December 21, 2023

**BANGLADESH UNIVERSITY OF ENGINEERING AND TECHNOLOGY**
**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**
**CSE310 (Compiler Sessional), July 2023 Term**
**Assignment on Syntax and Semantic Analysis**
**All Sessional Sections**

# 1 Introduction

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 function implementation in this assignment. To do so, we will build a parser with the help of Lex (Flex) and Yacc (Bison).

# 2 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`, `a < b < c`.
- No `break` statement and `switch-case` statement will be used.

# 3 Tasks

You have to complete the following tasks in this assignment.

## 3.1 Syntax Analysis

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

- Incorporate the grammar given in the file "BisonAssignmentGrammar.PDF" along with this document in your Yacc file.

- Modify your Lex file from the previous assignment to use it with your Yacc file. Remove all the symbol table insertions from the Lex file.
- Use a `SymbolInfo` pointer to pass information from lexical analyzer to parser when needed. For example, if your lexical analyzer detects an identifier, it will return a token named ID and pass its symbol and type using a `SymbolInfo` pointer as the attribute of the token. On the other hand, in the case of semicolons, it will only return the token as the parser does not need any more information.

  You can implement this in two ways: either redefine the type of `yylval (YYSTYPE)` in parser and associate `yylval` with new type in the scanner, or use `%union` field in the parser.
- Handle any ambiguity in the given grammar (For example, `if-else`, you can find a solution in page 188-189 of flex-bison manual). Your Yacc file should compile with zero (0) conflicts.
- Insert all the identifiers in the symbol table when they are declared in the input file. For example, if you find `int a, b, c;` then insert `a`, `b`, and `c` in the symbol table. You can do this in the grammar rule of declaration.
- When a grammar matches the input from the C code, it should 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.?
- As rules are matched against the lines of the source code, print the matching rules in the (`log.txt`) file. At the same time keep generating and storing the parse tree for the code. Finally, when parsing of the input C code is finished, print the parse tree in (`parsetree.txt`) file. See the supplied sample output to find the formatting of this file.
- Print symbol table when a scope exists (in the `log.txt` file). This means to print the symbol table just before the `exitScope` function of the symbol table is called.
- Print well-formed syntax error messages with line number (in an `error.txt` file).
- Print the symbol table after finishing parsing (in the `log.txt` file).

**Bonus Task:** Incorporate error recovery in your parser. Go through the Yacc manual for a better understanding of error recovery. You might need to use Yacc's predefined token error (token number 256) for this purpose.

## 4 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 have to perform the following semantic checks:

- Generate 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 the operands of the modulus operator should be integers.
- During a function call all the arguments should be consistent with the function definition.
- A void function cannot be called as a part of an expression.

- **Type Conversion:** You have to perform some type-conversion. For example, you have to generate error/warning message if floating point number is assigned to an integer type variable. Also, the result of RELOP and LOGICOP operation should be an integer.

- **Uniqueness Checking:** You should check whether a variable used in an expression is declared or not. Also, check whether there are multiple declarations of variables with the same ID in the same scope.

- **Array Index:** You have to check whether there is an index used with array and vice versa.

- **Function Parameter:** Check whether a function is called with appropriate number of parameters with appropriate types. Function definitions should also be consistent with declaration if there is any. Besides that, a function call cannot be made with non-function type identifier. To implement this task, you can add necessary fields in the `SymbolInfo` class as required but try to avoid redundant fields.

## 5 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 take another class to hold the above-mentioned fields and add a reference of that class in the `SymbolInfo` class for convenience. Note that this is just a guideline, you are free to implement otherwise.
- As a part of the semantic analysis you have to 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.

**Bonus Task:** Report an error if there is any invalid scoping of the function.

## 6 Input

The input will be a C source program in `.c` extension. The filename will have to be given from the command line.

## 7 Output

In this assignment, there will be three output files. One file, `parsetree.txt`, will contain the parse tree corresponding to the source code. Another file, `error.txt` will contain error messages with line numbers. The other file, `log.txt` will contain various log messages as lexical analysis and parsing is performed. Also print the line count and number of errors at the end of the log file. For more clarification about input-output check the supplied sample I/O files.

## 8 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.

## 9 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 which is to be named as your 7 digit student ID. Do not miss this point.
2. Put the Lex file named as `<your student id>.l` and Yacc file named as `<your student id>.y` containing your code. Also put additional C files or header files that are necessary to compile your Lex file. Do not put any input/output file, any shell script, the generated `lex.yy.c` file, or any executable file in this folder.
3. Compress the folder in a zip file which should be named as your 7 digit student id.
4. Submit the zip file within the Deadline.

## 10 Deadline

The submission deadline is set at **January 21 (Sunday), 2024, 11:00 PM**. There will be no extension of this deadline. In case you can not 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).