Spring Semester 2020-2021

Lab Report for CSN-362 (Compiler Laboratory)

Submitted by

Shubhang Tripathi (18114074)

stripathi1@cs.iitr.ac.in

Submitted to

Dr. Pradumn K. Pandey

Department of Computer Science and Engineering

Indian Institute of Technology (IIT) Roorkee

May 30, 2021

Contents

Lab 0

1.1 Problem Statement

Write a lex program to identify tokens available in miniC.

1.2 Resources

- C
- flex

1.3 Files

- miniC.l - lex file with definitions for various kinds of tokens
- input.txt - sample input file

1.4 Implementation Details

The implmentation contains regular expressions for various lexemes like operator, separators and keywords, which just use OR nodes. These are of the form:

```
operators [!\+\*\-=\|<>\/]|\+\+|\-\-|==|!=|\+=|\-=|\*=|\/=|>=|<=

separators [\{\}\(\)\[\],;:]

keyword char|else|if|int|return|void|while
```

Regular definitions are used for normal character, digit and escape sequences:

```
char [a-zA-Z_]
digit [0-9]
escape_sequences \\.
```

These regular definitions are further used to define the chacter constants, string constants, number constants and float constants:

```
char_const \'([^\']|{escape_sequences})\'
string_const \"([^\"]|{escape_sequences})*\"
number_const {digit}+
float_const {digit}*\.{digit}+
```

A character constant has the form of either a byte which is not ' or an escape sequence inside two ' symbols. The approach for string_constant is similar, but with multiple entries allowed inside "

For single line comments, the pattern is just any line that startswith //.

For multi-line comments, the program uses the ability of flex to have states of the lexer itself. The program defines the start and end of multi-line comments and those act as entry and exit for the COMMENT state. In that state, everything just keeps getting added to the lexemes discovered. Thus on discovering a multi-line comment start, state is changed to COMMENT and *yymore()* upon parsing, which keeps adding to the lexeme. Once the end pattern is found, state is changed back, and parsing of other tokens continues.

```
multi_comment_start \/\*
multi_comment_end \*\/

%%

{multi_comment_start}          BEGIN(COMMENT);

<COMMENT>[^*\n]* {yymore();}
<COMMENT>\*+[^*/\n]* {yymore();}
<COMMENT>\n {yymore();}
<COMMENT>{multi_comment_end}       BEGIN(INITIAL);
```

The program gives desired output whenever some lexeme is found, and increase the count for the corresponding entity, which are kept as global variables.

All these counts along with the total lexeme count is displayed at the end.

1.5 Results

```
-> lex miniC.l
-> gcc lex.yy.c
-> ./a.out <input filename>
# output is put into a file : output.txt in the current folder
```

