2025 Haskell January Test
Recursive-Descent Parsing

THREE HOURS TOTAL

The maximum mark is 30.

The **2025 Haskell Programming Prize** will be awarded for the best overall solution, or solutions.

- Please make your swipe card visible on your desk.
- Work in the file named `Parser.hs` inside the `parser/src` subfolder of your Home folder. **Do not move any files**.
- You are **not permitted** to edit the folder structure, or edit the `hft.cabal`, `cabal.project`, `src/Lexer.hs`, or `src/Types.hs` files without having been directed by the Examiners; any changes will be reverted.
- You may add additional tests, which will be reflected in the final script (*like in the PPTs*), but these will not be assessed. However, any changes made to the code that cause the **original given** tests to not compile will incur a compilation penalty.

It is important that you comment out any code that does not compile before you complete the test. There will be a TWO MARK PENALTY applied to any program that does not compile.

Credit will be awarded throughout for style, clarity and the appropriate use of Haskell's various language features including list comprehensions, higher-order functions, functors, monads etc.

1 Introduction

A programming language parser takes as input the textual representation of a program (a `String`) and generates as output the internal representation of that program – the program's so-called *Abstract Syntax Tree* – here as a Haskell data type.

The objective of this exercise is to build a parser for a very simple language, akin to a stripped-down version of Kotlin or C. The parser will work by *recursive-descent*, essentially by encoding of the rules of the language's *grammar* (see below) as a recursive program for decomposing the program text (a string) into its components hierarchically. While hand-written recursive-descent parsers are rarely used in practice, writing one can be valuable for understanding how parsing tools operate.

1.1 Program structure

A program in the language is a *block* comprising one or more *statements*, each of which is either an *assignment* or a *while loop*. Multiple statements within a block are separated by semicolons (`;`).

An assignment is of the form `v = e` where `v` is a variable and `e` is an expression. Expressions can be thought of as 'sums of products', reflecting the fact that multiplication (`*`) has higher precedence than addition (`+`). The individual *atoms* forming the arguments to `+` and `*` are either variables, integer constants or bracketed subexpressions; both `+` and `*` are left associative. Thus, `x = 1+2*(z+3)+y` will be parsed as though it were bracketed `x = (1+(2*(z+3)))+y`.

A while loop is of the form `while c do {b}` where `c` is an expression (the 'condition') and `b` is a block, delimited by curly braces, comprising at least one statement, as above.

You will not be required to implement the language, so any details of a program's execution are irrelevant to the exercise. However, the following can be assumed for completeness:

- There are no booleans in the language, only integers: the integer `0` represents 'false' and any other integer represents 'true'. Thus, a while loop will only terminate if the 'condition' is the integer `0`.
- Any inputs to the program can be assumed to be pre-assigned to the special variables `in1`, `in2`, etc. and any results generated to be assigned to the special variables `res1`, `res2`, etc.

As an example, the following string represents a program that will compute the factorial of a given number passed to it via `in1`:

```
fact :: String
fact = "acc = 1;
        n = in1;
        while n {
          acc = acc * n;
          n = n + -1
        };
        res1 = acc"
```

Notice that subtraction has to be implemented using a combination of addition and negation. Note also that there is no semicolon at the end of a block; in other words, semicolons are *separators*.

1.2 Representation

The following types are included in `Types.hs` and should be self-explanatory from the above:

```
type Program = Block

type Block = [Stmt]
```

```
data Stmt = Asgn String Expr
          | While Expr Block
          deriving (Eq, Show)

data Expr = Add Expr Expr
          | Mul Expr Expr
          | Val Int
          | Var String
          deriving (Eq, Show)
```

The abstract syntax tree (type `Program`, equivalently `Block`) for the `fact` program above, suitably formatted to aid readability, is:

```
  [Asgn "acc" (Val 1),
   Asgn "n" (Var "in1"),
   While (Var "n") [
     Asgn "acc" (Mul (Var "acc") (Var "n")),
     Asgn "n" (Add (Var "n") (Val (-1)))
   ],
   Asgn "res1" (Var "acc")]
```

1.3 Parsing rules

The *grammar* for a programming language specifies what it means for a given program to be syntactically well-formed. Formally speaking, a parser for a language will 'accept" any input string (i.e. program) that conforms to the language's grammar; in practice, of course, it will also generate an abstract syntax tree representing the program as part of the parsing process.

