FIT2102 Programming Paradigms 2025

Assignment 2: Generating Parsers From BNF

Due Date: ~~Wed 22 Oct~~ Fri 24 Oct 11:55 pm, Melbourne time (AEDT)
Weighting: 30% of your final mark for the unit
Interview: SWOTVAC (Week 13) + first week of exams (Week 14): 27th Oct – 9th Nov
Overview: Students will work **independently** to create a parser for the BNF grammar **specification** using functional programming techniques. Programs will be implemented in Haskell. **The goal is to demonstrate a good understanding of functional programming techniques as explored throughout the unit**, including written documentation of the design decisions and features.

Submission Instructions

Submit a zipped file named `<studentNo>_<name>.zip` which extracts to a folder named `<studentNo>_<name>`

- **It must contain all the code that will be marked** including the **report** and ***all code files***
- You also need to include a report describing your design decisions. The report must be named `<studentNo>_<name>.pdf`.
- **No additional Haskell libraries should be used.**
    - The exception to this is that you may use additional libraries only for testing purposes.
- **Do not submit the `.stack-work`, `node_modules`, or the `.git` folders.**
- **Make sure the code you submit executes properly.**

The marking process will look something like this:

1. Extract `<studentNo>_<name>.zip`
2. Copy the **submission** folder contents into the assignment code bundle submission folder
3. Execute `stack build`, `stack test` (for automated testing) and `stack run`/`npm run dev` for front end

Please ensure that you test this process before submitting. Any issues during this process will make your marker unhappy and may result in a deduction in marks.

Late submissions will be penalised at 5% per calendar day, rounded up. Late submissions more than seven days will receive zero marks and no feedback.

Table of Contents

Introduction

In this assignment, we will develop a transpiler using Haskell that converts BNF grammar into Haskell code. The task involves parsing input based on a provided BNF grammar and generating Haskell code that corresponds to it. The BNF grammar will be parsed into an Algebraic Data Type (ADT), and you will use that ADT to generate Haskell code. A web page is provided, in which the BNF grammar and a string to parse will be sent through an HTML-based websocket connection to a Haskell backend server, the Haskell server will need to convert this BNF grammar into the corresponding Haskell code, use it to parse the corresponding string and return it back to the website. A code scaffold is provided which will handle the basic communication between the web page and your assignment code.

You are encouraged to utilise materials covered in previous weeks, including solutions for applied questions, to aid in the development of your transpiler. **You must reference or cite ideas and code constructs obtained from external sources**, as well as anything else you might find in your independent research, for this assignment.

The assignment is split up into Part A (parsing), Part B (converting to Haskell code) and Part C (BNF modifiers), Part D (adding extra API) and Part E (BNF validation). However, we do recommend completing Part A/Part B in tandem.

The language you will parse will be based on the BNF specification. It is important that you read the requirements of each exercise carefully to avoid unnecessary work.

Goals / Learning Outcomes

The purpose of this assignment is to highlight and apply the skills you have learned to a practical exercise (parsing):

- Use functional programming and parsing effectively
- Understand and be able to use key functional programming principles (higher-order functions, pure functions, immutable data structures, abstractions)
- Apply Haskell and FP techniques to parse non-trivial BNF parsing

Scope of Assignment

You are only required to **parse** an expression into the necessary data types and convert the result to Haskell code such that it can be run using Haskell. You will **not** be required to write your own Haskell interpreter.

Exercises (24 marks)

These exercises provide a structured approach for creating the beginnings of a transpiler.

- Part A: (6 marks) Parsing BNF Grammar
- Part B: (4 marks) Converting parsed BNF into Haskell Code
- Part C: (6 marks) Modifiers
- Part D: (3 marks) Adding button to save Haskell code file
- Part E: (5 marks) BNF Validation
- **(Extension) Part F:** extensions for bonus marks!

You must parse the input into an intermediary representation (ADT) such as an abstract syntax tree to receive marks. This will allow easy conversion between your ADT and Haskell code.