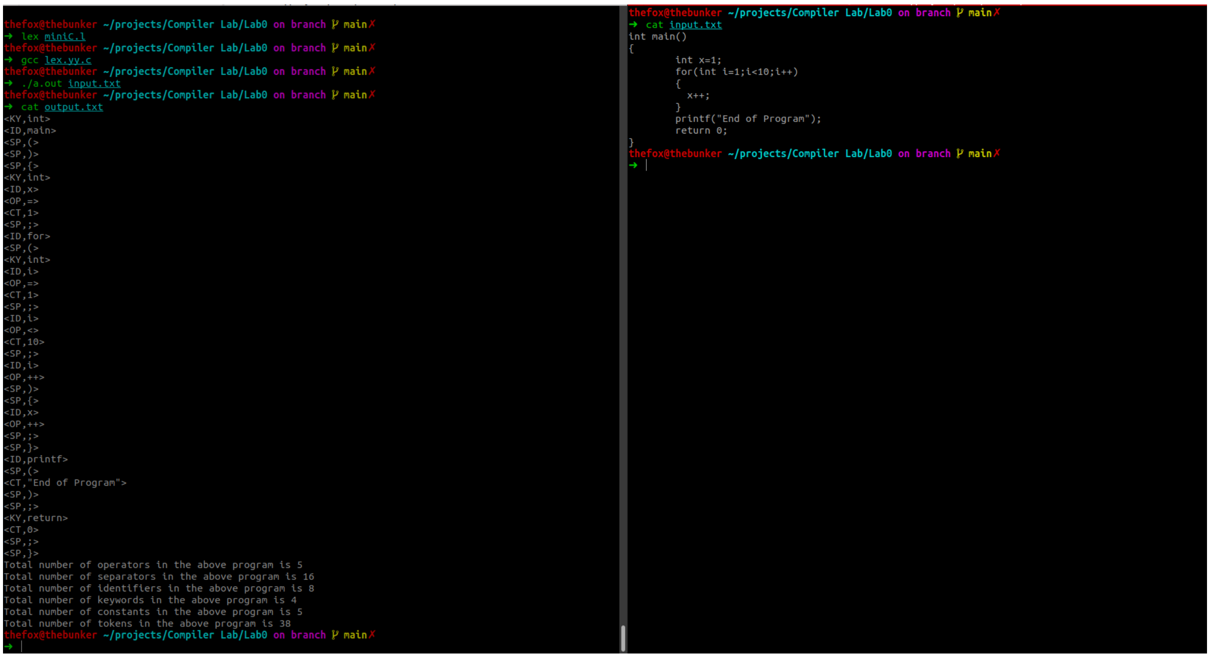

Figure 1: Results of the program for Lab0

Lab 1

2.1 Problem Statement

Write a program to identify tokens available in C/C++

2.2 Resources

- C
- flex

2.3 Files

- lab1.l - lex file with definitions for various kinds of tokens
- test.txt - sample input

2.4 Implementation Details

The program *lab1.lex* contains the required regular expressions used for identifying the tokens in an input file. These include using regular definitions for digit, normal characters, normal escape sequences. Keywords are specified by simply using an OR based expression of the keywords specified in the task expression. A similar method is used for operator and special symbols. For identifiers the program uses a character ([a-zA-Z_]), followed by 0 or more(upto 30)characters or digits. For integer constants, a digit followed by 0 or more digits. Character and string constants use ' and " as delimiters with any byte except these delimiters and escape sequences allowed inside them.

yyout for the program is set to NULL, so that lex itself does not output anything to the screen, and the actions for when a regular expression successfully matches is the one responsible for output. The program takes input from the user and will also work for files using pipes. The

2.5 Results

```
-> lex lab1.l
-> gcc lex.yy.c
-> ./a.out < <input filename>
```

Figure 2: Results of the program for Lab1

Lab 2

3.1 Problem Statement

Write a C program to recognize strings under 'a*', 'a*b+', 'abb'

3.2 Resources

- C

3.3 Files

- lab2.c - C code for the program

3.4 Implementation Details

For getting use input, the program reads into a buffer using the **fgets** function. This function returns a boolean indicating if anything was input or not. This return value is used to indicate when the program should stop execution.

The required regular expression patterns are stored in a global array. These are placed in between ' ^ ' (start of line) and '$'(end of line) . This is done as the regex engine of the C standard library will match any matching substring of the input, if these are not supplied.

The *main* function of the program then compiles these patterns into a *regex_t* structure by using the **regcomp** function. It uses the *REG_EXTENDED* flag to allow the use of extended POSIX regular expression syntax (needed in our case for the '+' specifier in the second regular expression). Then the program calls the aforementioned function to take input, exiting if none was found. After that it runs the input against all of the previously compiled regex patterns one by one, checking if a match was found, using the **regexec** function. If a match is found, a success message is printed specifying that the input was matched under which rule.

After the input loop is done, the compiled regular expressions are freed using the **regfree** function for cleanup.

