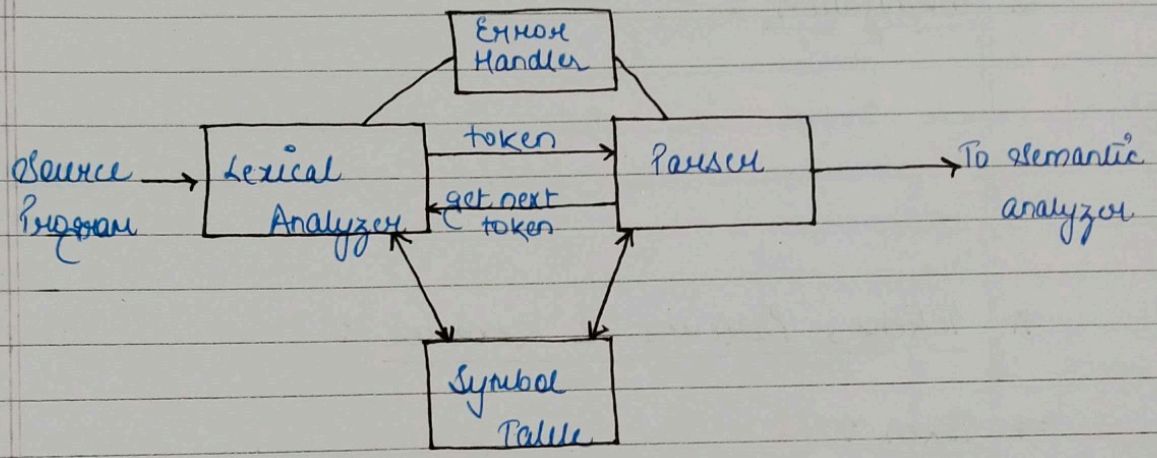
14th Jan '17.
Lecture-5

Unit-1

Lexical Analysis

Role of lexical analyzer :



- removes white spaces
- if there are more than 1 white spaces then, LA does compaction; i.e; if there are 3 blank spaces then LA will compact it & make it 1 blank space.
- removes the comment lines.

* Sometimes lexical analyzers are divided into cascade of 2 processes :

1 Scanning :

consist of simple processes that do not require tokenization of inputs such as deletion of comments, compaction of white spaces in one

2 Tokenization :

portion

LA ~~proper~~ is the more complex portion which produces the tokens from the output of scanner.

* Tokens, Patterns & Lexemes :

⇒ Token :

A token is a pair consisting of a token name & an optional attribute value.

The token name is an abstract symbol representing a kind of lexical unit.

Example : a particular keyword or a sequence of input characters denoting an identifier. ~~identifiers~~ identifiers, keywords, constants, literals, punctuations, tokens will be generated.

For operators tokens will be created but their ~~entry~~ entry will not be mentioned in symbol table.

int a ;

3 tokens for "int", "a", ";"

printf ("%d", &a) ;

① ② ③ ④ ⑤ ⑥ ⑦ ⑧

8 tokens for "printf", "(", "\"%d\"", ",", "&", "a", ")", ";"

&& | a | ;

3 tokens.

17th Jan'17

Lecture 6

⇒ Pattern :

A pattern is a description of a form that the lexemes of a token may take.

In the case of a keyword as a token the pattern is just the sequence of characters that form the keyword.

⇒ Lexeme :

A lexeme is a sequence of characters in the source program that match the pattern for a token & is identified by the lexical analyzer as an instance of that token.

Example : i >= j ? i : j ;

8 tokens

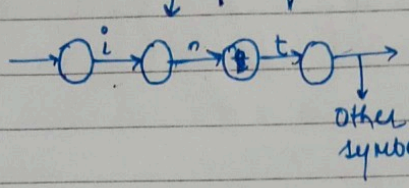
if i | > | j | 3 tokens

i | >= | j | ⇒ 3 tokens

Date: __/__/__
Page:

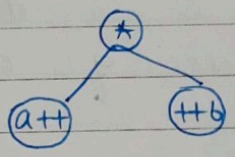
10th Jan' Lecture

int | a | = | 100 | ; 5 tokens



a | + + | + + | + + | b | ; | 6 tokens.
no lexical error
but syntax error (parse tree can't be made)

a | + + | * | + + | b | 5 tokens
no lexical & syntax error.

