Spring Semester 2020-2021

A Report for CSN-362 (Compiler Laboratory)

Submitted by

Ishan Pandey (18113061)

ipandey@cs.iitr.ac.in

Department of Computer Science and Engineering

Indian Institute of Technology (IIT) Roorkee

May 31, 2021

Contents

1 Assignment-0

1.1 Question

Write a lex program that take as input the code of miniC language and produce output as follows:
Input: int a = 0
Output:
<Token, Lexim >
<KY, int >
<ID, a >
<OP, = >
<CT, 0 >
<SP, ; >
Total number of tokens in the above program is 5.

1.2 Theory

The very first module of a compiler is lexical analyzer, which identify valid lexemes in the input code, and assign to them valid tokens of the corresponding language. The main job of the lexical analyzer is to group a sequence of characters into one lexeme and assign to it a valid token of the language. It also removes whitespaces and comments, which are not required for the compiler to convert into machine code. It interacts with the symbol table, entering each valid lexeme found along with its token name and token attribute. It interacts with Syntax Analyzer (second module of compiler) by serving it next lexeme and token continuously until input program ends.

1.3 Implementation Details

The tokens for each pattern is described in the question. So in this problem, whenever there is a matching of any lexeme with a pattern, the corresponding short-form of that taken and the lexeme found are printed in the `output.txt` file. We are also having count of each type of token and increment the corresponding count by 1 whenever the token is found. So finally we print out the sum of all the tokens types which essentially gives all the tokens found in that program. White spaces and other indentations are also handled in the code. When some indentation is encountered (like newline, white-space, etc), the indentation variable is incremented and no particular action is taken for that.

1.4 Instructions to run the code

- Using terminal, navigate to the L0 directory .
- The actual program implemented here is the `l0.l`. Run the command `lex l0.l` which will convert this lex program to C code with name `lex.yy.c`
- Compile the C code using the command `gcc lex.yy.c`.
- Before running the executable, make sure that your input is present in the file named `Input.txt`.
- Now run the executable using the command `./a.out`.

- The corresponding output can be seen in the file named `Output.txt`.