3.5 Results

```
-> gcc lab2.c
-> ./a.out
```

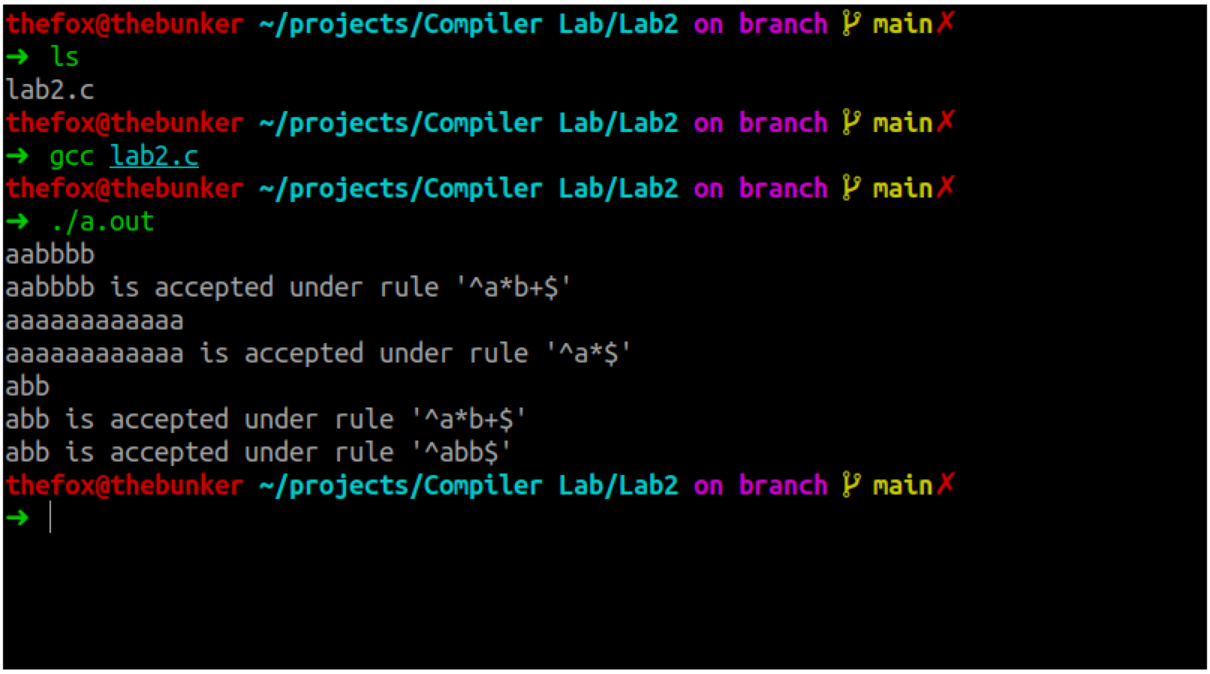

Figure 3: Results of the program for Lab2

Lab 3

4.1 Problem Statement

Write a C program to find first for non-terminals of any input grammar.

4.2 Resources

- C++

4.3 Files

- lab3.cpp - C++ code for the program

4.4 Implementation Details

Convention:

- Uppercase character are non-terminals
- Other chacters except '|' are terminals
- '|' is used to denote the start of a new body of the production for the same non-terminal
- ϵ is denoted as a space character (' ') in the first set. For productions of the form 'A $->$ ϵ', nothing is stored in the body of the production.

Productions are stored as a vector of characters, with the first character being the non-terminal on the RHS of the production rule and the remaining character to be the body of the production.

In order to parse the input grammar, a separate function is used. This consumes input line by line, then checks if the first character is uppercase or not, errors and exits in that case. If this check passes, the first character is placed into the vector. Then the first occurrence of '$->$' is found, marking the start of the body. Then the string is split into 2 parts if such an occurrence is found. Then the second part is iterated over, and all character are added to the vector until a '|' is found. At that point, the vector is added to the list of production rules. A new vector with the previously found non-character is created, which consumes the part after the '|', and so on till the end of the line. This is done until the number of productions specified by the user is reached.

Once the rules are parsed, the first sets are calculated for all the symbols. The algorithm for that is:

Algorithm 1 Calculation of FIRST Sets

Require: G, the input grammar
  function GET_FIRST(G)
    Initialize $FIRST(s) = \{\}$, for all $s \in N(non - terminals\ in\ grammar\ G)$
    Initialize $FIRST(s) = \{s\}$, for all $s \in T(terminals\ in\ grammar\ G)$
    repeat
      for each production $P- > X_1X_2...X_n \in G$ **do**
        $idx \leftarrow 0$
        repeat
          $idx \leftarrow idx\ +\ 1$
          $FIRST(P) \leftarrow FIRST(P)\ \bigcup \{FIRST(X_i)\ -\ \{\epsilon\}\}$
        until $\epsilon \in FIRST(X_i)$ **AND** $idx < n$
        if $idx\ ==\ n$ **AND** $\epsilon \in FIRST(X_n)$ **then**
          $FIRST(P) \leftarrow FIRST(P)\ \bigcup \{\epsilon\}$
        end if
      end for
    until $FIRST$ sets are changing
    return $FIRST$
  end function

For the $FIRST$ sets, the data structure used is a map from character to a set of characters.

