Parser Combinators

João Saraiva

2016/2017

Departamento de Informática
Universidade do Minho, Portugal

Parser: Type

A parser consumes the input string and produces an abstract syntax tree:

```
type Parser = [Char] -> Ast
```

Parser: Type

However, in a top-down parser each symbol consumes part of the input, leaving the rest for the next symbol:

$$S \rightarrow A\ B$$
$$A \rightarrow a+$$
$$B \rightarrow b^*$$

In this grammar A consumes the symbols a in the begging of a sentence, leaving bs to be parser by B.

Parser: Type

Thus, our new type reflects the partial consumption of he input by a grammar symbol:

```
type Parser = [Char] -> (Ast , [Char])
```

Parser: Type

But, a parser may fail!

Thus we use the list of successes technique where the empty list models the parsing failure.

```
type Parser = [Char] -> [ (Ast , [Char]) ]
```

Parser: Type

Finally, we parameterize our type so that it consumes any type of input (not only Char) and produces any tree as result.

```
type Parser s r = [s] -> [(r , [s])]
```

Basic Parsers: symbols

Let us start with a very simple parser:
it parser/consumes a symbol ‘a’ from the input.

```
symbola [] = []
symbola (x:xs) = if x == 'a' then [('a',xs)]
                          else []
```

Basic Parsers: symbols

That has type:

```
symbola :: [Char] -> [ (Char , [Char])]
symbola [] = []
symbola (x:xs) = if x == 'a' then [('a',xs)]
                 else []
```

Basic Parsers: symbols

Running ghci:

```
Parser> symbola "abc"
[('a',"bc")]
```

Basic Parsers: symbols

This function is a parser. And, we can rewrite its type as a Parser type as follows

```
symbola :: Parser Char Char
symbola [] = []
symbola (x:xs) = if x == 'a' then [('a',xs)]
                 else []
```

Basic Parsers: symbols

We can parameterize this function with the symbol we wish to parse from the input:

```
symbol :: Eq a => a -> Parser Char Char
Symbol s []                       = []
symbol s (x:xs) | s == x    = [(s,xs)]
                | otherwise = []
```

Basic Parsers: symbols

We can also define a very useful basic parser that parser a symbol from the input that satisfy a give predicate

```
satisfy :: (s -> Bool) -> Parser s s
satisfy p []                          = []
satisfy p (x:xs) | p x                = [(x,xs)]
                 | otherwise = []
```

Basic Parsers: symbols

Running ghci:

```
Parser> satisfy isDigit "12a"
[('1',"2a")]
```

Basic Parsers: symbols

Running ghci:

```
Parser> satisfy isDigit "12a"
[('1',"2a")]
```

Basic Parsers: tokens

But, we also need to parser a sequence of symbols, ie. a token, from the input.

```
token :: Eq s => [s] -> Parser s [s]
token t [] = []
token t inp = if   take (length t) inp == t
              then [(t,drop (length t) inp)]
              else []
```

Basic Parsers: succeeds

In order to model the empty production (that does not consume any symbol from the input) we define a parser that always succeeds:

```
succeed :: r -> Parser s r
succeed r inp = [ ( r , inp) ]
```

Parser Combinators: Or

Now, we need to define parser that combine the basic combinators. In grammars we use the BNF notation | to model alternatives of parsing.

S → “while”
  | “for

So, we need to define a combinator that expresses such a grammar “or”.

Parser Combinators: Or

We use the infix function **<|>** to model different parsing alternatives of the same input: we apply the two parsar to the input and we combine the list results by concatenating them:

```
(<|>) :: Parser s a -> Parser s a -> Parser s
a
(p <|> q) inp = p inp ++ q inp
```

Parser Combinators: Or

We use the infix function <|> to model different parsing alternatives of the same input: we apply the two parsers to the input and we combine their list results by concatenating them:

```
(<|>) :: Parser s a -> Parser s a -> Parser s
a
(p <|> q) inp = p inp ++ q inp
```

Parser Combinators: Or

```
pWhileFor =  token “for”
         <|> token “while”
```

Running ghci:

```
Parser> pWhileFor "while (x>0)"
[("while"," (x>0)")]
```

Parser Combinators: Or

```
pWhileFor =  token “for”
         <|> token “while”
```

Running ghci:

```
Parser> pWhileFor "while (x>0)"
[("while"," (x>0)")]
```

Parser Combinators: Or

We use the infix function <|> to model different parsing alternatives of the same input: we apply the two parsar to the input and we combine the list results by concatenating them:

```
(<|>) :: Parser s a -> Parser s a -> Parser s
a
(p <|> q) inp = p inp ++ q inp
```

Parser Combinators: Then

In grammars, the right-hand side of a production may include a sequence of symbols:

$$S \rightarrow A\ B$$
$$A \rightarrow a+$$
$$B \rightarrow b^*$$

Thus, we need to define a combinator parser that combines the sequential application of parsers.

Parser Combinators: Then

Because a parser may produce several solutions, we need to apply the first parser to the input. Then, to all produced results we apply the second parser (to the rest of input). Finally, we combine the results by tupling them:

```
(<*>)::Parser s a-> Parser s b-> Parser s
(a,b)
(p <*> r) inp = [ ((x,y),ys)
                | (x,xs)  <- p inp
                , (y,ys)  <- r xs
                ]
```

Parser Combinators: Then

Because a parser may produce several solutions, we need to apply the first parser to the input. Then, to all produced results we apply the second parser (to the rest of input). Finally, we combine the results by tupling them:

```
(<*>)::Parser s a-> Parser s b-> Parser s
(a,b)
(p <*> r) inp = [ ((x,y),ys)
                | (x,xs)  <- p inp
                , (y,ys)  <- r xs
                ]
```

Parser Combinators: Or

```
pSeq = symbol ‘a’ <*> symbol ‘b’ <*> symbol
‘c’
```

Running ghci:

```
Parser> pSeq "abcd"
[((('a','b'),'c'),"d")]
```

Parser Combinators: Then

By tupling the parsers' results may produce nested tuples which make handling them extremely difficult.