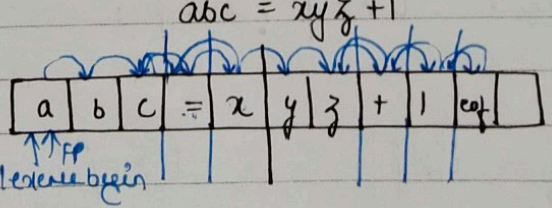


→ parse tree

Input buffering : to speed up the slow reading of source prgs. for tokenization.

We will introduce a 2 buffer scheme that handles large look ahead safely. We then consider an improvement involving sentinels that saves time checking for the end of buffer.

1. Buffer pairs :



18th Jan'17
Lecture-7

- Each buffer is of the same size N, and N is usually the size of a disk block. Using one system read command we can read N characters into a buffer.
- If fewer than N characters remain in the input file then a special character represented by eof marks the end of file & it is different from any possible character of the source program.
- 2 pointers to the input are maintained :

→ Pointer lexemeBegin : Marks the beginning of the current lexeme whose extent we are attempting to determine.

→ Pointer Forward : Scans ahead until a pattern match is found.

- Once the next lexeme is determined, forward is set to the character at its right end then after the lexeme is recorded as an attribute value of a token returned to the parser lexemeBegin is set to the character immediately after the lexeme is just found.

Date: __/__/__
Page:

- Advancing forward requires that we first test whether we have reached the end of one of the buffer & if so we must reload the other buffer from input & move forward to the beginning of the newly loaded buffer.

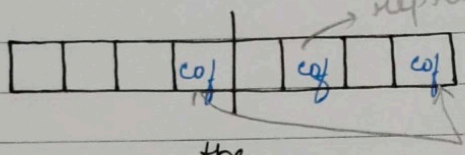
* Code to advance the forward pointer :

```
if (FP is at end of the 1st half)
{
    reload the 2nd half;
      FP = FP+1;
}
else if ( FP is at end of the 2nd half)
{
    reload the 1st half;
    move the FP to the beginning of the 1st half;
      FP = FP+1;
}
FP+1;
```

II. Buffer pair with sentinels : → in this FP matches with eof. whereas in previous FP match with size.

- In the previous method, for each character read we make 2 tests, one for the end of the buffer and other to determine what character is read.

- We can combine the buffer end test with the test for the current character if we extend each buffer to hold a sentinal character at the end. Sentinel is a special character that can not be part of the source program & the natural choice is the character eof.



* Code to advance ~~to~~ the FP :

```
if ( FP | = eof )
{
  if (FP is at the end of 1st half)
  {
    reload the 2nd half;
     FP = FP+1;
  }
  else if(FP is at the end of 2nd half)
  {
    reload 1st half;
    move to the beginning of the 1st half;
  }
  else
    terminate lexical analysis ;
}
```

Q. Find number of tokens in : int a[4][5];

for 1D arrays:

a[i] can be written as *(a+i)

for 2D arrays;

a[i][j] can be written as *(*(a+i)+j)

in int a[4][5];
replace with above form

| int | * | (| * | (| a | + | 4 |) | + | 5 |) | ; |

⇒ 13 tokens.

19th Jan'17
Lecture-8

Specification of tokens :

* Formal notations for RE :

1. Strings & Languages

2. Operations on languages :

a.) Union
b.) Concatination
c) Closure (Kleen's & Positive)

Example: L be the set of letters:

$L = \{A, B, C, \cdots Z, a, b, c \cdots z\}$

D be the set of digits:

$D = \{0, 1, \cdots 9\}$

LUD = language with 62 strings of length 1.

LD = set of 520 strings of length 2. each consist of 1 letter followed by 1 digit.

L^4 = set of all 4 letter strings.

L^* = set of all strings of letters including null.

$L(LUD)^*$ = set of all strings of letters & digits but beginning with letter.
It is an identifier.

D^+ = set of all strings of 1 or more digit, null is not present.

