CSE 4714/6714 – Programming Languages
Project 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 (`parser.[h,cpp]`) 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. This tree is composed of nodes that are linked together to form the tree, just like the trees that we have studied in class. The node types are determined by the productions in the example grammar (`grammar.pdf`):

```
<expr>
<term>
<factor>
<id>
<int_lit>
<nested_exp>
```

In Lab 3, we learned how to create a heterogenous list (`vector`) of nodes by creating an abstract base class, and then creating the node classes by inheriting this base class. The example parser uses this method, because the productions `<id>`, `<int_lit>`, and `<nested_exp>` are all types of `<factor>` (indicated above by coloring them red). Note how each of these productions inherit the base class `FactorNode`. This allows each `TermNode` to store a list of `<FactorNode*>`s, where each `FactorNode` is either an `IdNode`, `IntLitNode`, or `NestedExprNode`:

```
class TermNode {
. . .
  vector<FactorNode*> restFactors;
. . .
};
```

Note that each of these node types has a `printTo` method; the base class uses double dispatch to allow each node type to print using the << operator.

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;

...

// Start symbol is <expr>
root = expr();
```

See `parse_tree_nodes.[h,cpp]` 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:
    int _level = 0;           // recursion level of this node
    TermNode* firstTerm = nullptr;
    vector<int> restTermOps;  // TOK_ADD_OP or TOK_SUB_OP
    vector<TermNode*> restTerms;

    ExprNode(int level);
    ~ExprNode();
};
```

See `parse_tree_nodes.[h,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 entire tree can be printed with the following line of code (in `driver.cpp`):

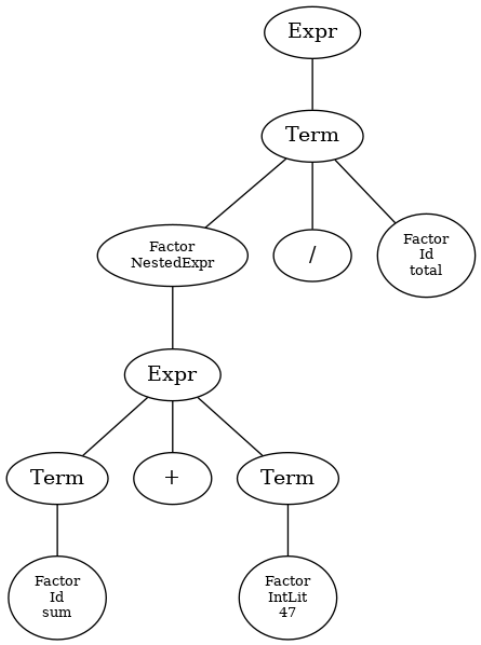
```
cout << *root << endl;
```

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 to visualize what the parse tree should be. The following are sample expressions and their parse trees, both what the parser prints out, and visualized as a tree:

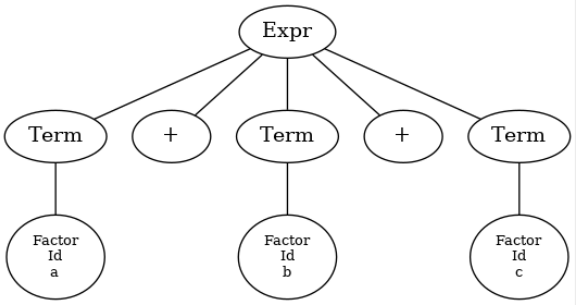
Expression: `echo "(sum + 47) / total" | parse.exe`

```
(expr
| (term
| | (factor
| | | (expr
| | | | (term
| | | | | (factor (IDENT: sum)
| | | | | factor)
| | | | term)
| | | +
| | | | (term
| | | | | (factor (INTLIT: 47)
| | | | | factor)
| | | | term)
| | | expr)
| | factor)
| /
| | (factor (IDENT: total)
| | factor)
| term)
expr)
```



