Introduction to Parsing and Parser Combinators with Haskell

Sunny Miglani
sm15504

Abstract

The aim of this essay is to introduce the process of Parsing generally, and in Haskell. Haskell is chosen due to its various available Parsing Libraries and its type system that allows the programmer create their own data types making it easier to have user defined Data Structures which are useful in the process of creating Parsers.

This essay shall act as a summary of the lecture series on Parsing given by Dr Nicholas Wu at the University of Bristol. The other aim of this is to create a summarised explanation of common Parser Combinators which are often described in a more scattered manner. To explain the various Parser combinators, an example Parser is used for an altered version of the *"while"* language[1] which can be found in the appendix of the essay.

At the end of this essay, the reader should understand Parsing and how to apply the knowledge gained to Haskell specifically, using Parser Combinators.

Parsing

Parsing is the process of analysing a string of symbols and keywords which follow a certain pre defined grammar. This process involves splitting the words into a set of internal symbols and tokens to create a tree structure (an Abstract Syntax Tree), which shows the relations between these tokens and symbols. This tree is optimised and converted into a set of instructions that a computer can understand.[2]

The process of parsing involves different kinds of smaller pieces of software, such as the Lexical Analyser, which converts the input string into a table with related internal symbols and keywords. This table is used as a reference by the "Syntactic Analyser" to create the actual tree. It also optimises the tree to allow faster running code with less memory consumed.The Lexical and Syntactic Analyser together, create the "Parser".

The "Translator" converts the tree into the instructions that a computer would follow. The input given into these softwares usually refers to a computer program, and the "grammar" refers to the rules the program's language has to follow and acts as the syntactical rules of the language.

Parsing and Haskell

A good way to explain the process of parsing is with the use of Haskell's type system. It allows the user to create their own data types which makes it easier to build an Abstract Syntax Tree for the Translator. Below is the general definition of a "Parser" in Haskell, and what it means to "Parse" an input.

```
> data Parser a = Parser (String -> [(String,a)])
              -- on the RHS of the Arrow
              -- String = Unparsed bits of String
              -- 'a' is the "parsed" bits of the initial input String

> parse :: Parser a -> (String -> [(String,a)])
> parse (Parser p) = p
```

The type "Parser" is a way of explaining to Haskell that a Parser takes in an input (a string) and creates a set of tuples of the unparsed value and the parsed value. Here, the *parsed* value is referred to with the type *'a'* which is the same as the type of the Parser it's created from. The *unparsed* value is the same type as the input, which in Haskell is a String.

The function *"parse"* unwraps the function "*String -> [(String,a)]"* from the type "Parser" and allows other functions to alter the data inside it.
Throughout this report, the words "*Parse*" and "*Parser*" will refer to these functions. An important fact to note is that in various Parser Libraries in Haskell (Such as Parsec and ReadP which shall be explained later) these functions may be defined differently to allow slightly more complex actions, but work on the same principle.

Parsing Libraries

There are many parsing libraries for Haskell, the report shall use "Parsec", "MegaParsec" and "ReadP" functions to explain Parsing and Parser Combinators. Each of these parsing libraries have similar functions and types which makes it easier to use these concepts regardless of the libraries being used.

These libraries are focused on Parser Combinators, which are higher order functions that take in two or more Parsers and create a new Parser. The Parser combinators in Haskell's library are usually instances of Monads, Applicatives or Functors which are particularly altered to be used for Parsers. The following section will explain Parsers and Parser combinators in more detail.

Parsers and Parser Combinators [3]

The first example of a Parser shall be "*failure*" which shows that a Parser need not always parse a result correctly and can sometimes fail.

```
> failure :: Parser a
> failure = Parser (\cs -> [])
```

The empty list donates a null list of (String,a) combinations and is therefore a "failed parse" as it was unable to create a type *'a'* from the input given to it. This Parser is meant to fail, and is slightly different to a Parser which actually fails, which would throw an error if it receives an incompatible type.