3. Regular Expressions :

Rules that define the regular expressions over some alphabet Σ and language that those expressions done.

a.) **Basis :**

2 rules that form the basis :

- "Null" is a regular expression and "$\mathcal{L}(\epsilon) = \{\epsilon\}$."
- if "a" is a symbol in Σ then, a is regular expression and "$\mathcal{L}(a) = \{a\}$." It means language of one string or length one with a in its position.

b.) **Induction rule :**

Suppose r and s are regular expression denoting languages $\mathcal{L}(r)$ & $\mathcal{L}(s)$.

r, or s is regular expression denoting language $\mathcal{L}(r) \cup \mathcal{L}(s)$.

r, s is a regular expression denoting language $\mathcal{L}(r)\,\mathcal{L}(s)$

r^* is a regular expression denoting $\mathcal{L}(r)^*$.

r is a regular expression denoting $\mathcal{L}(r)$.

4. Regular definations :

We may wish give names to certain RE and use those names in subsequent expressions as if the names were themselves symbols.

If Σ is an alphabet of basic symbols then a regular defination is a sequence of definations of the form :

$$d1 \rightarrow r1 \quad \Rightarrow \Sigma$$
$$d2 \rightarrow r2 \quad \Rightarrow \Sigma \cup d1$$
$$d3 \rightarrow r3$$
$$\vdots$$
$$dn \rightarrow rn$$

where,

each r_i is a RE over $\Sigma \cup \{d1 \cup d2 \cup \cdots d_{i-1}\}$

Example : Give a Regular defination for identifiers.

identifier → letter (letter/digit)*

letter → a|b|c|---|z|A|B|C|---|Z|

digit → 0|1|---|9|

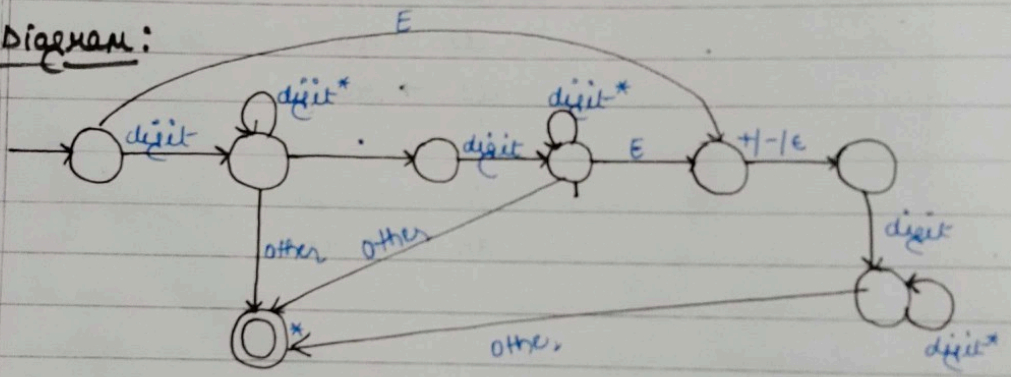
20th Jan '17

Lecture-9 **Example:** Give the regular defination for unsigned numbers (int & float).

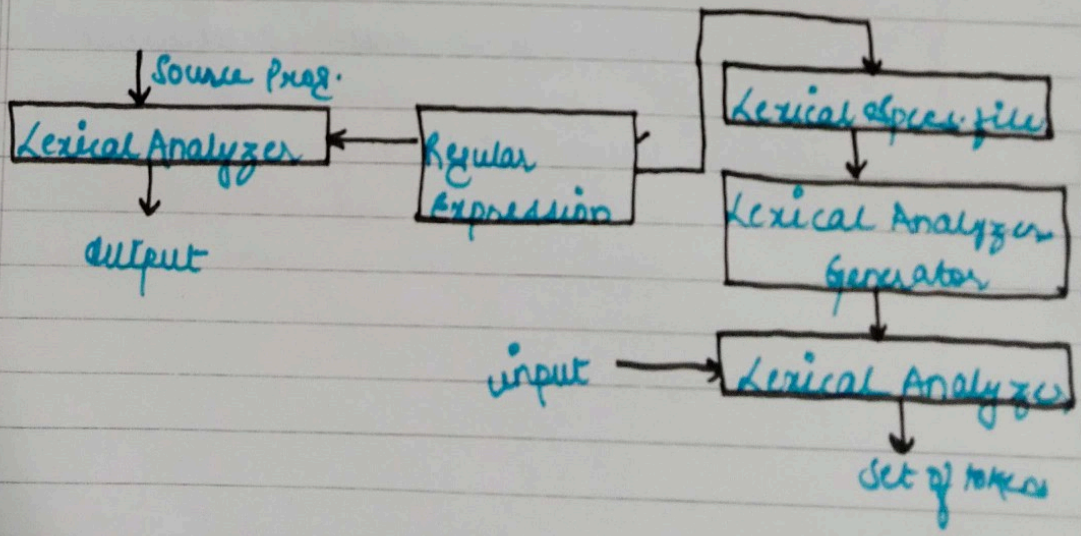
digit → 0|1|2|----|9
digits → digit (digit)*
optional fraction → . digits | ε
optional exponent → (E(+|-|ε) digits)|ε

