Parser Combinators

A program which decides for a give word w entscheidet, whether it can is in the language of a given context-free grammar G is called *parser* for G. In addition to this yes/no answer, it is useful that the parser yields further information, like a left-derivation or a syntax tree.

There are different approaches to implement parsers:

- **Parser-generators** like YACC (for C) or Happy (for Haskell) take a textual representation of the grammar and produce a parser program for this grammar.
- **Recursive-decent parsers** can be defined by alternating, recursive functions - one function for every non-terminal of the grammar.
- **Parser combinators** allow the definition of recursive-decent parsers in a grammar like notation. Hence parser combinators define parsers, which can directly be executed and applied as parser.

In the following we deal with the usage and implementation of such parser combinators.

Using Parser Combinators

A Parser (combinator) is a value of type `Parser a` and can be applied to a string by means of the function `parse`.

```
parse :: Parser a -> String -> Maybe a
```

The type a contains additional information, which are returned in case of a successful parse. A parser which only decides, whether a word can be derived is of type `Parser ()`. A simple parser, matching a given character is of type

```
char :: Char -> Parser ()
```

and can be used as follows:

```
ghci> parse (char 'a') "a"
Just ()
```

For other `Strings` we obtain `Nothing`.

```
ghci> parse (char 'a') ""
Nothing
ghci> parse (char 'a') "b"
Nothing
ghci> parse (char 'a') "ab"
Nothing
```

Simple parser can be combined to complex parsers by means of parser combinators. The first one is

```
(*>) :: Parser a -> Parser b -> Parser b
```

which executes two parsers after each other and returns the result of the second parser. Using this combinator we can define a parser which only matches the String "()":

```
parens = char '(' *> char ')'
```

The corresponding grammar (in Backus-Naur Form, BNF) is:

```
Parens ::= '(' ')'
```

At the moment we do not consider the result of a parser. However, there is also another combinator which yields the result of the first parser and drops the result of the second parser:

```
(<*) :: Parser a -> Parser b -> Parser a
```

Note that in both combinators the first parser argument is used before the second parser. Hence, in general it is not true, that `(<*) = flip (*>)`.

Since our parser at the moment only base on the atomic `char`-parser, every parser yields `()` as a result.

```
ghci> parse parens "()"
Just ()
```

Now lets extend the parser, such that it can parse arbitrary nested expressions with brackets. The corresponding BNF is:

```
Nested ::= '(' Nested ')' Nested
       |
```

Here the second alternative means, that an nested bracket expression may also be the empty word. Furthermore, the definition is recursively. The non-terminal `Nested` occurs in the right-hand side of the first rule.

In Haskell we can define this parser similarly:

```
nested
  =  char '(' *> nested *> char ')' *> nested
 <|> empty
```

We use the combinator

```
(<|>) :: Parser a -> Parser a -> Parser a
```

which combines two alternative parsers and

```
empty :: Parser ()
```

as a parser, which matches the empty word. Again we can test this parser using the function `parse`:

```
ghci> parse nested "((()(())))"
Just ()
ghci> parse nested "(()()"
Nothing
```

Hence, our parsers fulfill the following laws:

```
empty *> p  =  p
p <* empty  =  p
```

Furthermore, a distributive law holds for a sequence operator and the alternative:

```
(p <|> q) *> r  =  (p *> r) <|> (q *> r)
p *> (q <|> r)  =  (p *> q) <|> (p *> r)
```

All binary combinators are assosiative, for instance

```
(p <|> q) <|> r  =  p <|> (q <|> r)
```

and there is also a neutral element

```
failure :: Parser a
```

for the <|>-Kombinator. `failure` which does not match any string, i.e. it parses the empty language.

By means of the presented technique, every context-free grammar can be expressed with parser combinators. A problem, however, are grammar, which contain left-recursion. Translating the following grammar for the language _a*_

```
AStar ::= AStar 'a'
        |
```

in parser-combinators

```
aStar = aStar *> char 'a'
    <|> empty
```

we obtain a non-terminating parser, applying the parser by the `parse` function:

```
ghci> parse aStar "aaa"
*** Exception: stack overflow
```

As a solution, it is possible, to eliminate left-recursion. For instance, we can transform the grammar to > aStar = char ‘a’ *> aStar > <|> empty

Now `parse` terminates for all inputs

```
ghci> parse aStar "aaa"
Just ()
```

In general, the elimination of left-recursion can be difficult, see lecture on compiler construction.