You must add `deriving Show` to your ADT and all custom types your ADT contains. (Note that the skeleton code already has `deriving Show` on the `ADT` type for you, which you must not remove.) **You must not override this default Show instance as this will help us test your code.**

Your `Assignment.hs` file ***must*** exactly export the following:

- `bnfParser :: Parser ADT`
- `generateHaskellCode :: ADT -> String`
- `validate :: ADT -> [String]`
- Your `ADT` type

If your code does not have the aforementioned functions, with those **exact** types, you will get **zero marks for correctness**. If your code does not compile, you will also get **zero marks for correctness.**

Example Scripts

For each of these exercises, there will be a series of provided BNF grammar files. By running `stack test`, it will try to parse the BNF and save the output to a folder. This will generate Haskell code which you can manually view for correctness, or use your GHCi skills to try and load the file. During marking, we will be running your code on more complex examples than the provided example scripts; therefore, it is important you devise your own test cases to ensure your parser is valid on more complex BNFs. Feel free to share test cases with your peers. It will also produce a *git diff*, which is the difference between your output and the expected output. However, this requires installing the git command line tool, so please ensure that it is installed. Note that, each test case could have multiple possible correct outputs, depending if you have completed validation or not.

Furthermore, the more recommended way to test your code is to use `npm run dev` in combination with `stack run main` can be used to run the webpage with a live editor, running your code in real-time.

Part A (6 marks): Parsing BNF Grammar

Your task is to implement a parser that parses the BNF grammar and produces an Algebraic Data Type (ADT) that represents the structure of the grammar.

```
bnfParser :: Parser ADT
```

BNF grammars define a set of rules for constructing valid strings in a language. Each grammar consists of:

- **nonterminal symbols**: These represent abstract elements in the language (e.g., `<expression>`, `<term>`, `<factor>`). A nonterminal must start with a lowercase letter and may be followed by any combination of letters, digits or underscores (`_`). In particular, it cannot contain whitespace or the `>` character.
- **Terminal symbols**: These are the concrete tokens in the language (e.g., numbers, operators, keywords). Terminals will not contain any double quotes and you do **not** need to consider escape characters.
- **Production rules**: These define how nonterminal symbols can be expanded into sequences of nonterminals and terminals. Each rule must be on its own line.

Leading or trailing (non-newline) whitespaces of any of the components above should be ignored. Between any two nonterminal symbols, terminals, or macros, there must be **at least one** (non-newline) whitespace. There may be 1 or more non-newline whitespace characters before and after the `::=` and `|`. Any blank lines in the grammar should be **ignored**.

```
<expression> ::= <term> | <term> "+" <expression>
<term>       ::= <factor> | <factor> "*" <term>
<factor>     ::= "(" <expression> ")" | <number>
<number>     ::= [int]
```

In addition to the standard BNF rules, you also need to define *macros*. These macros are special symbols representing predefined patterns that can be used within the grammar to simplify its structure.

- `[int]` represents an integer.
- `[alpha]` represents an alphabetical string.
- `[newline]` represents a new line.

There may be edge cases to this parsing, but if the requirement is not specifically mentioned in this document, it will **not** be tested by us, so make sure your parser has reasonable behaviour, and does not crash.

Part B (4 marks): Converting to Haskell Code

The second part of this task requires you to convert your ADT into a Haskell representation. The resulting Haskell file must be formatted such that it is indented correctly, where the rules are defined in the same order as the grammar.

```
generateHaskellCode :: ADT -> String
```

Creating the Types

Each production rule in the grammar should be matched with a specific Haskell data type, ensuring that the structure of the grammar is preserved in the type system.

```
<expression> ::= <term> | <term> "+" <expression>
<term>       ::= <factor> | <factor> "*" <term>
<factor>     ::= "(" <expression> ")" | <number>
<number>     ::= [int]
```

Each nonterminal rule gets a data type:

- For each **nonterminal** (e.g., `<expression>`, `<term>`, `<factor>`), you need to define a corresponding Haskell data type. The name of the data type should be the capitalised name of the nonterminal. There are two alternatives depending on the number of alternatives:

Use `data` for multiple alternatives or multiple fields:

- If a nonterminal has multiple alternatives (different ways it can be constructed, separated by `|` in the BNF grammar), or it has an alternative with multiple fields, define a **numbered constructor** for each alternative. This number must also exist for **`data`** definitions with a single alternative.
- For example, if `<expression>` can be formed in two ways, as `<term>` or `<expression> "+" <term>`, the constructors would be `Expression1` and `Expression2`.
- **Indentation Rule**:
    - Each `|` (pipe symbol) must **align exactly** with the `=` sign on the line above it. This ensures clean, readable code.
    - **`deriving Show`** must be intended by exactly 4 spaces.

