PA7: Zombie-C Compiler (Due 12/1 @ 11:59PM)

The GitHub Classroom link for this assignment is: https://classroom.github.com/a/ZMrl2yxL

In this lab, we will be writing a compiler for Zombie-C, a high-level programming language for the zombie control programs we used in PA6. There are three main parts to this lab:

1. Get the lexer/parser to correctly read in the tokens and match the grammar of Zombie-C.
2. Create the Node classes for the AST.
3. Generate the zombie assembly code given an AST.

We use flex/bison in this assignment.

Windows users will need to install Cygwin. Download the 32-bit version of Cygwin (**NOT** 64-bit) and manually install it. When it asks you which packages to install, you must manually pick flex and bison (they aren't selected by default).

Mac users don't need to install anything. But don't forget to set the working directory in your scheme.

For this assignment, you'll be going back to using the "tests" target instead of "main."

For the command line arguments, if argc == 2 or 3, then argv[1] is the input .zc file. If we want to enable optimization, then argc == 3 and argv[2] is the string `-o`.

For this lab, it will be very helpful to look at the "Introduction to Compilers" lecture. You also will want to refer to the calc examples on Blackboard.

Note: You don't have to write your own unit tests for this lab. However, you won't pass any of the student/graded tests until you've at least implemented some code generation in Part 3)

Note 2: Because of the way flex/bison work, smart pointers won't work on this lab.

Part 1: Scanner and Parser

The flex input file is Zombie.l, and the bison input file is Zombie.y. You will be editing these files exclusively during this part of the lab. You'll have to deduce the grammar of Zombie-C based on the given files.

two_statements.zc

First, get two_statements.zc to parse properly. All the words you see in ZombieC files are keywords, and so you need to define them as tokens in Zombie.l/y. To get two_statements.zc to parse properly, you need a "forward" token. Remember, this means adding both a token declaration in Zombie.y and the regular expression pattern in Zombie.l

Once you have a token for the forward keyword, add an additional type of `statement` in Zombie.y. Instead of a `statement` just being `rotate`, as it is right now, add `forward` as an option as well. Whenever you add a new grammar rule, I strongly encourage to add an action that outputs debug text.

Next, in Zombie.l add a pattern for single-line (C++-style) comments. A single-line comment is `"//"` followed by 0 or more characters of anything, ending with a `\n`. Because comments are not part of AST, you don't need to declare a token to go with it. Just put the regular expression, and for the action, increment `gLineNum` (as done for the current `\n` pattern in Zombie.l).

Next, add support for multi-statement blocks in Zombie.y. A block is one or more statements (use the "list" pattern like discussed in lecture).

If everything works, you should successfully parse two_statements.zc, and have debug output like this (you'll still fail all unit tests, though):

```
Numeric value of 0
Rotate command
Single statement
Forward command
Multiple statements
Main entry point found!
```

stationary.zc

Next, get stationary.zc working. It's the simplest file with an `if` statement. In ZombieC, ifs must always have elses, you must use braces, and there are no else ifs. So, all if statements are:

```
if (boolean)
{
    block
}
else
{
    block
}
```

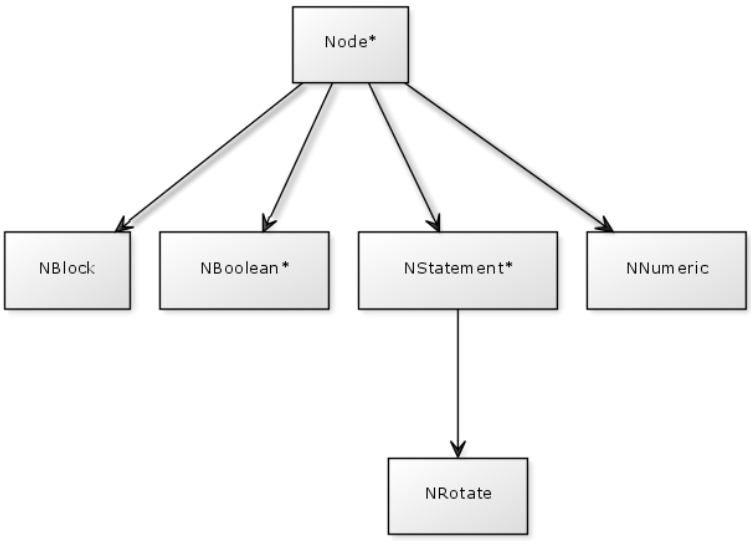
If combined with else is a statement. This requires creating a boolean grammar rule, that matches the different type of boolean checks including is_zombie(numeric), is_human(numeric), etc. This also means you'll need corresponding tokens.

The Rest

Once you have stationary.zc getting parsed properly, you're well on your way to getting the parsing done! I would suggest the following order of getting the remaining files to successfully parse: search.zc, michonne.zc, and the_governor.zc.

Part 2: The AST

Now that the Zombie-C files are parsing properly, we can generate the Abstract Syntax Tree. If you open Node.h, here are the current classes. The (*) signifies an abstract class:



You should not need to add any additional abstract classes to successfully build an AST. All the other classes you will make for the AST will derive from either `NBoolean` or `NStatement`. Depending on the node, it may need to take a pointer to another node type in its constructor. Or, if there are no additional parameters to apply to the node, then it may just use the default constructor.