The class of of context-free grammars, which can be parsed with parser combinators is exactly the union of all *LL(k)* grammars for all natural numbers k. Hence, parser combinators allow an arbitrary look-ahead, which may however be inefficient in some situations. We will see this later, when we look at the implementation.

Now, lets investigate some further combinators, which allow the computation of further syntax information. The simplest is

```
yield :: a -> Parser a
```

`yield x` is a parser, which matches the empty word and returns `x` as parse information.

As a more complex example, we extend our parser for bracket expressions with the maximal nesting depth:

```
nesting :: Parser Int
nesting
  =  (\m n -> max (m+1) n)
      <$> (char '(' *> nesting <* char ')')
      <*> nesting
 <|> yield 0
```

```
ghci> parse nesting "((()(())))"
Just 3
ghci> parse nesting ""
Just 0
ghci> parse nesting "(()()"
Nothing
```

The operator `<$>` allows the application of a function to parser, and we already know this function from `Applicative`

In the definition of `nesting` we used two new combinators

```
(<$>) :: (a -> b) -> Parser a -> Parser b
```

applies a function to the result of a parser. From the parsing perspective, the parser stays unchanged, but yields a different result.

We now see, that parsers are applicative functors and the functions have already the same names. We obtain the following instances:

```
instance Functor Parser where
  fmap = (<$>)
```

with

```
id <$> p  =  p
f <$> (g <$> p)  =  (f . g) <$> p
```

and an applicative instance, where

```
pure = yield
```

and

```
(<*>) :: Parser (a -> b) -> Parser a -> Parser b
```

Let us consider another example for a parser with result, using the following combinators:

```
anyChar :: Parser Char
check   :: (a -> Bool) -> Parser a -> Parser a
```

The parser `anyChar c` reads a single character and returns this character. The function `check` modifies a parser such that it checks whether the parse result fulfils a predicate. If this is the case, the parse result stays unchanged. Otherwise the parser fails.

Hence we can express the `char` parser as follows:

```
char :: Char -> Parser ()
char c = check (c==) anyChar *> empty
```

The final `*> empty` is used for converting the type to `Parser ()`.

An advantage of parser combinators is, that we can define new combinators on top of existing parsers. For instance, we can easily define a combinator `many`, which repeats the application of another parser an arbitrary number of times.

```
many :: Parser a -> Parser [a]
many p  =  (:) <$> p <*> many p
       <|> yield []
```

Hence, this operator behaves like a Kleene star and a string matches a parser `many p` if it matches the empty string, or an arbitrary repetition of parser p. The parse results are returned as a list. Note, however, that it does not make sense to apply the combinator `many` to a parser, which also matches the empty string, since this parser could be applied again and again, resulting in an infinite list and hence, parsing will not terminate.

Using this parser combinator, we can define a parser for palindroms:

```
palindrom
  =  check (\ (u,v) -> u == reverse v)
  $  (,)
 <$> many anyChar
 <*> many anyChar
```

This parser first recognizes two arbitrary words `u` and `v` and then tests, whether `u` is the reverse of ‘v

```
ghci> parse palindrom "anna"
Just ("an","na")
ghci> parse palindrom "otto"
Just ("ot","to")
```

However, this parser does not recognise palindroms with an odd number of characters, which can be easily fixed as an exercise.

This example shows, that the class of languages recognisable by our grammars is quite large. Palindroms without marked middle are a context-free language. But they cannot be recognized by means of a deterministic push-down automaton. The language is inherent ambiguous. The reason is, that a push-down can not recognise the middle of the word and has to guess. However, our parser combinators are able to parse this language, but they search for the middle and the resulting parser is not very efficient.

Later we will however see that our parser combinators are also able to recognise languages which are not context-free.

Implementation

We now implement these parser in Haskell.