The next Parser is "*produce*" which creates a Parser from any given input, and allows us to combine various types and create Parsers for them.

```
> produce :: a -> Parser a
> produce x = Parser (\ts -> [(ts,x)])
```

The following Parser is more true to the previously described system of a Parser and is meant to extract or "parse" the first character of an input

```
> item :: Parser Char -- "Parses" the first character in a string
> item = \inp -> case inp of
>                  []     -> []
>                  (x:xs) -> [(xs,x)]
```

This Parser "parses" the value '*x*' and keeps '*xs*' (the remainder of the input string) as the "unparsed" value. This is a relatively simple Parser and doesn't truly represent the complexity of a Parser used to parse actual data.

The next example is more complicated than the previous ones but introduces many important topics into this report. The Parser is going to be created for the data type "*Term*" whose grammar is given in BNF form below. (All these terms shall be explained after giving the example)

BNF Representation [3]

term :: = number | '(' expr ')'

```
> data Term = Number Int
>           | Parens Expr
```

The Parser for this datatype is :

```
> term :: Parser Term
> term = (Number <$> number) <|> (Parens <$ tok "(" <*> expr <* tok ")")
```

The above definitions seem extremely confusing but can be simplified after explaining a few terms. First, BNF Representation refers to the "Backus-Naur Form" which is the

common notation used to represent a grammar for any language. In this case the language is of the type "*Term*". According to this grammar, a '*Term*' can either be a '*Number Int*' or an '*Expr*' in Parenthesis (Expr leads to the choice of Term again and is therefore recursive).
The symbols "<$>" , "<|>", "<$","<*" are the Parser Combinators mentioned previously; these are the notations used by Haskell to identify the combinators. Each shall be defined and explained in the next section with examples.

Parser Combinator Definitions [3]

The <|> Combinator : The Associative Binary Operation

```
(<|>) :: Alternative f => f a -> f a -> f a
```

The <|> combinator is part of the Alternative class, which is a "monoid" on the Applicative Functors.

The Alternative class is defined as

```
> class Applicative f => Alternative f where
> empty :: f a
> (<|>) :: f a -> f a -> f a
```

Since Alternative is a Monoid, it follows the monoid laws [4]:

```
> empty <|> x = x
> x <|> empty = x
> (x <|> y) <|> z = x <|> (y <|> z)
```

Given below is an implementation of the Alternative class with Maybe, which is easier to explain

```
instance Alternative Maybe where
empty = Nothing
Just x <|> q = Just x
Nothing <|> q = q
```

Here "<|>" can be thought of the "choice" symbol, which allows the code to choose between the different options based on if the options lead to a failure. In this particular example, a failure is considered "*Nothing*" and due to the order of precedence in the operator (left precedence) the output is based on the result of the LHS. If he LHS returns a value which is not "*Nothing*" then we give the output of the operator as that value regardless of the RHS, but if we fail in the LHS, the output is the RHS regardless of the output of RHS failing or passing.

Explanation of <|> with relation to Parsers is given below; this is the definition which shows how the function handles parses.

```
instance Alternative Parser where
  empty = failure
  (<|>) = orElse

orElse :: Parser a-> Parser a -> Parser a
orElse (Parser px) (Parser py) = Parser ( \ts ->
       case px ts of
           [] -> py ts
           xs -> xs)
```

A breakdown of this code would be :

The combinator takes in 2 Parsers to create a new third Parser. It chooses between the initial two Parsers by looking at the result given to it from the left Parser applied to the input. Here the lambda function gives the input as *"ts"*, and the "case of" *px ts* gives us the output from the application of Parser '*px*' on input *'ts'*. This on returning an empty set automatically means that the output result of this combinator is the result of the applying Parser *'py'* on input *'ts'*. In case the initial Parser does not return an empty list but returns a list represented by the variable '*xs*'; the output for this combinator is automatically *'xs'*

The type of output remains as a Parser due to the lambda function acting as the "function" inside the Parser.