1.5 Results

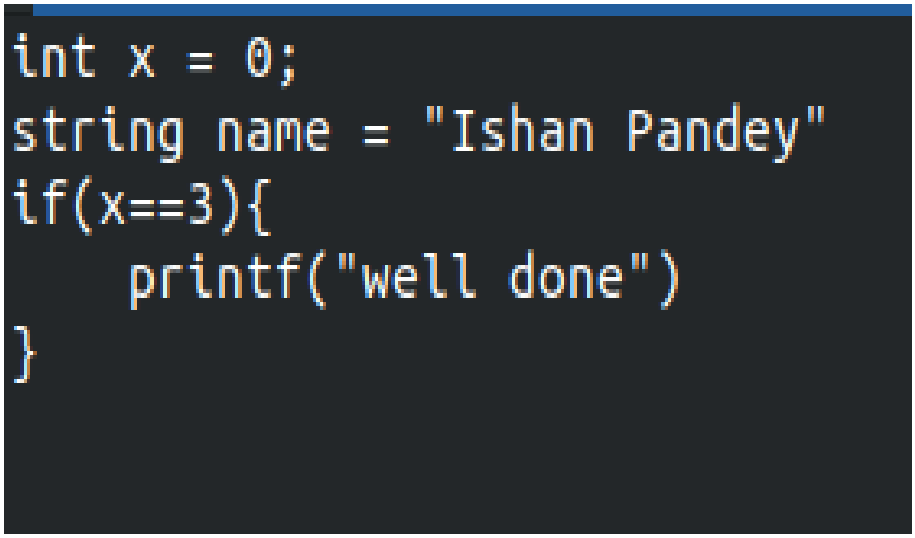


Figure 1: input file for L0

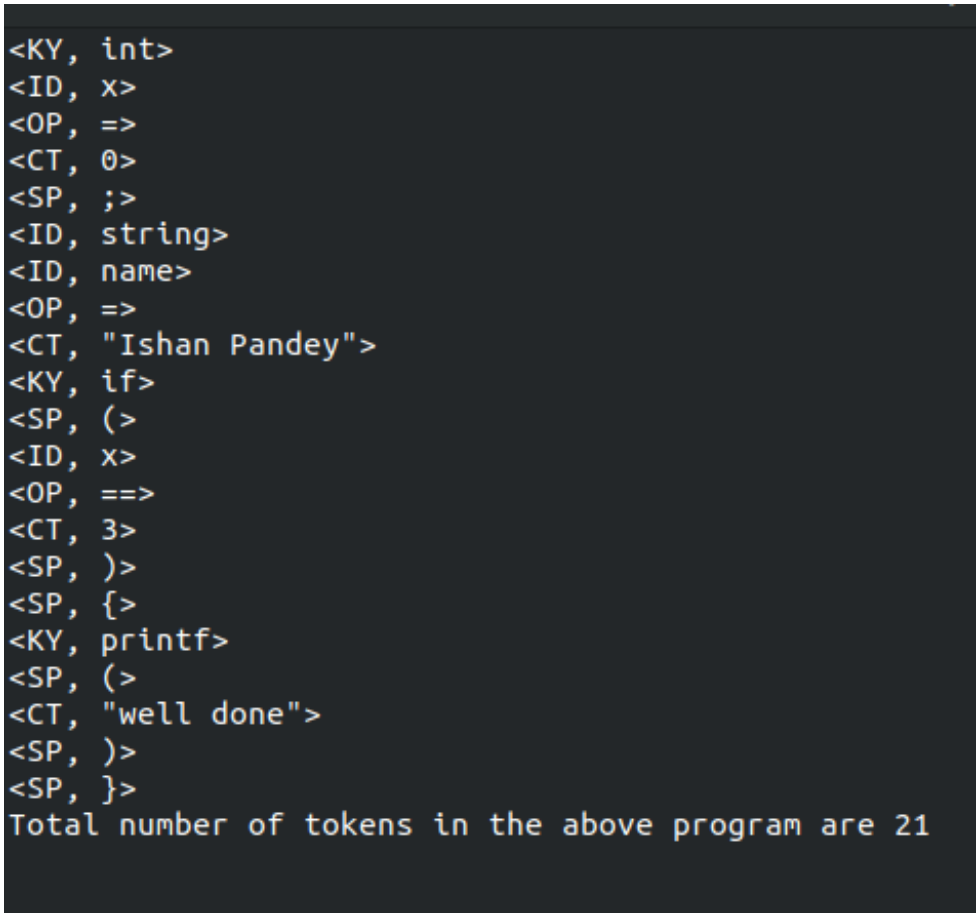


Figure 2: Output file for L0

2 Assignment-1

2.1 Question

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

2.2 Theory

The very first module of a compiler is lexical analyzer, which identify valid lexemes in the input code, and assign to them valid tokens of the corresponding language. The main job of the lexical analyzer is to group a sequence of characters into one lexeme and assign to it a valid token of the language. It also removes whitespaces and comments, which are not required for the compiler to convert into machine code. It interacts with the symbol table, entering each valid lexeme found along with its token name and token attribute. It interacts with Syntax Analyzer (second module of compiler) by serving it next lexeme and token continuously until input program ends.

2.3 Implementation Details

Lexical programs can be made easily using lex code. We are required to write the pattern corresponding to a given token and whenever a lexeme matching that pattern is found in the input, the corresponding action for that token is executed.

In our problem, we are required to write into the output file the corresponding name of the token which is found in the input code for a given lexeme. So we simply write that name in the output file. And when a newline character is encountered in the input code, newline character is written in the output as well. This generates an output file, which has same structure like input code, with space between each consecutive token (corresponding to the lexeme found in the input), and newline character corresponding to each newline character in the input code.

2.4 Instructions to run the code

- Using terminal, navigate to the L1 directory .
- The actual program implemented here is the `l1.l`. Run the command `lex l1.l` which will convert this lex program to C code with name `lex.yy.c`
- Compile the C code using the command `gcc lex.yy.c`.
- Before running the executable, make sure that your input is present in the file named `Input.txt`.
- Now run the executable using the command `./a.out`.
- The corresponding output can be seen in the file named `Output.txt`.

