Homework – Intermediate Representation

Goal:

We wish to continue implementing the intent and purpose of the BNF by reading the intermediate representation stage.

The goal is to store multiple intermediate representations and then later reuse them.

Purposes:

- Practice implementing another's work described in text (often much more difficult than it would seem at first glance).
- Gain experience automating parts of a compiler.
- Continue building upon pre-existing code
- Prepare for later translation to end point language.
- Implement assignment operator.
- Implement a symbol table.
- Implement procedures which require storing multiple in

Background:

Ambiguous parsing and operator precedence required encoding this information into productions. For example, the productions when parsed ensures that 2 + 3 x 4 performs 3 x 4 first. However, utilizing these productions and its associated LL(1) table into generating an intermediate representation is not straightforward. Ultimately, a clean intermediate presentation (IR) is desired to help prepare for eventual output.

Assignment:

The assignment comes in multiple parts. Start with the file irassignment.txt. The first several dozen expressions in that file are of this form:

```
num var1 = 3*4
num var2 = 12 / 6
num var3 = 12/6
num var4 = 5 + 4 * 3
num var5 = 12 + 34 * 45
num var6 = (4 / 5)
num var7 = 2 * 2 + 5 * 5
```

For each, your goal is to generate an internal IR, store the result related to the variable, then display the answer.

To start, these expressions need additional productions. Implementing these new expressions is far from straightforward. For example, a variable name can be found left of an assignment, but a number cannot be found left of an assignment. A variable name and a number can be found right of an assignment. This significantly complicates the logic. Updated productions are as follows on the next page. NonTerminals will start with an uppercase letter. Terminals will start with either a lowercase letter or will be a symbol. Note that the nonterminal `procedure` must match exactly the word procedure.

```
Goal                     LineFull
LineFull                 VarType  VarTypeAfter
LineFull                 LineVarName
LineFull                 negnum  Power'  MultDiv'  AddSub'
LineFull                 Parens  Power'  MultDiv'  AddSub'
LineFull                 result  GTerm
LineFull                 }
VarTypeAfter             LineVarName
VarTypeAfter             procedure  name  ProcedureParams  {
LineVarName              name  LineVarNameRemaining
LineVarNameRemaining     =  Expr
LineVarNameRemaining     PowerAndRightOp  MultDiv'  AddSub'
LineVarNameRemaining     MultAndRightOp  AddSub'
LineVarNameRemaining     DivAndRightOp  AddSub'
LineVarNameRemaining     AddSub'
ProcedureParams          (  Params  )
Params                   VarType  name  MoreParams
Params                   ϵ
MoreParams               ,  VarType  name  MoreParams
MoreParams               ϵ
VarType                  num
VarType                  ish
Expr                     LTermAddSub  AddSub'
LTermAddSub              LTermMultDiv  MultDiv'
LTermMultDiv             LTermPower  Power'
RTermAddSub              RTermMultDiv  MultDiv'
RTermMultDiv             RTermPower  Power'
AddSub'                  +  RTermAddSub  AddSub'
AddSub'                  -  RTermAddSub  AddSub'
AddSub'                  ϵ
MultDiv'                 MultAndRightOp
MultDiv'                 DivAndRightOp
MultDiv'                 ϵ
MultAndRightOp           *  RTermMultDiv  MultDiv'
DivAndRightOp            /  RTermMultDiv  MultDiv'
Power'                   PowerAndRightOp
Power'                   ϵ
PowerAndRightOp          ^  RTermPower  Power'
LTermPower               GTerm
LTermPower               negnum_value
LTermPower               negish_value
LTermPower               negname
RTermPower               GTerm
GTerm                    Parens
GTerm                    PosVal
GTerm                    SpaceNegVal
Parens                   (  Expr  )
PosVal                   num_value
PosVal                   ish_value
PosVal                   name
SpaceNegVal              spacenegnum_value
SpaceNegVal              spacenegish_value
SpaceNegVal              spacenegname
// Coming soon, productions for calling functions
```

Next, store the variables in a symbol table. The symbol table should have the following methods:

- Lookup(name)
- Insert(name, record)

The record contains all information relevant to the variable you deem necessary. For this assignment, the record should contain the variable's type. For this assignment, it may be helpful to also assign it a reference to its IR.

Creating your own IR

Multiple avenues exist for creating your own IR. You are welcome to experiment and implement your own approach. Be warned that many approaches seem straightforward at first, but edge cases can ruin the implementation (I've spent more time than I want to admit down rabbit holes in this area). Below are three approaches you can use.

1) **Store production parse tree:**
   This approach has you creating the tree as you parse productions. You cannot start a traversal until the entire expression is parsed and the tree is built. Additionally, your code must first identify within productions operator terminals and right-operand non-terminals placeholders. This information must be encoded somehow in the resulting tree. Once the tree is fully built, you can traverse it with a modified post-order traversal. Normally post-order traversal is: recursively go left, recursively go right, perform action. But these are not binary trees and don't follow the same process. The modified traversal is: Start by 1) recursively going down in order all branches to the left of any operator branch (if any exist), then 2) process a right operand production branch (if one exists), then 3) process an operator branch (if one exists), and finally 4) process all remaining branches (if any exist). Every time a terminal is encountered, put it on a stack. This stack can be your IR, or you can use the stack to build a simple binary tree.