The following rules define the grammar of the language describe above. It is not necessary to understand all the nuances of grammar theory in this exercise, but the items enclosed in 'single quotes' are *terminal*s, i.e. those strings that can appear verbatim in a program. The items in ⟨*angle brackets*⟩ are *non-terminals* which specify the structural components of the language and how they fit together. The `::=` can be read as "is defined to be" and `|` as "or". The symbol ϵ will be explained below.

⟨*block*⟩ ::= ⟨*stmt*⟩ | ⟨*stmt*⟩ '`;`' ⟨*block*⟩

⟨*stmt*⟩ ::= ⟨*ident*⟩ '`=`' ⟨*expr*⟩ | '`while`' ⟨*expr*⟩ ⟨*body*⟩

⟨*body*⟩ ::= '`{`' ⟨*block*⟩ '`}`'

⟨*expr*⟩ ::= ⟨*term*⟩ ⟨*expr'*⟩

⟨*expr'*⟩ ::= '`+`' ⟨*term*⟩ ⟨*expr'*⟩ | ϵ

⟨*term*⟩ ::= ⟨*atom*⟩ ⟨*term'*⟩

⟨*term'*⟩ ::= '`*`' ⟨*atom*⟩ ⟨*term'*⟩ | ϵ

⟨*atom*⟩ ::= ⟨*nat*⟩ | ⟨*ident*⟩ | '`(`' ⟨*expr*⟩ '`)`'

⟨*nat*⟩ ::= `[0-9]+` | '`-`' `[0-9]+`

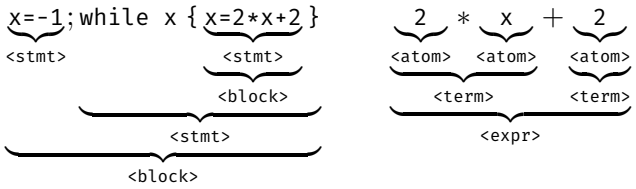
⟨*ident*⟩ ::= `[a-zA-Z][a-zA-Z0-9]*`

The ⟨*block*⟩ rule states, for example, that a block is a statement on its own or a statement followed by a semicolon and a block – note the inherent recursion. In other words, it comprises either a single statement or multiple statements separated by semicolons.

Similarly, an expression is a term **optionally** followed by a `'+'` character followed by a term followed by an ⟨*expr*'⟩. Note that '$|\,\epsilon$' implements the 'optional' part of the rule: the parser is allowed to "do nothing" if it can't see a `'+'`. Thus, an expression is either a term on its own or multiple terms separated by `'+'` symbols. Exactly the same principle applies to terms, but with `'*'` separating atoms.

It is not necessary to understand the last two lines in detail as they are considering tokenisation, which is done for you (see below), but the notation `"[0-9]+"` means a string comprising one or more occurrences of the digits `'0'` to `'9'`, e.g. `1492`. The notation `"[a-zA-Z][a-zA-Z0-9]*"` means an alphabetic character (`'a'`...`'z'` or `'A'`...`'Z'`) followed by zero or more occurrences of a character which is either an alphabetic character or digit, e.g. `x`, `B52`, `sec3B`.

As an example, here is how the program text `"x=-1; while x {x=2*x+2}"` is broken down; the breakdown of the expression `"2*x+2"` is shown separately on the right:



For completeness, the atom `-1` is formed by the terminal symbol `'-'` followed by a `'1'`.

The parsing rules can be thought of as an encoding of the above grammar, so building a recursive-descent parser is simply(!) a question of implementing the grammar rules in Haskell, and augmenting them so that the abstract syntax tree is computed as you go along.

Referring back to the Haskell representation above, a ⟨*term*⟩ will be encoded using `Mul`, an ⟨*expr*⟩ will be encoded using `Add`, each ⟨*stmt*⟩ will be encoded either as an `Asgn` or a `While`, a ⟨*block*⟩ will be encoded as a list of `Stmt`, and so on. A separate copy of this grammar can be found alongside this file.

