FIT2102

Programming Paradigms Assignment 2 - Report

Benjamin Leong Tjen Ho
32809824

Approach to Assignment

My general approach to this assignment was to first understand how the bits and pieces of the assignment specifications align and connect with each other. I came to the conclusion that our Parsers had to have some sort of reference in order to flow in a structured and controlled manner. In the sections below, I will illustrate how I used BNF and my very own custom Data Types to achieve this structure.

Not only that, I made sure to use my knowledge of Parsers gained from the weeks in this unit (particularly Week 9 - 11). The practice I had during those weeks were definitely crucial as without them, I would not have been able to start at all.

Structure of Code

I decided to split my code into 2 parts, each part in a single file. Hence, I will also split this section into 2 parts, 1 for HelperFile.hs and 1 for LambdaParser.hs.

HelperFile.hs

This file only contains custom Data Types, minor lambda expressions (as Builders) and helper functions which were used in LambdaParser.hs. I chose to only contain functions/content that I personally find contribute the least to the main Parsers in LambdaParser.hs. This is so that LambdaParser.hs will only contain the more important Parser functions.

Onto the general structure of the file, I split the file into Parts according to the Parts of this assignment. At the start of the file contains helper functions which were useful throughout both files such as isTok, chain, and betwBrac. Within each part, the format is as follows:

- Data Types (if any)
- Exercises
    - Lambda Expressions as Builders (e.g., True, False, head, etc.)
    - Helper functions (Most commonly, Chaining functions)

LambdaParser.hs

For each Exercise, the format is as follows:

- Parser Combinators
- Parser for individual segments
- Piecing together segments using Parser Combinators (and the chain function)
- The final Parser(s)

Parsing

BNF Grammar

```
<params>        ::= <lambda> 1*<alpha> <dot>
<param>         ::= <lambda> <alpha> <dot>
<lambdaExpr>    ::= 1*<longForm> | 1*<shortForm>
-- LongLambda Section
<longBase>      ::= <openBrac> <param> 1*<longBody> <closeBrac>
<longBody>      ::= 1*<longInner>
<longInner>     ::= <longLambda>
                  | <alpha>
                  | <openBrac> <longBody> <closeBrac>
<longLambda>    ::= 1*longBase
                  | <openBrac> <longLambda> <closeBrac>
-- ShortLambda Section
<shortBase>     ::= <openBrac> <shortBase> <closeBrac>
                  | <params> 1*<shortBody>
<shortBody>     ::= 1*<shortInner>
<shortInner>    ::= <shortLambda>
                  | <alpha>
                  | <openBrac> <shortBody> <closeBrac>
<shortLambda>   ::= 1*<shortBase>
                  | <openBrac> <shortLambda> <closeBrac>
-- Terminals Section
<lambda>        ::= 'λ'   <spaces>
<alpha>         ::= [a-z] <spaces>
<dot>           ::= '.'   <spaces>
<openBrac>      ::= '('   <spaces>
<closeBrac>     ::= ')'   <spaces>
<spaces>        ::= ' '   <spaces> | ''
```

I had two different but similar approaches for my tasks, one for Part 1, one for Part 2 and 3. Part 1 makes use of a full-fledged defined BNF while Part 2 and 3 made use of custom Data Types (more on that in **Haskell Features**).

Part 1

For this approach the idea was basically to create Parsers according to the structure of the BNF. This meant that for every non-terminal there was a Parser while every terminal represented a character or string to be parsed (or read). With this idea in mind, once I had created a BNF, I could easily create my Parsers! Right? Well not exactly.

My BNF as shown above does include non-terminals that come after `1*`. The `1*` symbol means the non-terminal must occur at least once. This meant that I had to check for that terminal (character or string) more than once since it can occur many times. Since Haskell is

a pure functional programming language, there was no way I could do this imperatively. Therefore, recursion was the way to go (and is like a best friend in this assignment). How do we deal with this? We make use of Parser Combinators!

Part 2 and 3

With the use of custom Data Types, I could approach the exercises in a more mathematical mindset. Similar to the Calculator Parser given in Tutorial 11, I set up the Data Structure such that I could visualise the precedence of operations. With that, I can structure my Parsers according to the precedence of operations and deal with the input as intended.

But just like in Part 1, we have an issue with looping/recursion and hence, Parser Combinators are used once more.

Parser Combinators

How does using Parser Combinators achieve a recursive effect? With the help of the chain function, I can parse some fixed Parser 1 or more times, and chain the results using a Parser Combinator. This allows me to check using some parser more than once without the use of imperative code, hence, recursion is achieved!

Parser's Typeclasses

The parsers and parser combinators were constructed using the Functor, Applicative, and Monad typeclasses. This is because the typeclasses mentioned have a ton of super useful predefined functions (e.g., fmap, bind, apply) that makes parsing so much easier.

Functional Programming

My programming style has always been to modularise everything! No matter the programming language, I tend to be able to identify repeating patterns and modularised functions where possible. It was no different for Haskell, if anything, it was much easier with a Functional Programming Language. Modularisation kills off the need of large repetition and allows me to reuse helpful blocks of code/relatively long lines by simply calling a predefined function.

