Department of Computer Science and Engineering

Course Code: CSE 420	**Credits: 1.5**
Course Name: Compiler Design	**Semester:** **Spring 2025**

1 Introduction

In the last assignment, we have constructed a syntax analyzer to for a subset of C language.

In this assignment, we will construct a symbol-table. A symbol-table is a data structure maintained by compilers in order to store information about the occurrence of various entities such as identifiers, objects, function names etc. Information of different entities may include type, value, scope etc. At the starting phase of constructing a compiler, we will construct a symbol-table which maintains a list of hash tables where each hash table contains information of symbols encountered in a scope of the source program.

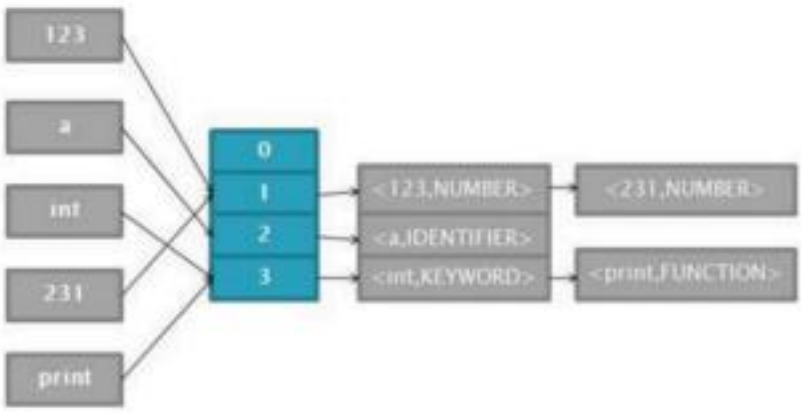

Figure 1: Symbol Table Using Hashing.

2 Symbol Table

In this assignment we will construct a symbol table that can store type and scope information of a symbol found in the source program. If the source program consists of a single scope then we can use a hash table to store information of different symbols. Hashing is done using the symbol as key. *Figure 1* illustrates such a symbol table. Now consider the following C code:

```
1. int a,b,c;
2. int func(int x) {
3.   int t=0;
4.   if(x==1){
5.        int a=0;
6.        t=1;
7.   }
8.   return t;
9. }
10.int main(){
11.  int x=2;
12.  func(x);
13.  return 0;
14.}
```

To successfully compile this program, we need to store the scope information.

Suppose that we are currently dealing with *line no 6*. In that case, global variables *a* , *b* and *c* , function parameter *x* , function *func*'s local variable *t* and the variable *a* declared within the if block, are visible. Moreover the variable a declared within the if block hides the global variable *a*. How can we store symbols in a symbol-table which can help us to handle scope easily?

One way is to maintain a separate hash table for each scope. We call each hash table a Scope-Table . In our Symbol-Table , we will maintain a list of scope-tables. You can also think of the list of scope-tables as a stack of scope-tables. Whenever we enter a new block, we create a new scope-table and insert it at the top of the list. Whenever we find a new symbol within that block of the source code, we insert it in the newly created scope-table i.e. the scope-table at the top of the stack. When we need to get the information of a symbol, at first we will search at the topmost scope-table.

If we could not find it there, we would search in its parent scope-table and so on. When a block ends we simply pop the topmost scope table. Suppose we give a unique number to each block, 1 to global scope, 2 to the function func, 3 to the if block and 4 to the main function. *Figure 2* illustrates the state of the symbol table when we are in *line no. 6* of the given code.

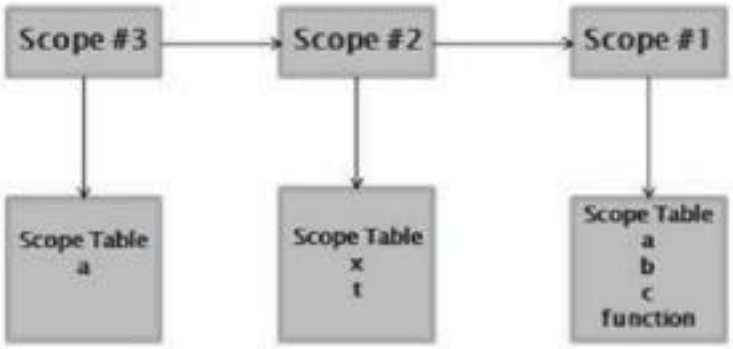

Figure 2: Symbol Table with scope handling.