4.5 Results

```
-> g++ lab3.cpp
-> ./a.out
```

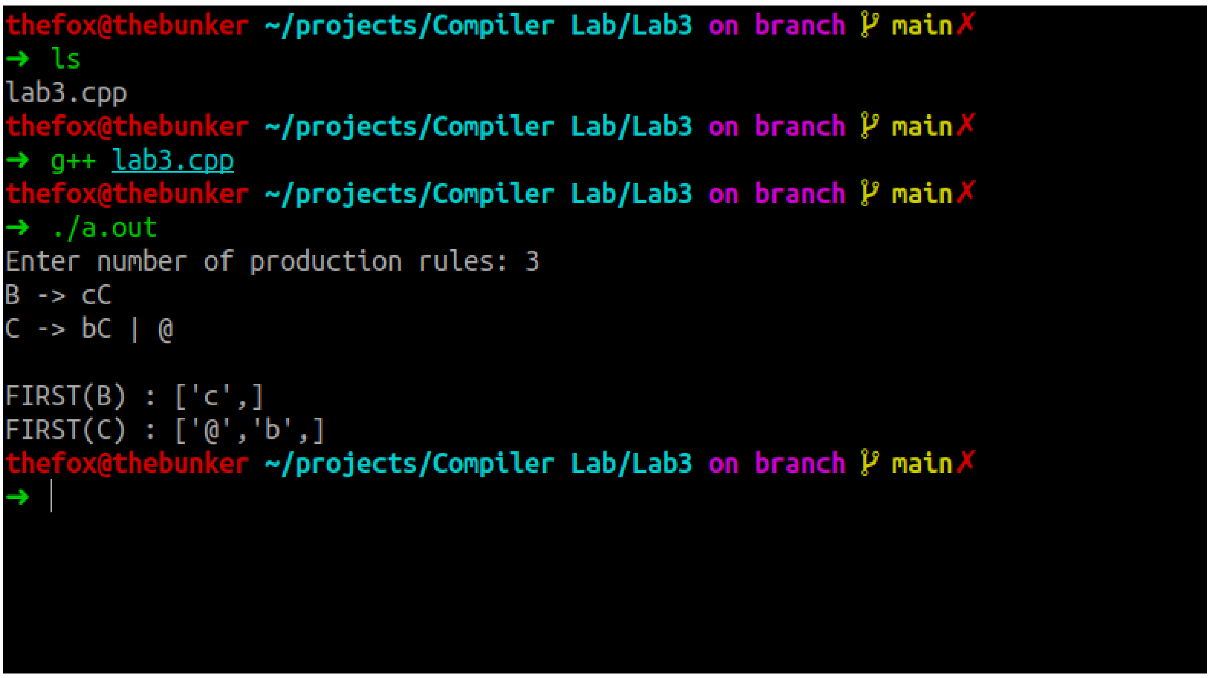

Figure 4: Results of the program for Lab3

Lab 4

5.1 Problem Statement

Write a C program for constructing of LL(1) parsing.

5.2 Resources

- C++

5.3 Files

- lab4.cpp - C++ code for the program

5.4 Implementation Details

This program uses the following grammar:

```
S -> TB
B -> +TB | ϵ
T -> FC
C -> *FC | ϵ
F -> (E) | i
```

In order to produce an LL(1) parsing table, the *FIRST* and *FOLLOW* sets are calculated first.

The calculation of the *FIRST* sets is same as discussed in the previous section[1].

