Vellore Institute Of Technology
Chennai

Compiler Design Lab

Lab date : 10 January 2024

L45 + L46

Profesor: Sureshkumar WI

Harish Thangaraj
21BRS1033

I affirm that all the work presented against this assignment is done by me with limited Internet reference and can be tested on the logic and concepts if necessary .

Experiment - 1 : Phases of a compiler

There are 6 primary phases of a compiler and 2 auxiliary phases as follows :

1. Lexical analyser :

The first phase of the compiler which recieves a stream of code and breaks down into tokens or otherwise known as Lexemes. This is performed with the help of a DFA where the lexer keeps track of the current state and the transitions. The lexer continuous this process until the full source code is tokenized. These tokens are then fed into the second phase for further processing.

2. Syntax analysis :

The phase of the compiler is also known as parsing where the tokens fed from the previous face are verified whether they conforms to a specified grammar. During parsing, an abstract syntax tree is constructed as the output of the phase.

3. Semantic analysis :

As the name suggests, the meaning of the source code is checked here. Every programming language has semantic rules that needs to be followed and the compiler here is responsible for upholding them. For example, a used undeclared variable in a c program will throw an error in this phase.

4. Intermediate code generator :

In layman's terms, the purpose of compiler is to covert source code into machine understandable code for processing. Till this point, the previous three phases can be grouped as analysis phase where the source code was analysed and prepared for the conversion. The fourth phase here is responsible for generating the intermediate level code that can be easily converted into machine level code.

5. Code optimizer :

The fifth phase involves optimizing (minimizing) the code. For instance, variables are reduced, eliminates redundant code blocks etc for efficient processing and improving memory usage. The optimized code is fed to the last and final stage of the compiler.

6. Target code generator :

This last phase involves generating low level code that can be executed the hardware components available.

i. Error handler :

This is an auxiliary component of the compiler and is associated with every stage mentioned before. The error handler checks and throws error messages if encountered.

ii. Symbol table :

The symbol table holds data regarding every operator, variable, functions and other symbols used in the program. This component is also associated with the above mentioned 6 stages and is dynamically updated as the program is processed.

Let us demonstrate the phases of compiler with the help of a regular expression :

Expression considered : a = b + (c * d)

Lexical analysis :

Symbol table

#	Token	Name	Type	Size	Scope
1	identifier 1	a	real	4	-
2	assigment operator	=	-	-	-
3	identifier 2	b	real	4	-
4	Arithmetic operator	+	-	-	-
5	Left parenthesis	(	-	-	-
6	identifier 3	c	real	4	-
7	arithmetic operator	*	-	-	-
8	identifier 4	d	real	4	-
9	Right parenthesis	)	-	-	-

Figure 1: Tokenization

Syntax analysis :

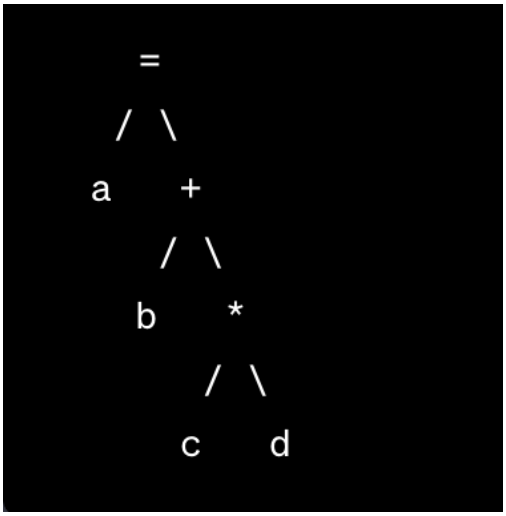


Figure 2: Abstract syntax tree

Semantic analysis :

Here the meaning of the source code is checked. For example, b * (c + d) = a despite passing the syntax phase is an incorrect declaration that will be caught.

Intermediate code generation :

t1 = c * d
t2 = b + t1
a = t2

Code optimizer :

t1 = c * d
a = b + t1

Code generator :

LOAD c
MUL d
STORE t1
LOAD b
ADD t1
STORE a

Experiment - 2 : Implementation of Deterministic Finite Automaton (DFA) from regular grammar using C language.

