Zabir Raihan
32353057

---

1. Design of the Code

High-Level Description of Approach
The parser is constructed using a bottom-up approach, defining the smallest possible parsers first and then combining them to create larger, more complex parsers. The parser combinators allow parsers to be combined in a modular and declarative way.

High-Level Structure of Code
The code is structured into several sections, each dealing with a specific part of the JavaScript syntax. The first section defines a series of basic parsers and combinators, including parsers for specific characters and tokens, and combinators for sequences and alternatives. The following sections define parsers for different types of expressions and statements.
Finally there is a section dedicated to pretty printing the various parsed parts.

Code Architecture Choices
The use of small, modular functions in combination with parser combinators allows for a clean and readable codebase. This makes it easy to extend the parser with new features or modify existing ones. For example, if we want to add integer division arithmetic expression, we will just have to add the parsing function specifically for it and add it to the end of the data types, unified comparison parser and pretty printer without affecting anything else. The same applies if lets say we want to add support for lambda expressions (which was done as a bonus attempt). Functions were split into smaller functions in a reasonable manner, with the goal being to make the code understandable just through function names but also in intents of reusability.

---

2. Parsing

BNF Grammar
The BNF grammar for the JavaScript subset is implicitly defined by the structure of the parsers and combinators. For example, the `expr` parser, which parses expressions, is defined as a choice between different types of expressions, each parsed by its own parser.

Usage of Parser Combinators
Parser combinators are used extensively to construct complex parsers from simpler ones. For instance, the `sepBy` combinator is used to parse comma-separated lists of values, and the binary operators are parsed using a generic `binaryOp` combinator.

Choices Made in Creating Parsers
The choice to use parser combinators was motivated by their expressive power and modularity. They provide a declarative way to describe the grammar of the language and can be easily composed to create more complex parsers. Monadic parser combinators, in particular, offered a high level of abstraction and made it possible to define parsers in a way that closely resembles the structure of Javascript

Construction Using Functor, Applicative, and Monad Typeclasses
Parsers in this design heavily leverage the `Functor`, `Applicative`, and `Monad` typeclasses. For instance, the use of `<*>` (an applicative operation) in `sepBy1` and `<$>` (fmap) in various parsers indicates how these typeclasses provide the necessary operations to combine and sequence parsers effectively.
The Functor typeclass is used to define how to apply a function to the result of a parser (fmap). The Applicative typeclass is used to define how to apply a parsed function to a parsed argument (<*>), and how to construct a parser that always succeeds with a given value (pure).

---

3. Functional Programming

The code is written in a declarative style, with small, modular functions that are composed together to create more complex functions. This style of programming is characteristic of functional programming and is well-suited to the task of parsing. All functions used for parsing are pure. Declarative style has been used throughout to make the parser easy to read and reason about. For instance, the definition of `arithExpr` simply expresses that an arithmetic expression can be any one of addition, subtraction, multiplication, division, or power operation, without detailing the steps to parse each one.

---

4. Haskell Language Features Used

Typeclasses, Custom Types and pattern matching
The Haskell typeclass system is used to define the behavior of the `Parser` type. Custom types are used for almost everything to represent different kinds of expressions and statements, allowing for a type-safe way of dealing with the different elements of the JavaScript syntax. Haskell's pattern matching has been used extensively too. The type of input was observed and depending on the input, specific functions were binded to the input. This enhanced readability and potential of extensions.

Higher-Order Functions
The code frequently uses higher-order functions like `binaryOp` which takes functions as arguments. The pervasive usage of operators from the Functor (`<$>`) and Applicative (`<*>`) typeclasses, as seen in `sepBy1`, demonstrates how these abstractions simplify complex operations.

Function Composition
A hallmark of Haskell, function composition, is evident throughout. By chaining together small operations, we build up more complex behaviors without nested calls or verbose interim variables. This is particularly evident in parsers such as `ternaryExpr`, which sequences multiple parsers, or in the `conditional` parser which also does something similar. Recursion was used for example to carry forward indent levels so we could ensure every child block is properly indented.

—

Bonus Features

Evaluating Arithmetic, Comparative and Logical Expressions:
When any of these expressions are detected, functions for evaluating them have been implemented. The evaluation is not too complex at the moment; it is not able to evaluate combination of the different types of expressions. But within the same type of expression, it can complete complex calculation such as a long chain of arithmetic operations or a chain of boolean logic.
To deep dive a bit for example in evaluating arithmetic expressions,
the function evalArith to evaluate arithmetic operations represented by the ArithExpr datatype and return results of type Maybe JSValue. For each arithmetic operation like Add, Sub, etc., it uses local helper functions (add, sub, etc.) to perform the actual computation on JSInt values. The function liftA2 is used to combine two Maybe values by lifting these helper functions, enabling arithmetic operations if both operands evaluate to a valid value. If either operand evaluates to Nothing, the entire operation returns Nothing.

An ExerciseD was created in `Assignment.hs` to showcase this. The inputs and outputs were placed in javascript/inputs/D and javascript/output/D.

Parsing Lambda expressions:
The code was updated to be able to handle lambda expressions. It was possible to integrate into a constant declaration as well. Much of this is fairly simple and not very complex but that was possible due to the modular nature of the existing code. All that was needed was to add specific logic for lambdas and the rest was just adding it to the data types. Several of the helper functions in conjunction with the power of Applicative to achieve this.
An ExerciseE was created in `Assignment.hs` to showcase this. The inputs and outputs were placed in javascript/inputs/E and javascript/output/E.

The outputs for both Exercise D and Exercise E were generated using `stack test`, ie it is possible to add more inputs and the program will parse and pretty print accordingly.

For both the bonus features, Spec.hs was edited to add Exercise D and Exercise E.