A Project Report on

LEXICAL ANALYSIS

Submitted to the Department of Information Technology

For the partial fulfilment of the course System Programming Lab (IT-651) in Information Technology

by

Sneha Dhar
Rajat Kumar Agarwal
Shantanu Saurabh

Roll number: 111308037/38/45

Registration number: 110813038/39/48 of 2013-17

Under the supervision of

Dr. CHANDAN GIRI

Department of Information Technology

INDIAN INSTITUTE OF ENGINEERING SCIENCE AND TECHNOLOGY, SHIBPUR

May, 2016

Contents

1. INTRODUCTION

The purpose of a lexical analyser is to read the input string one character at a time and produces a stream of tokens as output, where a token consists of a terminal symbol along with additional information in the form of attribute values. Tokens are written as tuples enclosed between a pair of small brackets.

To implement a lexical analyser by hand, it helps to start lexeme specifications, and then create a DFA or a description for the lexemes of each token. We can then write code to identify each occurrence of each lexeme on the input and to return information about the identified token.

2. APPROACH:

1. Creation of Deterministic Finite Automata.
2. Define the regular language from the given lexical specifications.
3. Define our own language that could understand basic data types, the different keywords which we'll use as well as the identifiers, whitespaces, delimiters and literals.
4. Convert the regular expression into a non-deterministic finite automaton and then further convert it into Deterministic Finite Automaton.
5. Implement the DFA using a 2D array.

This approach is called Table driven implementation of DFA.

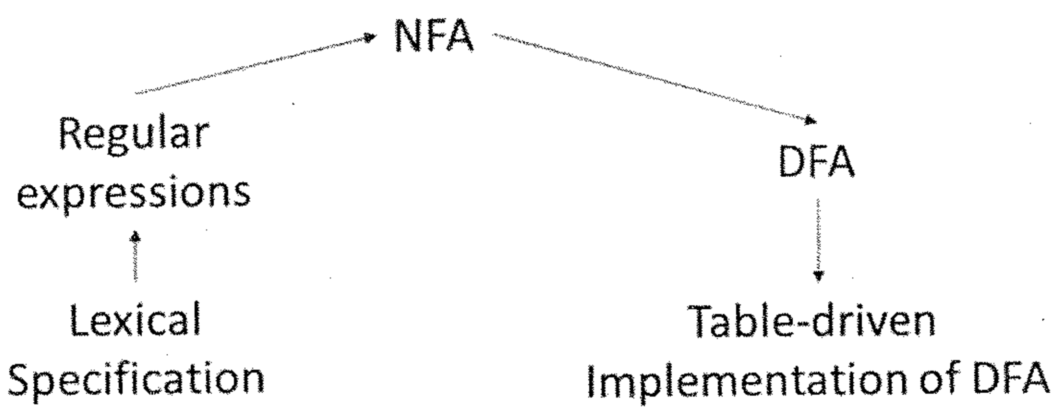


Fig 1: Steps of implementation of Lexical specifications via DFA

Following is a Regular Expression for identifying the identifiers for the language.

The rule states that:

- Token should start with a letter.
- Letter, digits and underscores are allowed
- It cannot end with an underscore.

letter → A | B | . . . | Z | a | b | . . . | z

digit → 0 | 1 | 2 | . . . | 9

und → '_'

id → letter (letter | digit | underscore)* | (letter | digit)

These are the rules for production of grammar.

Now this regular expression is converted to its equivalent NFA and further into the corresponding DFA .We show here the DFA for the following rules.

