Chapter 13

This is just the preamble, a few imports:

```
{-# LANGUAGE LambdaCase #-}

import Control.Applicative (Alternative (..))
import Control.Monad (void)
import Data.Char (
  isAlpha,
  isAlphaNum,
  isDigit,
  isLower,
  isSpace,
  isUpper,
 )
import System.IO (hSetEcho, stdin)
```

Here we define what a parser is and implement its Functor, Applicative and Monad classes.

```
newtype Parser a = P {parse :: String -> Maybe (a, String)}

item :: Parser Char
item = P $ \case
    [] -> Nothing
    (c : cs) -> Just (c, cs)

instance Functor Parser where
    fmap :: (a -> b) -> Parser a -> Parser b
    fmap f p = P $ \s -> do
        (r, s') <- parse p s
        return (f r, s')

instance Applicative Parser where
    pure :: a -> Parser a
    pure x = P $ \s -> Just (x, s)
    (<*>) :: Parser (a -> b) -> Parser a -> Parser b
    pf <*> px = P $ \s -> do
        (f, s') <- parse pf s
        parse (f <$> px) s'

three :: Parser String
three = g <$> item <*> item <*> item
  where
    g x y z = x : y : [z]

instance Monad Parser where
    (>>=) :: Parser a -> (a -> Parser b) -> Parser b
    px >>= f = P $ \s -> do
        (x, s') <- parse px s
        parse (f x) s'
```

The same silly parser implemented two ways:

```
threeM :: Parser String
threeM = do
    c0 <- item
    c1 <- item
    c2 <- item
    return $ c0 : c1 : [c2]

threeM' :: Parser String
threeM' = sequence [item, item, item]
```

A parser is also an Alternative Functor:

```
instance Alternative Parser where
    empty :: Parser a
    empty = P $ const Nothing
    (<|>) :: Parser a -> Parser a -> Parser a
    pl <|> pr = P $ \s -> case parse pl s of
        Nothing -> parse pr s
        l -> l
```

We define some basic (atomic) parsers:

```
sat :: (Char -> Bool) -> Parser Char
sat p = do
    c <- item
    if p c then return c else empty

digit :: Parser Char
digit = sat isDigit

lower :: Parser Char
lower = sat isLower

upper :: Parser Char
upper = sat isUpper

letter :: Parser Char
letter = sat isAlpha

alphanum :: Parser Char
alphanum = sat isAlphaNum

char :: Char -> Parser Char
char c = sat (== c)

string :: String -> Parser String
string "" = return ""
string s@(x : xs) = do
    char x
    string xs
    return s
```

And some more advanced parsers:

```
ident :: Parser String
ident = (:) <$> lower <*> many alphanum

nat :: Parser Int
nat = read <$> some digit

space :: Parser ()
space = void $ many $ sat isSpace

int :: Parser Int
int = (\n -> -n) <$> (char '-' *> nat) <|> nat

token :: Parser a -> Parser a
token p = space *> p <* space

identifier :: Parser String
identifier = token ident

natural :: Parser Int
natural = token nat

integer :: Parser Int
integer = token int

symbol :: String -> Parser String
symbol s = token $ string s

nats :: Parser [Int]
{-
nats = do
    symbol "["
    n <- natural
    ns <- many $ (\_ n -> n) <$> symbol "," <*> natural
    symbol "]"
    return $ n : ns
-}
nats = symbol "[" *> ((:) <$> natural <*> many (symbol "," *> natural)) <* symbol "]"

zero :: Parser Int
zero = P $ \s -> Just (0, s)

one :: Parser Int
one = P $ \s -> Just (1, s)
```

And we define the parser for expressions now:

```
expr :: Parser Int
expr = (+) <$> term <*> (symbol "+" *> expr <|> zero)

term :: Parser Int
term = (*) <$> factor <*> (symbol "*" *> term <|> one)

factor :: Parser Int
factor = symbol "(" *> expr <* symbol ")" <|> natural
```

Here is some code for displaying the calculator:

```
box :: [String]
box =
    [ "+---------------+"
    , "|               |"
    , "+---+---+---+---+"
    , "| q | c | d | = |"
    , "+---+---+---+---+"
    , "| 1 | 2 | 3 | + |"
    , "+---+---+---+---+"
    , "| 4 | 5 | 6 | - |"
    , "+---+---+---+---+"
    , "| 7 | 8 | 9 | * |"
    , "+---+---+---+---+"
    , "| 0 | ( | ) | / |"
    , "+---+---+---+---+"
    ]

buttons :: String
buttons = standard ++ extra
  where
    standard = "qcd=123+456-789*0()/"
    extra = "QCD \ESC\BS\DEL\n"

cls :: IO ()
cls = putStr "\ESC[2J"

type Pos = (Int, Int)

writeat :: Pos -> String -> IO ()
writeat p xs = do
    goto p
    putStr xs

goto :: Pos -> IO ()
goto (x, y) = putStr $ "\ESC[" ++ show y ++ ";" ++ show x ++ "H"

getCh :: IO Char
getCh = do
    hSetEcho stdin False
    x <- getChar
    hSetEcho stdin True
    return x

showbox :: IO ()
showbox = sequence_ [writeat (1, y) b | (y, b) <- zip [1 ..] box]

display :: [Char] -> IO ()
display s = do
    writeat (3, 2) $ replicate 13 ' '
    writeat (3, 2) $ reverse $ take 13 $ reverse s
```