A neat thing about Functional Programming is composing smaller functions together to achieve a much greater effect. I would say I used composition in almost every exercise in the assignment. This is especially useful when using custom Data Types. If my output doesn't match the expected (data) type, I can just smack another function in there using composition and voila, the function works perfectly.

Haskell Language Features Used

Custom Data Types

As mentioned above, I created my own custom Data Types when working on Part 2 and 3 of the Assignment. The advantage of using custom Data Types is that I get to decide how the values are structured and hence can manipulate them according to my needs. In this case, the data structures are designed to parse lambda expressions (as Builders) that represent logical and mathematical statements, and lists.

Figuring out how to properly construct my custom Data Types was no easy task. However, once I figured out how to properly use them, it made things a lot clearer due to the abstraction the data types offer. I could easily see the conversions from string to custom data types to Builder to Lambda.

Types and Higher Order Functions

Besides that, I also did create some types in order to make my life a bit easier when writing type hints. I can just write `Chainer` instead of `Builder -> Builder -> Builder`. Speaking of Chainer, the use of higher order functions meant that I could easily carry out complex operations without the need of defining complex code blocks. Essentially, there was no need to reinvent the wheel when I could easily just use useful (existing) functions as input into other functions to get out an even more useful function.

Fmap, Bind, Apply and other built-in functions

I already talked about why I used functions like fmap, bind and apply in **Parsing**. To elaborate, these functions allow me to parse in a much more flexible manner. Sometimes I want to parse several things but only wish to take the result of the second parser. That's where the right and left apply function comes in handy. I can determine which specific parser I want the result from and the code ignores everything else.

On the topic of parsing, there are also some built-in functions which were helpful when it came to dealing with the Parser wrapping. My favourites were liftA2 and liftA3. They accept values wrapped in a Parser and make use of the value inside which completely removes the need of a do block. This saves me so much time and space (lines).

Description of Extensions

I have implemented 2 extensions; The Factorial function as well as some list functions.

Factorial

I intended to create a Parser that takes in a string representing a factorial call (e.g., 5!) and returns the lambda expression equivalent to the string's factorial call, not its resulting value (yet).

Since my Factorial implementation is a recursive function, I made use of the Z-Combinator which we have learnt in Week 5 (Lambda Calculus). We couldn't use the Y-Combinator directly since it would have caused a stack overflow due to infinite "loops". The Z-Combinator works for Factorial since it only takes in one parameter (i.e., n). (What if there were more than one? That would be in the next section.)

As for all recursive functions, they require a base case. It's no mystery the base case for factorial would be when n is 0. But that begs the question of how exactly do we form our lambda expression for Factorial. After some tinkering, I realised that from Part 2, I have some logical operators to form a Lambda function for Factorial!

```
λfn.IF n==0 THEN 1 ELSE (f n-1)
```

(Pseudo-code for factorial as a Lambda Expression)

On the topic of reusing existing functions, I also decided to use the Parsers bracArith and number to represent my n value when parsing the string. This worked perfectly since they both returned a lambda expression that holds a numerical value.

Combining the n value and the factorial function, my Parser was basically completed. All that I had to do was to put everything together. Once again, bind and left apply are super convenient for checking for spaces and verifying the input string (if it ends with '!').

List Functions (foldr, map)

So how about recursive functions with more than one parameter such as foldr and map? To figure out how to implement this, I had to first understand the Z-Combinator. From my understanding, the reason the Z-Combinator stops when it should not the Y-Combinator is because of its extra parameter (let's refer to it as v).

```
Z-Comb = λf.(λx.f(λv.xxv))(λx.f(λv.xxv))
Y-Comb = λf.(λx.f(xx))(λx.f(xx))
```

The added inner parameter means that for every recursive call, the function will actually input the next value to check for its stopping condition. This determines whether or not more recursive calls are needed and if it's not, the function stops calling itself.

Okay, now that we know this, how does it relate to multi-parameter recursive functions? The combinator above only works for functions like Factorial where it accepts 1 parameter only. So to make it work for functions with let's say 2 parameters, we simply need to add 2 inner parameters instead of 1! For 3 parameters, add 3!

```
Z2-Comb = λm.(λx.m(λfl.xxfl))(λx.fl(λv.xxfl))
Z3-Comb = λm.(λx.m(λfil.xxfil))(λx.fil(λv.xxfil))
```

With working combinators, we just need to get a lambda expression for our functions, a suitable Parser to piece things together, and we're good to go! Using the same idea as Factorial, I used my logical expressions to form a base case for my recursive functions.

```
map = λmfl.IF (isNull l)       -- Base case/Stop condition
          THEN null
          ELSE cons (f (head l)) (m f (rest l))
```

```
foldr = λmfil.IF (isNull l)    -- Base case/Stop condition
          THEN i
          ELSE f (head l) (m f i (rest l)
```

The Parser used a similar idea as Factorial but this time, the Parser verifies the validity of the string first since the function names (foldr/map) appear before their inputs. Sounding like a broken record, I reused some of the super useful functions such as >>, *> (built-in) and stringTok.