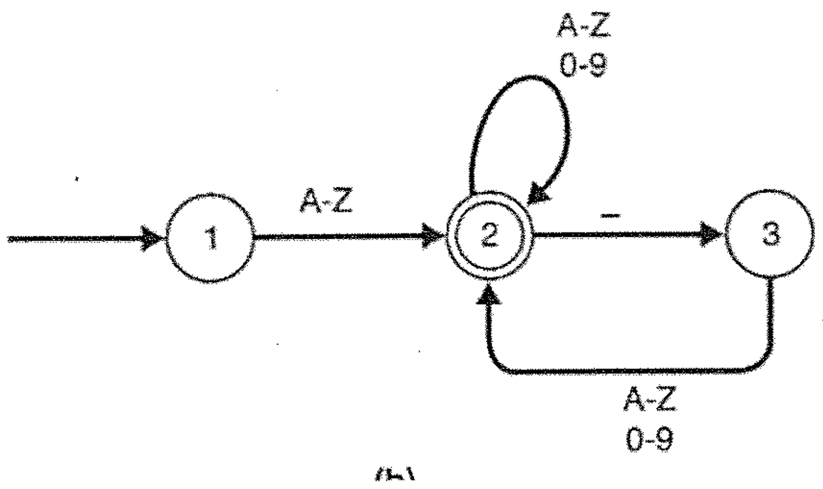


Fig 2: Equivalent DFA of above grammar.

This DFA is implemented as a table (a 2D array) where rows represent states while columns represent the different symbols of the grammar. The Entries corresponding to rows and columns show the transitions from the current state to a different state depending on the symbol encountered.

State	A-Z	0-9	-	
1	2			{starting state}
2	2	2	3	{final state}
3	2	2		

Fig 3: Table driven implementation of DFA

This is also known as Delta function.

Following the same methodology we laid some rules for the entire language and drew the equivalent DFA and created the corresponding table for the same.