1.4 Tokenisation

A parser typically works by first turning a given input string into a list of *tokens* (`[Token]`), with each token representing a valid 'terminal' symbol in the language, e.g. `+`, `(`, `while` etc., a variable, e.g. `v`, `thx1138`), or a non-negative integer constant, e.g. 0, 10, 3544. The following data type for representing tokens should be self-explanatory (`LBrace` and `RBrace` represent { and } respectively):

```
data Token = Eq | Plus | Minus | Times | LParen | RParen | LBrace | RBrace |
             Semi | Nat Int | Ident String | WhileTok
           deriving (Show, Eq)
```

For example, the tokens generated from the `fact` function above are as follows:

```
[Ident "acc",Eq,Nat 1,Semi,Ident "n",Eq,Ident "in1",Semi, WhileTok,
 Ident "n",LBrace,Ident "acc",Eq,Ident "acc",Times,Ident "n",Semi,
 Ident "n",Eq,Ident "n",Plus,Minus,Nat 1,RBrace,Semi,Ident "res1",
 Eq,Ident "acc"]
```

A predefined tokeniser is provided, so you will not need to write one yourself.

1.5 Handling errors

One of the challenges in the design of a parser is reporting meaningful error messages, should the program fail to conform to the language's grammar. Unlike the interim test, we will not assume that the input to the parser is *well-formed* in the above sense; this makes the logic more complex.

The idea is to have a parser take a list of `Tokens` and return an object of type `Error` should something go wrong and a pair of type `([Token], a)` if the parse successfully yielded an object of type `a`. To support both outcomes we'll use the `Either` type, hence the following type synonym, which can be found in `Types.hs`:

```
type Parser a = [Token] -> Either Error ([Token], a)
```

Here, `a` denotes the type of object being parsed, e.g. an assignment (`Asgn`), expression (`Expr`) etc., and the `[Token]` is list of tokens remaining, having successfully parsed the 'a'. For example, if the input is the string `"x=1; y=2"`, which tokenises to `[Ident "x", Eq, Val 1, Semi, Ident "y", Eq, Val 2]`, then an attempt to parse a statement using a parser of type `Parser Stmt` will return `Right ([Semi, Ident "y", Eq, Val 2], Asgn "x" (Num 1))`. Note that the first element of the pair is the list of tokens following the first statement and the second element of the pair is the parsed `Stmt` obtained from the tokens `[Ident "x", Eq, Val 1]`.

If a parse fails for any reason then the resulting `Error` will be wrapped in a `Left`. The various errors that can occur are given by the following, which is also defined in `Types.hs`:

```
data Error = BadChar Char
           | StmtNotFound (Maybe Token)
           | ExprNotFound (Maybe Token)
           | IntNotFound (Maybe Token)
           | Unexpected (Maybe Token) Token
           | UnparsedInput [Token]
           deriving (Eq, Show)
```

Each constructor contains enough information about the input for a meaningful error message to be manufactured. Further details of these errors will become apparent, but some examples, with explanations are shown below:

Input string	Error	Explanation
`x=1?  y=2`	`BadChar '?'`	Illegal character `'?'`
`x=1; 4y=2`	`StmtNotFound (Just (Nat 4))`	A statement (an assignment) was expected, but a 4 (token `Nat 4`) was found.
`x={1; y=2`	`ExprNotFound (Just LBrace)`	An expression was expected but left brace (`LBrace`) was found.
`x=(1; y=2`	`Unexpected (Just Semi) RParen`	A closing bracket (`RParen`) was expected, but a semicolon (`Semi`) was found.
`x=1 y=2`	`UnparsedInput [Ident "y",Eq,Nat 2]`	The statement `x=1` defines a valid program in its own right, but more text was found (`[[Ident "y",Eq,Nat 2]`).
`x=`	`ExprNotFound Nothing`	An expression was expected but there was no input after the '`=`'.
`x=-y`	`IntNotFound (Just (Ident "y"))`	`-` is not an operator, but forms part of a negative integer literal.

Recall that the input is a `String` and the error object will be wrapped in a `Left`.

2 What to do

There are three parts to the exercise. Parts II and III both build on Part I so you should complete Part I before moving on. If you get stuck in Part II you can attempt Part III, although there is considerably less hand-holding there than in the first two parts.