1. Algorithm :

1.1 Construct DFA:

i. Take user input for the number of states and the number of alphabets

ii. Construct the transition table dfa based on user input for transitions from each state on reading 'a' and 'b'.

iii. Print the DFA transition table.

1.2 Validation Function (validate):

i. Take a string as input along with its length (n).

ii. For each character, determine the next state based on the current state and the character read.

iii. Update the current state for the next iteration.

iv. If no valid transition is found, return -1 (indicating that the string is not accepted). v. Return the final state after processing the entire string.

1.3 Main Function (main):

Call the construct dfa function to construct the DFA.

Take user input for the length of the string (n) and the string itself (str).

Call the validate function to check if the input string is accepted by the DFA.

Print the final state after processing the string.

If the final state is the state specified (an accepting state), print "Accepted!"; otherwise, print "Not accepted!".

2. Source code :

```c
/*
Name : Harish Thangaraj
Reg_No : 21BRS1033
Slot : L45+L46
Language considered : L = { a^n b^m ; (n)mod 2=0, m>=1 }
*/

// Including header files
#include <stdio.h>
#include <string.h>

#define MAX_STATES 10
#define MAX_ALPHABETS 2

// Global declaration of dfa array and its size variables
int dfa[MAX_STATES][MAX_ALPHABETS];
int no_states, no_alphabets, accept_state;
int state = 0;

// Consructing a dfa
int construct_dfa()
{

    // Initializing the table
    printf("Enter No. of states");
    scanf("%d", &no_states);
    printf("Enter No. of alphabets");
    scanf("%d", &no_alphabets);
    printf("\n");
      printf("Enter acceptance state");
    scanf("%d", &accept_state);
    printf("\n");

    // Taking user input for transitions

    printf("Consider numerical states and enter the transitions\n");
    printf("If no transition exists, specify -1\n");
    printf("\n");

    for (int i = 0; i < no_states; i++)
    {
        for (int j = 0; j < no_alphabets; j++)
        {
            printf("Enter transition from %d on reading %c : ", i, (j == 0) ? 'a' : 'b');
            scanf("%d", &dfa[i][j]);
        }
    }

    // Printing the transition table
    printf("\nDFA Transition Table:\n");

    printf("States\t| Input 'a'\t| Input 'b'\n");
    printf("--------|------------|------------\n");
```

Figure 3: Source code - 1

```
    }
    result = validate(str, n);
    printf("%d", result);
    if (result == accept_state)
    {
        printf("Accepted!");
    }
    else
    {
        printf("Not accepted!");
    }
}
```