2.5 Results

Input.txt

```
int main()
{
int x=1;
for(int i=1;i<10;i++)
{
x++;
}
printf("End of Program");
return 0;
}
```

Ouput.txt

```
keyword identifier special_character special_character
special_character
keyword identifier special_character constant special_character
keyword special_character keyword identifier special_character constant
    special_character identifier operator constant
    special_character identifier operator special_character
special_character
identifier operator special_character
special_character
identifier special_character string special_character special_character
keyword constant special_character
special_character
```

Figure 3: Output of Solution 1

3 Assignment-2

3.1 Question

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

3.2 Theory

Regular Expressions(RE) are formal notation for generating patterns. Each pattern matches a set of strings, so regular expressions serve as names for a set of strings. Programming language tokens can be described by regular languages. RE basically comprises of various symbols (the alphabets of the strings) and various type of operation on these symbols like union, concatenation, kleene closure, etc. So using regular expression, we can specify a set of strings following a specific pattern in an abstract and compact way using regualr expression.

Finite automata(FA) are formal (or abstract) machines for recognizing patterns. The finite automata or finite state machine is an abstract machine which have five elements or tuple i.e finite set of states, set of input symbols, start state, set of final state and transition function for determining on which state to move upon an incoming input symbol from a given state.

Regular Expression and Finite Automata are inter-related. The language accepted by an finite automata can be easily described by Regular Expressions. It is the most effective way to represent any language. Hence, if we are asked to determine if given string is generated by a RE, we can find this by showing that the corresponding finite state machine accepts that particular string.

3.3 Implementation Details

The Regular Expression is first converted in the corresponding Deterministic Finite Automata (DFA) and then the DFA is implemented using the switch statement. This DFA has 8 states in total, out of which the state4 is not a final state, and all other are final states. In code, we have a state variable, which changes upon each input symbol. If the final state where the input ends is in the set 1, 2, 3, 5, 6, 7, 8, the input string is considered as acceptable, otherwise not.

3.4 Instructions to run the code

- Using terminal, navigate to the L2 directory .
- The actual program implemented here is the `l2.c`. Run the command `gcc l2.c` to compile this C code.
- Now run the executable using the command `./a.out`.
- The program first ask you to enter the string you want to test.
- If the string matches any of the regualr expression given in the question, that regualar expression will be printed. Otherwise, a message about the string not being accepted by any regualar expression is printed.

3.5 Results

Compiling the program with the GNU c++ compiler and running the executable and providing the string to be tested as input, we will get to know if the string is accepted or not, and also under which regular expression.

