CSE 4713/6713 – 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 `productions.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:

```
<expr> → <term> {( + | - ) <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.cpp 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;
```

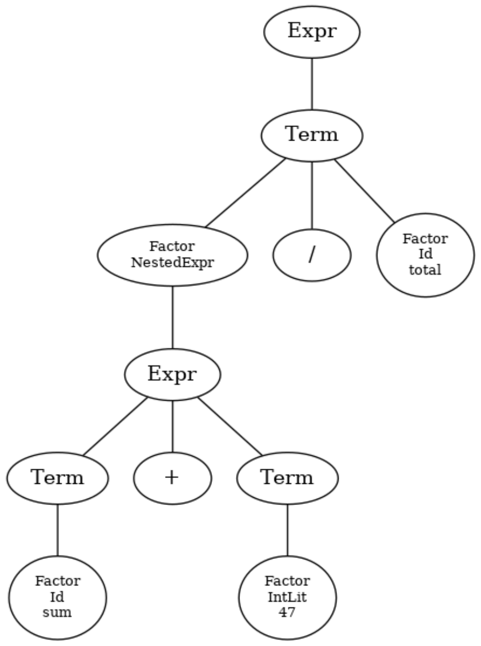
After a successful parse of an arithmetic expression, the expression can be output by calling `<<` with the root. The parser can print the entire tree with the following line of code.

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

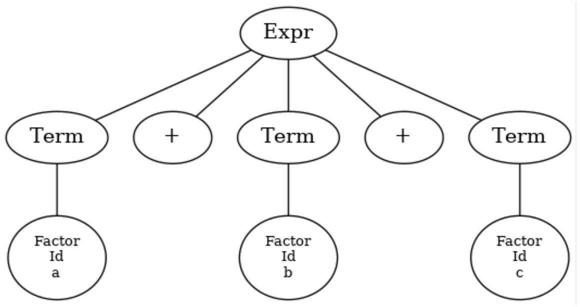
Before the example parser program ends, it deletes the root pointer. The destructor for the `ExprNode` in turn calls the destructors for the nodes in the parse tree. The `delete` methods for each node of the parse tree should use `cout` statements to show the order of the deletions.

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.

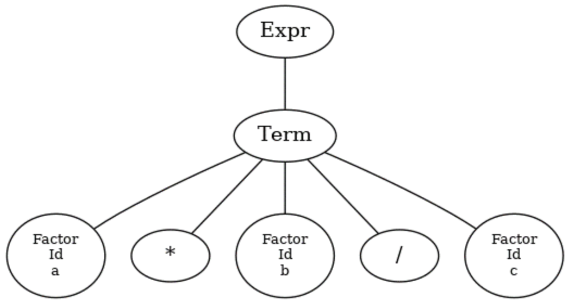
Expression: `(sum + 47) / total`



Expression: `a + b + c`



Expression: `a * b / c`



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

TIPS Parser and Parse Tree

The first step to 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` and `parse_tree_nodes.cpp` files in the `Part_3_Starting_Point.zip` file. You will need to add to / modify these files.

Use your recursive descent parser from Part 2 to parse a TIPS program. While your parser is parsing the input, create the nodes of the parse tree instead of just 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.). 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 `.exe` 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.

Grading

This is an individual assignment. Do not show / share your code with anyone else. You are responsible for debugging your own code. You are also responsible for testing your program so that the parser will function as expected with ***any*** input.

Correct output for test files:	80 points
Correct submission of deliverable:	10 points
new and delete operations appear correct:	10 points

Submissions that are late will be penalized 10 points for each day past the due date/time.

Any code that does not compile / run in the Windows Subsystem for Linux will receive a grade of 0.

You will not have access to the test files before submitting your program. I recommend that you create both valid and invalid TIPS programs to test your parser before submitting your code.