3 Tasks

3.1 Symbol Table Generation

We have already used a **symbol_info** class to build our syntax analyzer.
Now, You have to implement two more classes.

- **scope_table**: This class implements a *hash table*. You may need an array (**array of pointers of symbol_info type**) and a hash function. The hash function will determine the corresponding bucket no. the sumbol_info object will go into. The hash function would be *any suitable hash function to your choice*. The hash function will take the name of the symbol as input. Don't forget to keep the hashvalue in the range of the bucketsize.

  You will also need a pointer of **scope_table** type object named **parent_scope** as a member variable so that you can maintain a list of scope tables in the symbol table. Also give each table a **unique id**.
  You also need to add the following functionalities to your Scope Table:
  ❑ **Insert**: Insert into symbol table if already not inserted in this scope table. Return type of this function should be boolean indicating whether insertion is successful or not. ❑ **Look up**: Search the hash table for a particular symbol. Return a symbol_info pointer. ❑ **Delete**: Delete an entry from the scope table. Return true in case of successful deletion and false otherwise.
  ❑ **Print**: Print the scope table in the logfile.

  You should also **write a constructor** that takes an integer n as a parameter and allocates n buckets for the hash table. You should also **write a destructor** to deallocate memory.

  The function signatures are already given in the **scope_table.h** file. Complete them.

❖ **symbol_table**: This class implements **a list of scope tables**. The class should have a **pointer of scope_table** type which indicates the current scopetable. This class should contain the following functionalities:

❑ **Enter Scope**: Create a new scope_table and make it current one. Also make the previous current table as its parent_scope_table.

❑ **Exit Scope**: Remove the current scope_table.

❑ **Insert**: Insert a symbol in current scope_table. Return true for successful insertion and false otherwise. This should call the Insert function of the current scope_table. ❑ **Remove**: Remove a symbol from current scope_table. Return true for successful removal and false otherwise. This should call the remove function of the current scope_table.

❑ **Look up**: Look up a symbol in the symbol_table. At first search in the current scope_table then search in its parent scope table and so on. Return a pointer to the symbol_info object representing the searched symbol.

❑ **Print Current scope_table:** Print the current scope_table to logfile.

❑ **Print All scope_table**: Print all the scope_table(s) currently in the symbol_table.

3.2 Syntax Analysis with Symbol Insertions

We have already done syntax analysis in the 2nd assignment. Now, you have to modify the action parts for the grammar rules, where necessary. 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. Find out in the grammar rules where the declarations of variables are handled. You also need to insert function names in your symbol table in a similar way.

There can be three types of identifiers in your code.

1. **Normal Variable**: Store the variable name and datatype (int, float etc)
2. **Array Variables**: Store the variable name, datatype and array size.
3. **Function Names**: Store the function name, return type (find out which types are allowed in the grammar), number of parameters and types of each parameter. (Find out where the parameters and their types are handled in the grammar)

You may need to *modify the **symbol_info class*** to store this information about each symbol.

4 Input

The input will be a file with .c extension containing a c source program. File name will be given from the command line.

5 Output

In this assignment, there will be one output file. The output file should be named as **<Your_student_ID>_log.txt**. This will contain **matching grammar rules, and corresponding segment of source code** as the previous assignment. Print the line count at the end of the log file.

Additionally-

- Print the ID of the new scope_table when you enter a new scope.
- Print which scope_table is being removed from the scope_stack when you are exiting a scope
- When you exit a scope print the current state of the symbol table.
- To print the symbol table, you need to print the scope tables in the order they should appear in the scope stack. For each bucket, print the symbols and relevant information of the symbols.

For more clarification about input-output check the supplied sample I/O files given in the lab folder. You are highly encouraged to produce output exactly like the sample one.

6 Submission

1. In your local machine create a new folder whose **name is your student id.** 2. Put the lex file named as **<your_student_id>.l**, the Yacc file **<your_student_id>.y** and a script named **script.sh** (modifying with your own filenames), the **symbol_info.h** file, the **scope_table.h** file, the **symbol_table.h** a suitable input file names **input.c** in a folder **named with your student id**. **DO NOT** put any output file, generated lex.yy.c/y.tab.h/y.tab.c file or any executable file in this folder.
3. Compress the folder in a **.zip file** which should be **named as your student id**.
4. Submit the .zip file.

Failure to follow these instructions will result in penalty.