COMP3258

Functional Programming

Tutorial Session 7: Parser Combinators

Parser

```
data Parser a = P (String → [(a,String)])
```


```
parse :: Parser a → String → [(a,String)]
parse (P p) inp  =  p inp
```

```
parse :: Parser a → (String → [(a,String)])
parse (P p) =  p
```

Primitive Combinators

```
> parse item "abc"
[('a',"bc")]

> parse failure "abc"
[]

> parse (return 1) "abc"
[(1,"abc")]

> parse (item +++ return 'd') "abc"
[('a',"bc")]

> parse (failure +++ return 'd') "abc"
[('d',"abc")]
```

Do-notation

```
foo :: Parser (Char, Char)
foo = do x ← item
         item
         y ← item
         return (x, y)
```

```
> parse foo "abc"
[(('a','c'),"")]
```

- Each line of code must begin at the same column (same indentation level) in the block of do notation.
- In the first line, x <- item invokes item to parse a character, and x is the name of the result.
- In the second line, item is equivalent to _ <- item, which means the result is discarded because we don't care about it.
- In the last line, it returns a pair of x and y.
- In the sequence, if any parser fails, the whole parser fails. In this example, if the string only contains two characters, it will fail on the third parser, and return an empty list finally.

More Combinators

```
sat :: (Char → Bool) → Parser Char
sat p = do x ← item
           if (p x) then return x else failure

digit :: Parser Char
digit = sat isDigit

char :: Char → Parser Char
char x = sat (x ==)

-- Repeat zero or more times
many :: Parser a → Parser [a]

-- Repeat one or more times:
many1 :: Parser a → Parser [a]
```

Question 1

Define a parser `firstAlpha :: Parser Char` that gets the first alphabetic character (`a-z, A-Z`) in a string.

Hint: use `isAlpha` in `Data.Char`

```
-- > parse firstAlpha "aaa"
-- [('a',"aa")]

-- > parse firstAlpha "11111aaa"
-- [('a',"aa")]

firstAlpha :: Parser Char
firstAlpha = do many (sat (not . isAlpha))
                item
```

isAlpha :: Char -> Bool # Source

Selects alphabetic Unicode characters (lower-case, upper-case and title-case letters, plus letters of caseless scripts and modifiers letters). This function is equivalent to `isLetter`.

Question 2

Define a parser `manyN :: Int → Parser a → Parser [a]` that applies a parser for the specified number of times.

```
-- > parse (manyN 0 item) "abbbbbb"
-- [("","abbbbbb")]

-- > parse (manyN 3 item) "abbbbbb"
-- [("abb","bbbb")]

-- > parse (manyN 2 digit) "11ab"
-- [("11","ab")]

-- > parse (manyN 3 digit) "11ab"
-- []
```

```
manyN :: Int → Parser a → Parser [a]
manyN 0 p = return []
manyN n p = do x ← p
               xs ← manyN (n-1) p
               return (x: xs)
```

Question 3: Parsing Expressions

```
data Expr = Val Int
          | Add Expr Expr
          | Mul Expr Expr

pExpr :: Parser Expr
pExpr = undefined

> parse pExpr "1+2"
[(Add (Val 1) (Val 2),"")]
```

$$
\begin{aligned}
expr &:= term\,'+'\,expr \mid term \\
term &:= factor\,'*'\,term \mid factor \\
factor &:= '('\,expr\,')' \mid integer
\end{aligned}
$$

Question 3: Step 1

Define three functions pExpr, pTerm and pFactor for the three cases above.
The result should be an Expr so that they all have type Parser Expr.

```
pExpr :: Parser Expr
pExpr = undefined

pTerm :: Parser Expr
pTerm = undefined

pFactor :: Parser Expr
pFactor = undefined
```

$$
\begin{aligned}
expr &:= term\,'+'\,expr \mid term \\
term &:= factor\,'*'\,term \mid factor \\
factor &:= '('\,expr\,')' \mid integer
\end{aligned}
$$

Question 3: Step 2

Start with the factor case, it has two alternatives: an expression in a pair of parentheses (pPara), or an integer (pInt).

```
pFactor :: Parser Expr
pFactor = pPara +++ pInt
    where pPara = undefined
          pInt  = undefined
```

$$
\begin{aligned}
expr &:= term\,'+'\,expr \mid term \\
term &:= factor\,'*'\,term \mid factor \\
factor &:= \text{'('}\,expr\,\text{')'} \mid integer
\end{aligned}
$$

Question 3: Step 2

In Parsing.hs, there's a parser for parsing integers called integer. It uses the token function to get rid of spaces. We use it for our pInt:

```
pInt  = do x ← integer
           return $ Val x
```

$$
\begin{aligned}
expr &:= term\,'+'\,expr \mid term \\
term &:= factor\,'*'\,term \mid factor \\
factor &:= \text{'('}expr\text{')'} \mid integer
\end{aligned}
$$