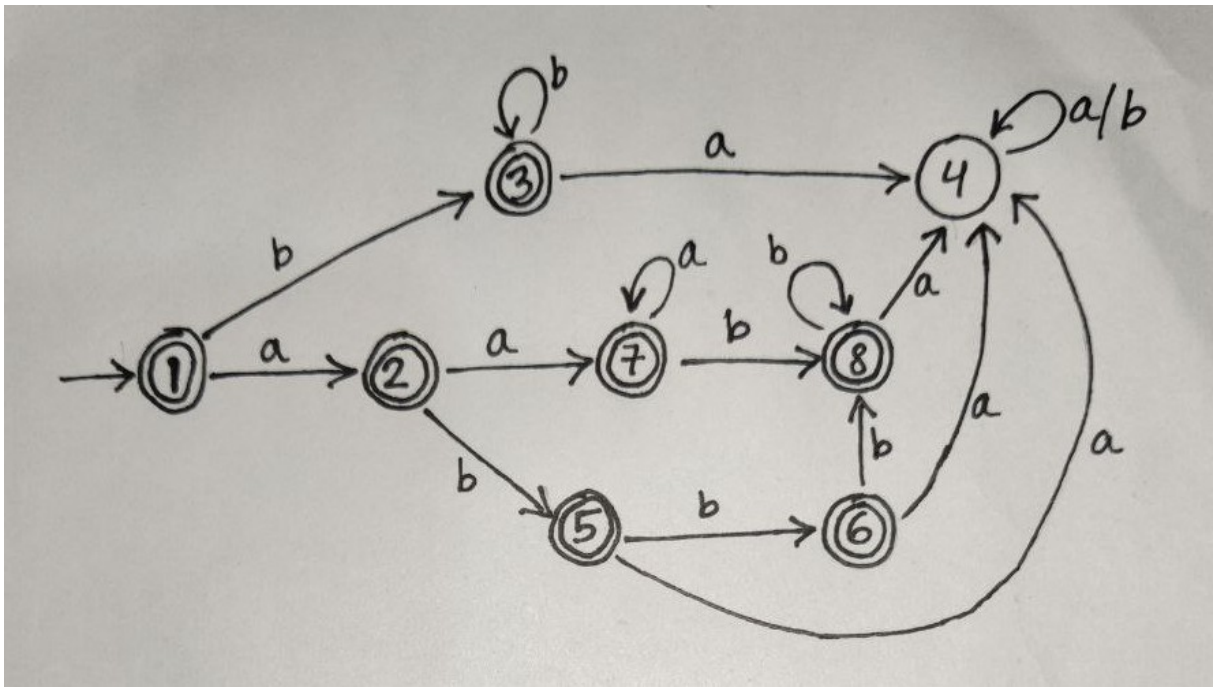


Figure 4: Combined DFA for RE in Asg2

```
ishan@ishan-Nitro-AN515-51: ~/Dev/Compiler_lab/L2
File  Edit  View  Search  Terminal  Help
(base) ishan@ishan-Nitro-AN515-51:~/Dev/Compiler_lab/L2$ ./a.out
Enter a string: aabbbb
Output: aabbbb is accepted under rule 'a*b+'
(base) ishan@ishan-Nitro-AN515-51:~/Dev/Compiler_lab/L2$ ./a.out
Enter a string: aaaaaa
Output: aaaaaa is accepted under rule 'a*'
(base) ishan@ishan-Nitro-AN515-51:~/Dev/Compiler_lab/L2$ ./a.out
Enter a string: abb
Output: abb is accepted under rule 'abb'
(base) ishan@ishan-Nitro-AN515-51:~/Dev/Compiler_lab/L2$ ./a.out
Enter a string: abbbabbabb
Output: abbbabbabb is not recognised by any pattern.
(base) ishan@ishan-Nitro-AN515-51:~/Dev/Compiler_lab/L2$ ./a.out
Enter a string: bbbaaa
Output: bbbaaa is not recognised by any pattern.
(base) ishan@ishan-Nitro-AN515-51:~/Dev/Compiler_lab/L2$
```

Figure 5: Output of Solution 2.

4 Assignment-3

4.1 Question

Write a C/C++ program to find the set FIRST of all non-terminal symbols of any input grammar.

4.2 Theory

Context free grammar is a formal grammar which is used to generate all possible strings in a given formal language. It can be defined using four components i.e a set of terminal symbols, a set of non-terminal symbols, a set of production rules and a start symbol. It differ from regular expression in the way that the regular expressions help to describe all the strings of a regular language while the context free grammar helps to define all possible strings of a context free language.

FIRST(A) is the set of all terminals which appear at the first position of all the strings that can be derived from the production of A's. If null string can be derived from the production of A, ε is also added in the FIRST(A) set.

4.3 Implementation Details

First of all I found out all those non-terminals which can generate null strings. This can be done by first finding all the non-terminals which has a production of type "A - ε" in the production set. Then, we iterate on other productions and identify all those patterns which has the form "A - Y1.Y2.....Yn" and where Y1, Y2 ... Yn are all nullable. These terminals are then also marked as nullable.

Now to find out all terminals in the FIRST(A), we take all productions of A, and add to FIRST(A) all terminals which appear like "A - a.X". Now for production of the form "A - B.C...", we also add FIRST(B) to FIRST(A). If B is nullable, we also add FIRST(C) to FIRST(A) and so on.

