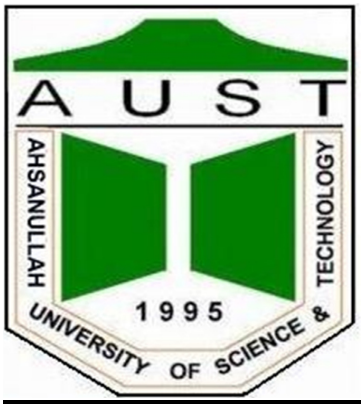
AHSANULLAH UNIVERSITY OF SCIENCE & TECHNOLOGY

Department of Computer Science and Engineering



Course No: CSE 4108

Course Name: Artificial Intelligence Lab

Section: B Lab Group: B2

Semester: Fall 2020

Report on Project: A Simple Compiler Based on Scanning and Filtering, Lexical Analysis, Symbol Table Construction and Management, Detecting Simple Syntax Errors

Submitted to:

Mr. Md. Aminur Rahman

Assistant Professor

Department of CSE, AUST

Submitted By:

Name: Md. Abir Hossain

ID: 170204106

Introduction

A compiler is a special program that processes statements written in a particular programming language and turns them into machine language or "code" that a computer's processor uses. Typically, a programmer writes language statements in a language such as Pascal or C one line at a time using an editor. The file that is created contains what are called the source statements. The programmer then runs the appropriate language compiler, specifying the name of the file that contains the source statements.

When executing (running), the compiler first parses (or analyzes) all of the language statements syntactically one after the other and then, in one or more successive stages or "passes", builds the output code, making sure that statements that refer to other statements are referred to correctly in the final code. Traditionally, the output of the compilation has been called object code or sometimes an object module. (Note that the term "object" here is not related to object oriented programming.) The object code is machine code that the processor can execute one instruction at a time.

A compiler works with what are sometimes called 3GL and higher-level languages. An assembler works on programs written using a processor's assembler language. In this project, we will build a basic compiler in 4 steps.

Step 0: Input Step

This is the initial step where user will input in a file. The name of the file. This file will be used as the input for the next step. The sample input which is being used to demonstrate the whole project is below:

```
/* A program fragment*/

float x1 = 3.125;;;
/* Definition of function f1 */
double f1(float a, int int x)
{if(x<x1)
double z;;
double z;
```

```
else z = 0.01;}}
else return z;
}
/* Beginning of main */
int main()
{{{{
    for(;;);
int n1; double z;
n1=25; z=f1(n1);}
```

Output Snippet:

```
input file:
/* A program fragment*/

float x1 = 3.125;;;
/* Definition of function f1 */
double f1(float a, int int x)
{if(x<x1)
double z;;
double z;
else z = 0.01;}}
else return z;
}
/* Beginning of main */
int main()
{{{{
    for(;;);
int n1; double z;
n1=25; z=f1(n1);}
```

Step 1: Scanning and Filtering a Source Program:

In this step, first the input file will be scanned and then, the comments and white spaces will be filtered. There are two type of comments, multi-line comment and single line comment. Both kinds of comments will be filtered in this step.

Output Snippet:

```
after deleting comment and adding line number
1
2
3       float x1 = 3.125 ; ; ;
4
5       double f1 ( float a , int int x )
6       { if ( x < x1 )
7       double z ; ;
8       double z ;
9       else z = 0.01 ; } }
10      else return z ;
11      }
12
13      int main ( )
14      { { { {
15      for ( ; ; ) ;
16      int n1 ; double z ;
17      n1 = 25 ; z = f1 ( n1 ) ; }
18
```

Step 2: Lexical Analysis:

In this step, we will separates out the valid tokens from the input. We will do this in 2 steps. In the first step, we will separate the lexemes. Then in the second step, we will categorized them, such as kw for keyword, id for identifier, num for number, par for parenthesis, sep for separator, op for operator and un for unknown.

Output Snippet:

```
after tokenizing:
1
2
3       [kw] [float] [id] [x1] [op] [=] [num] [3.125] [sep] [;] [sep] [;] [sep] [;]
4
5       [kw] [double] [id] [f1] [par] [(] [kw] [float] [id] [a] [sep] [,] [kw] [int] [kw] [int] [id] [x] [par] [)]
6       [par] [{] [kw] [if] [par] [(] [id] [x] [op] [<] [id] [x1] [par] [)]
7       [kw] [double] [id] [z] [sep] [;] [sep] [;]
8       [kw] [double] [id] [z] [sep] [;]
9       [kw] [else] [id] [z] [op] [=] [num] [0.01] [sep] [;] [par] [}] [par] [}]
10      [kw] [else] [kw] [return] [id] [z] [sep] [;]
11      [par] [}]
12
13      [kw] [int] [id] [main] [par] [(] [par] [)]
14      [par] [{] [par] [{] [par] [{] [par] [{]
15      [kw] [for] [par] [(] [sep] [;] [sep] [;] [par] [)] [sep] [;]
16      [kw] [int] [id] [n1] [sep] [;] [kw] [double] [id] [z] [sep] [;]
17      [id] [n1] [op] [=] [num] [25] [sep] [;] [id] [z] [op] [=] [id] [f1] [par] [(] [id] [n1] [par] [)] [sep] [;]
 [par] [}]
```

Step 3: Symbol Table Construction and Management:

In this step, we will construct a symbol table using hash map, the table in which all the identifiers will be stored along with information about them. There are 3 steps in this code. In the first step, after complete recognition of all the lexemes only identifiers are kept in pairs for formation of Symbol Tables. In the second step, we will generate the symbol table.

Output Snippet:

```
Symbol Table
 0              f1             func           double         global         ~
 6              main           func           int            global         ~
 7              x              var            int            f1             ~
 10             n1             var            int            main           25
 11             z              var            double         main           f1
 18             x1             var            float          global         3.125
 19             a              var            float          f1             ~
 20             z              var            double         f1             0.01
```

Step 4: Detecting Simple Syntax Errors:

In this final step of making a basic compiler, we will find out the simple syntax errors, like duplicate subsequent keywords, unbalanced curly braces, unmatched else, undeclared identifiers, duplicate identifiers etc. We will do this in 3 steps. In the first step, we will remove comments and add line number. In the second step, we will tokenize the lexemes line by line. Then in the last step, we will identify the errors.

Output Snippet:

```
Simple Syntex Error Detection:
line: 3 sep Duplicate token ;
line: 5 kw Duplicate token int
line: 7 sep Duplicate token ;
line: 8 id duplicate variable z
line: 9 par unmatched }
line: 10 kw unmatched else
line: 11 par unmatched }
PS D:\4.1\CSE 4129 Formal Languages and Compilers\lab\project>
```