Use `newtype` for single-alternative and single-field rules:

- If a nonterminal has **only one possible alternative and one field**, represent it using `newtype`. The constructor should not be numbered.

```
data Expression = Expression1 Term
                | Expression2 Term String Expression
    deriving Show

data Term = Term1 Factor
          | Term2 Factor String Term
    deriving Show

data Factor = Factor1 String Expression String
            | Factor2 Number
    deriving Show

newtype Number = Number Int
    deriving Show
```

You do not need to worry about the generated types conflicting with any built-in Haskell types.

Creating the Parsers

You do not need to validate the parsers are correct and terminate (yet - *subtle foreshadowing*). However, you need to ensure that the produced code is syntactically correct.

Each rule gets a parser:

- For each **nonterminal** in the grammar (e.g., `<expression>`, `<term>`, `<factor>`), define a corresponding **parser** function.
- Each parser reflects the structure of its associated nonterminal in the BNF.

Use `<|>` for multiple alternatives:

- If a nonterminal has **multiple alternatives** (different ways it can be constructed, separated by `|` in the BNF grammar), use the `<|>` operator to represent these alternatives.

- **Numbered Constructors**: Use the numbered constructors (e.g., `Expression1`, `Expression2`) for different alternatives in the parser, just as you would in the data types.
- For readability, align the beginning of each `<|>` (alternation operator) **exactly with the `=` sign** on the line above it.

Use `<$>` and `<*>` for sequencing parsers:

- Use `<$>` to apply constructors to the result of a parser and `<*>` to sequence parsers that must occur in a specific order.

Use the `string` function for terminals, wrapped in brackets.

Macros:

- `[int]` will become `int`
- `[alpha]` will become `(some alpha)`
- `[newline]` will become `(is '\n')`

```
expression :: Parser Expression
expression = Expression1 <$> term
           <|> Expression2 <$> term <*> (string "+") <*> expression

term :: Parser Term
term = Term1 <$> factor
     <|> Term2 <$> factor <*> (string "*") <*> term

factor :: Parser Factor
factor = Factor1 <$> (string "(") <*> expression <*> (string ")")
       <|> Factor2 <$> number

number :: Parser Number
number = Number <$> int
```

The expected output for the entire parser is shown below. At the end of the file, there should be **exactly one extra new line**.

```
 data Expression = Expression1 Term
                 | Expression2 Term String Expression
     deriving Show
```

```haskell
data Term = Term1 Factor
          | Term2 Factor String Term
    deriving Show

data Factor = Factor1 String Expression String
            | Factor2 Number
    deriving Show

newtype Number = Number Int
    deriving Show

expression :: Parser Expression
expression = Expression1 <$> term
           <|> Expression2 <$> term <*> (string "+") <*> expression

term :: Parser Term
term = Term1 <$> factor
     <|> Term2 <$> factor <*> (string "*") <*> term

factor :: Parser Factor
factor = Factor1 <$> (string "(") <*> expression <*> (string ")")
       <|> Factor2 <$> number

number :: Parser Number
number = Number <$> int
```

You do not need to worry about the generated parsers conflicting with any builtin Haskell functions/constants.

Part C (6 marks): Modifiers & Parameterised Rules

Modifiers

The parser and Haskell code generator should now be updated to support regex-like modifiers `tok`, `*`, `+`, and `?` (3 marks) and variable rules (3 marks), which can be applied to terminals, nonterminals, and macros.

- The `tok` modifier is used to remove trailing whitespace: `tok` *`expression`*
    - When `tok` is applied to a single terminal string, the resulting parser should be `(stringTok <string>)`. For example, `tok "+"` should generate `(stringTok "+")`.