4.4 Instructions to run the code

- Using terminal, navigate to the L3 directory .
- The actual program implemented here is the `l3.cpp`. Run the command `g++ l3.cpp` to compile this C++ code.
- Now run the executable using the command `./a.out`.
- The program first ask you to enter the number of productions you will enter for the grammar (5 in this case).
- Now write the productions. If a symbol has more than 1 production, you can either enter them separately or they can be separated using the pipe symbol (i.e |).
- Once you have all the productions, the program will print the FIRST of all non-terminals of that grammar.

4.5 Results

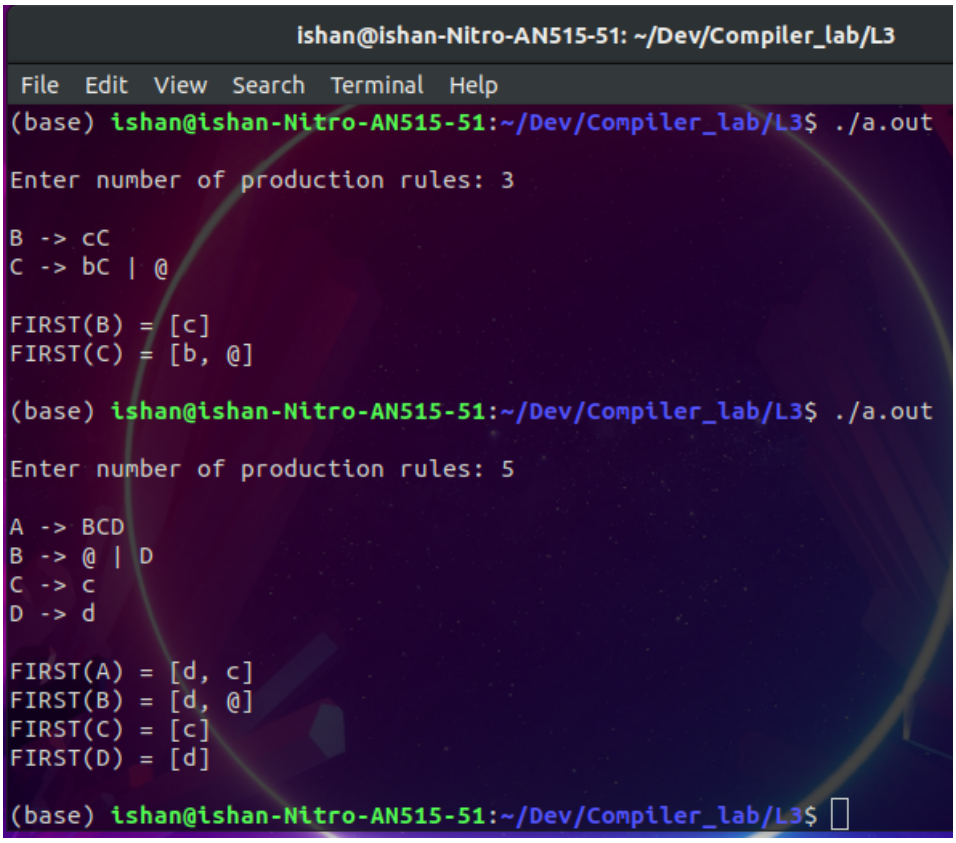


Figure 6: Output of Solution 3

5 Assignment-4

5.1 Question

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

5.2 Theory

A parser is the second part of compiler and its job is to check if the syntax of the code which the user is compiling is correct or not. There are two types of parsers which are majorly used i.e top-down and bottom-up parsers. LL(1) parsers comes under top-down parsing category.
Here the 1st L represents that the scanning of the input will be done from Left to Right manner and second L shows that in this Parsing technique we are going to use Left most Derivation Tree, and finally the 1 represents the number of look ahead, means how many symbols are you going to see when you want to make a decision.
The main step involved in making of LL(1) parsers is to make the LL(1) predictive parsing table which predicts what production to used when there is a non-terminal at the top of stack and a terminal at the top of input string. The table is constructed using FIRST and FOLLOW sets of non-terminal. All the Null Productions of the Grammars will go under the Follow elements and the remaining productions will lie under the elements of First set. Once the table is constructed, the input string is consumed step by step and relevant steps from the predictive parsing table are chosen to consume the complete input string.

