Name: Teh Jia Xuan	Student ID: 32844700
Assignment 2: Javascript Parsing	

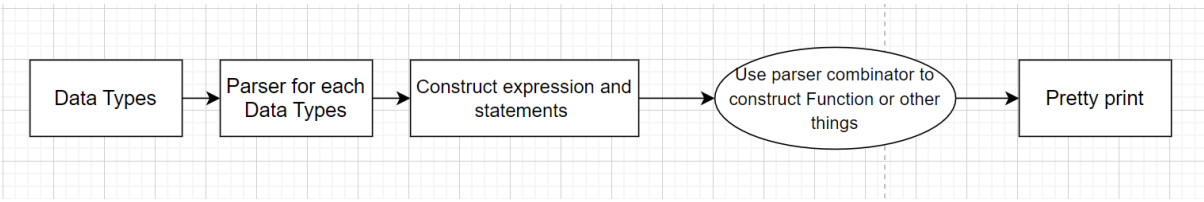
Code Design

Description of Approach

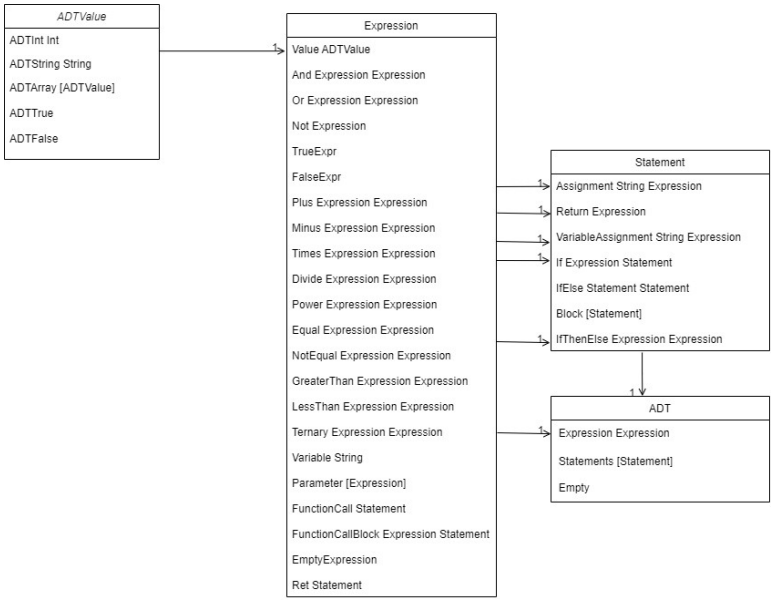
- Parser: check and parse valid javascript code
- Pretty print: print proper format javascript code
- Focused on creating readable and functional code
- Maintainability, readability and extensibility

Structure of code

- Every data type has its own constructor of parser
- Use parser combinator to constructor some data types like block and if
- Block and if used in other higher hierarchy parsers like function
- Pretty print the parser in proper format



Code architecture



- ADTValue are used to store int, string and boolean
- Expression can access ADTValue by using Value
- Statement can access to expression
- Expression can have nested statements like nested if and nested function

Parsing

BNF grammar

<ADTValue> ::= ADTInt <Int>

| ADTString <String>

| ADTArray [<ADTValue>]

| ADTTrue

| ADTFalse

<Expression> ::= And <Expression> “&&” <Expression>

| Or <Expression> “||” <Expression>

| Not “!”<Expression>

| TrueExpr

| FalseExpr

| Plus <Expression> “+” <Expression>

| Minus <Expression> “-“ <Expression>

| Times <Expression> “*” <Expression>

| Divide <Expression> “/” <Expression>

| Power <Expression> “**” <Expression>

| Equal <Expression> “===” <Expression>

| NotEqual <Expression> “!==” <Expression>

| GreaterThan <Expression> “>” <Expression>

| LessThan <Expression> “<” <Expression>

| Ternary <Expression> “?” <Expression> “:” <Expression>

| Value <ADTValue>

| Variable <String>

| Parameter [<Expression>]

| FunctionCall “(” <Statement> “)”

| FunctionCallBlock “(” <Expression> “)” “{”<Statement> “}”

| EmptyExpression

| Ret <Statement>

<Statement> ::= Assignment <String> <Expression>

| Return <Expression>

| VariableAssignment <String> <Expression>

| If “(” <Expression> “)” “{” <Statement> “}”

| IfElse <Statement> <Statement>

| Block “{” [<Statement>] “}”

