Functional programming, Seminar No. 8

Danya Rogozin
Institute for Information Transmission Problems, RAS
Serokell OÜ
Higher School of Economics
The Department of Computer Science

Type of parser

What is a type of function which parses an integer?

```
parseInteger :: String -> Bool
```

Not quite, we would like to get an `Integer`

The second variant. But such a parser can fail.

```
parseInteger :: String -> Integer
```

```
parseInteger :: String -> Maybe Integer
```

Now. How do you parse two integers separated by space? Or by any number of spaces? Or comma-separated list of integers? Or the same list inside square brackets [] with any number of spaces between elements?

The solution is the following:

```
parseInteger :: String -> Maybe (Integer, String)
```

Type of parser

But instead of this:

```
parseInteger :: String -> Maybe (Integer, String)
```

We introduce `Parser` with the `newtype` wrapper in order to have useful instances:

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

Some usage examples

The parser combinator type:

```
newtype Parser a = Parser { runP :: String -> Maybe (a, String)
```

Parsing functions

```
parseInteger :: Parser Integer
  -- String -> Maybe (Integer, String)
```

How to use it and which behaviour we want?

```
runP :: Parser a -> String -> Maybe (a, String)
ghci> runP parseInteger "5"
Just (5, "") :: Maybe (Integer, String)

ghci> runP parseInteger "42x7"
Just (42, "x7") :: Maybe (Integer, String)

ghci> runP parseInteger "abc"
Nothing :: Maybe (Integer, String)
```

Now, idea and how to do it

- The key idea of parser combinators: implement manually very simple parsers, implement combinators for combining parser and then implement more complex parsers by combining simpler ones.
- Haskell provides combinators through standard type classes. And it's convenient to use them. What we need is just:

```
instance Functor     Parser
  -- replace parser value
instance Applicative Parser
  -- run parsers sequentially one after another
instance Monad       Parser
  -- same as above but with monadic capabilities
instance Alternative Parser
  -- allows one to choose parser
```

Primitive parsers. `ok`

```haskell
newtype Parser a =
  Parser { runP :: String -> Maybe (a, String) }

-- always succeeds without consuming any input
ok :: Parser ()
ok = Parser $ \s -> Just ((), s)
```

Primitive parsers. `isnot`

```haskell
newtype Parser a =
  Parser { runP :: String -> Maybe (a, String) }

-- fails w/o consuming any input if given parser succeeds,
-- and succeeds if given parser fails
isnot :: Parser a -> Parser ()
isnot parser =
  Parser $ \s -> case runP parser s of
                   Just _  -> Nothing
                   Nothing -> Just ((), s)
```

Primitive parsers. `eof`

```haskell
newtype Parser a =
  Parser { runP :: String -> Maybe (a, String) }

-- succeeds only at the end of input stream
eof :: Parser ()
eof = Parser $ \s -> case s of
    [] -> Just ((), "")
    _  -> Nothing
```

Primitive parsers. ok

```haskell
newtype Parser a =
  Parser { runP :: String -> Maybe (a, String) }

-- consumes only single character and
-- returns it if predicate is true
satisfy :: (Char -> Bool) -> Parser Char
satisfy p = Parser $ \s -> case s of
  []     -> Nothing
  (x:xs) -> if p x then Just (x, xs) else Nothing
```

Combining

```haskell
-- always fails without consuming any input
notok :: Parser ()
notok = isnot ok

-- consumes given character and returns it
char :: Char -> Parser Char
char c = satisfy (== c)

-- consumes any character or any digit only
anyChar, digit :: Parser Char
anyChar = satisfy (const True)
digit   = satisfy isDigit
```

Combining

```
ghci> runP eof ""
Just ((),"")
ghci> runP eof "aba"
Nothing
ghci> runP (char 'a') "aba"
Just ('a',"ba")
ghci> runP (char 'x') "aba"
Nothing
```

Instances

The `Functor` instance

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

instance Functor Parser where
  fmap :: (a -> b) -> Parser a -> Parser b
  fmap f (Parser parser) = Parser (fmap (first f) . parser)

first :: (a -> b) -> (a, c) -> (b, c)
first f (a, c) = (f a, c)
```

Instances

The `Applicative` instance