The most common use of this Parser Combinator would be for the choice of a Parser when the data type being parsed could have many different translations.
For example for the BNF grammar data type :

```
b ::= true | false | not b | b opb b | a opr a
```

Converted into Haskell's data structure :

```
> data BType = BoolC Bool | Not BType | OpBExpr OpB BType BType | OpRExpr OpR AType AType
```

We say that the Parser "*bType*" could parse true for any of the Parsers given (all separated by a <|>)

```
> bType :: Parser BType
> bType = BoolC <$> boolP
>       <|> Not <$> bType
>       <|> OpBExpr <$> opB <*> bType <*> bType
>       <|> OpRExpr <$> opR <*> aType <*> aType
```

This holds the order of precedence as top down (or left to right if given horizontally).

The <$> Combinator:

The <$> is generally defined as :

```
(<$>) :: Functor f => (a -> b) -> f a -> f b
 fmap :: (a->b) -> fa -> fb
```

<$> and fmap are very similar in their definitions, *fmap* takes in a function that changes a value of type *'a'* into type *'b'* and a functor of type *'f a'* then changes all the values in the functor into type *'f b'*s.

Due to the similarity of the definitions of fmap and <*$*>, we can say that they do the same functions. In Fact,we can write:

```
> f <$> xs  instead of writing  > fmap f xs
```

This combinator also has a variant (<$) which is defined as :

```
> (<$) :: Functor f =>  a -> f b -> f a
```

Here the combinator takes in an input value and another value wrapped in a functor 'f' and replaces the value in the functor with the value 'a'. This can also be thought of as it wraps the functor 'f' around the value 'a' and ignores 'f b'

```
> ($>) :: Functor f => f a -> b -> f b has a similar application.
```

The application of "<$>" is quite simple, as it can just replace any use of "fmap", but the use of "<$" is more complicated and is shown and explained below:

```
> opB :: Parser OpB
> opB = AND <$ tok "&"
>       <|> OR <$ tok "|"
```

Here the symbols "<$" map the type '*AND'* over the result of the Parser '*tok'* (where *AND* has the type *OpB*) allowing us to give the ouput in the correct type as needed by Haskell (Type *OpB*)

This is often used just to wrap functors around data so that it's easier to use and pass around through the program.

The <*> Combinator :

The Applicative class in Haskell refers to Control.Applicative which acts as an intermediate between a Functor and a Monad, and is more widespread amongst Haskell users due to its advantages such as:

- The availability of more instances (increases usability) [5]
- Instances of "Applicative" can perform analysis of computations before they are executed and thus are more optimized

The Applicative functors are : (minimal complete definition)

```
> pure :: a -> f a
> (<*>) :: f (a->b) -> f a -> f b
> (*>) :: f a -> f b -> f b
> (<*) :: f a -> f b -> f a
```

Pure wraps the given data into the type of the instance of Applicative

From the definition of the (<*>) operator we can get a general idea of how it works. The definition if the operation is very similar to the definition of the <$> combinator, but the key difference is the fact that <*> inputs only functors *"f (a->b)"* whereas the <$> inputs any function of type *(a->b)*. This creates the difference of where to use and apply the <*> combinator.

For the use of the <*> operator with Parsers we define it as :

```
> instance Applicative Parser where
> -- pure :: a -> Parser a
> pure p = Parser p
> -- Parser a <*> Parser b = Parser a is applied then p of b is applied then
> --a Parser is returned
> Parser pf <*> Parser px = Parser (\ts ->
>                           [(ts'',f x) | (ts',f) <- pf ts,
>                                         (ts'',x)<- px ts'])
```

The <*> combinator creates a new Parser with a combination of *"pf"* and *"px"* by taking in an input *"ts"* and applying Parser *'pf'* on it to create values *ts'* and *'f'*. Then *px* is applied to *ts'* to create a tuple *(ts'', x)*. A combination of *(ts'', f x)* is given out as the result of the Parser created by <*>.It allows us to sequentially apply two Parsers on one set of data without seemingly losing any data.

