Zabir Raihan
32353057

—

Please check README for some instructions on running the parsers.

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.

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 next 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.

---

2. Parsing

BNF Grammar (this has been excluded from word count as per EdStem)

<Program> ::= <Stmt> | <Program> <Stmt>

<Stmt> ::= <StmtConst> | <StmtIf> | <StmtFuncCall> | <StmtReturn> |
<StmtFuncDecl> | <StmtBlock>

<StmtConst> ::= "const" <VarName> "=" <Expr> ";"

```
<StmtIf> ::= "if" "(" <Expr> ")" "{" <Stmt> "}" ["else" "{" <Stmt> "}"]
<StmtFuncCall> ::= <VarName> "(" [<Expr> {"," <Expr>}] ")" ";"
<StmtReturn> ::= "return" <Expr> ";"
<StmtFuncDecl> ::= "function" <VarName> "(" [<VarName> {"," <VarName>}] ")" "{"
<Stmt> "}"

<StmtBlock> ::= "{" <Stmt> {<Stmt>} "}"
<Expr> ::= <FuncCallExpr> | <JsVal> | <Arithmetic> | <Logical> | <Comparison> |
<TernaryOp> | <LambdaFunc>

<FuncCallExpr> ::= <FuncCall>
<JsVal> ::= <JSValue>
<Arithmetic> ::= <Add> | <Sub> | <Mul> | <Div> | <Pow>
<Logical> ::= <LAnd> | <LOr> | <LNot> | <LBool>
<Comparison> ::= <Equals> | <NotEquals> | <GreaterThan> | <LessThan>
<TernaryOp> ::= "(" <Expr> "?" <Expr> ":" <Expr> ")"
<LambdaFunc> ::= "(" [<VarName> {"," <VarName>}] ")" "=>" <Expr>
<FuncCall> ::= <VarName> "(" [<Expr> {"," <Expr>}] ")"
<JSValue> ::= <JSInt> | <JSString> | <JSBool> | <JSList> | <JsVariable>
<JSInt> ::= "-"? <Natural> | <Natural>
<JSString> ::= "\"" [^"]* "\""
<JSBool> ::= "true" | "false"
<JSList> ::= "[" [<Expr> {"," <Expr>}] "]"
<JsVariable> ::= <VarName>
<Add> ::= <Expr> "+" <Expr>
<Sub> ::= <Expr> "-" <Expr>
<Mul> ::= <Expr> "*" <Expr>
<Div> ::= <Expr> "/" <Expr>
<Pow> ::= <Expr> "**" <Expr>
<LAnd> ::= <Expr> "&&" <Expr>
<LOr> ::= <Expr> "||" <Expr>
<LNot> ::= "!" <Expr>
<LBool> ::= <Expr>
<Equals> ::= <Expr> "===" <Expr>
<NotEquals> ::= <Expr> "!==" <Expr>
<GreaterThan> ::= <Expr> ">" <Expr>
<LessThan> ::= <Expr> "<" <Expr>
<VarName> ::= [a-zA-Z] [a-zA-Z0-9]*
<Natural> ::= [0-9]+
```

Usage of Parser Combinators

Parser combinators are used extensively to construct complex parsers from simpler ones. For instance, the binary operators are parsed using a generic `binary0p` combinator. This has been used in all kinds of expressions in my parser to be able to

systematically parse two sides of a binary operator as well as any valid operator that may appear between the two sides.

Construction and choices in Using Functor, Applicative, and Monad Typeclasses
Parsers in this design heavily leverage the `Functor`, `Applicative` typeclasses. For example, `<$>` in various parsers allowed me to apply parsing functions to expressions wrapped in certain data types or functors. For example when I parse a block, The **Block** constructor is applied to the result of **stmts** using the **<$>** operator in a clean anc concise way. This makes the code more readable and easier to understand, as it clearly shows the intent of applying the constructor to the parsed statements.

---

3. Functional Programming

The code is written in a declarative style, with small, modular functions that are composed together to create more complex functions. For example, a **bracketed** function has been created so we can modularize our code to be able to parse any kind of brackets. Then it is used by **roundBracketed** or **squareBracketed** functions which are very good at demonstrating their purpose.

Small functions have been composed together to construct larger parsers. For example For example, the string function is defined as `traverse is` where these functions are chained together to be able to apply the `is` function to every element in the string. This made it easy to reduce a lot of repetition in code.

The code also heavily uses point free style of code. For example:
`evalLogicExpr = evalExpr logicExpr evalLogic`
The nice thing here is that we are able to define functions without mentioning their parameters. This makes the code concise and it works because it doesnt make a difference whatever the parameter is

---

4. Haskell Language Features Used

The Haskell typeclass system is used to define the behavior of the `Parser` type. Custom types are used for almost everything for, 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 (like pretty printing patterns) were binded to the input. This enhanced readability and potential of extensions.
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. And we also have a `evalUnifiedExpr` function to enable interoperability between different types of expressions.
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.