A function `mHead` – a "safe head" function that will prove useful when dealing with errors – is defined in the skeleton as follows:

```
mHead :: [a] -> Maybe a
mHead (x : _) = Just x
mHead _ = Nothing
```

Various inputs and token lists are defined in `Examples.hs` for testing and will be referred to below.

2.1 Part I

For this part of the exercise you should ignore both bracketed expressions, e.g. `x*(1+y)`, and while loops; these will be considered in Part III. Furthermore, expressions will be restricted to be variables (e.g. `Ident "x"`) or values (e.g. `Val (-1)`, `Val 224`); expressions involving addition and multiplication will be considered in Part II.

Because of the above restriction, expressions, terms and atoms in the grammar start the same. Therefore, the following definitions suffice for the term and expression parsing functions in Part I:

```
parseTerm :: Parser Expr
parseTerm = parseAtom

parseExpr :: Parser Expr
parseExpr = parseTerm
```

You should not modify these until you attempt Part II; however, you must be careful to use exactly the function required by the grammar in your code, so that your changes in Part II automatically work later.

1. Define a function `checkTok :: Token -> [Token] -> Either Error [Token]` that will check to see whether a given token appears at the head of a given list of tokens. If so, it should return the remaining tokens in the list, wrapped up in a `Right`. If the tokens don't match then it should return an `Unexpected` error, wrapped in a `Left`. The `Unexpected` constructor has two arguments: 1. '`Just`' the token at the head of the given list, or `Nothing` if the list was empty, and 2. the token that was expected. For example:

   ```
   ghci> checkTok Plus [Plus,Nat 4,RBrace]
   Right [Nat 4,RBrace]
   ghci> checkTok Plus [Minus,Nat 4,RBrace]
   Left (Unexpected (Just Minus) Plus)
   ghci> checkTok Plus []
   Left (Unexpected Nothing Plus)
   ```

   Hint: use `mHead` to acquire the unexpected token in the failure case.

   [2 Marks]

2. Define a function `parseAtom :: Parser Expr` that will parse an atom, noting that atoms are just a particular kind of expression.

   Note that the type signature is equivalent to `parseAtom :: [Token] -> Either Error ([Token], Expr)`, so the function takes a list of `Token`s and return either an error (`Left ...`) or a successfully parsed atom together with the tokens that follow those representing the atom (`Right ...`).

Referring back to the grammar, parsing a `Ident` token produces a `Var` atom and parsing a `Nat` token produces a `Val` atom. A `Minus` followed immediately by a `Nat` token is to be treated as one thing: the atom (`Val`) produced should be the negation of the integer within the `Nat`; you can use `negate` for this. If a `Minus` is seen without a corresponding `Nat`, this must produce a special error – `IntNotFound`. In all other cases it is not possible to parse an atom, so the function should return an `ExprNotFound` error. The argument to `ExprNotFound` and `IntNotFound` is of type `Maybe Token`, i.e. `Nothing` if the token list was null and the first token in the list otherwise (use `mHead`). Recall that bracketed subexpressions should be ignored for now, even though they are also atoms according to the grammar.

Some predefined token lists of the form `atomToksX` are provided in `Examples.hs` for testing `parseAtom`. For example:

```
ghci> parseAtom atomToks1
Right ([Plus,Ident "y",RBrace],Var "x")
ghci> parseAtom atomToks2
Right ([Plus,Ident "y",RBrace],Val 2)
ghci> parseAtom atomToks3
Right ([Plus,Ident "y",RBrace],Val (-2))
ghci> parseAtom atomToks4
Left (ExprNotFound (Just Plus))
```

[2 Marks]

3. Define a function `parseStmt :: Parser Stmt` that will parse a statement. As above, since while loops will be added in Part III, all statements at this point will be assignments.

   The token sequence for a valid assignment statement looks like this: `[Ident v, Eq, et1, ..., etN, etc.]` where the tokens `et1, ..., etN`[1] collectively represent an expression.

