CMPUT 274—Assignment 4 (Fall 2025)

R. Hackman

Due Date: Friday June 28th, 8:00PM

Per course policy you are allowed to engage in *reasonable* collaboration with your classmates. You must include in the comments of your assignment solutions a list of any students you collaborated with on that particular question.

For each assignment you may not use any Python features or functions not discussed in class! This means, for example, you may not use any built-in functions or methods we have not discussed. If you are in doubt, you may ask on the discussion forum — however remember if you are including your code, or even your plan to solve a question, you must make a private post. Use of disallowed features will result in a grade of zero on the question(s) in which they were used.

All functions you write must have a complete function specification as presented in the course notes. Failure to provide function specifications for functions will lead to the loss of marks.

Unless explicitly forbidden by a question, you are always allowed to write additional helper functions that may help you solve a problem.

`fimpl` - **F**un **Im**perative **P**rogramming **L**anguage

In this and the following assignment you will be developing your own interpreter for your new programming language: `fimpl`.

We will build the interpreter for our language step-by-step. For the first few questions of this assignment we will be working towards only supporting the arithmetic expressions of our language, not the entire language yet. In order to understand our language we present a *grammar* below in terms of symbols and production rules. Symbols defined by production rules defines the valid order in which strings can appear in our language. Strings appearing not in an order prescribed by our grammar are not part of our language (and thus a syntax error!). The production rules for a symbol will be in the form:

```
Symbol1 : A
        | B
        | C
```

Which can be read as "An A, or a B, or a C is a Symbol1". We can call these the alternatives that make up the symbol Symbol1. Symbols are often defined recursively in terms of themselves or other symbols. For example, if we wanted to define a symbol MyWord that matches all strings that end in `'x'` and have any number of `'y'` characters before the `'x'` that set of rules might look like this:

```
MyWord : 'x'
       | 'y'MyWord
```

This rule can be read as "a MyWord is the character `'x'`, or it is the character `'y'` followed by a MyWord." From this rule we can *produce* infinite valid strings that fit the rule (hence the name production rules). Initially the only information we can determine is that the string `'x'` is a valid MyWord, so that is our only possible valid MyWord. Then, however, we notice that the string `'y'` followed by a MyWord is a valid MyWord, so `'yx'` is a valid MyWord since it is a `'y'` followed by a MyWord (since `'x'` is a valid MyWord). Now that we have defined `'yx'` to be a vaid MyWord that means that `'yyx'` is also a valid MyWord, since it is the character `'y'` followed by a valid MyWord. This logic follows and through *inductive reasoning* any number of `'y'`s followed by a single `'x'` is a valid MyWord.

Note: Text in rules that is in quotes represents that string literal, text not in quotes represents a symbol, that can be replaced with any of that symbol's production rules.

Now, we provide the grammar for only the `fimpl` expressions, which are written in prefix notation using the tokens `'add'`, `'sub'`, `'mul'` and `'div'` for the addition, subtraction, multiplication, and integer division operators respectively.

```
Expr  : Atom
      | BinOp Expr Expr

BinOp : 'add'
      | 'sub'
      | 'mul'
      | 'div'

Atom  : Integer
      | Identifier
```

Instead of providing formal definitions of `Integer`, `Float`, and `Identifier` in our grammar (which to do quickly would require learning more grammar syntax) instead you will be provided an informal definition below.

- An integer is a sequence of characters that *may* begin with the character `-`, followed by one or more digits.
- An Identifier is a sequence of characters that begins with a lowercase letter followed by any number of alphabetic characters (uppercase or lowercase letters), and is not already a keyword in our language

It is worth noting that by the definition of our expression grammar above `fimpl` expressions are written in *Polish notation* also known as *prefix notation.* This is different from the notation we are used to for arithmetic expressions which is called *infix notation.* One of the benefits of prefix notation is that we do not need to rely on operator precedence rules or parentheses to tell us the order in which expressions should be evaluated, instead it is encoded directly in the order in which the expressions are written.

