



AHSANULLAH UNIVERSITY OF SCIENCE AND TECHNOLOGY
Department of Computer Science and Engineering

CSE4130: Formal Languages & Compilers Lab

Fall 2020

PROJECT REPORT

**A Simple Compiler Based on Scanning and Filtering,
Lexical Analysis, Symbol Table Construction and
Management, Detecting Simple Syntax Errors in addition
to Use of CFGs for Parsing**

Lab Section: A1

Submitted To

Mr. Md. Aminur Rahman

Assistant Professor

Department of CSE, AUST

Submitted By

Samiul Islam Niloy

Student ID: 170204016

October 9, 2021

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 and extra 1 step in the beginning for input and 1 step at the end for to show the use of CFG (Context Free Grammar) for parsing which is unrelated to the compiler but related to the sessions we were taught in the lab.

Step 0: Input Step

This is the initial step where user will input in the console and will be saved in a file. The name of the file where input will be saved is **input.txt**. 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:

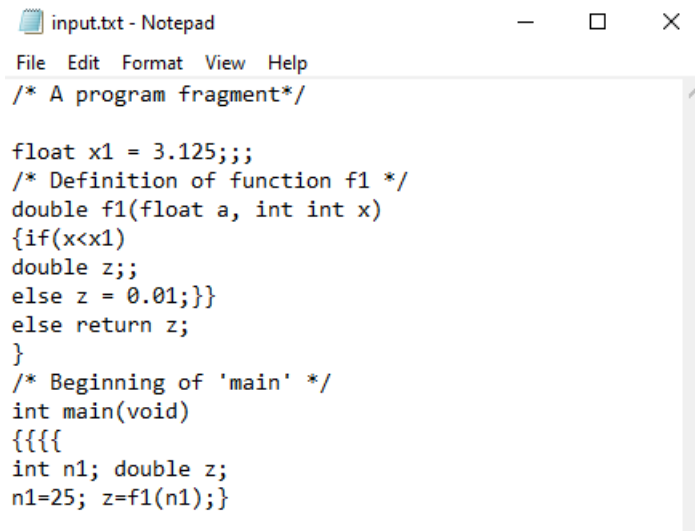
```
1 /* A program fragment*/
2
3 float x1 = 3.125;;;
4 /* Definition of function f1 */
5 double f1(float a, int int x)
6 {if(x<x1)
7 double z;;
8 else z = 0.01;}}
9 else return z;
10 }
11 /* Beginning of 'main' */
12 int main(void)
13 {{{{
14 int n1; double z;
15 n1=25; z=f1(n1);}
```

Please Enter The Input. If you want to stop input please EOF or CTRL+Z:

```
/* A program fragment*/

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

Figure 1: Entering input in the console



```

input.txt - Notepad
File Edit Format View Help
/* A program fragment*/

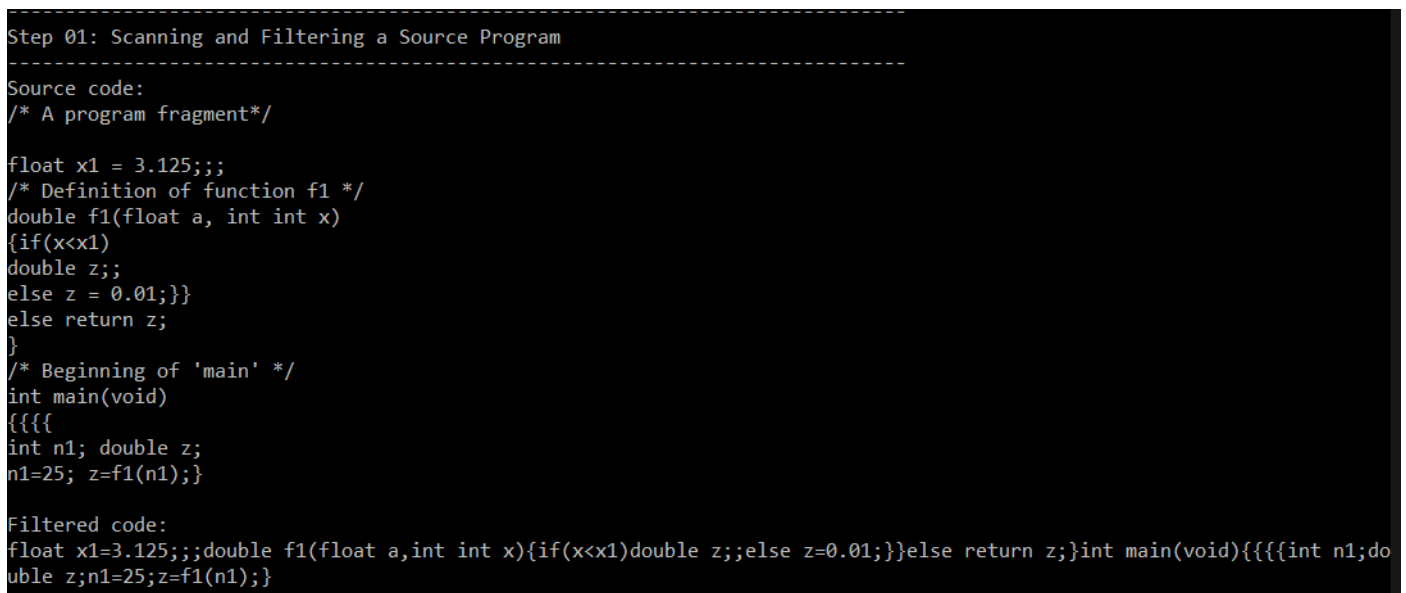
float x1 = 3.125;;
/* Definition of function f1 */
double f1(float a, int int x)
{if(x<x1)
double z;;
else z = 0.01;}}
else return z;
}
/* Beginning of 'main' */
int main(void)
{{{
int n1; double z;
n1=25; z=f1(n1);}

```

Figure 2: Elements in **input.txt** file after entering the input

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. After filtering, the filtered output will be saved in **filtered.txt** file.