   There are two ways to proceed: you can either check for an `Ident` at the head of the token list and use `checkTok` to check that the next token is `Eq`, or you can check that the first two tokens are an Ident and an `Eq` in a single pattern match. Either way, if successful the next step is to parse an expression from the tokens `et1, ..., etN` that follow the `Eq`. If the expression parses successfully then the result will be of the form `Right (etc., e)` where `e` is the parsed expression and `etc.` is the list of tokens that remain after parsing `e` (this could be empty).

   If the assignment statement is invalid for any reason then the result will be of the form `Left err` where `err` is an `Error` that summarises what went wrong.

   Some predefined token lists of the form `asgnToksX` are provided in `Examples.hs` for testing `parseStmt`. For example:

```
ghci> parseStmt asgnToks1
Right ([RBrace],Asgn "x" (Val 1))
ghci> parseStmt asgnToks2
Right ([],Asgn "x" (Var "y"))
ghci> parseStmt asgnToks3
Left (StmtNotFound (Just (Nat 4)))
ghci> parseStmt asgnToks4
Left (ExprNotFound (Just RBrace))
-- There are two possible errors for asgnToks5:
ghci> parseStmt asgnToks5
Left (Unexpected (Just Times) Eq)
```

[1] In Part I the N is either 1 or 2, but the example shows the general situation: this should not affect your `parseStmt`.

```
-- Or:
ghci> parseStmt asgnToks5
Left (StmtNotFound (Just (Ident "x")))
```

Note that `asgnToks5 = [Ident "x",Times,Ident "y"]` begins with an `Ident` but the next token is not an `Eq`. Depending on how you define `parseStmt` the error returned can indicate either an unexpected token or a missing statement – both are acceptable for the purposes of testing.

Hint: `Either` is both a functor and a monad, so can you use `fmap` or `do`?

[4 Marks]

2.2 Part II

The restriction on expressions and terms is now removed so that they can include addition (`+`) and multiplication (`*`); the corresponding tokens are `Add` and `Mul` respectively.

Recall from the grammar that an expression is either a single term or multiple terms separated by a `+`. Similarly, a term is either a single atom or multiple atoms separated by a `*`. Note the similarity!

1. Define a function `parseTerm :: Parser Expr` that uses `parseAtom` to parse a term. You should use the grammar rules to help: first try parsing an atom representing the first (left-most) argument of the multiplication; if that succeeds then pass the atom and the remaining tokens to a helper function ($\langle$*term'*$\rangle$ in the grammar). The first argument of that function is an *accumulating parameter*, `t` say, and the second the current list of tokens, `toks`, say. It should do the following:

   - If the first token in `toks` is a `Times` then use `parseAtom` recursively to parse the second argument to the multiplication. If that succeeds, giving an atom `x` and remaining tokens `toks'`, say, then recursively invoke the helper function with the accumulator `Mul t x` and tokens `toks'`. In short, if the term is valid then each `Times` token causes an additional `Mul` expression to be accumulated in a manner that reflects the left-associativity of `*`.
   - If the first token is not a `Times` then we're done: `t` is the required term and `toks` the remaining tokens. Note that this corresponds to the ϵ symbol in the grammar, which allows us to stop parsing a larger term if the `Times` is missing.
   - If at any point an error is found (`Left`) then that error must be returned as the result.

   Some predefined token lists of the form `termToksX` are provided in `Examples.hs` for testing `parseTerm`. For example:

```
ghci> parseTerm termToks1
Right ([RBrace],Mul (Var "x") (Val 3))
ghci> parseTerm termToks2
Right ([],Mul (Mul (Var "x") (Var "y")) (Val 2))
ghci> parseTerm termToks3
Left (ExprNotFound (Just Semi))
ghci> parseTerm termToks4
Left (ExprNotFound Nothing)
```

   Hint: Again, recall that `Either` is a monad. **[4 Marks]**

2. Define a function `parseExpr :: Parser Expr` that works identically to `parseTerm` except that it looks for `Plus` tokens instead of `Times` and uses `parseTerm` instead of `parseAtom` to perform the recursive parsing.

   Given that the two functions above have identical structure, can you think of a nice way to capture that structure as a single function that can be parameterised in two different ways to define `parseExpr` and `parseTerm`? Two of the three marks for `parseExpr` are reserved for this.