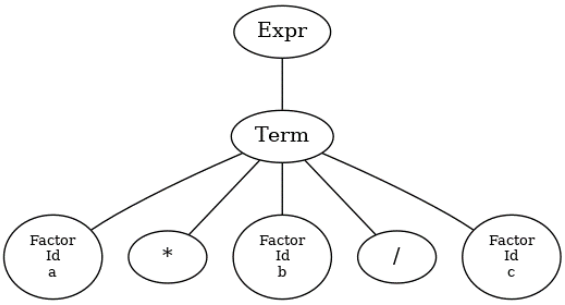
Expression: `echo "a + b + c" | parse.exe`

```
(expr
| (term
| | (factor (IDENT: a)
| | factor)
| term)
+
| (term
| | (factor (IDENT: b)
| | factor)
| term)
+
| (term
| | (factor (IDENT: c)
| | factor)
| term)
expr)
```



Expression: `echo "a * b / c" | parse.exe`

```
(expr
| (term
| | (factor (IDENT: a)
| | factor)
| *
| | (factor (IDENT: b)
| | factor)
| /
| | (factor (IDENT: c)
| | factor)
| term)
expr)
```



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

TIPS Parser and Parse Tree

Part 3 builds on Part 2, so the first step is to make sure that Part 2 runs correctly. Copy your code for Part 2 to a new directory.

Copy the code for the example parse tree nodes (`parse_tree_nodes.[h,cpp]`) into the new directory. Modify the `makefile` so that compiling the parser depends on the source code for the driver (`driver.cpp`), the parser (`parser.[h,cpp]`), and the parse tree nodes. The `makefile` given in the example code can serve as a model.

The next step in building a parse tree for TIPS programs is to create classes that represent the nodes in the parse tree. The classes in the `parse_tree_nodes.[h,cpp]` in the example code are a starting point.

You will need a node type, implemented as a C++ class, for every kind of production parsed in Part 2. Below are the production names that your professor has used; your names may of course differ:

```
<program>
<block>
<statement>
<assignment_stmt>
<compound_stmt>
<if_stmt>
<while_stmt>
<read_stmt>
<write_stmt>
<expression>
<simple_exp>
<term>
<factor>
<int_lit>
<float_lit>
<ident>
<nested_exp>
<not>
<minus>
```

As discussed above, `<term>` nodes contain a list of `<factor>`s, so `<factor>` is an abstract base class, inherited by the different kinds of factors (colored red). In a similar way, a `<compound_stmt>` contains a list of `<statements>`s, and therefore `<statement>` is also an abstract base class, inherited by the different kinds of statements (colored blue). The other productions (colored black) don't have to inherit anything.

Your Professor's Approach

Your professor worked this lab using the same hybrid bottom-up and top-down approach I recommended for Part 2. Of course, this is not the only possible approach, but I can recommend it.

I started with the simplest factor: `INTLIT`. I modified `driver.cpp` so the parser built up a factor first:

```
root = factor();
```

Then, I tested the code using commands like this:

```
$ echo "13" | parse.exe
```

I then moved on to `<float_lit>` and `<ident>`. `<not>` and `<minus>` are the first `<factor>`s that contain other `<factors>`s. I skipped `<nested_exp>` until later.

Then, I implemented `<term>`, which, as in the example code, must keep a list of `<factor>`s and factor operators (`*`, `/`, `AND`). I continued bottom-up with `<simple_exp>` and then `<expression>`. Last, I implemented

<nested_exp>. I carefully unit tested everything before moving to the next production. When this was finished, all possible TIPS expressions could be parsed and stored in parse trees.

Once these were thoroughly tested, I worked downwards, starting with <program>. I made a copy of the 1-hello.pas file, which I called 1-test.pas, and modified this as necessary to test only what I was doing. I used the command-line switch -p (see example driver.cpp) to visualize the parsing, and -d to ensure that deletions were correct. I worked downwards from <program> to <block> to <statement> (which must be an abstract class, like <factor>), then to <compound_stmt>, then to <write_stmt>. I then added the additional statements, one at a time. When working on one production, I first stubbed out sub-productions, just looking for a successful print statement before any other functionality. Again, it is very important to work systematically, unit testing everything carefully before proceeding with anything else.

