CSCC24 2025 Summer Assignment 3

Due: July 21 11:59PM
This assignment is worth 10% of the course grade.

In this assignment, you will practice functional-logic programming in Curry and implement recursive descent parsing in Haskell.

As usual, you should aim for reasonably efficient algorithms and reasonably organized, comprehensible code.

Code correctness (mostly auto-testing) is worth 90% of the marks; code quality is worth 10%.

Question 1: Assignment 1 but in Curry [5 marks]

Re-implement `consume` from Assignment 1, but this time in Curry, and using Curry's built-in non-determinism instead of a list to represent having multiple or no answers. Hence the type signature becomes:

```
consume :: R -> String -> String
```

It follows that *xs* is in the language of *r* iff the empty string is among the answers from `consume r xs`:

```
inRL :: R -> String -> Bool
inRL r xs = consume r xs =:= ""
```

There are advanced requirements that are worth a minority of marks:

- `inRL r xs where xs free` should generate all and only strings that match *r*. It should terminate if there are only finitely many such strings (if *r* does not use `Star`).

  You may fulfill this easily. If the base cases `Eps` and `Single` work, it is very likely that the general case also works.

  Automarking will not test infinite cases with `Star`.

  There is an `upTo` function in REDef.hs to help you generate a finite approximation of `Star`, e.g., `upTo 3 r` is equivalent to $\epsilon + r + rr + rrr$. Then you can test `inRL (upTo 3 r) xs where xs free` for fun, and it is finite.

- `consume (Star r) xs` should terminate, e.g., `consume (Star (Single 'c')) "cc"`

Question 2: Parsing [10 marks]

"A mad scientist was inspired by Python, Javascript, and Haskell. This was what happened."

In this assignment, we will write a parser for a language syntax described below. Here is part of the grammar in EBNF; ambiguities and omissions are resolved in words after. The start symbol is `expr`.

```
1  expr ::= lambda | ifelse
2  lambda ::= var "=>" expr                       -- inspired by JS
3  ifelse ::= arith [ "if" expr "else" ifelse ] -- inspired by Python but "enhanced"
4  arith ::= arith op arith                       -- source of ambiguity
5          | unary
6  unary ::= { "-" } app
7  app ::= app atom | atom                        -- inspired by Haskell, f x y z
8  atom ::= var
9         | "(" expr ")"
10 op ::= "+" | "-" | "*" | "/"
```

The following points resolve ambiguities and omissions:

- `var` is `identifier` from ParserLib, noting that `if` and `else` are reserved words.
- Line 4 is a source of ambiguity and left recursion that you need to fix. You need to implement an unambiguous, terminating parser based on these operator precedence levels, from highest to lowest:

operator	associativity
`* /`	left
`+ -`	left

- Whitespaces around tokens are possible.

Further explanations and hints:

- Parsing does not check types. The parser accepts what we may think of as ill-typed inputs, as long as they are allowed by the syntax.
- Line 3 is inspired by Python if-else expressions. And as the rule suggests, after an "`else`" you can have yet another if-else expression too! An example is in the sample test cases.

  But wait, there's more! Although Python fails to recognize the beauty of it, since the test condition is protected by a pair of matching "`if`" and "`else`" acting as parentheses, there is no ambiguity in allowing any expression, even yet another if-else expression! An example is in the sample test cases.
- Line 6 is unary prefix negation. Note that zero or more minus signs are possible, e.g., "`- -x`". (Using `operator` from ParserLib, "`--x`" doesn't qualify. This is intended.)
- Line 7 is another source of left recursion. But it is just trying to say that function application is left-associative, with operands being `atom`.

Implement the parser as `expr` in Parser.hs. The abstract syntax tree to build is defined by `Expr` in Hajspy.hs. My tests will only test `expr` directly or via `run` in testParser.hs. You are free to organize your helper parsers.