Since `parse` is the only function working with parsers, we can try to use the type of this function for the implementation:

```
type Parser a = String -> Maybe a

parse :: Parser a -> String -> Maybe a
parse p = p
```

However, this implementation is too simple, as we see, when we try to implement `(*>)`:

```
(*>) :: Parser a -> Parser b -> Parser b
p *> q = \s -> p s ??? q s
```

We cannot give a reasonable implementation, since the second parser may not be applied to the whole string, as the first parser. Instead, we need the information, how much input was consumed by the first parser. This reminds to the state parser and we try a similar type:

```
type Parser a = String -> Maybe (a,String)
```

Now, every parser yields in addition to the parse result, the remaining string, for further parsing. Hence, we have to modify the definition of `parse`:

```
parse :: Parser a -> String -> Maybe a
parse p s = case p s of
```

```
                Just (x,"") -> Just x
                _           -> Nothing
```

`parse` only yileds a result, if the parser consumes the whole input, i.e. the remaining string is empty. Now it is easy to define `*>`:

```
(*>) :: Parser a -> Parser b -> Parser b
p *> q = \s -> case p s of
                  Just (_,s') -> q s'
                  Nothing     -> Nothing
```

We ignore the result of the first parser and simply return the result of the second parser applied to the string remaining from the first parse.

Now, let us define the other combinators:

```
empty :: Parser ()
empty = \s -> Just ((),s)

char :: Char -> Parser ()
char x (c:cs) | x == c    = Just (c,cs)
              | otherwise = Nothing
```

For the definition of `<|>` for alternativs, we first parse with the first parser. If this parser fails, we try the second parser:

```
(<|>) :: Parser a -> Parser a -> Parser a
p <|> q = \s -> case p s of
                   Just xs -> Just xs
                   Nothing -> q s
```

However, this implementation does not fulfil the distributivity law, as we already expected, since we use `Mabye` as the incorrect `MonadPlus` instance:

```
test1  =  (empty <|> char 'a') *> char 'b'

test2  =  (empty *> char 'b')
      <|> (char 'a' *> char 'b')
```

The first parser does not recognise `"ab"`, while the second does.

```
ghci> parser test1 "ab"
Nothing
ghci> parser test2 "ab"
Just ()
```

To obtain a correct search, we have to use a correct type, which correctly implements the `laws` for `MonadPlus`, which is for instance the list type:

```
type Parser a = String -> [(a,String)]
```

The continuation based variant of Maybe `CMaybe` would also work.

Using lists, a parser can produce several results and for each result, a different part of the input can be consumed, i.e. every result has its own remaining string.

As a consequence the `parse`-function has to test whether there is a result, which returns the empty string. This can also be another than the first parser result within the list.

```
parse :: Parser a -> String -> Maybe a
parse p s = case filter (null.snd) $ p s of
              (x,_):_ -> Just x
              _       -> Nothing
```

If there exist more than one successful result, we simply return the first one.

Reengineering the definitions, we obtain:

```
empty :: Parser ()
empty = \s -> [((),s)]

char :: Char -> Parser ()
char x (c:cs) | x == c = [((),cs)]
char x _               = []

anyChar :: Parser Char
anyChar []     = []
anyChar (c:cs) = [(c,cs)]

check :: (a->Bool) -> Parser a -> Parser a
check ok p = filter (ok . fst) . p

char :: Char -> Parser ()
char c = check (c==) anyChar *> empty

(*>) :: Parser a -> Parser b -> Parser b
p *> q =
  \s -> [ xs | (_,s') <- p s, xs <- q s' ]
```

This can be generalised by the operator `(<*>)`:

```
(<*>) :: Parser (a->b) -> Parser a -> Parser b
p <*> q =
  \s -> [ (f x,s2) | (f,s1) <- p s,
                     (x,s2) <- q s1 ]
```

and we also have

```
(<*) :: Parser a -> Parser b -> Parser a
p <* q = const <$> p <*> q
```

`<$>` can be defined by `<*>`:

```
(<$>) :: (a -> b) -> Parser a -> Parser b
f <$> p = yield f <*> p

yield :: a -> Parser a
yield x = \s -> [(x,s)]
```

Using these combinators becomes clear from the following example:

```
ghci> let c = yield const
ghci> :t c
Parser (a -> b -> a)
ghci> let a = c <*> anyChar
ghci> :t a
Parser (b -> Char)
ghci> let ab = a <*> anyChar
ghci> :t ab
Parser Char
ghci> ab "abc"
[('a',"c")]
```

The `<*>` combinator successively applies the function `const` to the results of the two `anyChar` Parsers. So the result is the symbol `'a'` with remaining input `"c"`.

