-- Read an input string into a list of tokens.
lexTokens :: String -> [Token]
lexTokens input = lexNoPrefix (findToken input)

-- Drop any non-lexed prefix of the input. This language
-- is so simple that we can just use dropWhile.
findToken :: String -> String
findToken = dropWhile isSpace

-- Lex an input string, assuming that the first thing
-- in the string (if anything) is a token (as opposed to
-- whitespace).
lexNoPrefix :: String -> [Token]
lexNoPrefix []     = []
lexNoPrefix (c:cs) = token : lexTokens rest
  where
    (token, rest) = lex1 c cs

-- Given the first character and the rest of the input string,
-- lex one token, returning the remainder of the input string.
lex1 :: Char -> String -> (Token, String)
lex1 c cs
  | isDigit c,
  | (more_digs, rest) <- span isDigit cs
  | Just n <- readMaybe (c:more_digs)
  | (NumT n, rest)
  = (NumT n, rest)

-- lex the operators
lex1 '}' cs = (MinusT, cs)
lex1 '*' cs = (TimesT, cs)
```haskell
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:   putStrLn "Enter a prefix expression:"
123:   expr_string <- getLine
124:   when (expr_string == "quit") exitSuccess
125:   catch (do value <- evaluate (evalString expr_string)
126:                  print value)
127:          (\ (SomeException e) -> print e)
128:   main
```