And code for controlling and running the calculator:

```haskell
calc :: String -> IO ()
calc s = do
    display s
    c <- getCh
    if c `elem` buttons
        then process c s
        else do
            beep
            calc s

beep :: IO ()
beep = putStr "\BEL"

process :: Char -> String -> IO ()
process c s
    | c `elem` "qQ\ESC" = quit
    | c `elem` "d D\BS\DEL" = delete s
    | c `elem` "=\n" = eval s
    | c `elem` "cC" = clear
    | otherwise = press c s

quit :: IO ()
quit = goto (1, 14)

delete :: String -> IO ()
delete [] = calc []
delete s = calc $ init s

eval :: String -> IO ()
eval s = case parse expr s of
    Just (n, []) -> calc $ show n
    _ -> do
        beep
        calc s

clear :: IO ()
clear = calc []

press :: Char -> String -> IO ()
press c s = calc $ s ++ [c]

run :: IO ()
run = do
    cls
    showbox
    clear
```

Exercises

Exercise 1

Define a parser `comment :: Parser ()` for ordinary Haskell comments that begin with the symbol `--` and extend to the end of the current line, which is represented by the control character `'\n'`.

```haskell
comment :: Parser ()
comment = void $ string "--" *> many (sat (/= '\n')) *> char '\n'
```

Exercise 2

Using our second grammar for arithmetic expressions, draw the two possible parse trees for the expression `2+3+4`.

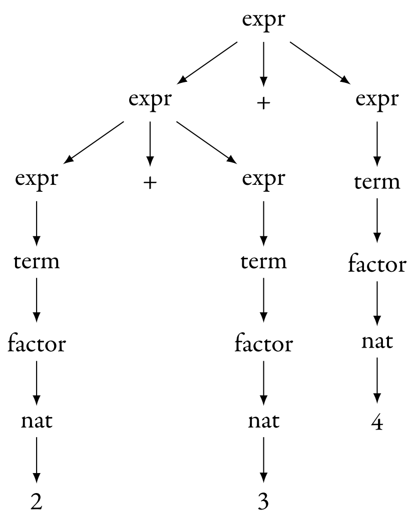

Figure 1: First Parse Tree

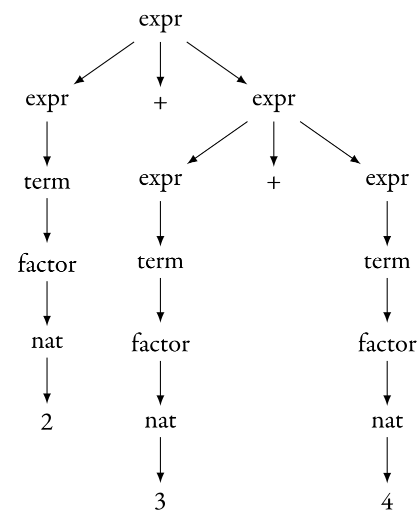

Figure 2: Second Parse Tree

Exercise 3
Using our third grammar for arithmetic expessions, draw the parse trees for the expressions `2+3`, `2*3*4` and `(2+3)+4`

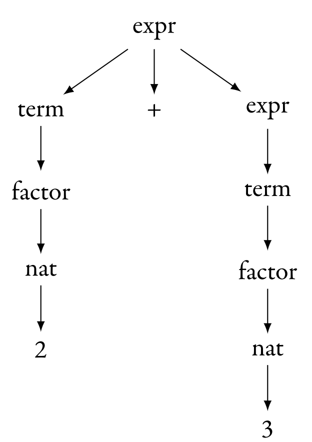

Figure 3: Parse Tree for 2 + 3

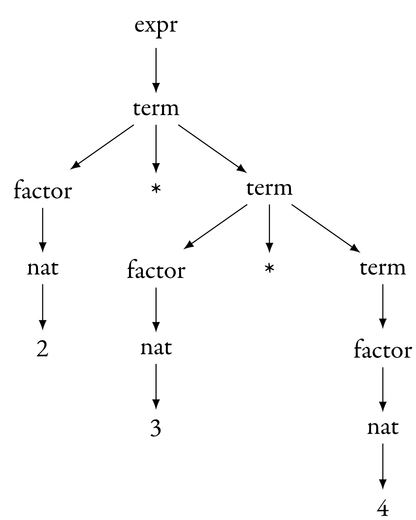

Figure 4: Parse Tree for $2 * 3 * 4$

Exercise 4: Explain Why the final simplification of the grammar for arithmetic expressions has a dramatic effect on the efficiency of the resulting parser. Hint: begin by considering how an expression comprising a single number would be parsed if this simplification step had not been made.