Additional Requirements

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;
```

You can read more about overloading the << operator at

https://www.tutorialspoint.com/cplusplus/input_output_operators_overloading.htm

However, the example code generally shows what is needed for this assignment.

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. When the global variable printDelete is true, 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.

Additional Hints

I've seen cases where the Windows Subsystem only operates in a small window that only shows about 30 lines of code at a time. Working Part 3 (or Part 2) through such a tiny keyhole will make everything *much* harder. It may be that only some virtualization systems provide a reasonably sized window. If this affects you, I recommend installing Cygwin instead.

Visual Studio Code has a setting called Files: Auto Save. If set to afterDelay, then everything that is typed into a buffer is immediately saved to disk. It is great to skip the step of having to explicitly save files every time they are modified.

Some random Unix commands are very useful for the assignments. I recommend you try them and see what they do.

```
$ cat front.in | parse.exe
$ cat front.in | parse.exe -p -t
$ cat front.in | parse.exe -p -t | grep -v FOUND
$ parse ../Part-3-Test_Cases/8-mult_table.pas | grep compound_stmt
$ cat 8-mult_table.correct| grep compound_stmt
$ man grep
```

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

Written Report

Write a report about your experience. The report should cover the following items:

1. Describe, in a few sentences, the development process you used during this assignment. In what order did you develop the parsing functions? How many functions did you write?
2. Which nodes where the most difficult to write? Why were they difficult? Which nodes were the easiest to write? Why were they easy?
3. The code now prints up to two parse trees: the first as the parsing occurs, and the second as the parse tree nodes are traversed via the overloaded << operator. Discuss what insight you gained into the running of this complex program from the printout of each tree.
4. Describe, in a few sentences, any sources you consulted during your development of this assignment.
5. In addition to the requirements specified above, the report should be neat, well structured, grammatically correct, and use words that are spelled correctly.

List your *name*, *netID*, and the *date* at the top of your report. Otherwise, the report may use any format and fonts. The report does not have any specific length requirement. Completeness is valued more than length, and excessive length could result in a lower grade.

Written Report Grading Rubric

Criteria	Ratings				Points
Development Process	**rank 1: Mastery** Engaging sentences. Correct and clear writing.	**rank 2: Good** Adequate sentences. Presentation covers material but has minor writing or structural errors.	**rank 3: Developing** Sentences leave out some minor material or are poorly structured. Recurring writing issues. Details not as in depth as required. Poor formatting.	**rank 4: Lacking** No sentences or significant issues. Incomplete, too short or too long, or repeated, significant writing issues. An obvious slap-dash effort.	25
Writing Nodes	**rank 1: Mastery** Productions given, formatting clear and easy to read. Metasymbols are clearly indicated. Productions are correct.	**rank 2: Good** Productions given and are correct, but have minor formatting or structural errors.	**rank 3: Developing** Productions are not correct, or have major formatting issues. Metasymbols are not indicated. Not enough productions are given. Productions are not in proper EBNF format. Details not as in depth as required.	**rank 4: Lacking** No productions or significant issues. An obvious slap-dash effort.	25
Parse Trees	**rank 1: Mastery** Engaging answer. Correct and clear writing. Complete answers.	**rank 2: Good** Adequate answer. Presentation covers material but has minor writing or structural errors.	**rank 3: Developing** Answers leave out some minor material or are poorly structured. Recurring writing issues. Details not as in depth as required. Poor formatting.	**rank 4: Lacking** No answers or significant issues. Incomplete, too short or long, or repeated, significant writing issues. An obvious slap-dash effort.	25
Overall Quality	**rank 1: Mastery** Clearly formatted, contains all required elements, attractively laid out.	**rank 2: Good** Some minor formatting issues or missing elements, but report is still easy to follow.	**rank 3: Developing** Significant issues with formatting, or many missing elements. Report is not so easy to follow.	**rank 4: Lacking** No demonstration of mastery. Significant formatting issues, or many missing elements. An obvious slap-dash effort.	25

Deliverable

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.

Feel free to assist other students on this lab assignment. If you need additional help, please contact the TA or attend their office hours.

The sample input / output files provided with the assignment are a good starting point for testing your program. You are responsible for testing your program so that the lexical analyzer will function as expected with ***any*** input.