    - When `tok` is applied to a single nonterminal symbol or a macro, the resulting parser should be `(tok parser)` where *`parser`* is the original parser for that symbol. For example, `tok [alpha]` should generate `(tok (some alpha))`, and `tok <nonterminal>` should generate `(tok nonterminal)`. Note that you need to enclose the `tok` inside brackets.
    - There must be at least one (non-newline) whitespace character after `tok` and before the expression.
- `*` indicates zero or more repetitions: *`expression`*`*`. *`expression`*`*` should generate the parser `(many parser)`.
- `+` indicates one or more repetitions: *`expression`*`+`. *`expression`*`+` should generate the parser `(some parser)`.
- `?` indicates zero or one occurrences: *`expression`*`?`. *`expression`*`?` should generate the parser `(optional parser)`.

Each modifier may be applied to a single symbol (terminal, nonterminal, or macro). You **do not** need to handle

- modifiers applied to a sequence of multiple symbols (e.g. `tok (<a> <b>)`, or `(<a> <b>)*`
- modifiers applied to an entire alternation (e.g. `tok (<a> | <b>)`, `(<a> | <b>)*`)
- nested modifiers such as `(tok <a>)*`

Parameterised Rules

Rules may now take **parameters**. A rule definition is in the form:

`<name(a, b, …)> ::=` *`expression`*

- Parameters must be **single unique lowercase letters**.
- Parameters are **in scope only within the RHS** of the rule.
- A parameter reference is written as `[a]`.
- Every parameter reference on the right-hand side **must** appear in the left-hand side parameter list.
    - This condition should be checked immediately while parsing the right-hand side, not delayed until after the whole rule is parsed.
- You must ignore any trailing and leading whitespace around the commas `,`.

You may then apply a parameterised rule later in the grammar:

- `<pair(a, b)> ::= [a] "," [b]`

- `<numberPair> ::= <pair([int], [int])>`

Only a single symbol or a single symbol with a modifier applied to it can be used as the argument for a parameterised rule. For example, the following are valid:

- `<pair(<nonterminal>, [int])>`
- `<pair("a", tok [int])>`
- `<pair(<nonterminal>*, "a"+)>`
- `<pair([int], <pair("a", "b")>)>`
- `<pair([x], [x])>` (where `x` is a defined parameter)

but the following is invalid:

- `<pair("a" "b")>`

You will **not** be tested on and do not have to deal with the case where a parameterised rule is applied with the incorrect number of arguments, such as `<pair>`, `<pair("a")>`, or `<pair("a", "b", "c")>`.

For a parameterised rule definition `<name(a, b, …)> ::= expression`:

- The generated data type should have 1 type variable per parameter, and each type variable should have the same name as the parameter.
- Instead of generating a parser, it should generate a function that takes in one parameter (a parser) for each parameter in the rule, with the same name. The function would then return the parser for the parameterised nonterminal. The type of this function should be `Parser a -> Parser b -> … -> Parser (Name a b …)`.
- When the parameterised nonterminal is used `<name(expression_1, expression_2, …)>`, it should generate the parser `(name parser_1 parser_2 …)` where `parser_i` is the corresponding parser for `expression_i`.

