Homework – LL(1) Parser to Intermediate Representation

Goal:

Implement a functioning Intermediate Representation (IR) and demonstrate optimization.

Purposes:

- Move the compiler further along, in preparation for final output.
- Understand the relationship between parse trees and IR trees.
- Supplying meta data to nodes
- Begin optimization of IR.

Instructions:

The LL(1) parser enables us to easily accept or reject lines of code as valid. However, we have not yet done anything with the valid lines of code. This assignment translated that logic into an IR.

Use your prior LL(1) parser engine. As you accept productions, build an appropriate IR using a strategy listed in "Creating an IR" below. For all valid lines of code, display 3 items: 1) "Valid": [the line of code], 2) the post-order notation version of the code as it should be executed, and 3) the optimized version where all possible parts of arithmetic are computed. For all invalid lines of code, display simply "Invalid".

Examples:

```
Valid: 2 * 2 + 5 * 5 ..... 2 2 * 5 5 * + ..... 29
Valid: 4 * 3 / 2 + 1 ..... 4 3 * 2 / 1 + ..... 7
Valid: 4 * 3 / 2 + one ..... 4 3 * 2 / one + ..... 6 one +
Valid: four * three / two + one ..... four three * two / one + ..... four three * two / one +
Valid: (((2+3)*4)+(7+(8/2))) ..... 2 3 + 4 * 7 8 2 / + + ..... 31
Valid: (((2+3)*4)+(7+(var1/2))) ..... 2 3 + 4 * 7 var1 2 / + + ..... 20 7 var1 2 / + +
Valid: 5 / 0 ..... 5 / 0 ..... [error]
Invalid: (a+b)+(c*d))
```

Creating an 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) Strategic production tree traversal:

This approach requires you create the parse 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 (RONT). As an example, suppose a production is:

```
AddSub'   ->   +  RTermAddSub  AddSub'
```

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

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 here, follow a modified process:

Upon entering a node, determine an order of branch traversals. 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) from left-to-right.

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 in which you can proceed to your post-order traversal and optimizations.

2) Create a simpler binary tree during the LL(1) parser process:

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 (RONTs).

This implementation placed within the existing LL(1) parser algorithm. An additional stack holding integers is required. In the LL(1) parser, add the following logic when items are added or removed from the stack:

- If a non-operator terminal (such as a number or a variable name) would be consumed off the LL(1) stack, one of two options will occur:
    - If this is the first node of the output tree, create a node and put that token in it. Make this node the current focus node.
    - If the current focus node contains a RONT 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 the 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. This node's data will act as a placeholder to be overwritten later. 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 accomplish this is just prior to consuming the RONT on the stack, record the LL(1) parser 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 LL(1) stack, then check the LL(1) stack's size and compare it with the integer stack's top. If the two values match, then this represents that a prior right operand branch has finished and a new branch has started. 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 (e.g., * can be priority 1, + can be priority 2, and * is "higher" precedence). From here, use an operator stack and an output queue. As tokens are processed from left to right of your expression, check if it's a value or variable, and if so, add it to the back of the queue. If the token is an operator, then one-by-one remove all operators of higher or equal 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.

Optimizations:

Detect if an operator has two numerical operands. If so, have your compiler perform that computation and simply the IR. If the entire IR can be simplified into a single number, do so. If only part of the IR can be simplified, then simplify only that part.

Detect if the operator is a / and the right operand is a 0. If so, this is a divide-by-zero error, and have the compiler state it's an error.

Verification:

- Verify that you can parse the all the inputs provided in the .txt file according to the instructions.
- Ensure your code displays all required outputs.

Please reach out to me if you have any questions!