FIT2102 Assignment 2 Report
Wong Yuan Yi 32845944

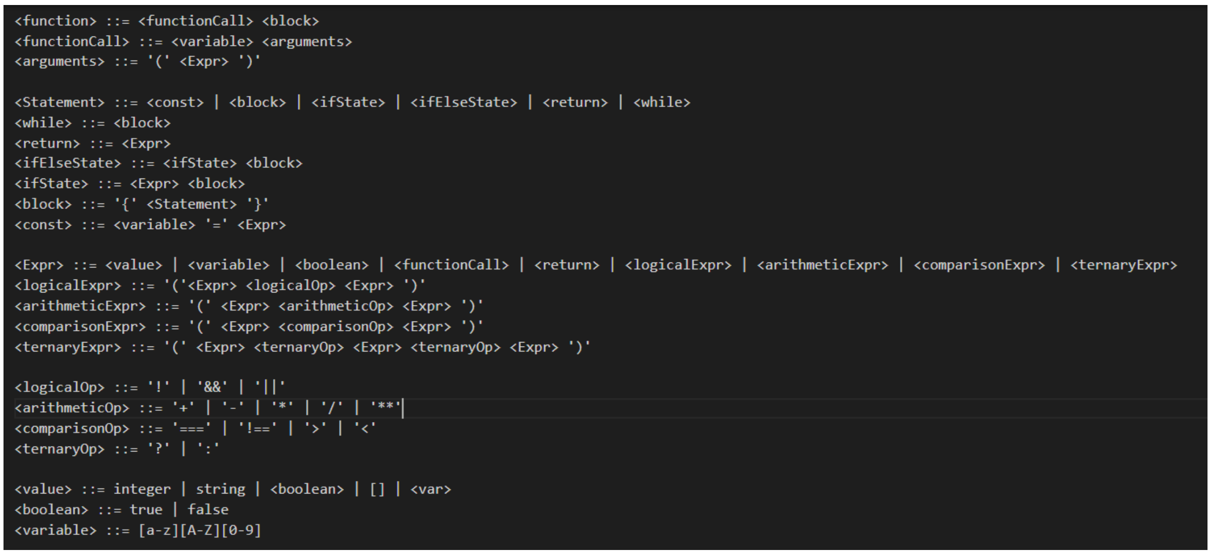
Design of Code

I have developed my JavaScript Parser by using an ADT to store different data types after parsing as well as implemented the show instance for each data type to pretty print the parsed data type. When designing parsers, small modular helper parsers were first created and then combined using parser combinators to ease the process. Part A parses data types and expressions, Part B parses block statements, while Part C parses functions. This ADT allows better maintainability and expansive capabilities, while the parser combinators simplifies the parsing process.

All helper parsers were defined in Parser.hs, while all ADT parsers were defined in Assignment.hs. The top of Assignment.hs are the ADT and show instance declarations, followed by ADT parsers arranged from top to bottom based on each exercise, lastly followed by the given parser code for each Part (A, B, C) that combines the parsers from each exercise for easy interpretability.

