January 2025 CSE 310: Compiler Sessional
Assignment 3: Syntax and Semantic Analysis

Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology

May 2025

Overview

In the previous assignment, we have constructed a lexical analyzer to generate token streams. In this assignment, we will construct the last part of the front end of a compiler for a subset of the C language. That means we will perform syntax analysis and semantic analysis with a grammar rule containing rule actions. To do so, we will build a parser with the help of ANTLR4, a robust parser generator that produces parsers capable of constructing and traversing parse trees.

Change Log

Description	Updated By	Timestamp
Assignment declared	ABS & NTD	24 May, 2025, 11:00 PM
Naming format updated (section 8)	NTD	11 June, 2025, 6:45 PM

1 Language

Our chosen subset of the C language has the following characteristics:

- There can be multiple functions. No two functions will have the same name. A function needs to be defined or declared before it is called. Also, a function and a global variable cannot have the same symbol.
- There will be no pre-processing directives like `#include` or `#define`.
- Variables can be declared at suitable places inside a function. Variables can also be declared in the global scope.
- All the operators used in the previous assignment are included. Precedence and associativity rules are as per standard. Although we will ignore consecutive logical operators or consecutive relational operators like, `a & b & c` or `a < b < c`.
- No break statement and switch-case statement will be used.

2 Tasks

You have to complete the following tasks in this assignment.

2.1 Syntax Analysis

For the syntax analysis part you have to do the following tasks:

- Use the grammar provided in the file `C8086Parser.g4` as skeleton to start writing grammar file. Ensure the grammar name matches the file name.
- In your previous assignment, you did symbol table insertions in lexer rules. Now there will be no symbol table insertions from the lexer rules as these will be handled in the parser. Some lexer rules are provided in the given `C8086Lexer.g4` which you can use as your starting point.
- Use ANTLR's token attributes to write logging information. For example, if your lexer detects an identifier, it will return a token named ID to parser. You can access the attributes using syntax like `$ID.text` or `$ID->getText()`.
- Resolve any ambiguities in the given grammar (e.g., if-else conflicts) by using ANTLR's precedence rules. Ensure the grammar compiles without errors using the `antlr4` command.
- Insert all identifiers into the symbol table when they are declared in the input file. For example, if you find `int a, b, c;`, insert `a`, `b`, and `c` into the symbol table. Implement this in the parser's variable declaration rule.
- When a grammar rule matches the input from the C code, print the matching rule in the correct order in an output file (`log.txt`). For each grammar rule matched with some portion of the code, print the rule along with the relevant portion of the code. See the given log files for clarity.
- Print the symbol table when a scope exits (in the `log.txt` file). This means printing the symbol table just before the scope is removed.

- Print well-formed syntax error messages with line numbers in `error` files.
- Print the symbol table after finishing parsing (in the `log.txt` file).

It is better to look at the given I/O files for clarity than just looking at instructions.

2.2 Semantic Analysis

In this part, you have to perform the following tasks:

- **Type Checking:** You have to perform different types of type checking in this part. You may implement these checks in your grammar rule actions and perform the following semantic checks:
  - Generate an error message if operands of an assignment operator are not consistent with each other. Note that the second operand of the assignment operator will be an expression that may contain numbers, variables, function calls etc.
  - Generate an error message if the index of an array is not an integer.
  - Both operands of the modulus operator should be integers.
  - During a function call, all arguments should be consistent with the function definition.
  - A void function cannot be called as part of an expression.
- **Type Conversion:** You have to perform some type conversion. For example, generate an error/warning message if a floating-point number is assigned to an integer type variable. Also, the result of `RELOP` and `LOGICOP` operations should be an integer.
- **Uniqueness Checking:** Check whether a variable used in an expression is declared. Also, check whether there are multiple declarations of variables with the same ID in the same scope.
- **Array Index:** Check whether an index is used with an array and vice versa.
- **Function Parameter:** Check whether a function is called with the appropriate number of parameters with appropriate types. Function definitions should also be consistent with declarations if any exists. A function call cannot be made with a non-function type identifier. To implement this task, add necessary fields to a custom `SymbolInfo` class as required, try to but avoid redundant fields.

We always appreciate your new ideas. So, if you find any other errors than the ones mentioned and provided in sample I/O, incorporate that in your code but make sure that your output matches.

3 Handling Grammar Rules for Functions

For implementing the grammar rules of functions, you will need to add some fields in your `SymbolInfo` class as required.

- You will need extra fields to store the return type, parameter list, number of parameters, etc., in the `SymbolInfo` class for proper handling of functions. You can create another class to hold these fields and add a reference to that class in the `SymbolInfo` class for convenience. Note that this is just a guideline; you are free to implement otherwise.
- As part of the semantic analysis, match the function declaration and function definition and report an error if there is any mismatch in the return type, parameter number, parameter sequence, or parameter type.

4 Input

The input will be a C source program with a `.c` extension. The filename must be provided via the command line.

5 Output

In this assignment, there will be two output files. One file, `log.txt`, will contain matching grammar rules, the corresponding segment of source code, and symbol table entries as instructed in the previous sections. Another file, `error.txt`, will contain error messages with line numbers. For any detected error, print something like "Line no 5: Corresponding error message". Print the line count and number of errors at the end of the `log.txt` file. For more clarification about input-output, check the supplied sample I/O files.

6 Setup and Implementation

Use ANTLR4 to generate the lexer and parser. You may implement the parser in either Java or C++, following the setup instructions provided in the accompanying "Getting Started with ANTLR4" document. For C++ implementations, ensure the ANTLR4 C++ runtime is installed and properly linked, as described in the setup guide. For Java, ensure the ANTLR4 jar file is included in the classpath.

- Generate the lexer and parser using the command `antlr4 -Dlanguage=Cpp CSubset.g4` for C++ or `antlr4 CSubset.g4` for Java.
- Use a custom `SymbolInfo` class to manage symbol table entries, extended as needed for function-related information.

7 Submission

- Plagiarism is strongly prohibited. In case of plagiarism, −100% marks will be given.
- No submission after the deadline will be allowed.
- Deadline shall not be extended under any circumstances.

8 Submission

All submissions will be taken via the departmental Moodle site. Please follow the steps given below to submit your assignment. 10% **out of the total assignment marks will be allocated for the correct submission.**

1. In your local machine, create a new folder named with your 7-digit student ID. Do not miss this point.
2. Include the ANTLR grammar file named `C<your student id>.g4` (e.g `C2105193Lexer.g4`) containing your grammar. Also include additional C++ or Java or script files necessary to compile and run your parser. Do not include input/output files, generated lexer/parser files, or executables in this folder.

3. Compress the folder into a zip file named with your 7-digit student ID.
4. Submit the zip file within the deadline.

9 Deadline

The submission deadline is set at June 20, 2025, 11:55 PM. There will be no extension of this deadline. If you cannot access the Moodle site, send an email to any of your course teachers attaching the zip file mentioned in the earlier section (within the deadline, of course).