Thus we use a clever approach: the first parser returns a function as result, the second returns a value, and we combine the results of the two by applying the returned function to the returned value!

Parser Combinators: Then

```
(<*>) :: Parser s (a -> b)
      -> Parser s a
      -> Parser s b
(p <*> r) inp = [ (f v ,ys)
                | (f   ,xs)  <- p inp
                , (  v ,ys)  <- r xs
                ]
```

Parser Combinators: Apply

Now, we need a way to apply a function during parsing. We define the **<$>** combinator.

```
(<$>) :: (a -> r) -> Parser s a -> Parser s r
(f <$> p) inp = [ (f v , xs)
                | (v   , xs)  <- p inp
                ]
```

Parser Combinators in Practice

Consider for example the following grammar that expresses a list of spaces:

$$
\begin{aligned}
\text{Spaces} \rightarrow\ & \text{‘ ‘ Spaces} \\
|\ & \text{‘ ‘}
\end{aligned}
$$

We can express it as follows:

```
(<$>) :: (a -> r) -> Parser s a -> Parser s r
spaces =    f <$>  symbol ‘ ’ <*> spaces
        <|>  g <$>  symbol ‘ ‘
```

Parser Combinators in Practice

If we are interested in the spaces we just add them to a list:

```
f a b = a : b
g a   = [a]
```

If we want to discard them:

```
f _ _ = ()
g _   = ()
```

Parser Combinators in Practice

Exercise: re-write **spaces** so that it models a list of zero or more spaces

Exercise: re-write **token** so that it consumes spaces after the given token.

Parser Combinators in Practice

Exercise 1: re-write **spaces** so that it models a list of zero or more spaces

Exercise 2: re-write **token** so that it consumes spaces after the given token.

```
Spaces =  f   <$> symbol ‘ ‘ <*> spaces
      <|>   succeed ()
  Where f _ _ = ()

token t = (\r _ -> r) <$> token t <*> spaces’
```

Parser Combinators for EBNF

Exercise 3: Write the combinator **oneOrMore, and zeroOrMore,** that express the repetitions of a given paper, and have types:

```
oneOrMore , zeroOrMore :: Parser a b
                       -> Parser a [b]
```

Exercise 4: Re-write the parser **spaces** using the combinator **zeroOrMore**.

Parser Combinators for EBNF

Exercise 5: Write the combinator **optional** that expresses the optional operator of regular expressions a?.

```
optional :: Parser s r -> r -> Parser s r
```

Exercise 6: *Write the* parser **enclosedBy,** that expresses the parsing of a structure enclosed by some syntactic sugar (not contributing to the result of the parser):

```
enclosedBy :: Parser s a -> Parser s b
           -> Parser s c -> Parser s b
```

Parser Combinators for EBNF

Exercise 7: Write the combinators **separatedBy** and **followedBy** that expresses the parsing of a structured that is separated (followed) by some punctuation symbol. The parsing of such symbols do not contribute to the result.

```
SeparatedBy , followedBy :: Parser s a
                         -> Parser s b
                         -> Parser s [a]
```

Parser Combinators in Practice

A parser for Regular Expressions.

The result of the parser will be an AST defined as:

```
data RegExp sy  = Epsilon
                | Literal  sy
                | Or   (RegExp sy) (RegExp sy)
                | Then (RegExp sy) (RegExp sy)
                | Star      (RegExp sy)
                | OneOrMore (RegExp sy)
                | Optional  (RegExp sy)
```

Parser Combinators in Practice

Regular Expressions: non-felt recursive grammar

```
Expr      → ExprThen ‘|’ Expr
            | ExprThen
            |

ExprThen → Term ExprThen
            | Term

Term      → Factor ‘?’ | Factor ‘*’
            | Factor ‘+’ | Factor

Factor    →  <letterOrDig> | ‘<anyChar>‘
            | ‘(‘ Expr ‘)’
```

Parser Combinators in Practice

```
expr :: Parser Char (RegExp Char)
expr =  f  <$> termThen <*> symbol '|' <*> expr
    <|> id <$> termThen
    <|> succeed Epsilon
  where f l _ r = Or l r

termThen =  f  <$> term <*> termThen
        <|> id <$> term
  where f l r   = Then l r

term =  f  <$> factor <*> symbol '?'
    <|> g  <$> factor <*> symbol '*'
    <|> h  <$> factor <*> symbol '+'
    <|> id <$> factor
  where
     f e _ = Optional e
     g e _ = Star     e
     h e _ = Then     e (Star e)

factor =  f <$> satisfy (\x -> isDigit x || isAlpha x) -- letterOrDig
      <|> g <$> symbol '\'' <*> satisfy (\ x -> True) <*> symbol '\''
      <|> h <$> symbol '(' <*> expr <*> symbol ')'
  where
     f a     = Literal a
     g _ e _ = Literal e
     h _ e _ = e
```

Parser Combinators in Practice

```haskell
-- | Parser for regular expressions

parseRegExp :: [Char]                -- ^ Input symbols
            -> Maybe (RegExp Char)   -- ^ Regular expression
parseRegExp re = res
  where parsed_re = expr re
        res | parsed_re == [] = Nothing
            | otherwise       = Just (fst (head parsed_re))
```