It remains the definition of the combinator `<|>` for the definition of alternatives within the grammar. `<|>` applies both parsers to the input an appends their results.

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

The following example shows the results of it's application:

```
ghci> (empty <|> char 'a') "abc"
[((),"abc"),((),"bc")]
```

This parser either returns `()` without consuming a symbol from the input or reads an `'a'` from the input and returns the remaining string `"bc"`.

The parser `failure` behaves neutral with respect to `<|>` since it does not yield a result:

```
failure :: Parser a
failure _ = []
```

The definition of `<|>` ising lists now (in contrast to the `Maybe` version) fulfils the distributivity law. Hence, a parser tries all possible matchings by means of backtracking. This is, for instance, mecessary for the palindrom parser, which did not work in the `Maybe` variant.

Unfortunately, backtracking can result in efficiency problems in some sitations. As a solution, we provide an alternative alternativ combinator `<!>`, which does not introduce backtrack points for every possible mathcing of the first parser. Like the `Maybe` parser it only executes the second parser if the first parser did not succeed:

```
(<!>) :: Parser a -> Parser a -> Parser a
p <!> q = \s -> case p s of
                  [] -> q s
                  xs -> xs
```

Like `<|>` the operator `<!>` is also associative with neutral element `failure`. Instead of the distributivity law it holds, that:

```
yield x <!> p  =  yield x
```

In practice you usually use this operator, when you are sure that the second alternative will not succeed if the first one succeeds. Hence in non-overlapping cases.

The following parser for binary number in LSB representation (least significant bit first) demonstrates once again the usage of the defined combinators:

```
binary  =  (\b n -> 2*n + b) <$> bit <*> binary
       <!> yield 0

bit  =  char '0' *> yield 0
    <!> char '1' *> yield 1
```

The `binary` parser reads sequences of zeros and ones and computes the corresponding value of the binary number. For this, we apply the function `(\b n -> 2*n + b)` by means of `<$>` and `<*>` to the results of the parsers `bit` and `binary`. If no symbol is left, `binary` returns zero. These two alternatives exclude each other in this example and we use `<!>` instead of `<|>`.

Here are some example calls:

```
ghci> parse binary ""
Just 0
ghci> parse binary "0"
Just 0
ghci> parse binary "1"
Just 1
ghci> parse binary "10"
Just 1
ghci> parse binary "01"
Just 2
ghci> parse binary "110"
Just 3
ghci> parse binary "1101"
```

```
Just 11
ghci> parse binary "010101"
Just 42
```

A simple generalisation of the parser type is to use arbitrary lists of token instead of Strings:

```
type Parser tok a = [tok] -> [(a,[tok])]
```

All definition can easily be transferred. In practice this is especially useful, if the parse process is divided into a scanner and a parser (see Compiler construction).

So far we defined the parser type as a type synonym. Instead we should (like in many other cases before) use a `newtype` definition.

Parsing non-context-free languages

For parsing non-context-free languages, we can use different approaches. First of all, it is possible to parse the language $a^*b^*c^*$ and return (as some kind of AST) the number of the occurences of the different letters. Then we can use the `check` function to compare the results and finally project to one of these, to return the $n§:

```
anbncn :: Parser Int
anbncn =
  (\(x,_,_) -> x) <$>
 (check (\(n,m,k) -> n==m && m==k) ((,,) <$> star 'a' <*> star 'b' <*> star 'c'))
```

Here a `star` parser behaves similar to `many (char c)`, but returns the number of occurences instead of a list of `()`s:

```
star :: Char -> Parser Int
star c = (+1) <$> (char c *> star c)
     <|> yield 0
```

Anoth approach is the construction of a parser, which has an infinite number of rules. Therefore, we start with an enumeration of all accepted words:

epsilon | abc | aabbcc | aaabbbccc | ...

Simply enumerating all these rules would on the one hand produce a correct parser and all words of the form $a^nb^nc^n$ would be recognized. However, for words with different numbers of as, bs and cs, the parser would not terminate. The parser never knows, that all alternatives will only match longer words.

On the other hand, it is possible to define an equivalent (infinite) regular expression, which make the decision whether wether we continue parsing in dependence of the first letter beeing an

$$a$$

or

$$b$$

:

epsilon | a(bc | a(bbcc | a(bbbccc | a(bbbbcccc | . . .))))