5.3 Implementation Details

- First find the set FIRST and FOLLOW of every non-terminal in the grammar.
- Then build the LL(1) parsing table of form `table [non-terminal][terminal]`.
- for every production rule `non-terminal -> α`
  - add this production rule to `table[non-terminal][first(α)]`
  - if Nullable(α) = true, add rule to `table[non-terminal][follow(non-terminal)]`
- to parse the given string we construct a stack and a look-ahead pointer
  - when top of stack is non-terminal pop the top of the stack and push the string from `table[non-terminal][look-ahead-symbol]`.
  - when top of the stack is terminal and matches the look-ahead-symbol then pop the top of the stack and move the look-ahead, otherwise string doesn't matches the provided grammar.

5.4 Instructions to run the code

- Using terminal, navigate to the L4 directory .
- The actual program implemented here is the `l4.cpp`. Run the command `g++ l4.cpp` which will convert this cpp program to an executable file with name `a.exe` or `a.out` depending upon os.
- Now run the executable using the command `./a.out` or `a.exe`.
- The corresponding output can be seen on the terminal.
- The program first asks you to write the number of productions which you will enter.
- In the following line, write down the productions consecutively.

- The program first displays the FIRST, FOLLOW and the table content for the entered grammar and finally takes a input string which has to be evaluated for this parser.
- The final output is the content of stack and the input string parsed so far at each step.

5.5 Results

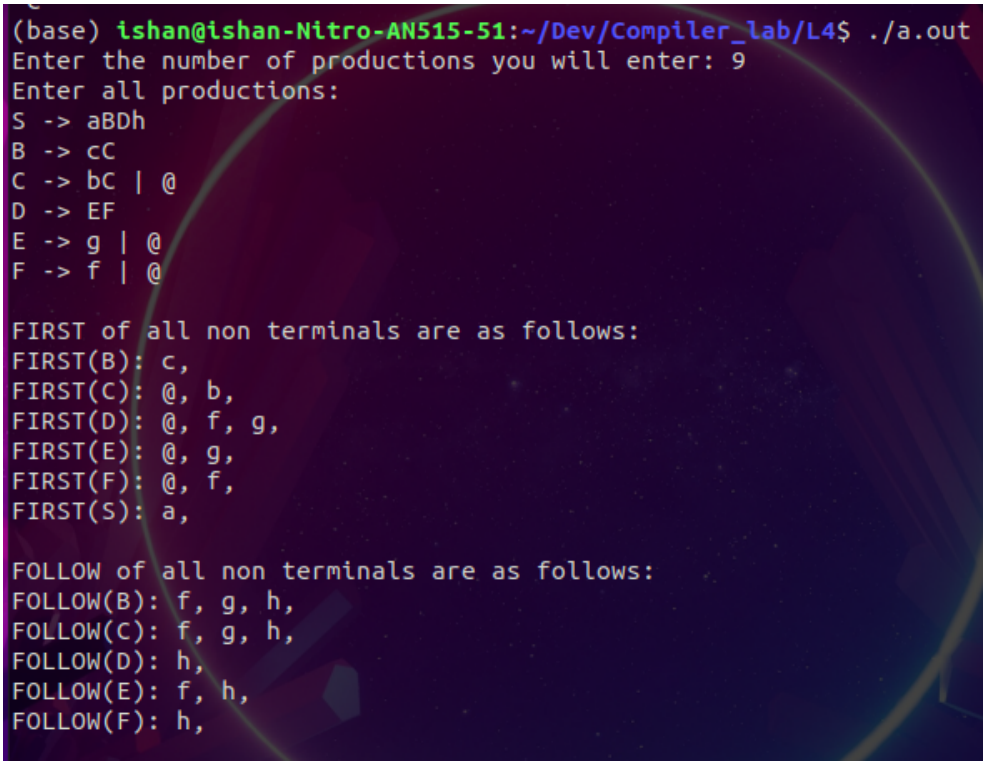


Figure 7: Output of Solution 4 : First and Follow sets

```
TABLE entries:
Non-terminal    Terminal        Production to use
B               c               cC
C               b               bC
C               f               @
C               g               @
C               h               @
D               f               EF
D               g               EF
D               h               EF
E               f               @
E               g               g
E               h               @
F               f               f
F               h               @
S               a               aBDh
```

