module LL where

data Token = NUMBER Int | PLUS | LPAREN | RPAREN | EOF
   deriving (Show, Eq)

data Exp = Lit Int
          | Plus Exp Exp
   deriving (Show, Eq)

-- LL(1) Parser that converts a token stream into an expression.
parse :: [Token] -> Maybe Exp
parse = parseT

-- Helper functions for 'parse'. These should really be in a "where" clause of 'parse', but this allows for easier debugging
parseT :: [Token] -> Maybe Exp
parseT all_ts@(NUMBER _:_) = case parseS all_ts of
   Just (exp, [EOF]) -> Just exp
   _                 -> Nothing
parseT all_ts@(LPAREN:_)   = case parseS all_ts of
   Just (exp, [EOF]) -> Just exp
   _                 -> Nothing
parseT _                   = Nothing

parseS :: [Token] -> Maybe (Exp, [Token])
parseS all_ts@(NUMBER _:_) = case parseE all_ts of
   Just (exp, ts) -> parseS' exp ts
parseS all_ts@(LPAREN:_)   = case parseE all_ts of
   Just (exp, ts) -> parseS' exp ts
parseS _                   = Nothing

parseS' :: Exp -> [Token] -> Maybe (Exp, [Token])
parseS' exp (PLUS:ts)     = case parseS ts of
   Just (exp', ts') -> Just (Plus exp exp', ts')
   _                -> Nothing
parseS' exp (EOF:_)      = Just (exp, ts)
parseS' _   _             = Nothing

parseE :: [Token] -> Maybe (Exp, [Token])
parseE (NUMBER n:ts) = Just (Lit n, ts)
parseE (LPAREN:ts)   = case parseS ts of
   Just (exp, RPAREN:ts') -> Just (exp, ts')
   _                      -> Nothing
parseE _             = Nothing

-- Test input for the source string: (1 + 2 + (3 + 4)) + 5

-- helpful for debugging
renderExp :: Exp -> String
renderExp (Lit n)       = show n
renderExp (Plus e1 e2@(Plus _ _)) = renderExp e1 ++ " + (" ++ renderExp e2 ++ ")"
renderExp (Plus e1 e2)  = renderExp e1 ++ " + " ++ renderExp e2