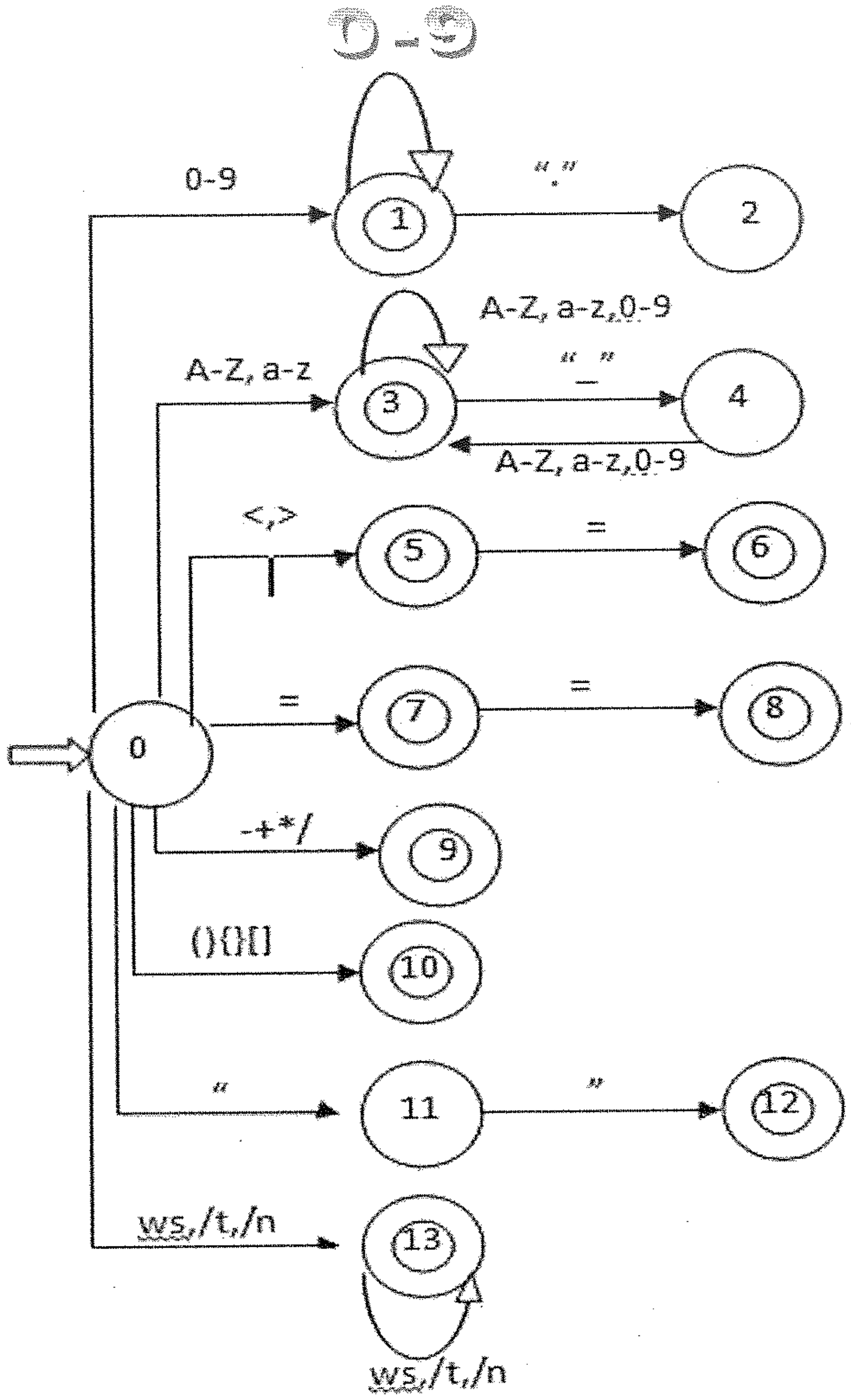


Fig 4: DFA for our programming language.

Symbols -> States	“0-9”	“.”	“a-z A-Z”	“_”	“< >”	“=”	“/*+-“	“{}[]()”	“”“”	“ /t/n”
→ 0	1		3		5	7	9	10	11	13
1*	1	2								
2	1									
3*	3		3	4						
4	3		3							
5*						6				
6*										
7*						8				
8*										
9*										
10*										
11									12	
12*										
13*										13

Table 1: DFA implementation for the lexical specifications of our programming language

3. SOURCE CODE

```
#include<stdio.h>
#include<string.h>
typedef enum{
    s_0_9=0,
    s_point=1,
    s_A_Z=2,
    s_underscore=3,
    s_rel_ops=4,
    s_equals=5,
    s_operators=6,
    s_delimiter=7,
    s_literal=8,
    s_whitespaces=9
}symbols;

int get_type(char y)
{
    int x = (int)y;
    //printf(" %d ",x);
    if(x>='0'&& x<='9') return 0;
    else if(x == '.') return 1;
    else if(x>='a' && x <='z'|| x>='A' && x <='Z')return 2;
    else if(x == 95) return 3;
    else if(x == 60||x == 62) return 4;
    else if(x == '=') return 5;
    else if(x == '+'||x == '-'||x == '*'||x == '/') return 6;
    else if(x == '{'||x =='}'||x =='('||x ==')'||x==59) return 7;
    else if(x== 34||x ==39) return 8;
    else if(x == 32||x==10||x==9) return 9;
    else return 100;
}
int symtab[14][10] = { {1,50,3,50,5,7,9,10,11,13},
                {1,2,50,50,50,50,50,50,50},
                    {1,50,50,50,50,50,50,50,50,50},
                    {3,50,3,4,50,50,50,50,50,50},
                    {3,50,3,50,50,50,50,50,50,50},
                {50,50,50,50,50,6,50,50,50,50},
                {50,50,50,50,50,50,50,50,50,50},
                {50,50,50,50,50,8,50,50,50,50},
                {50,50,50,50,50,50,50,50,50,50},
                {50,50,50,50,50,50,50,50,50,50},
                {50,50,50,50,50,50,50,50,50,50},
                {50,50,50,50,50,50,50,50,12,50},
                {50,50,50,50,50,50,50,50,50,50},
                {50,50,50,50,50,50,50,50,50,13}
 };
int is_acceptingstate(int state){
    //printf(" %d ",state);

return(state==1||state==3||state==5||state==6||state==7||state==8||s
tate==9||state==10||state==11||state==12||state==13)?1:0;
}
```

```
void findfinalstatename(char* str,int state) {

    switch(state)
    {

case 1: strcpy(str,"Constant");

return;

case 3: strcpy(str,"Identifier");