Without "left-factoring," the expression parser would be:

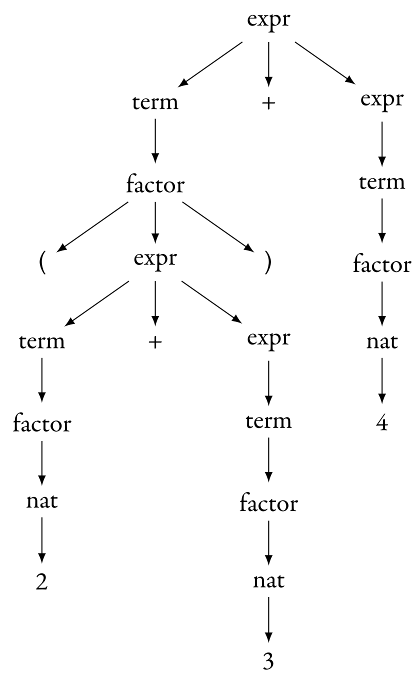

Figure 5: Parse Tree for $(2 * 3) + 4$

```
exprUnfactored :: Parser Int
exprUnfactored = (+) <$> term <* symbol "+" <*> expr <|> term
```

To parse only a natural number, first `term <* symbol "+" <*> expr` will be parsed, so `term` will be parsed completely and then the parser will fail at `symbol "+"`. Then the term will be parsed again and succeed. So the parser will parse the term twice.

Exercise 5: Define a suitable type `Expr` for arithmetic expressions and modify the parser for expressions to have type `expr :: Parser Expr`.

```
data Expr = T Term (Maybe Expr) deriving (Show)
data Term = F Factor (Maybe Term) deriving (Show)
data Factor = E Expr | N Int deriving (Show)

nothing :: Parser (Maybe a)
nothing = pure Nothing

expr' :: Parser Expr
expr' = T <$> term' <*> (Just <$> (symbol "+" *> expr') <|> nothing)

term' :: Parser Term
term' = F <$> factor' <*> (Just <$> (symbol "*" *> term') <|> nothing)

factor' :: Parser Factor
factor' = E <$> (symbol "(" *> expr' <* symbol ")") <|> N <$> natural
```

Exercise 6 Extend the parser `expr :: Parser Int` to support subtraction and division, and to use integer values rather than natural numbers, based upon the following revisions to the grammar:

⟨*expr*⟩ ::= ⟨*term*⟩ (‘+’ ⟨*expr*⟩ | ‘-’ ⟨*expr*⟩ | ⟨*empty*⟩)

⟨*term*⟩ ::= ⟨*factor*⟩ (‘*’ ⟨*term*⟩ | ‘/’ ⟨*term*⟩ | ⟨*empty*⟩)

⟨*factor*⟩ :: = ‘(’ ⟨*expr*⟩ ‘)’ | ⟨*int*⟩

As follows, switching to monadic parsing since the character we parse determines the function to apply.

```
exprI :: Parser Int
exprI = do
    t <- termI
    do
        op <- symbol "+" <|> symbol "-"
        e <- exprI
        case op of
            "+" -> return $ t + e
            "-" -> return $ t - e
        <|> return t

termI :: Parser Int
termI = do
    f <- factorI
    do
        op <- symbol "*" <|> symbol "/"
        t <- termI
        case op of
            "*" -> return $ f * t
            "/" -> return $ f `div` t
        <|> return f


factorI :: Parser Int
factorI = symbol "(" *> exprI <* symbol ")" <|> integer
```

Exercise 7 Further extend the grammar and parser for arithmetic expressions to support exponentiation `^`, which is assumed to associate to the right and have higher priority than multiplication and division, but lower priority than parentheses and numbers. For example, `2^3*4` means `(2^3)*4`. Hint: the new level of priority requires a new rule in the grammar.

```
exprE :: Parser Int
exprE = do
    t <- termE
    do
        op <- symbol "+" <|> symbol "-"
        e <- exprE
        case op of
            "+" -> return $ t + e
            "-" -> return $ t - e
        <|> return t

termE :: Parser Int
termE = do
    f <- factorE
    do
        op <- symbol "*" <|> symbol "/"
        t <- termE
        case op of
            "*" -> return $ f * t
            "/" -> return $ f `div` t
        <|> return f

factorE :: Parser Int
factorE = (^) <$> baseE <*> (symbol "^" *> powerE <|> one)

baseE :: Parser Int
baseE = symbol "(" *> exprE <* symbol ")" <|> integer

powerE :: Parser Int
powerE = baseE
```

Exercise 8 Consider expressions built up from natural numbers using a subtraction operator that is assumed to associate to the left.

1. Translate this description directly into a grammar.

   $\langle expr \rangle ::= \ \langle expr \rangle\, (\text{‘-’}\, \langle int \rangle \mid \langle empty \rangle)$

2. Implement this grammar as a parser `expr :: Parser Int`

   ```
   exprSub :: Parser Int
   exprSub = (-) <$> exprSub <*> (symbol "-" *> natural) <|> zero
   ```

3. What is the problem with this parser?

   Answer: It never terminates.

4. Show how it can be fixed. Hint: rewrite the parser using the repetition primitive `many` and the library function `foldl`.

   ```
   exprSub' :: Parser Int
   exprSub' = foldl (-) <$> natural <*> many (symbol "-" *> natural)
   ```