Parsing

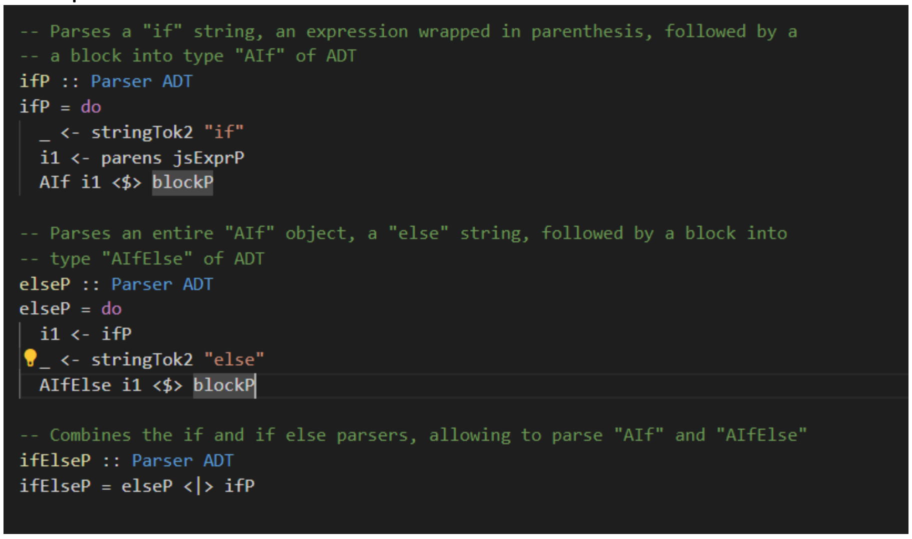
BNF Grammar:



```
<function> ::= <functionCall> <block>
<functionCall> ::= <variable> <arguments>
<arguments> ::= '(' <Expr> ')'

<Statement> ::= <const> | <block> | <ifState> | <ifElseState> | <return> | <while>
<while> ::= <block>
<return> ::= <Expr>
<ifElseState> ::= <ifState> <block>
<ifState> ::= <Expr> <block>
<block> ::= '{' <Statement> '}'
<const> ::= <variable> '=' <Expr>

<Expr> ::= <value> | <variable> | <boolean> | <functionCall> | <return> | <logicalExpr> | <arithmeticExpr> | <comparisonExpr> | <ternaryExpr>
<logicalExpr> ::= '('<Expr> <logicalOp> <Expr> ')'
<arithmeticExpr> ::= '(' <Expr> <arithmeticOp> <Expr> ')'
<comparisonExpr> ::= '(' <Expr> <comparisonOp> <Expr> ')'
<ternaryExpr> ::= '(' <Expr> <ternaryOp> <Expr> <ternaryOp> <Expr> ')'

<logicalOp> ::= '!' | '&&' | '||'
<arithmeticOp> ::= '+' | '-' | '*' | '/' | '**'
<comparisonOp> ::= '===' | '!==' | '>' | '<'
<ternaryOp> ::= '?' | ':'

<value> ::= integer | string | <boolean> | [] | <var>
<boolean> ::= true | false
<variable> ::= [a-z][A-Z][0-9]
```

Parser combinators were mainly used simplify a hard task by combining simple parsers. Parsers for each exercise were split and combined with combinators mainly <|> and “do” (>>=) because it is easier to focus on parsers for small parts and constructing complex parsers afterwards by combing smaller parsers. When combining using <|>, parsers that parse the same thing, but more will have higher precedence and therefore the order when combining parsers matters.

Example:



```haskell
-- Parses a "if" string, an expression wrapped in parenthesis, followed by a
-- a block into type "AIf" of ADT
ifP :: Parser ADT
ifP = do
  _ <- stringTok2 "if"
  i1 <- parens jsExprP
  AIf i1 <$> blockP

-- Parses an entire "AIf" object, a "else" string, followed by a block into
-- type "AIfElse" of ADT
elseP :: Parser ADT
elseP = do
  i1 <- ifP
  _ <- stringTok2 "else"
  AIfElse i1 <$> blockP

-- Combines the if and if else parsers, allowing to parse "AIf" and "AIfElse"
ifElseP :: Parser ADT
ifElseP = elseP <|> ifP
```

elseP must be before ifP because elseP also parses an if statement first, having ifP in front would parse an if statement, but the else statement will be left unparsed. Furthermore, parsing simple strings into operator objects were used because it can be easily combined later with other parsers to work with expressions into valid data types. Lastly, general parsers for matching strings and characters were used in order to match an input string, if it matches then an object will be constructed.

Functional Programming

For FP principles, small modular functions were used to handle specific tasks, this allows for better readability and easier development process.

Example modular functions:
Multiple functions defined for isTailRecursive() functions, some used to check Criteria 2, some for Criteria 3. This allowed to easily isolate different cases and check individually and combined together with “&&” to fit both criteria, this way it is much easier to identify errors and make changes to existing criteria or even add further criterias.

All functions and parsers do not manipulate ADT object values as well but returns a new updated ADT object. Recursion by pattern matching and mapping were used instead of looping due to the immutable data rule. Type hinting was also implemented to catch errors easily as well.

Haskell Language Features Used

For parsers they were of type Functor, Monad, Applicative, and Alternative. This enabled the use of higher order functions such as <$>, <*>, <|>, >>=. <$> and <*> allows for easily manipulating values of parsers and wrap them into new parsers for easier handling while <|> and >>= enables combining parsers easily. These functions were mostly used to extract, construct and combining parsers together easily.

Generally, a parser is mainly combined with other parsers using “do” instead of >>= for better readability because do easily shows which parser is first, and then constructing a data type with <$> instead of using pure as it is syntactic sugar. Parse outcomes or input also uses pattern matching instead of using multiple “case”s because of readability issues as multiple levels of indentations can be confusing, pattern matching is good to handle unwanted cases easily as well using “_” empty pattern match. Lastly, object destructuring was also used to access inner data types of an object easily.