Example Grammar

```
<program>    ::= <statement>*
<statement>  ::= "console.log(" <expression> ");" <comment>? [newline]
<expression> ::= <term> | <term> "+" <expression>
<term>       ::= <factor> | <factor> "*" <term>
<factor>     ::= tok <number> | tok <variable> | tok "(" <expression> tok ")"
<number>     ::= [int]
<variable>   ::= [alpha]
<pair(a, b)> ::= [a] "," [b]
<numberPair> ::= <pair([int], [int])>
<comment>    ::= " // " [alpha]
```

```haskell
newtype Program = Program [Statement]
    deriving Show

data Statement = Statement1 String Expression String (Maybe Comment) Char
    deriving Show

data Expression = Expression1 Term
                | Expression2 Term String Expression
    deriving Show

data Term = Term1 Factor
          | Term2 Factor String Term
    deriving Show

data Factor = Factor1 Number
            | Factor2 Variable
            | Factor3 String Expression String
    deriving Show

newtype Number = Number Int
    deriving Show

newtype Variable = Variable String
    deriving Show

data Pair a b = Pair1 a String b
    deriving Show

newtype NumberPair = NumberPair (Pair Int Int)
    deriving Show

data Comment = Comment1 String String
    deriving Show

program :: Parser Program
program = Program <$> (many statement)

statement :: Parser Statement
statement = Statement1 <$> (string "console.log(") <*> expression <*>
(string ");") <*> (optional comment) <*> (is '\n')

expression :: Parser Expression
expression = Expression1 <$> term
           <|> Expression2 <$> term <*> (string "+") <*> expression

term :: Parser Term
term = Term1 <$> factor
     <|> Term2 <$> factor <*> (string "*") <*> term
```

```
factor :: Parser Factor
factor = Factor1 <$> (tok number)
       <|> Factor2 <$> (tok variable)
       <|> Factor3 <$> (stringTok "(") <*> expression <*> (stringTok ")")

number :: Parser Number
number = Number <$> int

variable :: Parser Variable
variable = Variable <$> (some alpha)

pair :: Parser a -> Parser b -> Parser (Pair a b)
pair a b = Pair1 <$> a <*> (string ",") <*> b

numberPair :: Parser NumberPair
numberPair = NumberPair <$> (pair int int)

comment :: Parser Comment
comment = Comment1 <$> (string " // ") <*> (some alpha)
```

Part D (3 marks): Add a Save Button

This task involves changing the webpage to include extra capabilities, allowing a more feature-full UI. You **will not** be marked on the layout or ease of use of features, as long as they are clearly visible to your marker, e.g., a button should be clearly visible on the screen. This task will involve some light additions to both the HTML page and TypeScript code. This will likely involve creating an observable stream for the data, merging it into the subscription stream, and sending the information to the Haskell backend. The communicated information between the Haskell backend and the webpage will need to be updated to include additional information that the user wants the engine to achieve.

- A button must be added to the webpage for **saving,** where the generated Haskell file is saved **using** Haskell. The user does not need to be prompted for a file name, and the Haskell code should be saved according to the name of the first defined parser. A document-level comment must be added to the saved Haskell file formatted in ISO 8601 format for the current date and time: `YYYY-MM-DDTHH-MM-SS`. The function `getTime` is provided, which will provide you this time in an `IO String` format.

```
module Output where
.
```

```
.
. (Rest of Haskell)
```

Part E (5 marks): BNF Validation

In this task, you will implement a validation function to check for common issues in BNF grammars.

```
validate :: ADT -> [String]
```

You will extend your parser generator with a validation step that performs three types of checks on the input grammar before generating code. If any issues are found, a list of warnings should be returned.

- **Undefined nonterminals**: If a rule refers to a nonterminal that is never defined elsewhere in the grammar.
- **Duplicate rules**: If two rules define the same nonterminal.
- **Left recursion**: A rule is left-recursive if it immediately refers to itself as the first element of the sequence in any of the alternatives in the rule. This also applies when the nonterminal references itself indirectly via a cycle in left-most elements via other nonterminals.
    - The reason why this is a warning is because the generated parsers for left-recursive rules would go into an infinite loop.

The strings returned for each warning must be:

- For undefined nonterminals: `Undefined nonterminal: rule name`
- For duplicate rules: `Duplicate rule: rule name`
- For left recursion: `Left recursion in: rule name`

where *`rule name`* is the name of the rule.

The order of the warnings returned by `validate` do not matter.

To ensure you produce valid Haskell code, you must remove all BNF rules that produce these warnings. For all instances of duplicate rules, you must keep only the first instance of the rule. (This requires you to modify your `generateHaskellCode` function.)

Validation is not a one-time step; it must be performed **iteratively**. Resolving one problem can introduce others. For instance, eliminating a left-recursive rule might lead to a reference to a now-missing nonterminal. Since fixing one issue can introduce others, validation should continue until no new problems are found. However, the warnings returned by `validate` should only return the warnings that arose from the **first** iteration.

