Report for Assignment 2

Data structures

The main approach to parse the input then pretty printing it is through an intermediary ADT.

Mainly, every single exercise in this assignment introduces a new data type for this intermediary ADT, and they may have different number of parameters depending on how I structured my ADT type. How I defined the ADT type is essential because the nested ADT data should be stored, which I did through the parameters.

For example, ADT data of If defined:

```
        | JIf ADT ADT
```

This means I must store two things: the condition and the if statement body (both of type ADT) Hence, when parsing I will store the ADT of condition and ADT of Block.

```
        | Block [ADT]
```

Which, again, is another one of ADT data type.

In addition, the other information is that the parsed input is an If statement, as it is encapsulated as an If ADT.

All the ADTs follows from this data structuring.
This will make things easier for parsing and printing.

Parsing

The parsing involves CFG, which is also hinted in the definition of the data types of ADT. As stated above, An ADT can have another ADT as one of its parameters (which can theoretically be any one of the ADT data types defined).

Hence, it is important for me to properly define the parsers.

```
parseIf :: Parser ADT
parseIf = do
  spaces
  stringTok "if"
  spaces
  n1 <- parens parseExerciseA  -- condition, enforces a parenthesis around the
condition
  spaces
  n2 <- parseBlock  -- statement body
  return (JIf n1 n2)
```

parseIf is a parser of ADT type. Basically this parser makes use combinators such as spaces, stringTok, (parens and parseBlock of type ADT), and finally returns a ADT. These parser combinators allows reuse of code (by reusing these parser combinators) in different parsers. Hence, parsers can be built by different parser combinators collectively in order.

Notably, code for parseIf shows the importance of Monad in constructing a parser (with the monadic bind to sequence parsing actions.

```
parens p = Parens <$> (spaces *> charTok '(' *> spaces *> p <* spaces <* charTok
')' <* spaces)
```

Furthermore, the code for parens highlights the importance of Applicative and Functor in constructing a different parser.

Moreover, it can demonstrate that small modular functions are useful, because it can be used as a parser combinator for another parser function.

FP

```
-- Used to determine the number of return statemetns in the function.
countReturns :: [ADT] -> Int  -- counts the number of returns in the list of ADTs
countReturns = length . filter isReturn
  where
    isReturn (Return _) = True
    isReturn _ = False
```

```
-- Used to determine the number of return statemetns in the function.
countReturns :: [ADT] -> Int  -- counts the number of returns in the list of ADTs
countReturns adts = length (filter isReturn adts)
  where
    isReturn (Return _) = True
```

```
    isReturn _ = False
```

These two are the same function, where the first example demonstrates point free style. Using point free can make code more concise to read. Code can be made point free by composing functions.

Haskell features

The main typeclass for the assignment is Parser. It mainly allows for custom Functor/ Monad/Applicative definitions. This means I can use fmap to create new Parser instances (as is the defined functionality for fmap in Functor class), alternatives for shifting to next parser if it fails, or apply which allows applying parsers in sequence.

Extended

A converter from if then else to ternary expression has been implemented as parseTernaryAlt This involves the same steps as optimising function call.
Firstly, it shows that the modular parsing functions are very useful, because with some pattern matching and already defined parsing functions, I was able to parse a If else statement using parseIf, then saving the essentials as parameters (as noted in the beginning of this report), then encapsulating these information as a new Ternary wrapper.
Additionally, the pretty printing for this particular Ternary is redefined, as it can parse ADTs of a higher level in the grammar. (parsers for exercise B instead of just exercise A).