2) **Create a simpler binary tree as the LL(1) parser proceeds:**
   This process is not as pretty as the other two, but does allow for a quick jump to a binary tree containing only terminals of interest and removing unwanted non-terminals along the way. Your code must first identify within productions operator terminals and right-operand non-terminals placeholders. For the remainder of this section, a right-operand non-terminals will be called a RONT. As an example, suppose a production is:

   ```
   AddSub'                   +  RTermAddSub  AddSub'
   ```

   Then the `+` is a right-operand. The `RTermAddSub` is a right-operand non-terminal (RONT).

   Your code must also have a second stack to hold integer index values. From here, you need to proceed with a few rules, add the following logic when items are added or removed from the stack in the LL(1) processing code.

   - If a non-operator terminal (such as a number or a variable name) would be consumed off that stack, one of two options will occur:
     - If this is the first node of the tree, create a node and put that token in it. Make this node the current focus node.
     - If the current focus node contains a non-terminal placeholder value, overwrite its value with the current token. Keep the current focus on this node.

- If an operator would be consumed off the stack, one of two options will occur:
    - If the current focus node is the root node, then create a parent node relative to the current focus node, then place this operator token in the node. The parent's left link will point to the node it came from. Make this operator node the current focus node.
    - Otherwise, create a new node containing an operator value. This node will take the position in the tree that the current focus node did, and current focus node will become the left child of this new node. Make this node containing an operator value the current focus node.
- If a RONT is consumed by expanding it into its productions, then create a node as a new right child. That node will contain as data the RONT as a placeholder. Make this node the current focus node.

  Next, this code must remember when this RONT's full usage through the parsing is done, or in other words, when the right operand is done. The best way to do this is that just prior to consuming the RONT on the stack, record the stack's size and push that value in a second stack which holds integer values. That integer value is utilized in the next bullet point.
- When an item is consumed from the stack, if that item is not an operator terminal or a RONT, and this item is not putting values back on the stack, then check this stack's size and compare it with the integer stack's top. If the two values match, then this represents the right operand branch finishing. Pop the value off the integer stack. Send the current node focus up to the parent.

3) **Shunting Yard Algorithm -**
   This approach is straightforward, but has a problem described in the next paragraph. Your code must first identify all possible operator terminals and assign them a priority precedence number. From here, use an operator stack and an output queue. As tokens are processed from left to right of your expression, check it's a value or variable, if so, add it to the back of the queue. If the token is an operator, then one-by-one remove all operators of lower precedence in the stack and move each to the back of the queue, then place this operator on the stack. When the expression is parsed, move all remaining operators from the stack onto the back of the queue, one-by-one. The queue then has the post-order traversal.

   A problem with this approach is that it's duplicating work. The supplied productions had operator precedence built in. By using the Shunting Yard Algorithm, your code only uses the LL(1) parser to verify an expression is syntactically valid but ignores all its operator precedence logic, and then use the Shunting Yard Algorithm to create expressions based on operator precedence.

   A helpful characteristic of using the Shunting Yard Algorithm is that it allows for both fewer productions and more flexibility in designing productions. In practice, I have found that creating productions with operator precedence built-in to be surprisingly challenging, while creating productions for the Shunting Yard approach to be forgiving.

Supporting types:

The BNF didn't clearly specify that floating points are supported. Your next goal is to support the following:

```
ish var17 = 3.5
ish var18 = (3.5)
ish var19 = ((3.55))
ish var20 = 3.14159+2.718218
```

The `ish` is a 32-bit float data type. The "ish" means it's approximate, as floats (IEEE 754 numbers) often are.

Your symbol table should associate that these variables are not integers

Finding issues:

Verify that symbols cannot be reused:

```
num a = 5
num b = 3 + 4
num a = 16
```

Verify that invalid mathematical expressions are disallowed (find this in the IR, not in the productions)

```
num error = 5 / 0
```

Allow for variable reuse:

Examples:

```
num var1 = 3*4
num var2 = 12 / 6
num var3 = 12/6
num var4 = 5 + 4 * 3
num var5 = 12 + 34 * 45
num var49 = var1 + var2
num var50 = var1 - var2
num var51 = var2 - var1
num var52 = var1 * var1
num var53 = (var3 + var4) * (var5 + var3)
```

Copying values:

Values should copy

```
num orig = 4 + 2
num copy = orig
```

Procedures:

This last section represents a large addition to your homework. This section requires that you add more productions to support procedures, store and recall IRs, use parameter variables, support return values, and support levels of scope. Please reach out if you struggle here.

Note that you are validating lines of code, not entire procedures. Your parser must validate:
```
num procedure add(num a, num b) {
```
as a valid line of code.

The mechanism to support this is by treading each as its own labeled block. These labeled blocks will make a good analogy to assembly labels when that time comes. So for example, upon seeing an add procedure, your IR must know that the add block as started. It then continues storing IR, until the }, at which point it knows the add block has ended and another block will be starting.

You must support the following code. Your implementation to support methods at this time can be as clean or as hacky as you please. Properly handling return values not covered in this assignment:

```
num procedure add(num a, num b) {
  num result = a + b
  return result
}

num procedure mult(num a, num b) {
  num result = a * b
  return result
}

num a = 2
num b = 3
```

Extra Credit:

Parse and validate these lines of code. Further, prove you can execute the IR and compute the values for result1 through result8.

```
num result1 = add(a, b)
num result2 = mult(a, b)
num result3 = mult(a,a)
num result4 = mult(b,b)
num result5 = add(-a, -b)
num result6 = add(a, b) - mult(a, b)
num result7 = add(a,a) + mult(a,b) * mult(b,b)
//num result8 = add((add(a,b) + mult(b,a)) + 7
```

Verification:

Display the results of all main variables computed. Display errors if the line cannot be computed. Record a short video run through of your code.