For example, consider the following expression written in our typical infix notation

$$5 \times (3 + 2 \times 9)$$

We know that in this expression the first value to compute is 2×9 because it is the expression with the highest precedence in the parentheses. Following our precedence rules we then evaluate this as follows:

$$\begin{aligned}5 \times (3 + 2 \times 9) &= 5 \times (3 + 18)\\ &= 5 \times (21)\\ &= 105\end{aligned}$$

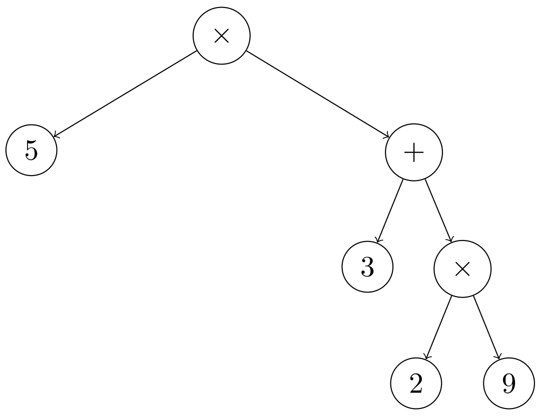
The equivalent prefix notation expression would be written as follows:

$$\times\ 5\ +\ 3\ \times\ 2\ 9$$

In this format the operands are the two expressions immediately following the operand, which themselves may be operators to be applied. Evaluation of this expression would then be performed by repeated applications of the operators which have final values to apply to. For example:

$$\begin{aligned}\times\ 5\ +\ 3\ \times\ 2\ 9\ &= \times\ 5\ +\ 3\ 18\\ &= \times\ 5\ 21\\ &= 105\end{aligned}$$

It is worth noting, and helpful to understanding the assignment to come that arithmetic expressions, which are themselves recursively defined, can be represented quite nicely with a tree. For example, below is a tree that represents the expression discussed above.



Given the definition of our grammar, and the understanding of prefix notation expressions, we can evaluate the sample `fimpl` expressions below:

```
sub 1 3                     ----> -2
add sub 2 3 div add 10 2 3 ----> 3
mul add -3 5 sub 1 -4      ----> 10
add 3 x                     ----> 3+x, we must know what x is to compute final value!
```

Part One

This assignment, which is part one of implementing our `fimpl` interpreter, is a series of functions to implement which will be used together to develop an interpreter for our grammar so far, which is only that of arithmetic expressions. Unlike previous assignments where each question was worth the same weight, questions in this assignment will be worth varying weight which is listed with each question, as several of the questions ask for functions which are quite easy for us to implement by now, but are necessary stepping stones to completing our larger goal.

1. **(10%)** Our `fimpl` programs, and currently our expressions, are ultimately just text. As such, we must first make steps to translate this text into a more usable form for processing. First we will write the function `strToTokenList` which takes a single string parameter and returns a `LList` of all the whitespace separated strings in our string. It is important to note that multiple adjacent whitespace may occur in a `fimpl` program and our function must take that into account.

   Hint: You may want to reuse the functions `strSplit`, and potentially `removeOccurrences` you wrote in A2 (however, `removeOccurrences` can easily be replaced with an application of `filter`).

   Consider the following examples of `strToTokenList` for reference.

   ```
   strToTokenList("mul add -3 5 sub 1 -4 ")
       -> LL("mul", "add", "-3", "5", "sub", "1", "-4")
   strToTokenList("add      -2\n\n   \t4")
       -> LL("add", "-2", "4")
   ```

2. **(10%)** Once we have our `LList` of tokens part of processing it is determining what type of token we have. Write the function `isIdentifier` which takes a single string parameter and returns `True` if that string is a valid `fimpl` identifier, and `False` otherwise. Reminder, the rules for a valid `fimpl` identifier are:

   - Is a string of only alphabetic characters
   - The first character must be a lowercase letter
   - It is not already a keyword in our language

   For this first part the only keywords to consider are those which represent our operators which are `"add"`, `"sub"`, `"mul"`, and `"div"`. However, do keep in mind you will be updating this function with more keywords as they are added to our grammar in future parts so you want to design the function in a way that is easy to update with more keywords to check against.

   Consider the following examples of `isIdentifier` for reference.