| IfThenElse “if” <Expression> “then” <Expression> “else” <Expression>

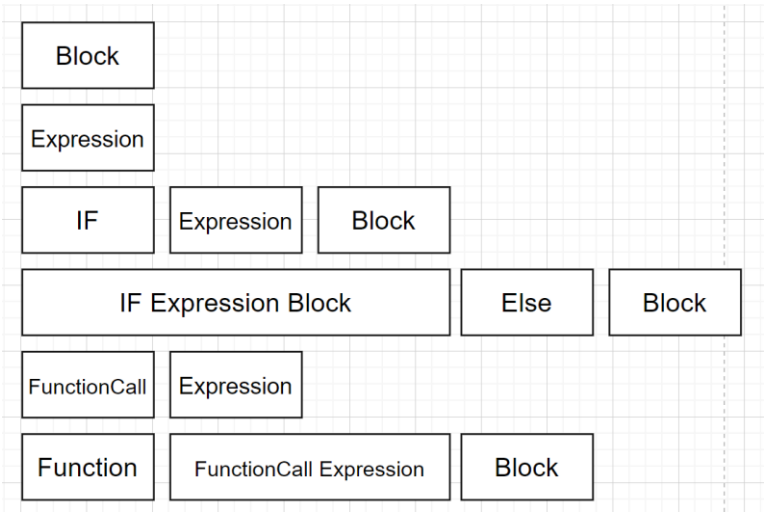
<ADT> ::= Empty

| Expression <Expression>

| Statements [<Statement>]

Usage of parser combinators

- Parser are created based on the BNF grammar
- Conciseness, parser for if statement is combined with If-else statement
- Efficiency, parser for function call is reused in function call block
- All expressions are combined under parsePartA is used in every statement that contains expressions.
- Simplicity, It helped me easier to debug and test when completing parsing as it is made up of multiple small parser
- produce informative error messages
- Able to reuse parsing components in different parser
- Challenges: different data types cannot parse together
- It requires me to swap the expression and statement data type in the provided BNF grammar



Choices

- Parser that can be combined is created before the complex one
- Parser combinators started with creating an individual parser like block and expression and combining them like Lego pieces into a more complex parser
- Challenge: Parsing failed when combining multiple parsers using alternative, even there is one parser is functional.
- Order of parsers in alternative is important, and it requires multiple tests to get it right.

How it creates

Functor

- fmap is used to apply ADT function to the value inside the functor
- modify the value without modify input

Applicative

- <*> is used to combine parsers and apply function
- *> is used for discarding left parser's result. It is widely used for removing spaces in my parser
- Used for sequencing and combining parser

Monad

- Bind operator is used in parser combinators to sequence parsers
- It allows me to connect multiple parsers
- Sequencing operator allows parsing actions in correct order
- Do notation, allows parsing in sequence -> readability
- Parsing based on previous result-> dependency

Functional Programming

Small modular functions & Composing small functions together

- Break down complex problems into smaller chucks
- Every function is to perform a specific task
- Keep all functions small and easier to read
- Smaller functions are easier to debug and reuse
- Testability, small functions are easier to test

Declarative style

- Expressive and easier to read
- Abstraction, hide complex code
- Functional programming, it uses pure functions

Point-Free style

- remove unnecessary code
- make code short and concise
- avoid explicit mention of arguments
- consistency, simplifies process of understanding

Haskell Features used

Typeclasses and Custom Types

- Safety, type errors are noticed at compile time
- Typeclasses ensure the operations applied to types that aligned
- Custom types able to abstract implementation details

Higher-Order Function, fmap, apply, bind

- Reusable, functions accept other functions as arguments
- substituting the result of constructor efficiently
- Pass function to construct another complex function
- Simple and elegant, fmap apply function to element without unwrapping it
- Apply effectively lifts function into functor
- Bind allows me to chain monadic actions
- Create maintainable code

Foldl

- Fold through array use function and accumulator
- Used to combine expression

Map

- Iterate through array and apply function

Function Composition

- Readability, easier to understand
- Avoid declaring intermediate variables
- Pure functions don't have side effects
- Combine pure function

Description of Extensions

Intended to implement

- I aimed to implement converting for loop to forEach
- Too complicated and time-consuming
- unable to identify the list and correctly iterate through them
- Move on to converting if/then/else to ternary

What I implement

- Implemented when it parses if then else it prints out ternary expression

Cool features

- It automatically identifies if/then/else
- Replace with ternary

Challenges when implementing and solution

- Challenge: The return type is not aligned with the ternary data type
- Solution: Adding another return type at expression in the BNF grammar
- Reuse the return parser and wrap it with return type at expression

Haskell features that are not covered in course

- I have used some functions that are not covered in course like unwords