The data structure for *FOLLOW* sets is the same as the *FIRST* sets, i.e. a map from character to set of characters. The algorithm for it's computation is:

Algorithm 2 Calculation of FOLLOW Sets

Require:
  G, the input grammar
  $FIRST$, the FIRST sets
  function GET_FOLLOW($G,\ FIRST$)
    Initialize $FOLLOW(s) = \{\}$, for all $s \in T \bigcup N\ in\ G$
    Initialize $FOLLOW(S) = \{\$\}$, for start symbol S in G
    repeat
      for each production $P- > X_1X_2...X_n \in G$ **do**
        $FOLLOW(X_n) \leftarrow FOLLOW(X_n) \bigcup FOLLOW(P)$
        $REST \leftarrow FOLLOW(P)$
        for i **in** n **to** 2 **do**
          $FOLLOW(X_{i-1}) \leftarrow FOLLOW(X_{i-1}) \bigcup \{FIRST(X_i)\ -\ \{\epsilon\}\}$
          if $\epsilon \in FIRST(X_i)$ **then**
            $FOLLOW(X_{i-1}) \leftarrow FOLLOW(X_{i-1}) \bigcup REST$
          end if
          $REST \leftarrow FOLLOW(X_{i-1})$
        end for
      end for
    until $FOLLOW$ sets are changing
    return $FOLLOW$
  end function

Once, these two are available, the construction of parsing table is done. This parsing table is defined as a map from a pair of character and another character, to an index (integer) in the grammar (vector of production rules).

The algorithm for that is:

Algorithm 3 Calculation of Parsing Table

Require:
  G, the input grammar
  $FIRST$, the FIRST sets
  $FOLLOW$, the FOLLOW sets
  function GET_TABLE($G,\ FIRST,\ FOLLOW$)
    $TABLE \leftarrow \{\}$
    for each production $P- > X_1X_2...X_n \in G$ **do**
      $is\_nullable \leftarrow true$
      $i \leftarrow 1$
      while $i\ <\ n$ **AND** $is\_nullable$ **do**
        $TABLE(P,\ s) \leftarrow i\quad \forall\ s\ \in \{FIRST(X_i)\ -\ \{\epsilon\}\}$
        $is\_nullable \leftarrow \epsilon \in FIRST(X_i)$
      end while
      if $is\_nullable$ **AND** $i\ ==\ n$ **then**
        $TABLE(P,\ s) \leftarrow i\quad \forall\ s\ \in FOLLOW(P)$
      end if
    end for
    return $TABLE$
  end function

Once the table is calculated, the parsing is done as follow:

Algorithm 4 Algorithm for LL(1) parsing

Require:
G, the input grammar
$TABLE$, the LL(1) parsing table
$INPUT$, the user input string to be parsed

$stack \leftarrow \{\$S\}$, where S is the start symbol of the Grammar G
while $stack \neq \phi$ **OR** $INPUT \neq \phi$ **do**
  if $stack.top \in T$ **AND** $stack.top\ ==\ INPUT.front$ **then**
    $stack.pop()$
    $INPUT \leftarrow INPUT[1:]$
  else if $TABLE(stack.top, INPUT.front) \neq \phi$ **then**
    $stack.pop()$
    $stack.push(TABLE(stack.top, INPUT.front))$
  else
    return parsing error
  end if
  if $stack\ ==\ \phi$ **AND** $INPUT\ ==\ \phi$ **then**
    return success
  end if
  return parsing error
end while