number → digits optional fraction optional exponent

Transition Diagram:



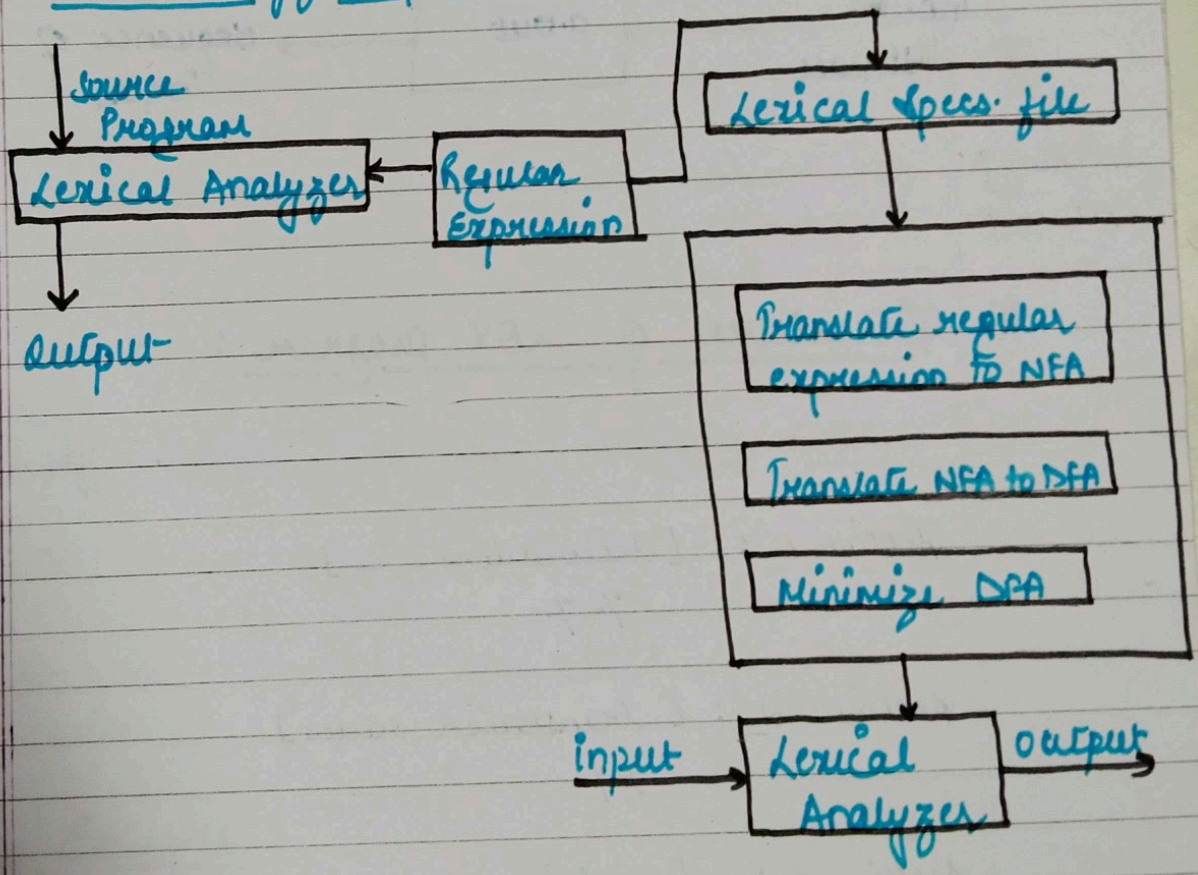
Elements of Lexical Analyzer :