The application of this in our Parser for the "while" language is as seen below :

```
> aType :: Parser AType
> aType = Var <$ tok "VAR" <*> stringP
>        <|> IntCon <$ tok "IntCon" <*> intParser
>        <|> Neg <$ tok "Neg" <*> aType
>        <|> OpAExpr <$ tok "OpAExpr" <*> opA <*> aType <*> aType
```

Here we see a combination of all previously mentioned Parsers such as the <|> , <$ and now we see the application of <*> in the final line given above. This allows us to parse the input through 3 different Parsers and would allow us to separate the tokens through the input for each of the corresponding Parsers.
Example of this would be providing an input of type '*opA*', and two of type *'aType'*. The final output of the Parser would be the separated tokens of types *'opA'* and types '*aType*'

The "many" and "some" Combinators

```
> many :: Alternative f => f a -> f [a]
> some :: Alternative f => f a -> f [a]
```

Many is a combinator that parses *zero* or more occurrences
Some is a combinator that parses *one* or more occurrences.

The combinators such as “*many*” and “*some*” are one of the few mutually recursive Parser combinator pairs that are defined in terms of each other. This can cause confusion when attempting to understand them. Similar mutual recursion is seen in simpler functions such as “*even*” and *“odd”*

```
> some :: Alternative f => f a -> f [a]
> some :: Parser a -> Parser [a]
> some v = (:) <$> v <*> (many v)


> many :: Alternative f => f a -> f [a]
> many v = some v <|> produce []
```

The best way to understand some and many is to look at the base case of this mutual recursion which is in the “*many v”* definition as “*produce []*”.

Run through the last but one iteration of the recursion, and we see that “*some v*” calls the base case in “*many v*” which creates an empty list (wrapped with a Parser). This is then “returned” to some V which maps the Parser “v” onto the empty list and is added to the list with the “(:)” operator.

(:) operator is meant to be “Cons” and is meant to add values into a list.

An example of the use of “many” and “some” in the While language is given below :

```
> stringP :: Parser String
> stringP = some (noneOf ("\n\r\"={}[],")) <* whitespace
```

This Parser parses an input of type “String”. It makes sure that the characters entered as the input are not key characters mean to be next line characters or parenthesis.

The use of some in this Parser is to run the Parser “*noneOf*” multiple times until it fails to parse, this allows the input to run through all the characters in a string. Applying noneOf without the use of “*some*” would parse only one character in the string and would leave the rest unparsed.

Bind : (>>=)

```
> bind :: Parser a -> (a-> Parser b) -> Parser b
> Parser px `bind` f = Parser (\ts ->
>                     concat[ parse (f x) ts’ | (ts’,x) <- px ts])
```

Bind first applies Parser ‘*px’* onto input *“ts”* to gain a value *(ts’,x)*. The *‘x’* from this value is used to create a Parser *(f x)* which is applied to *“ ts’ “*. This occurs for every parsed value from px ts. The bind combinator concatenates all these values together into a list. Which ends up in the form of [(a,String)] which is the form of output of a Parser. It uses the

lambda function) of Haskell to create the form String -> [(a,String)] which means that it creates the new Parser, Parser b.

Bind can be thought of as a very complicated “;” symbol from Java or C. It conceptually tells the compiler to allow the values before the symbol to be available after the symbol as well. In languages like C or Java, the variables are stored throughout a lot of lines of code unless defined otherwise (local variables, private variables etc). This is allowed by the “;” which tells the compiler that the data and the variables they’re represented by should be available in the next part of the program. It can be loosely thought that this is the application of “bind” in general (not specific to Parsers) that allows the data given to be used by multiple parts of a program.

The application of “bind” in general and Parsers is shown in the function “satisfy” :

Satisfy:

```
> satisfy :: (Char -> Bool) -> Parser Char
> satisfy p = item >>= (\c ->
>   if p c then
>    produce c
>    else failure)
```

The satisfy function takes in an input and only parses it if the given predicate (char->Bool) returns true once applied to the character. The use of this is shown in the following two combinators.