The *first* step should be to report, then remove duplicate non-terminals, ensuring you always keep the first rule of the duplicates, followed by checking the other validation rules.

You will not be tested on and do not have to worry about the following:

- Duplicate rules that also have other warnings, i.e. duplicate rules that also reference undefined nonterminals or are left-recursive.
- Left-recursive rules that use parameterised nonterminals. For example, if you have the rules `<pair(a, b)> ::= [a] "," [b]` and `<x> ::= <pair(<x>, [int])>`, `x` is left-recursive, but you do not need to report a warning.

For example, given the following grammar:

```
<start> ::= <expr>
<expr> ::= <term> "+" <expr> | <term>
<term> ::= <factor>
<factor> ::= <expr> "*" <factor> | <number>
<number> ::= [int]
<duplicated> ::= "a"
<duplicated> ::= "b"
```

The unordered set of warning strings should be

- `Left recursion in: expr`
- `Left recursion in: term`
- `Left recursion in: factor`
- `Duplicate rule: duplicated`

Note that the left recursion warnings in `expr`, `term` and `factor` are due to the cycle they form in their leftmost non-terminal rules: `expr` references `term`, which references `factor`, which references `expr`.

After removing the offending rules (`expr`, `term`, `factor`, `duplicated`), we would get this grammar:

```
<start> ::= <expr>
<number> ::= [int]
<duplicated> ::= "a"
```

Now, `start` uses an undefined nonterminal `expr`, so it would be removed, and the final Haskell code would be based on the below grammar:

```
        <number> ::= [int]
        <duplicated> ::= "a"
```

Part F (up to 6 bonus marks): Extension

Implement anything that is interesting, impressive, or otherwise “shows off” your understanding of Haskell, functional programming, and/or parsing.

To achieve the maximum amount of bonus marks, the feature should be similar in complexity to Part C. If you are in doubt about whether your extension is compelx enough, feel free to discuss with staff during consultations.

The bonus marks only apply to **this assignment,** and the final mark for this assignment is **capped** at 30 marks (100%). This means you cannot score more than 30 marks or 100%.

Some suggestions for extensions of varying complexity and difficulty:

- Add more Macros (up to 2 marks)
- Create BNF for our extended BNF (up to 2 marks)
- Support constructs such as `tok (<a> | <b>)`
- Support escape characters in strings (e.g. `"\t"`)
- Improve validation:
    - Ensure that parameterised nonterminals are called with the correct number of arguments
    - Detect left-recursive rules that use parameterised nonterminals
- **Comprehensive** test cases over the parser, converting to Haskell code, and validation
    - Warning: It is super hard to be comprehensive; stay away unless you love testing.
- Others

(Choosing one of the simpler suggestions to implement may not receive the maximum available marks.)

Report (2 marks)

You are required to provide a report in PDF format of max. 1000 words (markers will not mark beyond this word limit). Descriptions of extensions can use up to 200 words per extension feature. Your report does not need to be long, but we have increased the word limit for anyone who wants to use more words, e.g., to document important design decisions. *We expect your report to be of similar length to A1*

Make sure to summarise the intention of the code, and highlight the interesting parts and difficulties you encountered. Focus on the **"why", not the "how"**.

Additionally, just posting screenshots of code is **heavily discouraged**, unless it contains something of particular importance. Remember, markers will be looking at your code alongside your report, so we do not need to see your code twice.

Importantly, this report must include a description of why and how parser combinators helped you complete the parsing. In summary, your report should include the following sections:

- Design of the code (including data structures)
    - High-level description of approach
    - High-level structure of code
    - Code architecture choices
- Parsing
    - Usage of parser combinators
    - Choices made in creating parsers and parser combinators
    - How parsers and parser combinators were constructed using the Functor, Applicative, and Monad typeclasses
- Functional programming (focusing on the **why**)
    - Small modular functions
    - Composing small functions together
    - Declarative style (including point-free style)
- Haskell language features Used (focusing on the **why**)
    - Typeclasses and custom types
- Higher order functions, `fmap`, apply (`<*>`), bind (`>>=`) (if you used them)
    - Function composition
- Description of extensions (if applicable)
    - What you intended to implement
    - What you did implement
    - What is cool/interesting/complex about it
    - This may include using Haskell features that are not covered in course content

There is some overlap between the sections. You should **avoid** repeating descriptions or ideas in the report.