   [3 Marks]

3. Define a function `parseBlock :: Parser Block` that will parse a block comprising either a single statement, or multiple statements separated by semicolons.

   Hint: There are several ways you can define this for full marks, but arguably the most efficient approach is to define a helper function that takes the current list of tokens and an accumulating parameter that contains the list of statements parsed so far. Initially this will be `[]`.

   Some predefined token lists of the form `blockToksX` are provided in `Examples.hs` for testing `parseBlock`. For example:

```
ghci> parseBlock blockToks1
Right ([],[Asgn "y" (Var "x")])
ghci> parseBlock blockToks2
Right ([],[Asgn "x" (Val 4),Asgn "y" (Var "x")])
ghci> parseBlock blockToks3
Left (ExprNotFound (Just RParen))
-- For blockToks4, either:
ghci> parseBlock blockToks4
Left (StmtNotFound (Just (Ident "x")))
-- Or:
ghci> parseBlock blockToks4
Left (Unexpected (Just Times) Eq)
ghci> parseBlock blockToks5
Left (StmtNotFound Nothing)
```

   As with `parseStmt` there are two equally valid errors for `blockToks4`, which comprises an expression, rather than a statement.

   Hint: Having successfully parsed one statement, if the next token is a `Semi` then what follows must be a block for the parse to succeed (recursion?); if not, you're done.

   [4 Marks]

4. Define a function `parse :: String -> Either Error Program` that will parse a string representing the text of a program (equivalent to a `Block`). A function `tokenise` is provided in `Lexer.hs` with type `tokenise :: String -> Either Error [Token]`. You should use this to tokenise the input; if this succeeds then you should attempt to parse the tokens generated as a `Block`. For this top-level parse to succeed there must be no input text remaining having parsed the block; if there is then an `UnparsedInput` error must be returned.

   Some predefined token lists of the form `progStrX` are provided in `Examples.hs` for testing `parse`. For example (recall that programs are blocks):

```
ghci> parse progStr1
Right [Asgn "res1" (Val 0)]
ghci> parse progStr2
Right [Asgn "v1" (Val 1),Asgn "v2" (Val 2),
       Asgn "res1" (Add (Var "v1") (Mul (Var "v2") (Val 3)))]
ghci> parse progStr3
Left (BadChar '?')
ghci> parse progStr4
Left (StmtNotFound (Just (Nat 4)))
ghci> parse progStr5
Left (ExprNotFound (Just LBrace))
ghci> parse progStr6
Left (UnparsedInput [Ident "res1",Eq,Nat 2])
```

   [4 Marks]

2.3 Part III

1. Extend your `parseAtom` function to accommodate parenthesised sub-expressions. For example:

```
ghci> parseAtom parenToks1
Right ([],Var "x")
ghci> parseAtom parenToks2
Right ([],Mul (Var "x") (Var "y"))
```

and, using `parseExpr`:

```
ghci> parseExpr parenToks3
Right ([],Mul (Var "x") (Add (Val 2) (Var "y")))
ghci> parseExpr parenToks4
Left (ExprNotFound (Just RParen))
ghci> parseExpr parenToks5
Left (Unexpected Nothing RParen)
```

Hint: you might find `checkTok` useful.

[2 Marks]

2. Extend your `parseStmt` function to accommodate while loops. For example:

```
ghci> parse fact == factParse
True
ghci> parse badFact1
Left (Unexpected (Just LParen) LBrace)
ghci> parse badFact2
Left (StmtNotFound (Just RBrace))
ghci> parse badFact3
Left (Unexpected (Just (Ident "acc")) RBrace)
```

Hint: you might find `checkTok` useful.

[5 Marks]

Final note: A function `printParse :: String -> IO ()` is provided in the skeleton which will summarise the result of parsing a given program string. In particular, errors are "pretty-printed" similarly to how they might be in a 'production' compiler. You may wish to experiment with this. For example:

```
ghci> printParse progStr1
Parse successful...
[Asgn "x" (Val 0)]
ghci> printParse progStr3
Parse error: Unrecognised character: '?'
ghci> printParse progStr4
Parse error: Statement expected, but 4 found
ghci> printParse progStr5
Parse error: Expression expected, but { found
```

Good luck!