The “>>=” symbol here uses the output or result from the lambda function (\c->...) as the input for the function “item”. It binds the lambda function the input of “item”

The “noneOf” Combinator:

The noneOf combinator is defined as :

```
> noneOf :: [Char] -> Parser Char
> noneOf cs = satisfy (\c -> not (elem c cs))
```

noneOf isn’t exactly a combinator since it doesn’t take in any Parsers as input, but creates a Parser that is run on an input. The function takes in a list of characters (in the form of a string which is seen as the same in Haskell) and checks if the character is part of the given input (elem c cs), the return type of this check is then flipped using the “not” keyword and is then applied to satisfy and the result of satisfy is the output of the function.
As an example input, we take the use of noneOf in the “while” language Parser.

```
> stringP :: Parser String
> stringP = some (noneOf ("\n\r\"={}[],")) <* whitespace
```

Here the Parser takes in an input string and checks to make sure none of the characters in the string are reserved characters such as “\n” for newline, and “{}[]” for parenthesis.

Assuming this Parser *stringP* is applied to an input "SuperUltraTrueFriends" we can see that none of these letters are any of the reserved letters.
Matching *"S"* against the input string, we get a false from *"elem"* and a true from *"satisfy"* and so our final output returns true and the character is "parsed", this applied with the help of *"many"* can parse a large string.

A similar *"oneOf"* combinator is used to confirm that a character is part of a given defined reserved character list.

Haskell for Parsing

To show off the power of Haskell, the appendix of this report contains a whole Parser and data structure for the following BNF. The data structure has also been included to give an idea about how easy it is to create this in Haskell

BNF Form:

```
a ::= string | num | -a | a opa a
b ::= true | false | not b | b opb b | a opr a
opa ::= + | - | * | /
opb ::=  and | or
opr ::= > | <
S :: = string := a | skip | S1;S2 | (S) | if b then S1 else S2 | while b do S
```

Data Structure for the Tree in Haskell

```
> data AType = Var String | IntCon Integer | Neg AType | OpAExpr OpA AType AType deriving
(Show)
> data BType = BoolC Bool | Not BType | OpBExpr OpB BType BType | OpRExpr OpR AType AType
deriving (Show)
> data OpA = (:+:) | (:-:) | (:*:) | (:/:) deriving (Show)
> data OpB = AND | OR deriving (Show)
> data OpR = (:<:) | (:>:) deriving (Show)
> data Stmnt = Assign String AType
>            | Seq [Stmnt]
>            | If BType Stmnt Stmnt
>            | While BType Stmnt
>            | SKIP deriving(Show)
```

Parts of the code have been used as examples throughout this report and the rest is available in the appendix.

Conclusion

The Parser Combinators summarised in this essay have more applications in Haskell than just Parsers, their definitions are general enough that they can be applied to combine various complicated functions. With the help of these Parsers and the Parsing Libraries, it should now be possible to create and implement a simple Parser.

The applications of these Parsers go beyond parsing Languages; for example, they can be used to extract various fields and types of data from databases This implies that these Parsers while simplistic in code, are surprisingly powerful, which is the main advantage of using a language such as Haskell.

References / Bibliography :

Note: Any unreferenced part of this essay can be assumed to be part of Dr Nicholas Wu's lecture series on Parsing at University of Bristol.

1. "While" Language, Haskell Wikipedia (Website):
   - ➢ https://wiki.haskell.org/Parsing_a_simple_imperative_language
2. University of Bristol : Introduction to Computer Architecture COMS 12200
3. University of Bristol : Language Engineering COMS 222011
4. Monad Rules, Haskell Wikipedia (Website)
   - ➢ Specifically : "Other Monoidal Classes"
   - ➢ https://wiki.haskell.org/Typeclassopedia
5. Haskell, Control.Applicative Explanation (Webiste)
   - ➢ https://hackage.haskell.org/package/base-4.9.0.0/docs/Control-Applicative.html

Appendix :

The following is the Parser for the “While” language [1]

