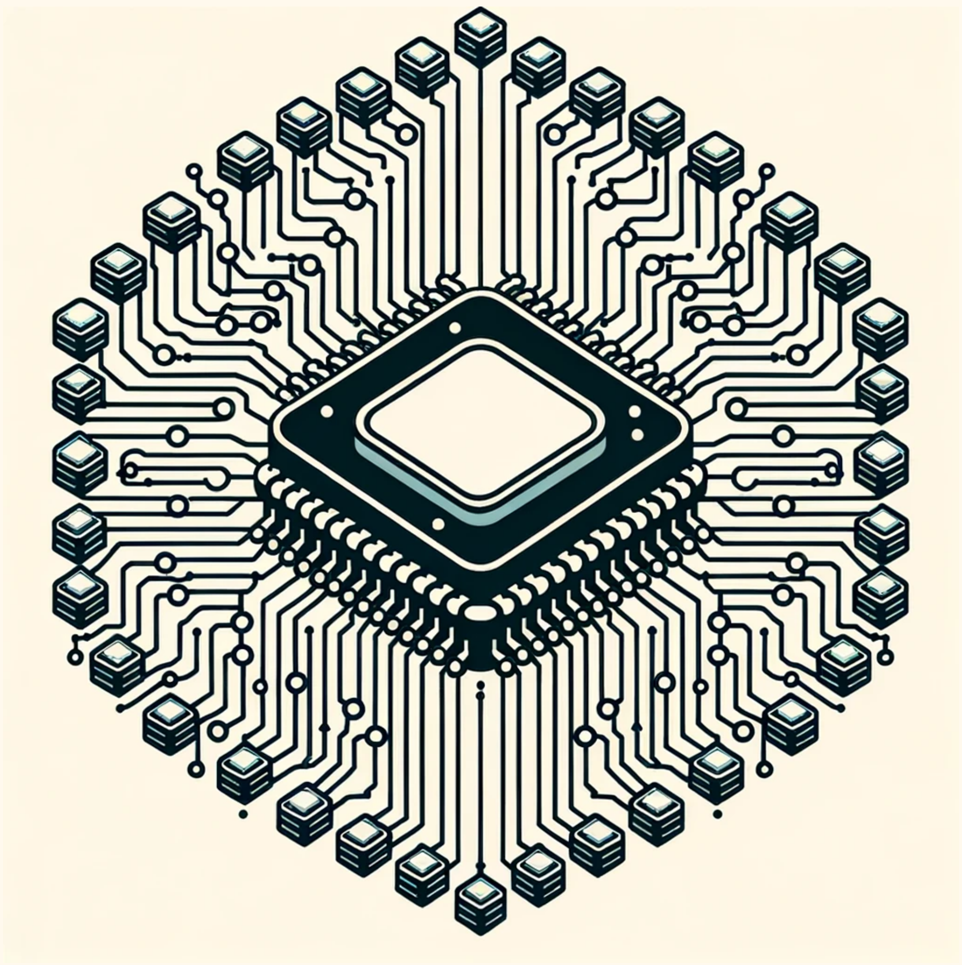
FIT2102 Tutorial 2

Assignment 2
Haskell Project - Parsing

—

By Myint Myat Thura (31861067)



1. Design of the code

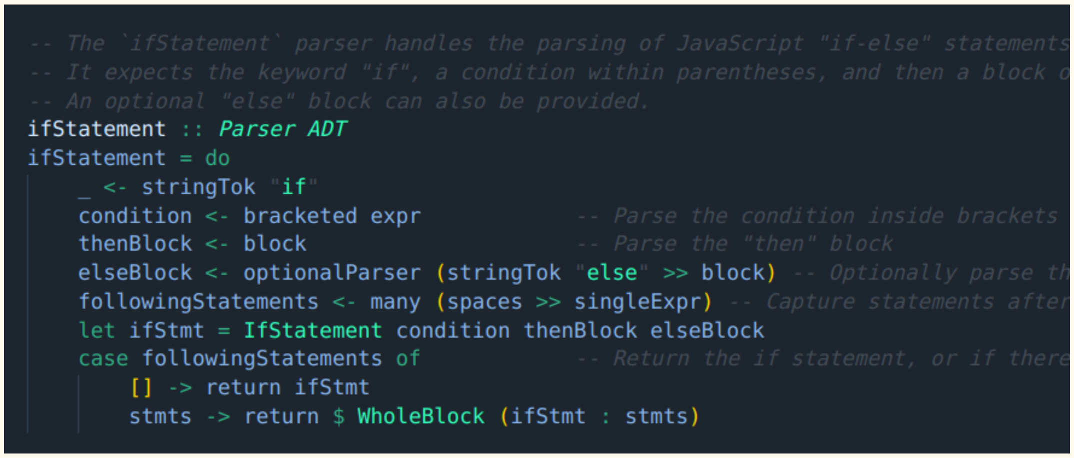
This Haskell project aims to parse a subset of JavaScript, transforming valid code snippets into an algebraic data type (ADT). Using Haskell's type system and functional paradigms, the ADT captures key JavaScript constructs. The architecture revolves around the Parser monad, using combinators to craft complex parsers from simpler units.