Figure 4: Source code - 2

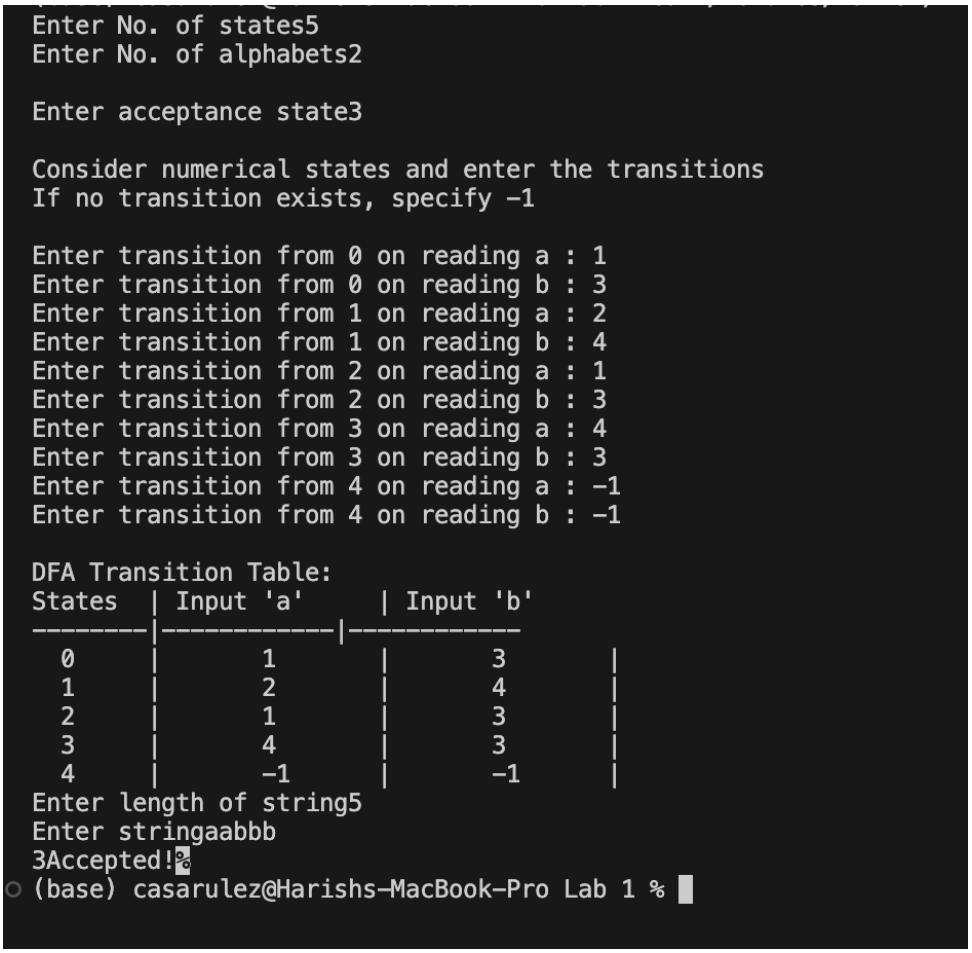
```
    for (int i = 0; i < no_states; i++)
    {
        printf("  %d\t|", i);
        for (int j = 0; j < no_alphabets; j++)
        {
            printf("\t%d\t|", dfa[i][j]);
        }
        printf("\n");
    }
    return accept_state;
}

int validate(char input[], int n)
{
    for (int a = 0; a < n; a++)
    {
        int found = 0;
        for (int j = 0; j < no_states; j++)
        {
            if (input[a] == 'a' && state == j)
            {
                state = dfa[j][0];
                found = 1;
                break;
            }
            if (input[a] == 'b' && state == j)
            {
                state = dfa[j][1];
                found = 1;
                break;
            }
        }
        if (!found)
        {
            // If no valid transition is found for the current character and state, reject the string
            return -1;
        }
    }
    return state;
}

// Primary code
int main()
{
    accept_state=construct_dfa();
    int n, result;
    printf("Enter length of string");
    scanf("%d", &n);
    char str[n];
    printf("Enter string");
    for (int i = 0; i < n; i++)
    {
        scanf(" %c", &str[i]);
```

Figure 5: Source code - 3

3. Ouput :


Enter No. of states5
Enter No. of alphabets2
Enter acceptance state3
Consider numerical states and enter the transitions
If no transition exists, specify -1
Enter transition from 0 on reading a : 1
Enter transition from 0 on reading b : 3
Enter transition from 1 on reading a : 2
Enter transition from 1 on reading b : 4
Enter transition from 2 on reading a : 1
Enter transition from 2 on reading b : 3
Enter transition from 3 on reading a : 4
Enter transition from 3 on reading b : 3
Enter transition from 4 on reading a : -1
Enter transition from 4 on reading b : -1
DFA Transition Table:
States | Input 'a' | Input 'b'
0 | 1 | 3
1 | 2 | 4
2 | 1 | 3
3 | 4 | 3
4 | -1 | -1
Enter length of string5
Enter stringaabbb
3Accepted!%
(base) casarulez@Harishs-MacBook-Pro Lab 1 %

```
Enter No. of states5
Enter No. of alphabets2

Enter acceptance state3

Consider numerical states and enter the transitions
If no transition exists, specify -1

Enter transition from 0 on reading a : 1
Enter transition from 0 on reading b : 3
Enter transition from 1 on reading a : 2
Enter transition from 1 on reading b : 4
Enter transition from 2 on reading a : 1
Enter transition from 2 on reading b : 3
Enter transition from 3 on reading a : 4
Enter transition from 3 on reading b : 3
Enter transition from 4 on reading a : -1
Enter transition from 4 on reading b : -1

DFA Transition Table:
States  | Input 'a'     | Input 'b'
--------|---------------|-----------
  0     |       1       |       3       |
  1     |       2       |       4       |
  2     |       1       |       3       |
  3     |       4       |       3       |
  4     |       -1      |       -1      |
Enter length of string4
Enter stringaaaa
2Not accepted!%
```