   ```
   isIdentifier("foo") -> True
   isIdentifier("Foo") -> False
   isIdentifier("add") -> False
   isIdentifier("xDIM") -> True
   ```

3. **(45%)** We are now prepared to begin the processing of a `fimpl` expression. Our first step will be to take a `LList` of tokens, such as one produced by `strToTokenList`, and produce an *expression tree* which is a tree that represents an arithmetic expression. For this we will define a data type `ExprTree` as follows:

```
An ExprTree is one of:
 - LL("ident", str)
 - LL("intLit", int)
 - LL("binOp", Operator, ExprTree, ExprTree)

An Operator is one of:
 - "add"
 - "mul"
 - "div"
 - "sub"
```

By this definition every `ExprTree` is a `LList` in one of three specific forms, and the first item of that `LList` tells us what type of expression we have

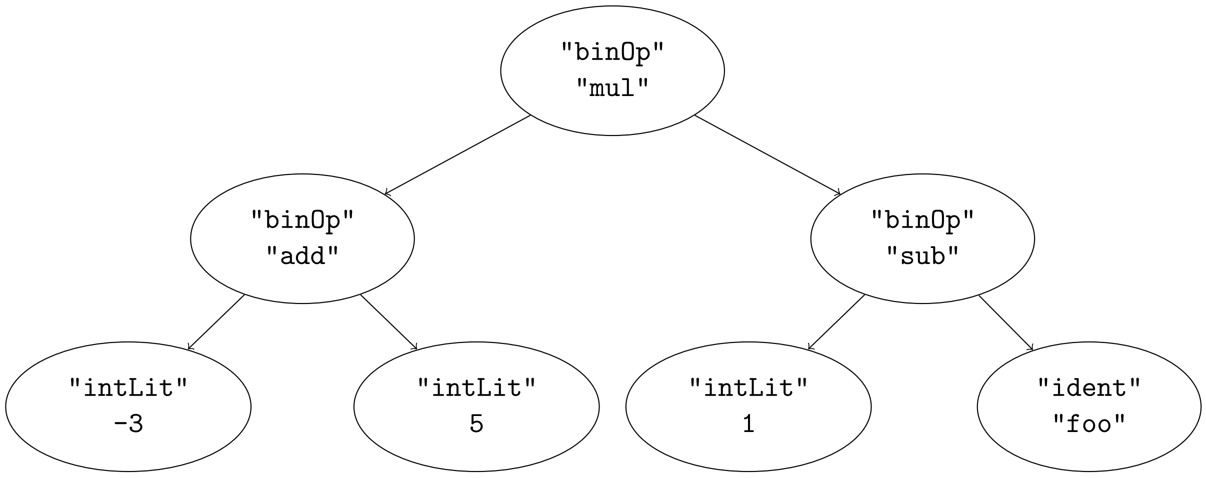
- For an identifier expression the first item in the `LList` is the string `"ident"` while the second item in the `LList` is the string that represents that identifier
- For an integer literal expression the first item in the `LList` is the string `"intLit"` while the second item in the `LList` is the actual integer that literal represents.
- For a binary operator expression the first item in the `LList` is the string literal `"binOp"`, the second item is a string which is one of our `fimpl` operators. The third and fourth items are themselves `ExprTrees` which represent the first and second operands of our operator respectively.

For example consider the following `LList` of tokens and the equivalent `ExprTree`:

```
LL("add", "3", "x")
 -> ("binOp", "add", ("intLit", 3), ("ident", "x"))

LL("mul", "add", "-3", "5", "sub", "1", "foo")
 -> ("binOp", "mul",
       ("binOp", "add",
           ("intLit", -3)
           ("intLit", 5)
       ),
       ("binOp", "sub",
           ("intLit", 1),
           ("ident", "foo")
       )
    )
```

To aid with understanding, a visual representation of the second ExprTree is provided.



You must write the function `tokensToExprTree` which takes a `LList` of tokens (i.e. one produced by `strToTokenList`), and returns a `LList` of two items, the first is the `ExprTree` that represents the first expression appearing in the `LList` of tokens and the second item is the updated `LList` of tokens in which all items that were used to make this `ExprTree` have been removed.