Figure 8: Output of Solution 4 : Table Entries

```
Enter the input string to test: acbgfh

The following are the parsing steps using stack and input:
STACK           INPUT
S               acbgfh$
aBDh            acbgfh$
BDh             cbgfh$
cCDh            cbgfh$
CDh             bgfh$
bCDh            bgfh$
CDh             gfh$
Dh              gfh$
EFh             gfh$
gFh             gfh$
Fh              fh$
fh              fh$
h               h$
```

Figure 9: Output of Solution 4 : Stack

6 Assignment-5

6.1 Question

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 Theory

Bottom-up Parsing during the left-right scan of the input constructs a right most derivation in reverse. Informally, a "handle" is a substring that matches body of a production, and whose reduction represents one step along the reverse of right most derivation. The handle is unique if the grammar is unambiguous.

LR Parsing also called **Shift-Reduce Parsing** is a type of bottom up parser whose working is based on right most derivation. It's main goal is to locate the handle and reduce it. LR parser uses DFA at its core for locating handles and has its own parsing table to select production rules corresponding to a reduction. It has four basic actions :

- Shift
- Reduce
- Accept
- Error

LR(0) parser is a kind of LR parser where there is no lookahead, it has to decide upon by considering what inputs have already been seen and used.

LR(0) items of grammar G is a production of G with dot at some position of the body. Various operations are performed on LR(0) items :

- **Closure** returns set of items that serve as a state for the DFA of LR parser.
- **Fuction GOTO** takes input a state and a symbol returns a new state which is resulted from due to transition from input state upon providing the input symbol.

LR(0) Cannonical items is the set of all states (sets of items) of the DFA of the LR parser. It is calculated using closure and goto operations.

SLR parser or Simple LR(1) parser is a kind of LR parser which uses a lookahead of one symbol. To tackle shift reduce conflicts, it has a rule that it will only reduce when the next symbol is in the follow set of the head of the production corresponding to that reduction. This was not possible in case of LR(0) parser because of no lookahead.

6.3 Implementation Details

My implementation consists of three classes for CFG, state (set of items) and slr.

- **CFG class** contains the implementation for cfg grammar. I have assumed that each symbol will be a single character and epsilon will be represented by . It contains 2 arrays or vectors for terminal and non terminal symbols, start symbol and a map to store production rules.
- **State class** contains integer prod_count to store number of items, an id to identify the state, map to store items and some helper functions to print and initialize state.
- **SLR class** contains most of the implementation. It contains pointer to cfg class to store grammar, vector items to store cannonical set of items, map goto_table to implement goto function, map id_state to check repetition of states, first and follow sets, prod_no to assign a number to each production and finally map reduce to map reduce-able states to corresponding production numbers. Moreover it contains helper functions to calculate closure, goto_table, cannonical set of items, first and follow sets, and finally function to print parsing table. It also contains some utility functions like split and stringify_state which aid helper functions.

First, I declared an instance of cfg class and put all the production rules, start symbol and all the terminal and non terminal symbols in it. Then, I create an instance of slr class and pass the instance of cfg class. It automatically, initializes all the values in slr class, adds a new production @->E, creates the first state, takes its closure and stores it in items.

Next, I called the method to calculate LR(0) cannonical set of items, I've used the same algorithm taught in class, it calculates all items and also fills the goto_table and reduce map.

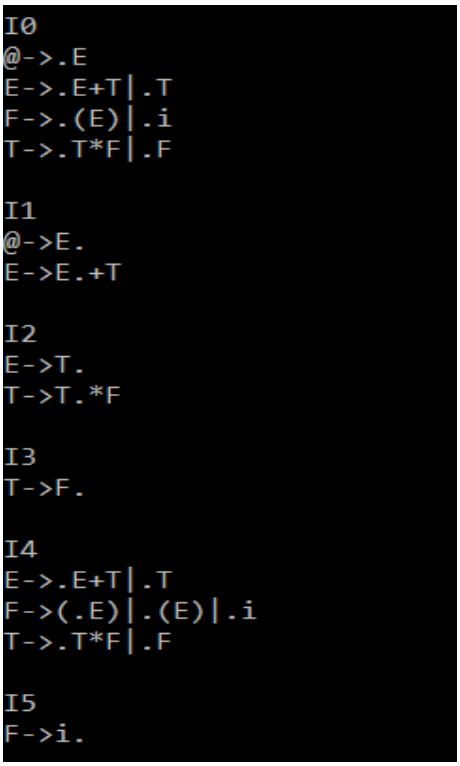
Next, I called the methods to calculate first and follow sets, again same algorithm taught in class has been used in these methods.