5.5 Results

Compiling and running the code:

```
-> g++ lab4.cpp
-> ./a.out
```

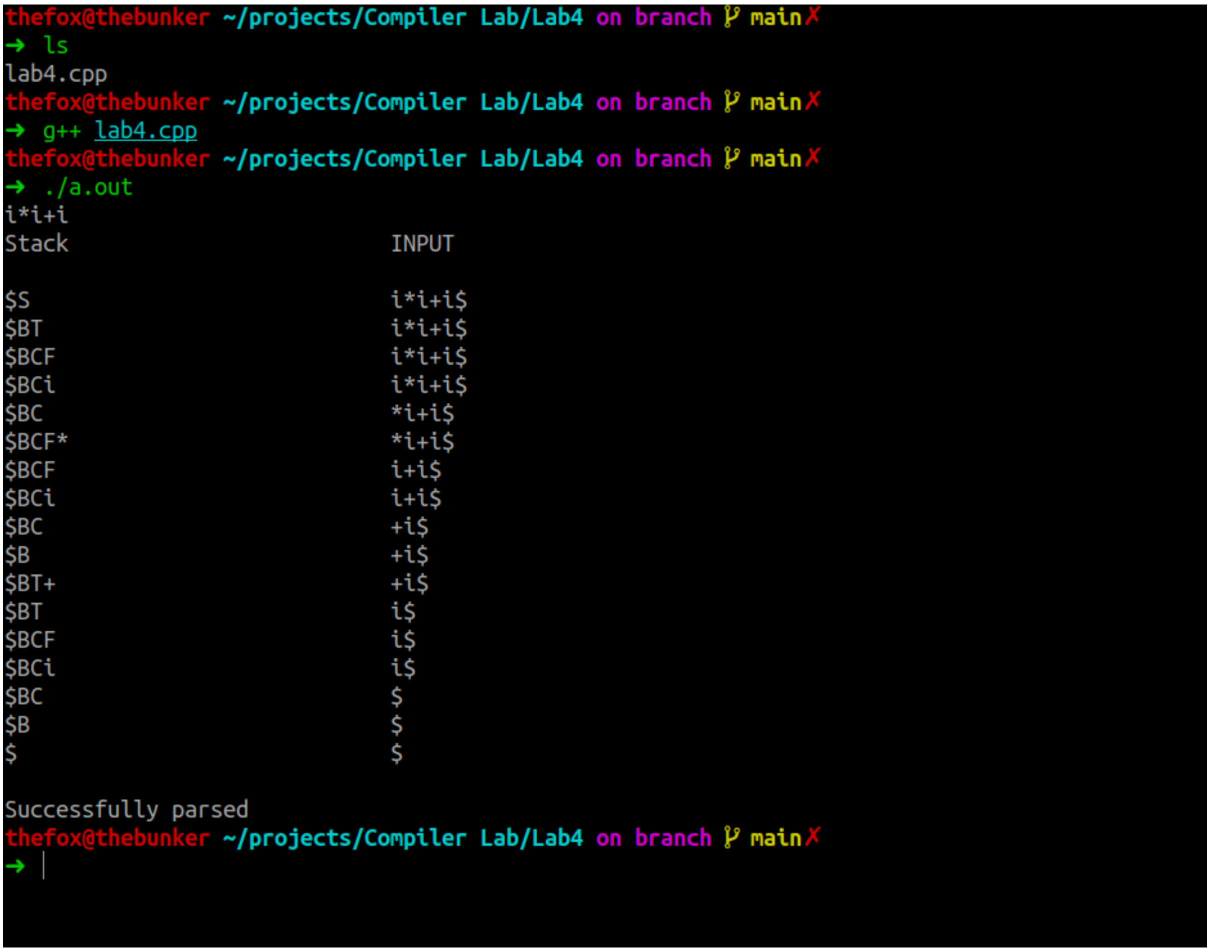

Figure 5: Results of the program for Lab4

Lab 5

6.1 Problem Statement

Write a C program to identify LR(0) cannonical items and produce LR(1) (SLR) parsing table.

E − > E + T | T

T − > T * F | F

F − > (E) | id

Two levels of output:

1. LR(0) set of items
2. Parsing Table

6.2 Resources

- C++

6.3 Files

- lab5.cpp - C++ code for the program

6.4 Implementation Details

In order to create the LR(0) items and the SLR(1) parsing table, firstly the FIRST and FOLLOW sets are needed which were explained previously[1 and 2]. The data structure used is also the same i.e. a map from character to set of characters.

For the parsing table, the **closure** function and **GOTO** function are needed. The Itemsets are stored in the form of a set of pairs. The first thing in the pair is a vector, which represents all symbols before the '.'. The second part is a list which represents all symbols after the '.'. The algorithm for *closure* and *GOTO* are:

Algorithm 5 closure function

Require:
  G, the input grammar
  set_of_items, the initial set of items
  function CLOSURE($set\_of\_items,\ G$)
    repeat
      for each item $A- > \alpha.B\beta \in set\_of\_items$ **do**
        if $B \in N$ **then**
          for each production rule $B- > \gamma \in G$ **do**
            $set\_of\_items \leftarrow set\_of\_items \bigcup \{B- > .\gamma\}$
          end for
        end if
      end for
    until set_of_items is changing
    return set_of_items
  end function

For caching purposes, the program also stores all found GOTO combinations in a map from a pair of ItemSet and character to another ItemSet.