Parser.lhs

```
> import Control.Monad
> import Control.Applicative

============================================================

> newtype Parser a = Parser (String -> [(String, a)])

> parse :: Parser a -> (String -> [(String, a)])
> parse (Parser p) = p

> item :: Parser Char
> item = Parser (\s ->
>   case s of
>        []     -> []
>        (x:xs) -> [(xs,x)])

> failure :: Parser a
> failure = Parser (\ts -> [])

> produce :: a -> Parser a
> produce x = Parser (\ts -> [(ts, x)])

> instance Applicative Parser where
>    pure x = produce x
>    Parser pf <*> Parser px = Parser (\ts -> [ (ts'', f x )| (ts', f) <- pf ts,
>                                                             (ts'', x) <- px ts'] )

> instance Functor Parser where
>    fmap f (Parser px) = Parser (\ts -> [ (ts', f x)  | (ts', x) <- px ts])

> instance Monad Parser where
>    return = produce
>    (Parser px) >>= f = Parser (\ts ->
>        concat [parse (f x) ts' | (ts', x) <- px ts])

> satisfy :: (Char -> Bool) -> Parser Char
```

```
> satisfy p = item >>= (\c ->
>   if p c then
>    produce c
>    else failure)

> char :: Char -> Parser Char
> char c = satisfy (c == )

> string :: String -> Parser String
> string [] = produce []
> string (c:cs) = char c >>= (\c' ->
>                string cs >>= (\cs' ->
>                produce (c:cs)))

> instance Alternative Parser where
>    empty = failure
>    (<|>) = orElse
>    many p = some p <|> produce []
>    some p = (:) <$> p <*> many p

> orElse :: Parser a -> Parser a -> Parser a
> orElse (Parser px) (Parser py) = Parser (\ts ->
>    case px ts of
>        [] -> py ts
>        xs -> xs)


> oneOf :: [Char] -> Parser Char
> oneOf s = satisfy (flip elem s)

> noneOf :: [Char] -> Parser Char
> noneOf cs = satisfy (\c -> not (elem c cs))



Grammar of the Language


a ::= string | num | -a | a opa a
b ::= true | false | not b | b opb b | a opr a
opa ::= + | - | * | /
opb ::=  and | or
opr ::= > | <

S :: = string := a | skip | S1;S2 | (S) | if b then S1 else S2 | while b do S



> data AType = Var String | IntCon Integer | Neg AType | OpAExpr OpA AType AType deriving
(Show)
> data BType = BoolC Bool | Not BType | OpBExpr OpB BType BType | OpRExpr OpR AType AType
deriving (Show)
> data OpA = (:+:) | (:-:) | (:*:) | (:/:) deriving (Show)
> data OpB = AND | OR deriving (Show)
```

```
> data OpR = (:<:) | (:>:) deriving (Show)
> data Stmnt = Assign String AType
>            | Seq [Stmnt]
>            | If BType Stmnt Stmnt
>            | While BType Stmnt
>            | SKIP deriving(Show)

> tok ::String -> Parser String
> tok t = string t <* whitespace

> whitespace :: Parser ()
> whitespace = many (oneOf " \t") *> pure ()

> aType :: Parser AType
> aType = Var <$> tok "VAR" <*> stringP
>       <|> IntCon <$> tok "IntCon" <*> intParser
>       <|> Neg <$> tok "Neg" <*> aType
>       <|> OpAExpr <$> tok "OpAExpr" <*> opA <*> aType <*> aType

> stringP :: Parser String
> stringP = some (noneOf ("\n\r\"={}[],")) <* whitespace

> intParser :: Parser Integer
> intParser = (some (oneOf ['0'..'9']) >>= produce.read)  <* whitespace

> bType :: Parser BType
> bType = BoolC <$> boolP
>       <|> Not <$> bType
>       <|> OpBExpr <$> opB <*> bType <*> bType
>       <|> OpRExpr <$> opR <*> aType <*> aType


> boolP :: Parser Bool
> boolP = (tok "TRUE" >> return (True))
>     <|> (tok "FALSE" >> return (False))


> opA :: Parser OpA
> opA = (:+:) <$ tok "+"
>    <|> (:-:) <$ tok "-"
>    <|> (:*:) <$ tok "*"
>    <|> (:/:) <$ tok "/"


> opB :: Parser OpB
> opB = AND <$ tok "&"
>       <|> OR <$ tok "|"

> opR :: Parser OpR
> opR = (:<:) <$ tok "<"
>     <|> (:>:) <$ tok ">"
```