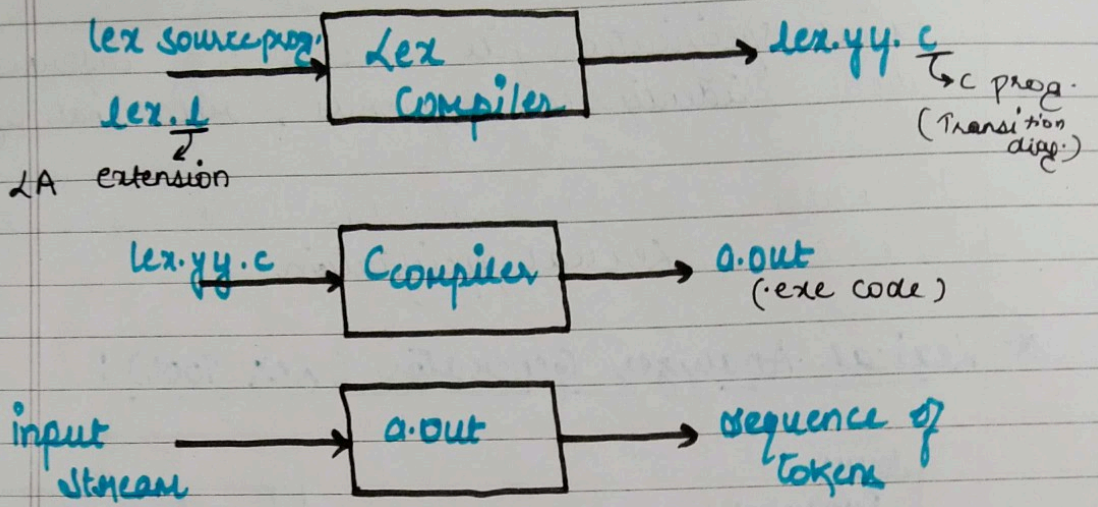
Lexical Specification file consists of information about identifiers, keywords, relational operators.

The rules of basic building blocks for a language are called Lexical specification.

* Lexical Analyzer Generator (Lex Tool):



* Use of Lex ?



21st Jan '17
Lecture-10

Structure of LEX program:

A LEX program is in following form:

Section 1 : [declarations ---]
%%

Section 2 : [translation rules]
%%

Section 3 : [auxiliary functions/ subroutines]

Date: __/__/__
Page:

Section 1 : [declaration ...]

This section includes variables, constants & regular definition.

Section 2 : [Translation rules]

The translation rules each have the form pattern and corresponding action.
Each pattern is a RE which may use the regular definations of the declaration section.
The actions are the fragments of code written in C.

Section 3 : [auxiliary functions / subroutines]

It holds whatever additional functions are used in the actions. Alternatively, these functions can be compiled seperately & can be loaded with LA.

Example :

```
%{
    #include <stdio.h>
    int count;
%}
    letter [A-Z a-z]
    digit [0-9]
%%
```

// section 1

```
IF { printf(" It is a KEYWORD %s", yytext); }
FOR { printf(" It is a KEYWORD %s", yytext); }
WHile { printf (" It is a KEYWORD %s ", yytext); }

{letter} ({letters}|{digit})* {printf("identifier %s", yytext);}

%%
```

// section 2

```
main()
{
    yylex();
}
```

// section 3

Q. Write a lex program to find the longest word in the input file & print the length of word.

```
%{
    #include<stdio.h>
    int k=0;
%}
%%

[a-z A-Z]+
{
    if (yyleng > k)
    {
        k=yyleng;
    }
}
%%

int main(int argc[], char **argv[])
{
    yyin = fopen("abc.txt", "r");
    yylex();
    printf("Largest : %d", k);
    printf("\n");
    return 0;
}
```

+ -> occurrences of a-z, A-Z can be more than 1 ... 0 is not inc.

24th Jan'17
Lecture 11

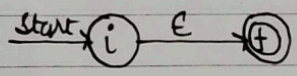
Thompson's Algorithm:

Input : A RE 'r' over alphabet 'Σ'.