Algorithm 6 GOTO function

Require:
  G, the input grammar
  set_of_items, an items sets
  $gotos$, the map of GOTO results found till now
  s, a symbol $s \in T \bigcup N$
  function CLOSURE($G,\ set\_of\_items,\ gotos,\ s$)
    $goto\_set \leftarrow \{\}$
    for each item $A- > \alpha.B\beta \in set\_of\_items$ **do**
      if $B\ ==\ s$ **then**
        $goto\_set \leftarrow goto\_set \bigcup \{A- > \alpha B.\beta\}$
      end if
    end for
    if $goto\_set \neq \phi$ **then**
      $gotos(set\_of\_items,\ s) \leftarrow closure(goto\_set)$
    end if
    return $closure(goto\_set)$
  end function

The construction of LR(0) states uses both the above mentioned functions. LR(0) states are a set of items. All the states are represented in a vector of itemsets. In the actual implementation both a set and a vector are needed. The set is used to check if an ItemSet was already found and the vector is used as the final storage mechanism, since indices are needed in the parsing table construction. The algorithm is:

Algorithm 7 calculation of LR(0) items

Require:

G, the input grammar
$gotos$, the map of GOTO results found till now
function GET_ITEMS($G,\ goto\_set$)
    $items\_set \leftarrow closure(\{S->.E\}, G)$
    repeat
        for each item $I \in items\_set$ **do**
            for each symbols $s \in T \bigcup N\ in\ G$ **do**
                $items\_set \leftarrow items\_set \bigcup GOTO(G,\ I,\ gotos,\ s)$
            end for
        end for
    until $items\_set$ is changing
    return $items\_set$
end function

Once, the ItemSets are calculated, the algorithm for creating the parsing table is performed:

Algorithm 8 calculation of SLR(1) parsing table

Require:

G, the input grammar
$items$, the ItemSets
$FOLLOW$, the FOLLOW sets
$gotos$, the map of GOTO results found till now

$table \leftarrow \{\}$
for i in 0 to $(\#items\ -\ 1)$ **do**
    for each production $P_j->X \in G$ **do**
        if $P_j->X. \in items[i]$ **then**
            $table(i,s) \leftarrow$ "$Reduce\ i$" $\forall s \in FOLLOW(P_j)$
        end if
        if $S->E.\ \in\ items[i]$ **then**
            $table(i,\$) \leftarrow$ "$Accept$"
        end if
    end for
    for j in 0 to $(\#items\ -\ 1)$ **do**
        for each $s \in T\ in\ G$ **do**
            if $gotos(items[i],\ s)\ ==\ items[j]$ **then**
                $table(i,s) \leftarrow$ "$Shift\ j$"
            end if
        end for
        for each $s \in N\ in\ G$ **do**
            if $gotos(items[i],\ s)\ ==\ items[j]$ **then**
                $table(i,s) \leftarrow$ "$Goto\ j$"
            end if
        end for
    end for
end for

Here $\#N$ represents the length of N.