The function specification is given below:

```
def tokensToExprTree(tokens):
  '''
  tokensToExprTree parses the first complete expression in the LList of tokens
                   into an ExprTree and returns a pair of the form
                   LL(ExprTree, LList) where the first item is the ExprTree
                   that represents the given expression, and the second item
                   is the parameter tokens but without any of the tokens that
                   were used to construct this expression

  tokens  - A LList of str
  returns - A LL(ExprTree, LList of Str)

  Examples:
    tokensToExprTree(LL("add", "-2", "4"))
      -> (('binOp',
            'add',
            ('intLit', -2),
            ('intLit', 4)),
          ())
    tokensToExprTree(LL("x", "5", "add", "3", "y"))
      -> (("ident", "x"),
           ("5", "add", "3", "y"))
    tokensToExprTree(LL("add", "x", "add", "5", "3", "mul", "10", "2"))
      -> (('binOp',
            'add',
             ('ident', 'x'),
             ('binOp', 'add',
               ('intLit', 5),
               ('intLit', 3)
             )
           )
           ("mul", "10", "2")
          )
  '''
```

Note how in the second and third examples the second item of the returned pair, which is meant to be the `LList` of tokens without the processed ones, is non-empty. This is the expected behaviour as the function `tokensToExprTree` is only meant to remove as many tokens as is necessary to form a complete `ExprTree` from the first tokens seen. How many tokens need to be processed depends on what tokens are seen. The fact that `tokensToExprTree` also returns the "new" `LList` of tokens is very important in order to make the function itself easy to implement.

In order to aid you in implementing your function a simple outline of what you must do is provided below.

- Is the first token an identifier?
  - Your returned `ExprTree` then should be an identifier node
  - Your returned tokens `LList` then should have that identifier removed
- Is the first token an integer? (**Hint:** Your `isIntegerStr` function from A2 will be useful!)
  - Your return `ExprTree` then should be an integer literal node. Remember that the second value in an integer literal node the second value should be the actual integer, not a string. As such your `strToInt` function from A2 will be useful!
  - Your returned tokens `LList` then should have that integer string removed
- Is the first token a binary operator?
  - The first item of the tokens `LList` is then the operator in question
  - The expression that immediately follows that operator is its left-hand operand, that means you need to construct an `ExprTree` from the tokens that follow this operand in the tokens list.
  - The expression that immediately follows the end of the expression that is our left-hand operand is our right-hand operand. That means you need to construct an `ExprTree` from the tokens that come *after* the tokens that were consumed while producing your left-hand operand.
  - **Hint:** You need to construct an `ExprTree` from a `LList` of tokens as well as produce a new `LList` of tokens that is everything after the expression you processed — do you have a function that can do that?
  - Your returned tokens `LList` should have binary operator token we found removed, as well as all of the tokens that were used in constructing our left-hand operand and our right-hand operand.

4. **(10%)** We are almost ready to evaluate a `fimpl` expression. First we will write another helper function, this function will be called `lookup`. This function takes two parameters, the first of which is a string and the second of which is a `LList` of pairs of the form `LList(str, int)`. The first item of each pair represents an identifier and the second item in each pair is the value that identifier corresponds to.

   Hint: this function is the same as `lookup` that you wrote in A2, except instead of having two parallel lists for names and values you now have one list of pairs, where each pair contains a name and the corresponding value.

   For this question write the function `lookup` which takes a string and a `LList of LL(str, int)` and returns the integer that corresponds to the given string parameter.

5. **(25%)** We are now ready to actually evaluate our `fimpl` expressions. Your job is to write the function `evalExpr` which has the following specification:

```
def evalExpr(eTree, defns):
  '''
  evalExpr returns the result of evaluating the given ExprTree eTree given
           the provided identifier definitions in the LList of pairs defns

  eTree   - an ExprTree
  defns   - a LList of LL(str, int)
  returns - int

  Examples:
    evalExpr(first(tokensToExprTree(strToTokenList("mul add -3 5 sub 1 -4 "))),
            LL()) -> 10

    evalExpr(first(tokensToExprTree(strToTokenList("add x 3"))),
            LL(LL("x", 2))) -> 5

    evalExpr(first(tokensToExprTree(strToTokenList("add x 3"))),
            LL(LL("x", -25))) -> -22
  '''
```

The majority of the work necessary to make `evalExpr` work has already been completed for us through the encoded of our expression into a tree. The steps to implement `evalExpr` are roughly listed below.

- Check if `eTree` is an integer literal node
  - If it is then simply return the integer it corresponds to.
- Check if `eTree` is an identifier node
  - If it is then lookup the value that is associated with that identifier in the `defns` parameter and return the result
- Check if `eTree` is a binary operator node
  - If it is then return the result of performing the operator specified by that node on the result of the left-hand operand and the result of the right-hand operand

Deliverables: All the functions specified in this assignment should be contained within one single Python file named `interp.py` which should be submitted to Canvas.