Question 3: Step 2

Then for the pPara, we parse an open parenthesis, an expression, and a close parenthesis in sequence. We use token function to remove leading and trailing spaces.

```
pPara = do token $ char '('
           e ← pExpr
           token $ char ')'
           return e
```

$$
\begin{aligned}
expr &:= term\,'+'\,expr \mid term \\
term &:= factor\,'*'\,term \mid factor \\
factor &:= \text{'('}expr\text{')'} \mid integer
\end{aligned}
$$

Question 3: Step 2

Then for the pPara, we parse an open parenthesis, an expression, and a close parenthesis in sequence. We use token function to remove leading and trailing spaces.

```
pPara = do token $ char '('
           e ← pExpr
           token $ char ')'
           return e
```

$$
\begin{aligned}
expr &:= term\,'+'\,expr \mid term \\
term &:= factor\,'*'\,term \mid factor \\
factor &:= \text{'('}expr\text{')'} \mid integer
\end{aligned}
$$

Question 3: Step 2

Start with the factor case, it has two alternatives: an expression in a pair of parentheses (pPara), or an integer (pInt).

```
pFactor :: Parser Expr
pFactor = pPara +++ pInt
    where pPara = do _ ← token $ char '('
                     e ← pExpr
                     _ ← token $ char ')'
                     return e
          pInt  = do x ← integer
                     return $ Val x
```

$$
\begin{aligned}
expr &:= term\,'+'\,expr \mid term \\
term &:= factor\,'*'\,term \mid factor \\
factor &:= '('expr')' \mid integer
\end{aligned}
$$

Question 3: Step 3

For the pTerm, we still have two alternatives: a multiplication of a factor and a term (pFactorTerm), or a single factor (pFactor).

Implement the pFactorTerm above.

```
pTerm :: Parser Expr
pTerm = pFactorTerm +++ pFactor
    where pFactorTerm = do
            f ← pFactor
            token $ char '*'
            t ← pTerm
            return $ Mul f t
```

$$
\begin{aligned}
expr &:= term\,'+'\,expr \mid term \\
term &:= factor\,'*'\,term \mid factor \\
factor &:= \text{'('}\,expr\,\text{')'} \mid integer
\end{aligned}
$$

Question 3: Step 4

We use the similar skeleton for the pExpr function:

```
pExpr :: Parser Expr
pExpr = pTermExpr +++ pTerm
    where pTermExpr = do
                t ← pTerm
                _ ← token $ char '+'
                e ← pExpr
                return $ Add t e
```

$$
\begin{aligned}
expr &:= term\,'+'\,expr \mid term \\
term &:= factor\,'*'\,term \mid factor \\
factor &:= '('expr')' \mid integer
\end{aligned}
$$

Question 4

sepBy works like some, although the consecutive parses of the parser are separated by the parse of another parser. This is particularly useful when parsing a list like structure such as “a, b, c, d”

```
>>> p = (many1 (char 'a')) `sepBy` token (char ',')
>>> parse p "aaa"
[(["aaa"],"")]
>>> parse p ""
[]
>>> parse p "aa, aaa , a"
[(["aa","aaa","a"],"")]
```

```
sepBy :: Parser a → Parser b → Parser [a]
sepBy p sp = do
   x ← p
   xs ← many (do sp
                 p)
   return (x:xs)
```

Question 5

Define a parser a non-empty list of integers, such as [1,-42,17], using sepBy.

```
>>> parse ints "[1,2,3,4]"
[([1,2,3,4],"")]
```

```haskell
ints :: Parser [Int]
ints = do
   char '['
   ns ← int `sepBy` (char ',')
   char ']'
   return ns
```

Question 6: Parse Url Query

- Parse url query: application/x-www-form-urlencoded
- format: “field1=value1&field2=value2”
- e.g., "foo=bar&a%21=b+c"

```
>>> parse p_query "foo=bar&a%21=b+c"
[(["foo",Just "bar"),("a!",Just "b c")],"")]
```

Question 6

```haskell
hexDigit = sat isHexDigit

p_query :: Parser [(String, Maybe String)]
p_query = p_pair `sepBy` (char '&')

p_pair :: Parser (String, Maybe String)
p_pair = do
    name ← many1 p_char
    value ← optionMaybe (char '=' >> many p_char)
    return (name, value)

p_char :: Parser Char
p_char = oneOf urlBaseChars +++ (char '+' >> return ' ') +++ p_hex
    where urlBaseChars = ['a'..'z']++['A'..'Z']++['0'..'9']++"$-_.!*'(),"

p_hex :: Parser Char
p_hex = do
    char '%'
    a ← hexDigit
    b ← hexDigit
    let ((d, _):_) = readHex [a,b]
    return . toEnum $ d
```

```haskell
option :: a → Parser a → Parser a
option x p = p +++ return x

optionMaybe :: Parser a → Parser (Maybe a)
optionMaybe p = option Nothing (liftM Just p)

oneOf cs = sat (\c → elem c cs)
```