Principles of Programming Languages 2017W, Functional Programming

Assignment 2: Parsers (4 points)

Parser combinators are a classical example from functional programming. Parser combinators allow building parsers as higher-order functions and by function composition. In this assignment your task is, first, to study a simple parser combinatory implementation and then implement some parsers. In Assignment 3 then we will use the parser combinators for implementing a parser for a simple Lisp-like language.

In the following, the parser combinator solution is explained step-by-step.

Parser as functions

A parser is a *function* that takes a list of tokens (which are Strings, compare Assignment 1) and returns a pair of a `Maybe` of some polymorphic type `a` and a list of remaining tokens. The list of remaining tokens is supposed to be used for continuing parsing. We define a type synonym `Parser a` for parsers which are functions from a list of tokens to a pair with a possible parse result and the remaining list of tokens (recall that type synonyms are fully equivalent type names):

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

That means a parser can be applied to a list of tokens and if it succeeds, will return a parse result in a `Just` and the rest of tokens remaining, or if the parser fails, will return `Nothing` and the original list of tokens. The following example shows how to use a parser, i.e., apply it to a list of tokens and then distinguish if it succeeded or failed:

```
parser :: Parser …
parser = …
case parser tokens of
  (Just result, restTokens) ->  -- handle parser succeeded
  (Nothing, tokens)         ->  -- handle parser failed
```

As an example, let us define a parser `anyToken` which accepts any token. If the input list contains at least one token, it will return the first token as the parse result. If the input list is empty, it will return (Nothing, []).

```
anyToken :: Parser String
anyToken (tkn:tkns)   = (Just tkn, tkns)
anyToken []           = (Nothing, [])
```

Here is the application of the parser to a list of tokens:

```
> anyToken ["Hello", "World"]
(Just "Hello",["World"])
```

Elementary combinators

Parsers are mainly built by combining simpler parsers. Recall that parsers are functions of type `[String] -> (Maybe a, [String])`. That means combining parsers is a sort of function composition.

For combining parsers we first need a *trivial* combinator `result` which will be used to return final parse results. The function result gets a value `x` of some type `a` and it a parser, thus a function. The

created parser then when applied will return the given value wrapped in a `Just` plus the original list of tokens unchanged.

```
result :: a -> Parser a
result x = \tkns -> (Just x, tkns)
```

For example with

```
result True
```

one gets a parser function which returns the `True` in a `Just` and the given tokens `tkns`

```
\tkns -> (Just True, tkns)
```

Thus, the type of `result True` is `Parser Bool`. The following example shows the application to tokens `["B","C","D"]`:

```
> (result True) ["B","C","D"]
(Just True, ["B","C","D"])
```

The most important combinator is `andThen` which creates a parser by putting two parsers in series. The created parser will first apply the first parser `p` which gives a first result. If this parser succeeds a function `fn` is applied to the parse result `x`. This function returns a parser which is the result of the combination. However, if the first fails, `Nothing` and the list of tokens is returned immediately. Note that the application of `fn` to `x` gives a parser (!) which is applied to the rest of tokens only if the first succeeds.

```
andThen :: Parser a -> (a -> Parser b) -> Parser b
andThen p fn =
    \tkns ->
        case p tkns of
            (Nothing, _)    -> (Nothing, tkns)
            (Just x, rest)  -> (fn x) rest   -- (fn x) returns a Parser b
```

Let us now use `result` and `andThen` for building a parser which accepts two tokens and combines them into one result string:

```
twoTokens  :: Parser String
twoTokens  = anyToken `andThen`
                  \t1 -> anyToken
                         `andThen`
                               \t2 -> result (t1 ++ t2)
```

Here is the application of the combined parser:

```
> twoTokens ["Hello", "World"]
(Just "HelloWorld", [])
```

Mapping and filtering

We allow mapping and filtering of parsers. A `mapTo` function has a parser `p` and a function `fn` as arguments and returns a parser which applies `p` and then returns the mapping of the parse result.

```
mapTo :: Parser a -> (a -> b) -> Parser b
mapTo p fn  = p `andThen` \a -> result (fn a)
```

Note that if parser `p` fails, `Nothing` will be returned immediately and the function `fn` is not applied.