This can be realized by means of our parser combinators as follows:

```
anbncn2 :: Parser Int
anbncn2 = let bs n = foldr (*>) (yield n) (replicate n (char 'b'))
              cs n = foldr (*>) (yield n) (replicate n (char 'c'))
              bcs n = bs n *> cs n
              abc n = char 'a' *> ((bcs n) <|> abc (n+1)) in
          yield 0 <|> abc 1
```

A third, even more elegant approach is first parsing the context-free language $a^n b^n$ and then construct an appropriate c^n parser. Hence, the c^n parser depends on the parse result of the first parser, which leads us to monadic parsers.

Monadic Parser

First of all we need a sequence operator for parser, which parameterizes the second parser by the parse result of the first parser. The combinator `*>=` passes the result of one parser to a function which yields another parser:

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

Using this operator, all parsers can be defined and the monadic parser is a simple extension of the previously defined state parser. For the the palindrome example, this looks like this:

```
palindrome =
  many anyChar *>= \u ->
  (empty <|> anyChar *> empty) *>
  word (reverse u)
```

After parsing the first half of the palindrome and returning the `String` as the monadic result, we construct a parser which expects the revers of this word.

```
word :: String -> Parser ()
word []     = empty
word (c:cs) = char c *> word cs
```

For palindromes of odd length it is necessary to conditionally accept one additional letter in the middle. Hence, we should add the rule `(empty <|> anyChar *> empty) *>` to its definition. Here the part `*> empty` is only necessary to construct the same type to both alternatives.

Since, the `word` combinator yields `()`, also the palindrome parser yields `()`:

```
ghci> parse palindrome "anna"
Just ()
ghci> parse palindrome "rentner"
Just ()
ghci> parse palindrome "hans"
Nothing
```

The combinator `*>=` has exactly the type of the monadic bind operator `>>=`. Furthermore, `yield` relates to `return`. Also the monad laws are fulfilled for the parser combinators and we can make parsers and instance of class `Monad`:

```
instance Monad Parser where
  return = yield
  (>>=)  = (*>=)
```

This instance allows the definition of parsers using the `do` notation. For instance, the parser for arbitrary nested brackets can be defined as follows:

```
nested
  =  do char '('
        nested
        char ')'
        nested
 <|> empty
```

Parser which are anotated after each other in the `do` block will be executed sequentially. The results of each parser can be accessed by using the left arrow :

```
nesting
  =  do char '('
        m <- nesting
        char ')'
        n <- nesting
        return (max (m+1) n)
 <|> return 0
```

Using `do` it is simple to access exactly the parser results we are interested in. Using special combinators like `*>` and `<*` is not necessary.

Finally, we can also define the palindrome parser in `do` notation:

```
palindrom = do u <- many anyChar
               empty <|> anyChar *> empty
               word (reverse u)
        <|> empty
```

It remains to give the implementation of the combinator `*>=`:

```
(*>=) :: Parser a -> (a-> Parser b) -> Parser b
p *>= f =
  \s -> [ (y,s2) | (x,s1) <- p s,
                   (y,s2) <- f x s1 ]
```

The result `x` of the parser `p` is passed to the function `f`. This application yields the second parser, which can be applied to the remaining input string.

The combinator `*>=` is the most powerful combinator. All other combinators can be expressed using this combinator. This relates the hierarchy between Functor, Applicative Functor and Monad. For instance the operator `<*>` can be defined as follows:

```
p <*> q  =  p *>= \f -> q *>= \x -> yield (f x)
```

As a further application of the monadic parser combinar, we again consider the language $a^n b^n c^n$.

```
abcn3 = do
  n <- anbn
  foldr (*>) empty (replicate n (char 'c'))
  return n

anbn = yield 0
   <|> (+1) <$> (char 'a' *> anbn <* char 'b')
```

Applikative Functor

We have already seen that `<$>` is the function `fmap` from the `Functor` class. Furthermore, `yield` and`*> =`correspond to the `Monad` operations`return` and `>>=`. Also `<*>' and its variants are abstracted in class`Applicative‘:

```
class Functor f => Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
  (*>)  :: f a -> f b -> f b
  (<*)  :: f a -> f b -> f a
```