return;

case 5: strcpy(str,"Relational");

return;

case 6: strcpy(str,"Relational");

return;

case 7: strcpy(str,"Assignment");

return;

        case 8: strcpy(str,"Relational");

return;

        case 9: strcpy(str,"Operator");

return;

case 10: strcpy(str,"Delimiter");

return;

case 11: strcpy(str,"Literal");

             return;
case 12: strcpy(str, "Literal Pair");
                 return;

case 13: strcpy(str,"WhiteSpace");

return;

default: strcpy(str,"Invalid");

    }

}
```

```
int main(){

        char input[1000],temp[20],statename[20];
        gets(input);
        int state = 0, prev_state = 0,i=0,j=0,flag = 0,flag2 = 0;
        symbols type;
        while(input[i] != '\0'){
                type = get_type(input[i]);
                if(input[i] == '.'|| input[i] == '_') flag2 = 1;

                temp[j++] = input[i];

prev_state = state;
                state = symtab[state][type];

                        if(is_acceptingstate(state))      flag = 1;
                else
                {
                if(flag == 1 && flag2 ==0)
                        {
                                i=i-1;
                                temp[--j] = '\0';
                                findfinalstatename(statename,prev_state);
                                printf("<\"%s\",\"%s\">\n",temp,statename);
                                memset(statename,'\0',20);

                                flag = 0;
                                j = 0;
                                state = 0;
                                prev_state = 0;
                                memset(temp,'\0',20);
                        }
                        else if(flag == 0 && flag2 == 0)
                        {
                                printf("<\"%s\",\"%s\">\n",temp,"Invalid");

                                flag = 0;
                                j = 0;
                                state = 0;
                                prev_state = 0;
                                memset(temp,'\0',20);


                        }
                        else if(flag2 == 1){}        }
                i++;
                flag2 = 0;

        }
        findfinalstatename(statename,state);
        printf("<\"%s\",\"%s\">\n",temp,statename);


 }
```