```

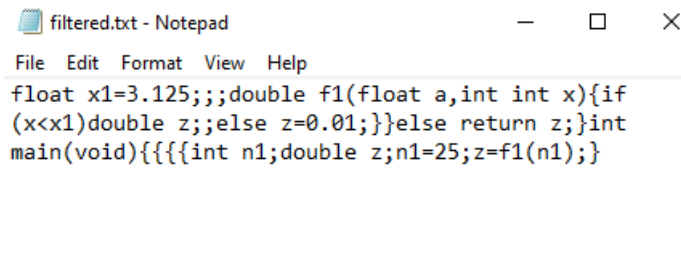
Step 01: Scanning and Filtering a Source Program
-----
Source code:
/* A program fragment*/

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

Filtered code:
float x1=3.125;;double f1(float a,int int x){if(x<x1)double z;;else z=0.01;}}else return z;}int main(void){{{{int n1;double z;n1=25;z=f1(n1);}

```

Figure 3: Output of step 1 (Scanning and Filtering a Source Program) in the console



```

float x1=3.125;;;double f1(float a,int int x){if
(x<x1)double z;;else z=0.01;}}else return z;}int
main(void){{{{int n1;double z;n1=25;z=f1(n1);}

```

Figure 4: Contents in **filtered.txt** file

Step 2: Lexical Analysis

In this step, we will separates out the valid tokens from the input(**filtered.txt**). 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 unkn for unknown.

```

-----
Step 02: Lexical Analysis
-----

Sample Input:
float x1=3.125;;;double f1(float a,int int x){if(x<x1)double z;;else z=0.01;}}else return z;}int main(void){{{{int n1;do
uble z;n1=25;z=f1(n1);}

Step 1:
float x1 = 3.125 ; ; ; double f1 ( float a , int int x ) { if ( x < x1 ) double z ; ; else z = 0.01 ; } } else ret
urn z ; } int main ( void ) { { { { int n1 ; double z ; n1 = 25 ; z = f1 ( n1 ) ; }

Step 2:
[kw float] [id x1] [op =] [num 3.125] [sep ;] [sep ;] [sep ;] [kw double] [id f1] [par (] [kw float] [id a] [sep ,] [kw
int] [kw int] [id x] [par )] [par {} [kw if] [par (] [id x] [op <] [id x1] [par )] [kw double] [id z] [sep ;] [sep ;] [k
w else] [id z] [op =] [num 0.01] [sep ;] [par {} [par {} [kw else] [kw return] [id z] [sep ;] [par {} [kw int] [id main]
[par (] [kw void] [par )] [par {} [par {} [par {} [par {} [kw int] [id n1] [sep ;] [kw double] [id z] [sep ;] [id n1] [
op =] [num 25] [sep ;] [id z] [op =] [id f1] [par (] [id n1] [par )] [sep ;] [par {}

```

Figure 5: Output of step 2 (Lexical Analysis) in the console

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. Finally, in the third step, we will modify the lexemes where the token streams (identifiers) will be replaced with numbers.