```haskell
newtype Parser a =
  Parser { runP :: String -> Maybe (a, String) }

instance Applicative Parser where
  pure :: a -> Parser a
  pure a = Parser $ \s -> Just (a, s)

  (<*>) :: Parser (a -> b) -> Parser a -> Parser b
  Parser pf <*> Parser pa = Parser $ \s -> case pf s of
      Nothing     -> Nothing
      Just (f, t) -> case pa t of
          Nothing     -> Nothing
          Just (a, r) -> Just (f a, r)
```

Instances

The `Monad` instance

```haskell
newtype Parser a =
  Parser { runP :: String -> Maybe (a, String) }

instance Monad Parser where
  (>>=) :: Parser a -> (a -> Parser b) -> Parser b
  Parser pf >>= k = Parser $ \s -> do
      (x, rest1) <- runP p1 s
      runP (k x) rest1

instance Alternative Parser where
  empty :: Parser a
  -- always fails
  (<|>) :: Parser a -> Parser a -> Parser a
  -- run first, if fails - run second
```

Simple parser combinators core

```haskell
-- type
newtype Parser a =
  Parser { runP :: String -> Maybe (a, String) }

-- parsers
eof, ok :: Parser ()
satisfy :: (Char -> Bool) -> Parser Char
empty   :: Parser a

-- combinators
pure  :: a -> Parser a
(<|>) :: Parser a -> Parser a -> Parser a          -- orElse
(>>=) :: Parser a -> (a -> Parser b) -> Parser b   -- andThen
```

Simple parser combinators core

```haskell
-- combinators
-- * Functor
fmap  :: (a -> b) -> Parser a -> Parser b
(<$)  :: a -> Parser b -> Parser a

-- * Applicative
(<*>) :: Parser (a -> b) -> Parser a -> Parser b
(<*)  :: Parser a -> Parser b -> Parser a
  -- run both in sequence, result of first

-- * Alternative
many  :: Parser a -> Parser [a]
some  :: Parser a -> Parser [a]

-- * Monadic
(>>=) :: Parser a -> (a -> Parser b) -> Parser b   -- andThen
```

Examples

```
ghci> runP (ord <$> char 'A') "A"
Just (65,"")

ghci> runP ((\x y -> [x, y]) <$> char 'a' <*> char 'b') "abc"
Just ("ab","c")
ghci> runP ((\x y -> [x, y]) <$> char 'a' <*> char 'b') "xxx"
Nothing

ghci> runP (char 'a' <* eof) "a"
Just ('a',"")
ghci> runP (char 'a' <* eof) "ab"
Nothing
```

Examples

```
ghci> runP (many $ char 'a') "aaabcd"
Just ("aaa","bcd")
ghci> runP (many $ char 'a') "xxx"
Just ("","xxx")
ghci> runP (some $ char 'a') "xxx"
Nothing
```

```
ghci> runP (many $ char 'a') "aaabcd"
Just ("aaa","bcd")
ghci> runP (many $ char 'a') "xxx"
Just ("","xxx")
ghci> runP (some $ char 'a') "xxx"
Nothing
```

Examples

```
ghci> runP (many $ char 'a') "aaabcd"
Just ("aaa","bcd")
ghci> runP (many $ char 'a') "xxx"
Just ("","xxx")
ghci> runP (some $ char 'a') "xxx"
Nothing
```

```
ghci> runP (many $ char 'a') "aaabcd"
Just ("aaa","bcd")
ghci> runP (many $ char 'a') "xxx"
Just ("","xxx")
ghci> runP (some $ char 'a') "xxx"
Nothing
```

Examples

```haskell
string :: String -> Parser String
  -- like 'char' but for string
oneOf  :: [String] -> Parser String
  -- parse first matched string from list

data Answer = Yes | No

yesP :: Parser Answer
yesP = Yes <$ oneOf ["y", "Y", "yes", "Yes", "ys"]

noP :: Parser Answer
noP = No <$ oneOf ["n", "N", "no", "No"]

answerP :: Parser Answer
answerP = yesP <|> noP
```