2. Parsing

BNF Grammar

The provided BNF serves as a precise guide for our parser, enabling structured error handling. Distinct error types, like (UnexpectedEof) and (UnexpectedChar), pinpoint input deviations from the BNF, offering clear feedback. This clarity, paired with the BNF's exactness, facilitates rigorous parser testing. Furthermore, by aligning with Haskell's robust type system, constructs like (Parser a) and (ParseResult a) ensure reliable and valid parsing, enhancing the system's overall dependability.

Use of Monads and/with Do Notations



```haskell
-- The `ifStatement` parser handles the parsing of JavaScript "if-else" statements
-- It expects the keyword "if", a condition within parentheses, and then a block o
-- An optional "else" block can also be provided.
ifStatement :: Parser ADT
ifStatement = do
    _ <- stringTok "if"
    condition <- bracketed expr              -- Parse the condition inside brackets
    thenBlock <- block                       -- Parse the "then" block
    elseBlock <- optionalParser (stringTok "else" >> block) -- Optionally parse th
    followingStatements <- many (spaces >> singleExpr) -- Capture statements after
    let ifStmt = IfStatement condition thenBlock elseBlock
    case followingStatements of              -- Return the if statement, or if there
        [] -> return ifStmt
        stmts -> return $ WholeBlock (ifStmt : stmts)
```

In parsing, sequence matters, and **monads** are perfect for representing this inherent order. Using **monads**, like in the **ifStatement** parser, ensures each part is recognized in the right sequence, from the "if" keyword to the condition and the blocks that follow. The use of binding, seen with the **“<-”** notation, allows us to capture critical outputs for later use, while discarding others that merely guide the parsing process but aren't necessary in the resulting data.

Parser Combinators elevate our parsing game. They encapsulate common parsing patterns, making our parser more modular.

The inherent flexibility in coding structures, like the **optional** "else" block, is elegantly handled using **applicatives**. They let us try a parsing path and, if it doesn't fit, gracefully fall back. This ensures our parser is both robust and adaptable.

The **many** combinator is our tool to address this. It's not about knowing in advance how many times a construct appears; it's about being ready to handle it as many times as it does.

Lastly, once the text is parsed, we're not done. We need to convert this recognized structure into meaningful data. This is where Haskell's strengths, like **pattern matching**, come into play. It lets us discern between different parsing outcomes and construct the appropriate data structures.

Together, these techniques form a sophisticated parser that's sequenced, modular, adaptable, and transformative, converting raw JavaScript text into a structured and meaningful abstract syntax tree.

3. Function Programming

In my code, I leveraged the core principles of functional programming to ensure clarity and maintainability. By designing small modular functions, like bracketed expr, I ensured each piece had a single responsibility, making it easier to test and debug. I took advantage of Haskell's power of composing small functions together, as seen in

“condition <- bracketed expr”

and

“stringTok "else" >> block”.

This composability allowed me to piece together simple parsers, achieving more complex behavior without adding confusion. Finally, I wrote in a declarative style using the **do** notation, eliminating the need for intricate loops or manual string handling. This approach, combined with some point-free style elements like optionalParser, focused on the parsing logic, making my code more concise and readable.

4. Haskell Language Features Used

Typeclasses and Custom Types: I used custom types, like the ADT, to represent different JavaScript constructs. The typeclasses allowed me to define generic behaviors, making the code more reusable.

My use of fmap, apply, and bind operators showcased Haskell's strength in handling functions. In lines like

condition <- bracketed expr

The bind (<-) operator helps sequence operations. The use of these functions allowed for more concise code while preserving strong type safety and functional purity.

The art of piecing together functions is at the heart of Haskell. In my code, function composition was evident in constructs like

stringTok "else" >> block

Where two parsing functions are seamlessly sequenced. This meant I could build complex behaviors from simpler building blocks, leading to more maintainable and understandable code.