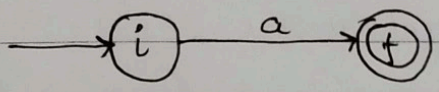
Output: An NFA 'N' accepting $L(r)$.

Method/Procedure :

a.) Basis : For expression ϵ construct the NFA

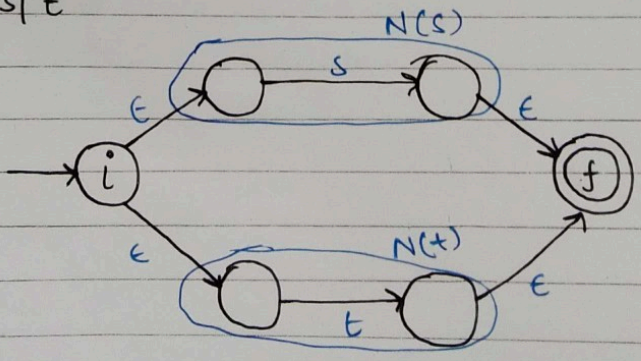


For an subexpression ~~is~~ 'a' in Σ.

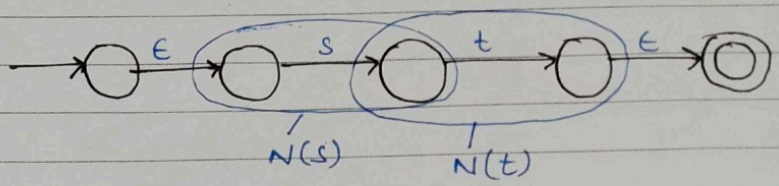


b.) Inductive : Suppose N(s) & N(t) are NFAs for REs s & t resp.

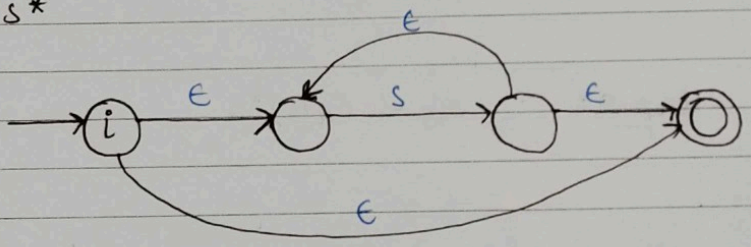
i.) $r = s|t$



ii) $r = st$



iii) $r = s^*$

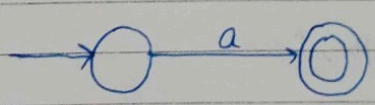


iv) Suppose $r = (s)$ then $L(r) = L(s)$ and then we can use the NFA N(s) as N(r).

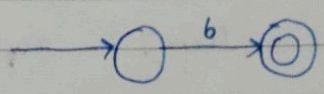
Ques. Use Thompson's construction algo. to construct NFA for given RE :

$$r = (a|b)^* abb$$

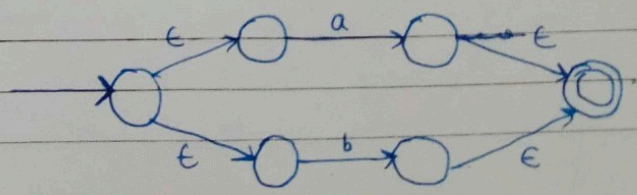
Step-1 $r_1 = a$.