Also look at the `%union` struct in Zombie.y. **You should not need to add any more types to the `%union`.**

Once you've created your Node subclasses, you can construct them in the relevant grammar actions. For instance, the action for the rotate is:

```
$$ = new NRotate($3);
```

Since `CodeGen` is a pure virtual function in the abstract classes, any derived class must implement it.

Most grammar actions will only require setting `$$` to a new instance of the node, as with `NRotate`. But there are a couple of special cases to watch out for.

The `main_loop` action needs to set the global `gMainBlock` pointer, and call `SetMainBlock` on it.

The other special case is `block`. When you match the first statement, you construct a new `NBlock`, and add the statement to this new `NBlock`'s list of statements. But for subsequent statements, rather than creating a new `NBlock`, you should add the new statement to the already existing `NBlock`.

Unfortunately, you can't test whether the AST built properly without adding the code gen code in part 3. But if the AST builds without crashing, that's a step in the right direction.

Part 3: Code Generation

Now that we have an AST, we can do code gen. **Your compiler should always output to a file called "out.zom"** (without any path).

As you start implementing this, you can comment out more and more test cases in the student tests file, and they should pass.

In SrcMain.cpp, after the `zompilerparse` call, if there's no compile error, the AST is valid. This is where you then generate the list of ZOM assembly commands to output.

We store the commands in a `struct` called `CodeContext`. This `struct` has a vector of strings. Each time you generate a new command, add it to the vector. The `CodeGen` function for each node takes this `CodeContext` by reference. Make a CodeContext as a local variable and pass it to the root node. This should then call CodeGen on every node in the AST, yielding the final vector with all the zom commands.

Only code gen if the lexical and syntax analysis was successful. `gMainBlock` will be `nullptr` if compilation failed, so first check to make sure that isn't the case. Then, call `CodeGen` on the main block as such:

```
gMainBlock->CodeGen(myContext);
```

Code Gen of basic.zc

First implement `CodeGen` in `NBlock`. What it needs to do is loop through all the statements in its member list, and call `CodeGen` on each statement. Once the loop finishes, you should check if the current block is the main block. If it is, this means that you need to add one final "goto,1" command to complete the main loop.

The implementation of `NRotate::CodeGen` is given. So, you don't have to worry about that for basic.zc.

The last thing you need to do is back in the main function. Once the `CodeGen` returns from the main block, you should have a `CodeContext` which has been populated with all the assembly commands. Output each command to a line in "out.zom."

You should pass the test for basic.zc if this works!

Code Gen of two_statements.zc

The code generation for the second file should only additionally require you to implement `NForward::CodeGen`, and you should pass this additional student test.

Code Gen of stationary.zc

For stationarcy.zc, you need to implement `CodeGen` for the most challenging statement in ZombieC: the if/else statement. With the if/else, you want the output code to be something like this:

1. Test flag
2. je to the if block
3. else block code
4. goto past the if block
5. if block code

The problem is that when we are generating code, we simply add the instruction to our vector. But when it's time to output the je, there's no way of knowing how many lines of code the else statement is going to be. Similarly, when it's time to output the `goto`, there's no way of knowing how long the if case code will be.

The way you can solve this is by keeping track of where the instruction for steps #2 and steps #4 are in the vector. Then, once the code for the if and else blocks has been generated, you can go back and fix up their jump locations to point to the correct line numbers.

After this, you should now pass stationary.zc.

The Other Files

Once you have the if/else working properly, you can go through the other files in the same order as before (search, michonne, and the_governor). If their outputs all line up, you're set mostly.

And a Little Bit of Optimization...

If ProcessCommandArgs gets an argc of 3 and argv[2] == "-o" you should enable optimization of goto statements.

In michonne.out.zom, there are gotos chained together. For instance, line 8 goes to line 10, which goes to line 12, which goes to line 15, which goes to line 1. But each of the statements in the chain could simply be "goto,1".

The problem is that when we initially add `goto` to the mOps vector, there is no way of knowing the `goto` is part of a chain that will lead back to the first line. So, we need to do this optimization after generating the code for the entire program.

An easy way to keep track of the gotos is by adding a `std::map<int,int>` of gotos to `CodeContext`. The key is the line number the `goto` statement is on, and the value is the line number the `goto` statement jumps to. Then every time a `goto` is ouput, you should add an entry to the map.

Then, when the `main_block` has been generated, you jump through the map locating any chains, replacing the `goto` chains with a `goto` directly to the line number at the end of the chain. So for example, instead of:

```
8. goto 10 -> 10. goto 12 -> 12. goto 15 -> 15. goto 1
```

We can simplify this chain to:

```
8. goto 1
```

Which is way more efficient!

Once you implement this, you should pass all graded tests on Travis. Don’t forget to fix your warnings.

Grading Rubric

Part	Points
Graded Tests	90
Code Quality	10
Total	**100**