4. EXPLANATION:

- All the possible lexmes have been defined as enumeration types so as to maintain code's simplicity.
- The function `get_type()` returns the corresponding number (enumeration) for a given character.
- If the symbol is unidentified, it returns 50 as error code.
- The DFA is implemented via `symtab[14][10]`. It contains entries specified in the Delta function (Table-1). All bank entries are filled with 50 as undefined state.
- The function `is_acceptingstate()` returns 1 if the state is accepting otherwise it returns 0.
- The function `findfinalstatename()` gives the name of accepted state(the class to which the tokens belong).

The Main function:

In our implementation we go on checking the string until we reach a non-accepting state. If non accepting state is reached then we remove the last character and display the corresponding token class to which the lexme belongs. The process continues till the end is reached.

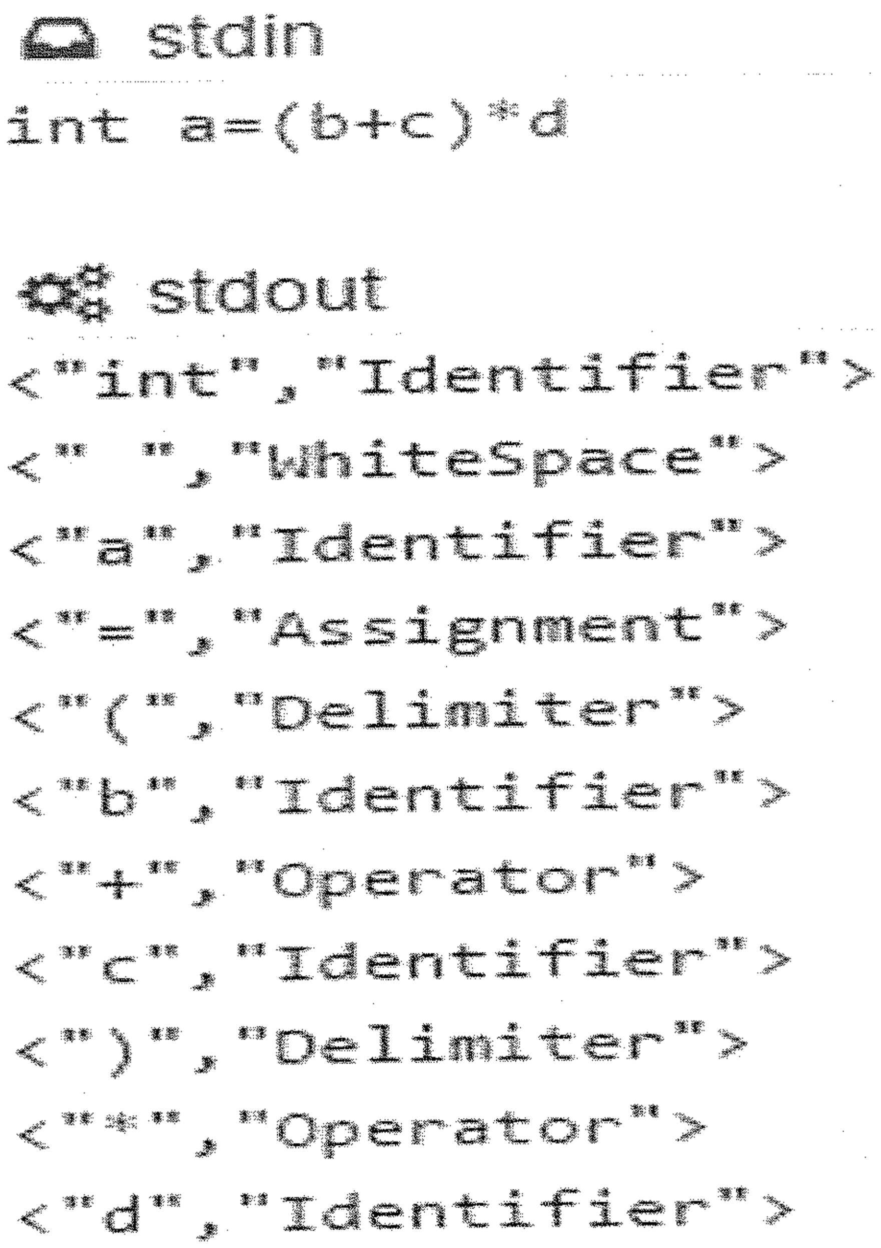
Two flags are used for this purpose. One is used to keep track if the accepted state has occurred in the last input. Other flag helps us to identify if we have seen the character '_' or '.' as these does not follow our regular implementation.

State and prev_state keeps track of which state is seen in current and previous input respectively.

Variables i, j, temp[], statename[] are temporary variables used in the program.

The input is stored in array input[].

