# LECTURE 10

Theory and Design of PL (CS 538)

February 24, 2020

# NEWS

# HW2: COMMENTS/QUESTIONS?

# HW3: IMPLEMENTING RUSE

- Big assignment: due in **three** weeks
- Three parts:
  1. Implementing an expression evaluator
  2. Implementing a parser
  3. Implementing a command-line REPL

*Start early!!!*

# HW3: RECOMMENDED ORDER

1. First part of WR3, do-notation
2. Evaluator (`Eval.hs`)
3. Parser (`Parser.hs`)
4. REPL (`Main.hs`)
5. Optional tests (`Tests.hs`)

# APPLICATIVE PARSING

# PARSER COMBINATORS: APPLICATIVE

- Applicative will let us combine multiple parsers:

```
instance Applicative Parser where
  -- pure :: a -> Parser a
  pure x = MkParser $ \str -> Just (x, str)

  -- (<*>) :: Parser (a -> b) -> Parser a -> Parser b
  parF <*> parA = MkParser $ \str ->
    case runParser parF str of                        -- run first
      Nothing -> Nothing                              -- first failed
      Just (f, str') ->                               -- first OK
        case runParser parA str' of                   -- run second
          Nothing -> Nothing                          -- second failed
          Just (v, str'') -> Just (f v, str'')        -- second OK
```

- Kind of sequencing: feed `str'` to second parser

# USEFUL ABBREVIATIONS: APPLICATIVE

- Get these definitions for free from Applicative

```
-- Mapping a two-argument function (shout2 to shout2Maybe)
-- Or: sequence two things, then apply function to results
liftA2 :: (Applicative f) => (a -> b -> c) -> f a -> f b -> f c
liftA2 fun x y = fun <$> x <*> y

-- Sequence two things, keep first result, forget second result
(<*) :: (Applicative f) => f a -> f b - > f a
(<*) = liftA2 (\x y -> x)

-- Sequence two things, forget first result, keep second result
(*>) :: (Applicative f) => f a -> f b - > f b
(*>) = liftA2 (\x y -> y)
```

# PARSER COMBINATORS: ALTERNATIVE

- Ordered choice: try two, take first one that works

```
orElseP :: Parser a -> Parser a -> Parser a
par1 `orElseP` par2 = MkParser $ \str ->
  let firstRes = runParser par1 str           -- Try parser 1
  in case firstRes of
    Nothing          -> runParser par2 str  -- Fail, try parser 2
    Just (val, str') -> Just (val, str')    -- OK, return parse 1
```

# ALTERNATIVE TYPECLASS

- Cleaning up, use the Alternative typeclass

```
instance Alternative Parser where
  -- (<|>) :: Parser a -> Parser a -> Parser a
  (<|>) = orElseP

  -- empty :: Parser a
  empty = failP

failP = MkParser $ \_ -> Nothing  -- Always fails (not emptyP!)
```

- Law: `empty <|> p === p <|> empty === p`
  - Monoid, but for things with a type parameter

# REPETITION

- Repeat a parser multiple times, gather results

```
manyP :: Parser a -> Parser [a]       -- zero-or-more times
manyP par = (manyP1 par) <|> emptyP
--               ^        ^    ^-------- zero times
--               |        +------------- or
--               +---------------------- one-or-more times

manyP1 :: Parser a -> Parser [a]      -- one-or-more times
manyP1 par = (:) <$> par <*> manyP par
--            ^       ^       ^--------- zero or more times
--            |       +----------------- parse one time
--            +------------------------- gather results
```

# SEPARATED BY

- Parse lists of things separated by something:

```
list = item {sep item}
```

- Parse out the items, while dropping separators

```
-- zero or more items
sepByP :: Parser a -> Parser b -> Parser [a]
sepByP item sep = (sepBy1P item sep) <|> emptyP

-- one or more items
sepBy1P :: Parser a -> Parser b -> Parser [a]
sepBy1P item sep = (:) <$> item <*> manyP (sep *> item)
```

# EXAMPLE: CALCULATOR

# CALCULATOR INPUTS

- Handle numbers, parentheses, plus, and minus
  - In HW3: you will do much more...

```
term = atom "+" term | atom "-" term | atom ;
atom = num | "(" term ")" ;
```

- Model grammars with two Haskell datatypes:

```haskell
data Term = Add Atom Term | Sub Atom Term | Single Atom
data Atom = Num Int | Parens Term

-- "Recursive descent" parser, grammar must be "left-factored"...
```

# FIRST: HELPER PARSERS

- Parse any number of spaces

```
spacesP :: Parser [Char]
spacesP = manyP spaceP
```

- Ignore spaces, then parse something

```
tokenP :: Parser a -> Parser a
tokenP par = spacesP *> par
```

# PARSING NUMBERS

- Numeric constants: nonempty string of digits

```
-- parse a string of digits into an Int
intP :: Parser Int
intP = read <$> manyP1 digit

-- ignore spaces before int, wrap in Num
numP :: Parser Atom
numP = Num <$> (tokenP intP)
```

# PARSING PARENS

- Parenthesized: term surround by parentheses

```
parenTermP :: Parser Atom
parenTermP = Parens <$>
    (tokenP $ charP '(') *> termP <* (tokenP $ charP ')')
```

# PARSING ATOMS

- Now, we are ready to parse atoms:

```
atomP :: Parser Atom
atomP =   numP
      <|> parenTermP
```

# PARSING TERMS

- We're finally ready to parse terms:

```
termP :: Parser Term
termP =  Add <$> atomP <* (tokenP (charP '+')) <*> termP
     <|> Sub <$> atomP <* (tokenP (charP '-')) <*> termP
     <|> Single <$> atomP
```

# EVALUATING ATOMS

- Atom evaluator: turn atoms into numbers

```
evalAtom :: Atom -> Int
evalAtom atom = case atom of
  Num n         -> n
  Parens term -> evalTerm term
```

# EVALUATING TERMS

- Term evaluator: turn terms into numbers

```
evalTerm :: Term -> Int
evalTerm term = case term of
  Add at term -> (evalAtom at) + (evalTerm term)
  Sub at term -> (evalAtom at) - (evalTerm term)
  Single atom -> evalAtom atom
```

# CAPPING IT ALL OFF

- Read input, parse, evaluate, print, and loop

```
runCalc :: String -> String
runCalc input = case getParse input of
  Nothing   -> "Parse failed"
  Just term -> show $ evalTerm term
```