CSE 4714/6714 – Programming Languages
Part 3 – Building a Parse Tree

The next step in creating an interpreter for TIPS is to create a parse tree of the input while checking the syntax of the input.

Example Code

Examine the files in `Part_3_Example.zip`. Note that each production function (see `parser.h`) now returns a pointer to the node it produces. The `term` function now returns a `TermNode*`. The `factor` function now returns a `FactorNode*`. These return values are used to construct the parse tree. The entire process is started in `driver.cpp` by creating a root pointer and then calling the first production method:

```
// Create the root of the parse tree
ExprNode* root = nullptr;

...

// Process <expr> production
root = expr();
```

See `parse_tree_nodes.h` in the example subdirectory for the classes that represent the nodes in the parse tree of the arithmetic grammar. In the arithmetic expression grammar, an expression is made up of one or more terms:

$$\text{<expr>} \rightarrow \text{<term>} \text{\{(+ | -) <term> \}}$$

The `ExprNode` class contains data members for the first term, the operators that connect remaining terms, and the remaining terms.

```
class ExprNode {
public:
    TermNode* firstTerm = nullptr;
    vector<int> restTermOps;  // TOK_ADD_OP or TOK_SUB_OP
    vector<TermNode*> restTerms;

    ~ExprNode();
};
```

See `parse_tree_nodes.h` for examples of overloaded operator `<<` for the node classes. These functions allow a node to be printed out using the standard C++ `<<` operator.
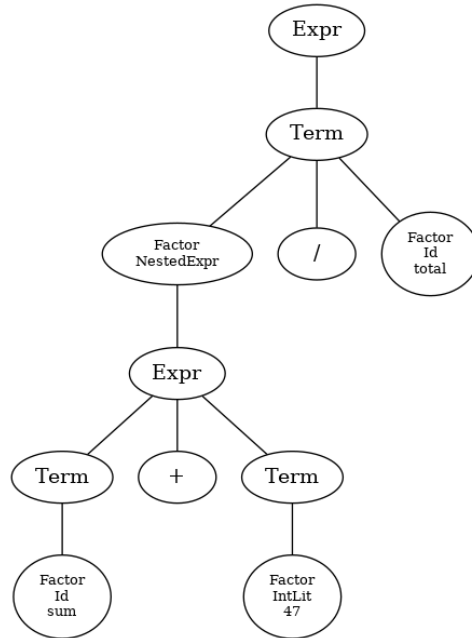
```
cout << anExprNode << endl;
```

After a successful parse of an arithmetic expression, the expression can be output by calling `<<` with the root. The parser can print the entire tree with the following line of code.
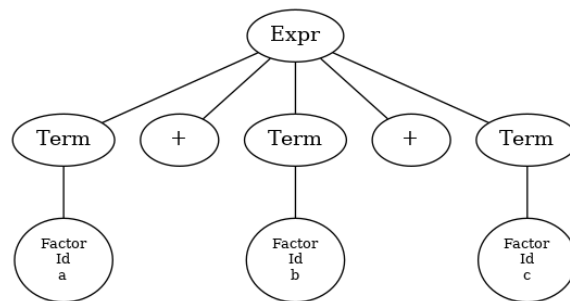
```
cout << *root << endl;
```

Before the example parser program ends, it deletes the root pointer. The destructor for the `ExprNode` in turn calls the destructors for the nodes in the parse tree. The `delete` methods for each node of the parse tree should use `cout` statements to show the order of the deletions.

It is helpful to be able to construct parse trees by hand in order to visualize what the parse tree should be. The following are sample expressions and their parse trees.
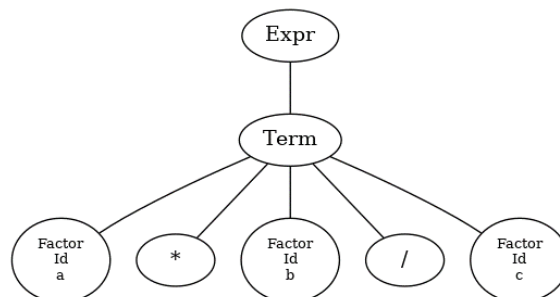
Expression: `(sum + 47) / total`

Expr — Term — (Factor NestedExpr) (/) (Factor Id total)
Factor NestedExpr — Expr — Term (+) Term
Term — Factor Id sum
Term — Factor IntLit 47

Expression: `a + b + c`

Expr — Term (+) Term (+) Term
Term — Factor Id a
Term — Factor Id b
Term — Factor Id c

Expression: `a * b / c`

Expr — Term — Factor Id a (*) Factor Id b (/) Factor Id c

Run these samples with the provided parser to see the expected behavior for the parser.

2

TIPS Parser and Parse Tree

Copy your code for Part 2 to a new directory. Modify the `makefile` so that compiling the driver depends on `parse_tree_nodes.h`. You can do this by adding the bold text below to line 34 of your `makefile`:

```
driver.o: driver.cpp productions.h parse_tree_nodes.h lexer.h
```

The next step in building a parse tree for TIPS programs is to create classes that represent the nodes (program, assignment, factor, read, write, etc.) in a parse tree. A set of classes to start with are in the `parse_tree_nodes.h` file. You will need to add to / modify this file.

Modify your code in `productions.h` so that while your parser is parsing the input, the code creates the nodes of the parse tree in addition to printing out the statements in the program.

Check your parse tree by printing out the tree after the TIPS program is successfully parsed. Overload the << operator for each node class so that nodes can be printed:

```
cout << anExprNode << endl;
```

See the Hints section below for links to resources on overloading the << operator.

If you defined the root of your parse tree as

```
ProgramNode* root;
```

you should be able to print the entire tree with the following line of code.

```
cout << *root << endl;
```

See the sample outputs to see the expected format of the printed parse tree.

Before the parser program ends, delete the root pointer. The delete methods for each node of the parse tree should use `cout` statements to show the order of the deletions. See the sample outputs for the expected messages.

NOTE: The behavior of your parser when it finds a syntax error should not change. The parser should print the line number, the error message, and exit.

The deliverable is a zip file of the source code files needed to build and execute your parser (`rules.l`, `productions.h`, `driver.cpp`, etc.). Do NOT include any of your own test cases (`.pas` or `.correct` files). Create a zip file named *netid*`_part_3.zip` and upload that file to the assignment. Do not include any files generated by the `makefile` in your submission. For example, your submission should not include any `.o` or executable files.

Hints

You can read more about overloading the << operator at
    https://www.tutorialspoint.com/cplusplus/input_output_operators_overloading.htm

In Part 4 you will implement an interpreter for your parse tree that executes the TIPS program.