6. OUTPUT:



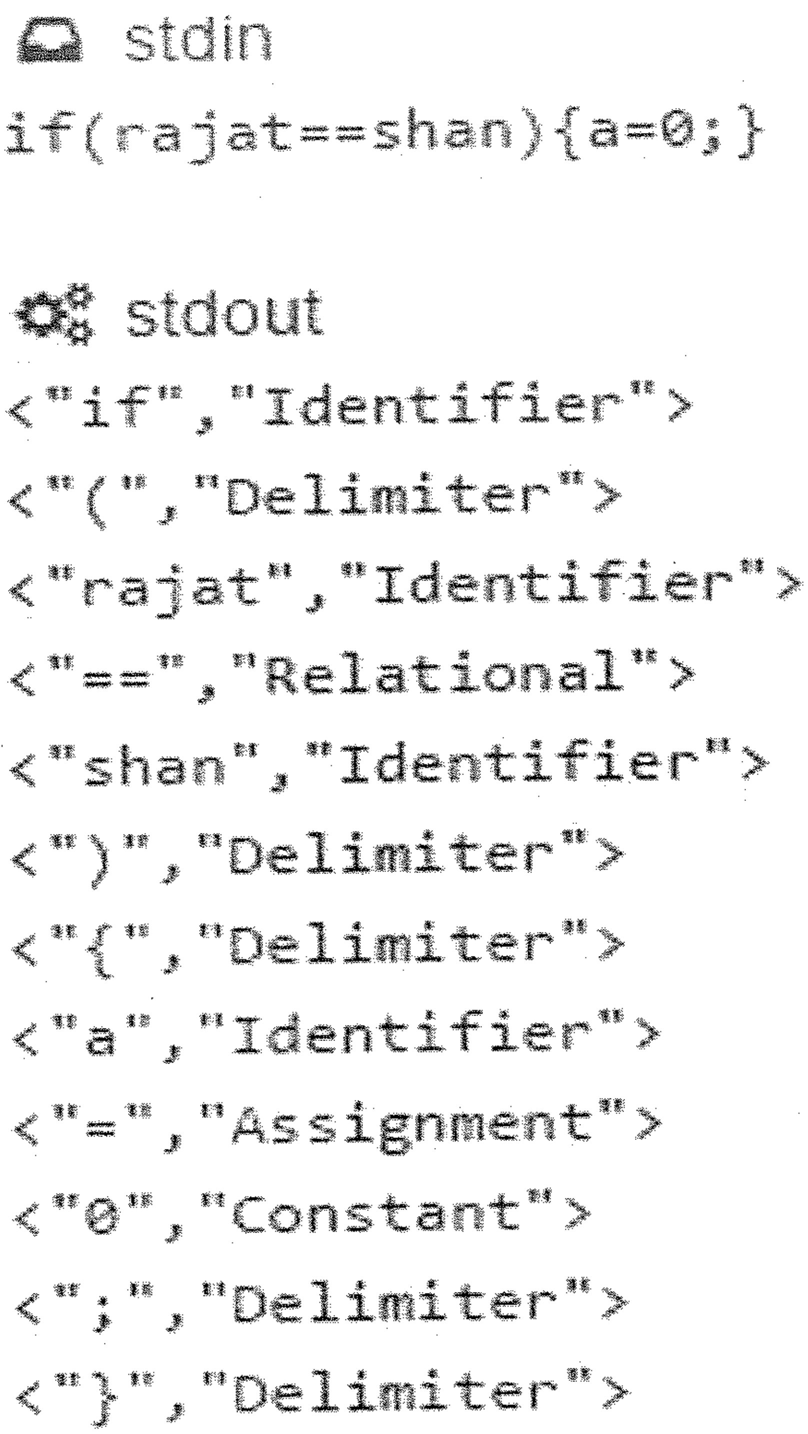
stdin

```
int a=(b+c)*d
```

stdout

```
<"int","Identifier">
<" ","WhiteSpace">
<"a","Identifier">
<"=","Assignment">
<"(","Delimiter">
<"b","Identifier">
<"+","Operator">
<"c","Identifier">
<")","Delimiter">
<"*","Operator">
<"d","Identifier">
```

Fig 5a: Output of the program lexx.c



```
stdin
if(rajat==shan){a=0;}

stdout
<"if","Identifier">
<"(","Delimiter">
<"rajat","Identifier">
<"==","Relational">
<"shan","Identifier">
<")","Delimiter">
<"{","Delimiter">
<"a","Identifier">
<"=","Assignment">
<"0","Constant">
<";","Delimiter">
<"}","Delimiter">
```

Fig 5b: Output of the program lexx.c

6. CONCLUSION AND FUTURE WORK:

The lexical analyser has been designed to take the input streams and generate the token as output which supports the minimal set of rules. It is based on finite automaton and can be built quickly and reliably by hand or with a lexical analysis generator.

A larger set of rules can be implemented in future to improve the performance of the compiler. Then the focus will shift to improving the running time of our analyser in order to achieve faster compilation. One of the directions it might take in future might involve making the whole syntax analyser and analyse the language to check the syntax.

7. REFERENCES:

- Book on "System Software" by Leland L. Beck and D. Manjula
- AHO, A., R. SETHI, and J. ULLMAN. 1986. *Compilers: Principles, Techniques, and Tools.* New York: Addison-Wesley.
- http://www.tutorialspoint.com/compiler_design/compiler_design_lexical_analysis.htm
- http://dragonbook.stanford.edu/lecture-notes/Stanford-CS143/03-Lexical-Analysis.pdf