Lastly, parsing table is printed using all the information we have gathered up till now. Production rules numbers are also printed to improve readability of parsing table.

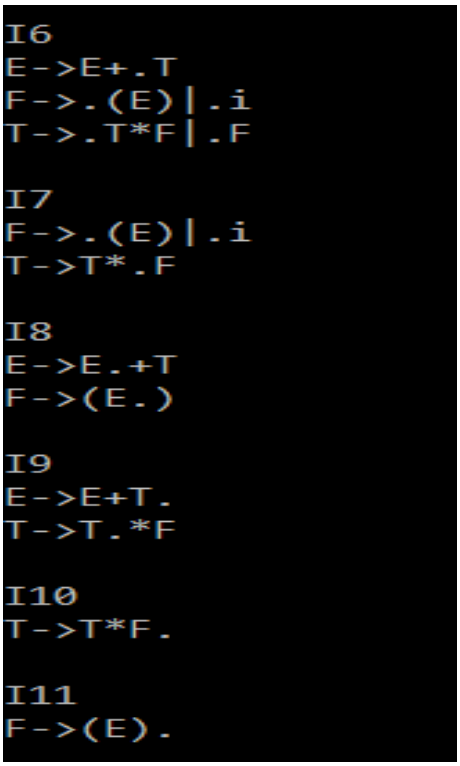
6.4 Instructions to run the code

- Using terminal, navigate to the L5 directory .
- The actual program implemented here is the `l5.cpp`. Run the command `g++ l5.cpp` which will convert this cpp program to an executable file with name `a.exe` or `a.out` depending upon os.
- Now run the executable using the command `./a.out` or `a.exe`.
- The corresponding output can be seen on the terminal.

6.5 Results



(a) LR(0) items



(b) LR(0) items

Figure 10: LR(0) cannonical set of items

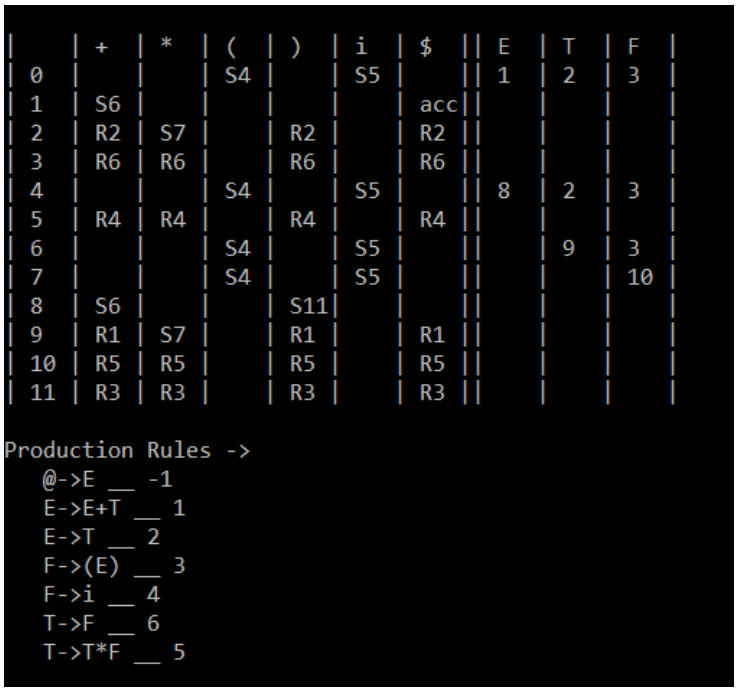


Figure 11: Output of Solution 5