Using `mapTo` we can for example map the parse result from `anyToken` to the length of the token. This gives a parser of type `Parser Int`.

```
anyTokenLength :: Parser Int
anyTokenLength = anyToken `mapTo` length
```

Here is the application of `anyTokenLength`:

```
> anyTokenLength ["Hello", "World"]
(Just 5,["World"])
```

Similarly, a function `filterBy` takes a parser `p` and a predicate `pred` as arguments. The result of `filterBy` is again a parser which works by first applying parser `p`. If this fails, `(Nothing, tkns)` is returned immediately. If the parser succeeds, the predicate `pred` is used to test the result. If the test fails, `(Nothing, tkns)` is returned, otherwise the initial parse result and the list of remaining tokens is returned. Note that if the predicate test fails, the original list of tokens is returned.

```
filterBy :: Parser a -> (a -> Bool) -> Parser a
filterBy p pred =
    \tkns ->
        case p tkns of
            (Nothing, rest) -> (Nothing, tkns)
            (Just x, rest)  -> if pred x then (Just x, rest)
                               else (Nothing, tkns)
```

We can use `filterBy` for writing a function which creates parsers of specific tokens. That means function `token` has a string `tkn` as parameter and will return a parser of type `Parser String` which uses `parseAnyToken` to parse any token and then filters those which are equal to `tkn`:

```
token :: String -> Parser String
token tkn =  anyToken `filterBy` (== tkn)
```

Using `token` we can now define parsers for just parsing opening and closing parenthesis tokens. Write parser `open` and `close` using `token` which parse opening and closing rounded bracket tokens.

```
open  :: Parser String
open = -- TODO

close :: Parser String
close = -- TODO
```

Similarily, we can define a parser `word` which accepts any token except opening and closing bracket tokens. Write a parser `word` using `filterBy` and `anyToken` which parses any token except opening and closing bracket tokens.

```
word :: Parser String
word = -- TODO
```

More combinators

We introduce more complex combinators for building parsers. The parser combinator `multiple` supports 0 to n parses of a parser `p`. The combined parser returns a list of parse results in a `Just`. The parser combinator `multiple` is recursive, applying the parser `p` until it fails. Upon failure, an empty list is returned as parse result, thus `multiple` always succeeds.

```
multiple :: Parser a -> Parser [a]
multiple p =
    \tkns ->
       let
```

```
        (mbP, pRest) = p tkns
    in
        case mbP of
            Nothing  -> (Just [], pRest)
            Just pR   ->
                case multiple p pRest of
                    (Just results, rest)  -> (Just (pR:results), rest)
                    (Nothing, rest)       -> (Nothing, rest)
```

The following definition shows the application of `multiple` for defining a parser which parses 0 to n tokens `"A"`:

```
multipleAs :: Parser [String]
multipleAs = multiple (token "A")
```

Here is the application of parser `multipleAs`:

```
> multipleAs ["A", "A", "A", "B", "C"]
(Just ["A","A","A"],["B","C"])
```

Another combinator is `orElse` which builds an or-combination of two parsers `p` and `q`. That means, first parser `p` is applied and, if it succeeds, the result is returned. If not, parser `q` is applied and its results is returned. Write parser combinator `orElse`:

```
orElse :: Parser a -> Parser a -> Parser a
orElse p q = undefined -- TODO
```

The following example shows how `orElse` is used to define a parser which accepts a token `"A"` or `"B"`:

```
aOrB = (token "A") `orElse` (token "B")
```

The following parser then accepts a series of `"A"`s or `"B"`s:

```
multipleAOrBs = multiple ((token "A") `orElse` (token "B"))
```

Here is the application of parser `multipleAOrBs`

```
> multipleAOrBs ["A", "B", "B", "A", "C", "D"]
(Just ["A","B","B","A"],["C","D"])
```

Parser for arithmetic expressions

Using the basic parsers and the parser combinators, your task is now to define a parser which accepts a list of tokens and creates a concrete syntax tree. The concrete syntax tree is defined by a data definition as follows:

```
data CST =
          Atom String
        | Op String [CST]
      deriving (Eq, Read, Show)
```

That means a `CST` is a type with two variants, either an `Atom` which has a field of type `String`, or an operation `Op` with a field of type `String` for the operator and a list operands which are again elements of type `CST`.

A parser `atom` is defined for parsing `Atom`s from word tokens. Recall, that the parser `word` from above parses word tokens. The result is mapped by `mapTo` to `Atom` elements:

```
atom :: Parser CST
atom = word `mapTo` (\w -> Atom w)
```

Define a parser `operation`

```
operation :: Parser CST
operation = -- TODO
```

for parsing `Op` elements from tokens using combinators `andThen`, `multiple`, `orElse`, `result`, `token` etc. as follows:

- first parse an open bracket
- then parse the operator which can either be a `"+"`, `"*"`, `"-"`, or `"/"` (use `orElse` and `token`)
- then parse a list of operands (use `multiple` and `expr` from below)
- then parse a closing bracket using `close`
- finally construct the resulting `Op` element

As a final step define a parser `expr`

```
expr :: Parser CST
expr = -- TODO
```

by an or-combination of `atom` and `operation`.

Use your parsers and tokenizer (from Assignment 1) for parsing arithmetic expressions in prefix notation and constructing concrete syntax trees. For example, input string

```
"( + x (- x))"
```

should give tokens

```
["(","+","x","(","-","x",")",")"]
```

and parse result

```
(Just (Op "+" [Atom "x",Op "-" [Atom "x"]]),[])
```