```

Step 03: Symbol Table Construction and Management
-----
Step 1:

[float] [id x1] [=] [3.125] [;] [;] [;] [double] [id f1] [(] [float] [id a] [,] [int] [int] [id x] []) [(] [if] [(] [id x] [<] [id x1] []) [d
ouble] [id z] [;] [;] [else] [id z] [=] [0.01] [;] [}] [}] [else] [return] [id z] [;] [}] [int] [id main] [(] [void] []) [(] [}] [}] [int] [id
n1] [;] [double] [id z] [;] [id n1] [=] [25] [;] [id z] [=] [id f1] [(] [id n1] []) [;] [}]

Step 2:

Sl No.   Name   Id Type   Data Type   Scope   Value
1        x1      var      float      global  3.125
2        f1      Func     double     global
3        a       var      float      f1
4        x       var      int        f1
5        x1      var      int        f1
6        z       var      double     f1      0.01
7        z       var      double     global
8        main   Func     int        global
9        n1      var      double     main    25
10       z       var      double     main

Step 3:

[float] [id 1] [=] [3.125] [;] [;] [;] [double] [id 2] [(] [float] [id 3] [,] [int] [int] [id 4] []) [(] [if] [(] [id 4] [<] [id 1] []) [doub
le] [id 6] [;] [;] [else] [id 6] [=] [0.01] [;] [}] [}] [else] [return] [id 7] [;] [}] [int] [id 8] [(] [void] []) [(] [}] [}] [int] [id
9] [;] [double] [id 10] [;] [id 9] [=] [25] [;] [id 10] [=] [id 2] [(] [id 9] []) [;] [}]

```

Figure 6: Output of step 3 (Symbol Table Construction and Management) in the console

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.

```
-----  
Step 04: Detecting Simple Syntax Errors  
-----  
  
/* A program fragment*/  
  
float x1 = 3.125;;;  
/* Definition of function f1 */  
double f1(float a, int int x)  
{if(x<x1)  
double z;;  
else z = 0.01;}}  
else return z;  
}  
/* Beginning of 'main' */  
int main(void)  
{{{{  
int n1; double z;  
n1=25; z=f1(n1);}
```

Figure 7: Output of step 4 (Detecting Simple Syntax Errors) input in the console

```
-----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     else z = 0.01 ; } }  
9     else return z ;  
10  }  
11  
12  int main ( void )  
13  {  
14  { { {  
15    int n1 ; double z ;  
16    n1 = 25 ; z = f1 ( n1 ) ; }  
17  }
```

Figure 8: Output of step 4 (Detecting Simple Syntax Errors) removing comments and adding line number

```
-----Error-----  
  
line no 3 - error type sp - error: ; - Duplicate token  
line no 5 - error type kw - error: int - Duplicate token  
line no 7 - error type sp - error: ; - Duplicate token  
line no 8 - error type par - error: } - Unbalanced Bracket  
line no 9 - error type kw - error: else - Unmatched else  
line no 9 - error type kw - error: return - Duplicate token  
line no 10 - error type par - error: } - Unbalanced Bracket  
line no 15 - error type par - error: { - Unbalanced Bracket
```

Figure 9: Output of step 4 (Detecting Simple Syntax Errors) identifying errors in the input code

Step 5: Use of CFGs for Parsing

This step is not related to making a compiler. But this was a vital session in our lab. So, this step is additionally merged with the complete compiler code. In this step, the content in **input.txt** file will be checked whether they are accepted by the given grammar or not. The grammar is given below:

```

<stat> → <asgn_stat> | <dscn_stat> | <loop_stat>
<asgn_stat> → id = <expn>
<expn> → <smpl_expn> <extn>
<extn> → <relop> <smpl_expn> | ε
<dscn_stat> → if ( <expn> ) <stat> <extn1>
<extn1> → else <stat> | ε
<loop_stat> → while ( <expn> ) <stat> | for ( <asgn_stat> ; <expn> ; <asgn_stat>
) <stat>
<relop> → == | != | <= | >= | > | <

```

Note: <smpl_expn> can be implemented using below grammar.

```

<Exp> → <Term> + <Term> | <Term> - <Term> | <Term>
<Term> → <Factor> * <Factor> | <Factor> / <Factor> | <Factor>
<Factor> → ( <Exp> ) | ID | NUM
ID → a|b|c|d|e
NUM → 0|1|2|...|9

```

Non-terminal symbols:

<Exp>, <Term>, <Factor>

Terminal symbols:

+, -, *, /, (,), a, b, c, d, e, 0, 1, 2, 3, ..., 9

Start symbol:

<Exp>

```

-----
Step 05: Use of CFGs for Parsing
-----
Rejected by Grammar.

Process returned 0 (0x0)   execution time : 7.992 s
Press any key to continue.

```

Figure 10: Output of step 5 (Use of CFGs for Parsing) in the console