6.5 Results

Compiling and running the code:

```
-> g++ lab5.cpp
-> ./a.out
```

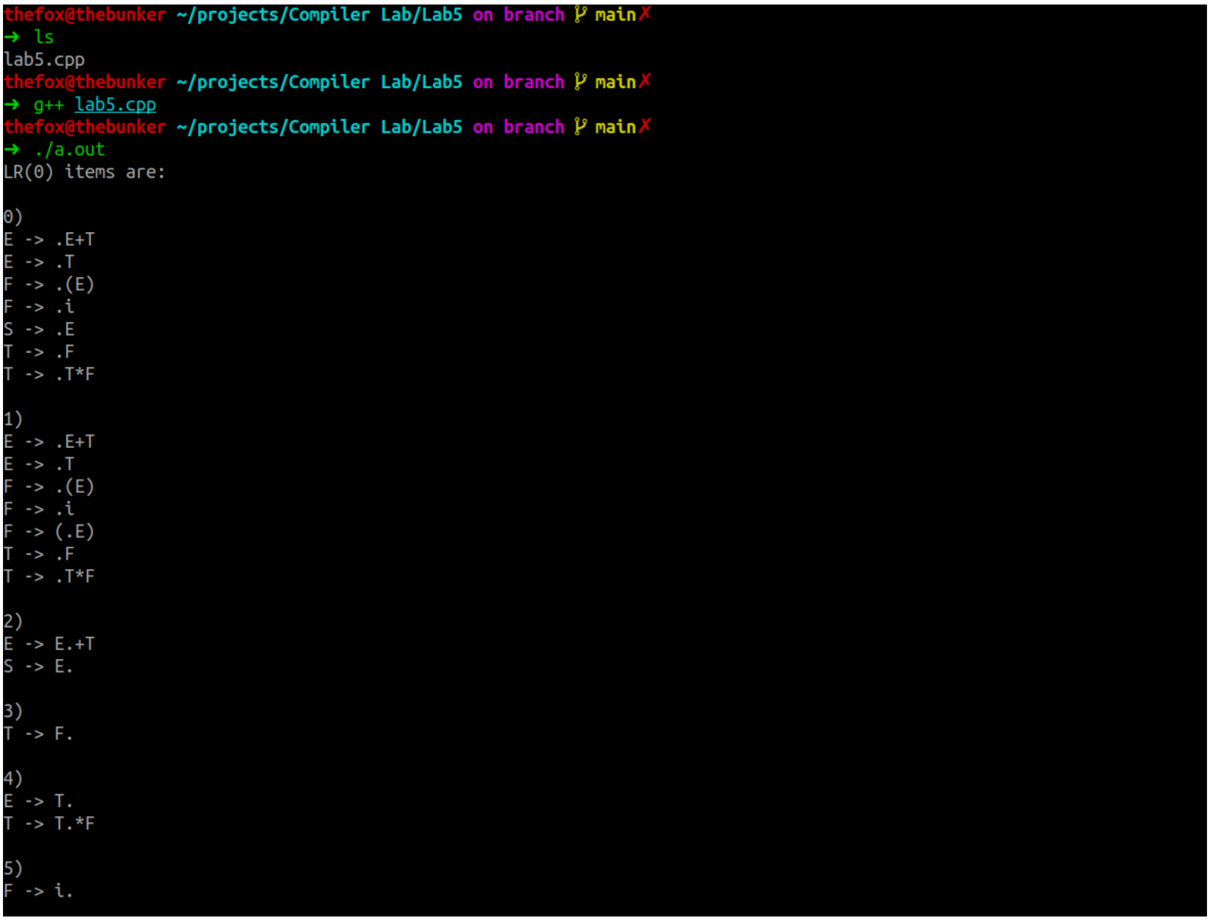

Figure 6: Results of the program for Lab5 - 1

```
6)
E -> E+.T
F -> .(E)
F -> .i
T -> .F
T -> .T*F

7)
E -> E+T.
T -> T.*F

8)
F -> .(E)
F -> .i
T -> T*.F

9)
T -> T*F.

10)
E -> E.+T
F -> (E.)

11)
F -> (E).

   State|         i|         )|         (|         *|         +|         $|         E|         T|         F
-------------------------------------------------------------------------------------------------------------
       0|       S-5|          |       S-1|          |          |          |         2|         4|         3
       1|       S-5|          |       S-1|          |          |          |        10|         4|         3
       2|          |          |          |          |       S-6|    Accept|          |          |
       3|          |       R-4|          |       R-4|       R-4|       R-4|          |          |
       4|          |       R-2|          |       S-8|       R-2|       R-2|          |          |
       5|          |       R-6|          |       R-6|       R-6|       R-6|          |          |
       6|       S-5|          |       S-1|          |          |          |          |         7|         3
       7|          |       R-1|          |       S-8|       R-1|       R-1|          |          |
       8|       S-5|          |       S-1|          |          |          |          |          |         9
       9|          |       R-3|          |       R-3|       R-3|       R-3|          |          |
      10|          |      S-11|          |          |       S-6|          |          |          |
      11|          |       R-5|          |       R-5|       R-5|       R-5|          |          |
thefox@thebunker ~/projects/Compiler Lab/Lab5 on branch main ✗
➜ |
```

Figure 7: Results of the program for Lab5 - 2