Code Quality (4 marks)

Code quality will relate more to how understandable your code is. You must have readable and **functional** code, commented when necessary. Readable code means that you keep your lines at a reasonable length (< 80 characters), that you provide comments above non-trivial functions, and that you comment sections of your code whose function may not be clear.

Your functions should all be small and modular, building up in complexity, and taking advantage of built-in functions or self-defined utility functions when possible. It should be easy to read and understand what each piece of your code is doing, and why it is useful. Do **not** reimplement library functions such as `map`, and use the appropriate library function when possible.

Your code should aim to re-use previous functions as much as possible, and not repeat work when possible.

Code quality includes your ADT and if it is well structured, i.e., does not have a bunch of repeated data types and follows a logical manner (the JSON example from the applied session is a good example of what an ADT should look like).

Marking breakdown

The main marking criteria for each parsing and pretty printing exercise consists of two parts: **correctness** and **FP style.** Both correctness and FP style will be worth 50% of the marks for each of the exercises, i.e., if your code passes all tests, you will get at least half marks for Exercise A, and Exercise B.

You will be provided with some sample input and tests for determining the validity of the outputted Haskell files. The sample inputs provided will **not** be exhaustive, you are **heavily** encouraged to add your own, perhaps covering edge cases.

Correctness

We will be running a series of tests which test each exercise, and depending on how many of the tests you pass, a proportion of marks will be awarded.

FP Style

FP style relates to if the code is done in a way that aligns with the unit content and functional programming.

You must apply concepts from the course. The important thing here is that you need to use what we have taught you effectively. For example, defining a new type and its `Monad` instance, but then never actually needing to use it will not give you marks. Note: using bind (`>>=`) for the sake of **using the** `Monad` when it is not needed will not count as "effective usage."

Most importantly, code that does not utilise Haskell's language features, and that attempts to code in a more imperative style, will not be awarded high marks.

Minimum Requirements

An estimate of a passing grade will be completing parts A and B. However, this will need to be accompanied by high code quality and a good report.

A higher mark will require parsing of the more difficult data structures, and modifications of the HTML page.

Changelog

- 1 Oct: Clarify definition of left-recursive cycles to make it more precise
- 4 Oct: Change ‘four types of checks’ to ‘three types of checks’ to be more accurate in Part E

- 4 Oct: Fix expected Haskell code output for example grammar in Part C (changed `Expression1 [Term]` to `Expression1 Term` and `Term1 [Factor]` to `Term1 Factor`)
- 7 Oct: Fix expected Haskell output for example grammar in Part B to match the `examples/expected_output/simple/1.hs` in the code bundle
- 13 Oct: Fix expected Haskell output for example grammar in Part C to match the `examples/expected_output/modifiers/1.hs` in the code bundle (fix type of `numberParser` to be `Parser NumberParser`)
- 15 Oct: Clarify whitespace parsing requirements in Part A (changed 'Between any two symbols there must be at least one (non-newline) whitespace.' to 'Between any two nonterminal symbols, terminals, or macros, there must be **at least one** (non-newline) whitespace. There may be 1 or more non-newline whitespace characters before and after the `::=` and `|`.'). You will not be tested on situations where there are 0 non-newline whitespace characters before or after the `::=` or `|`.
- 16 Oct: Remove the sentence 'These rules must then be displayed on the web page underneath the HTML Output.' from part E
- 17 Oct: Clarify interview dates
- 18 Oct: Clarify that in Part D, the generated Haskell code should be saved (not the 'parser output')
- 19 Oct: Change the `<pair([x], [x]?)>` example to `<pair([x], [x])>`. You will not be tested on `tok [x]`, `[x]*`, `[x]+`, or `[x]?`.
- 20 Oct: Extended due date to Friday
- 23 Oct: Changed 'Abstract Data Type' to 'Algebraic Data Type'