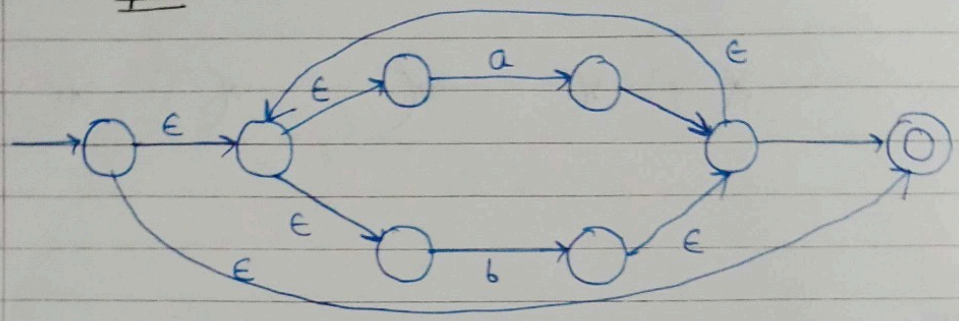
Step-2 : $r_2 = b$



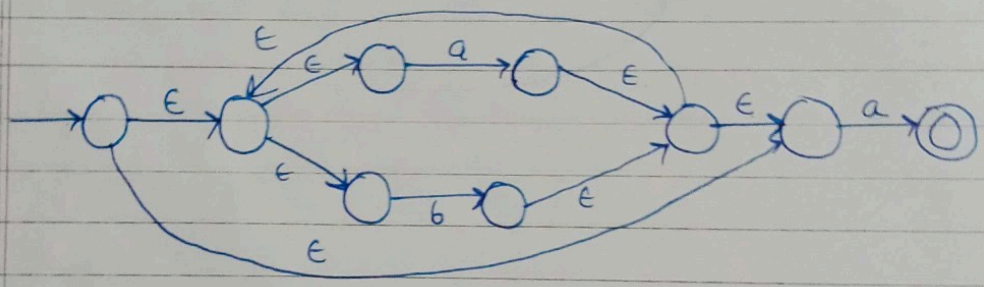
Step-3 : $r_3 = a|b$



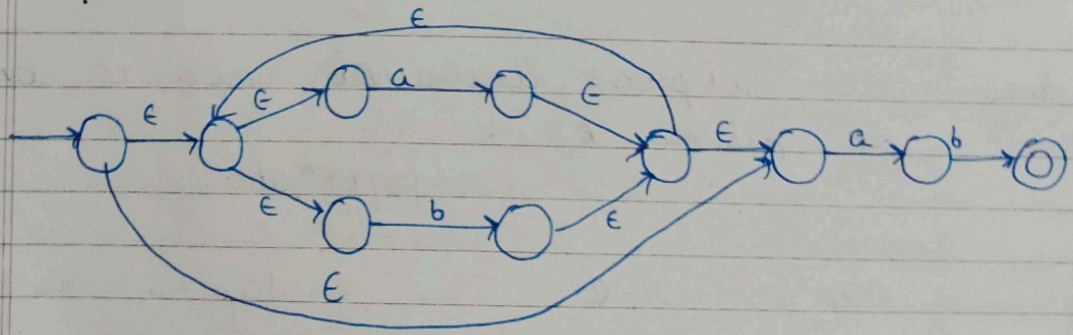
Step-4 : $r_4 = (a|b)^*$



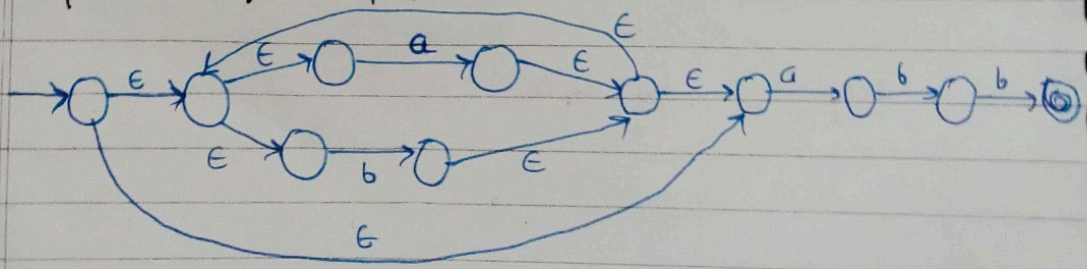
Step-5 : $r_5 = (a|b)^* a$



Step-6 : $r_6 = (a|b)^* a b$

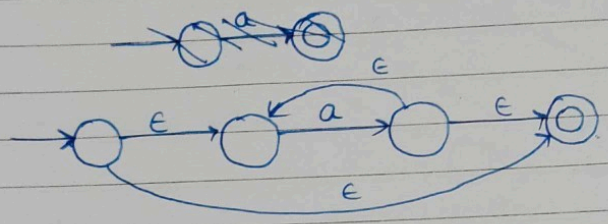


Step-7 : $r_7 = (a|b)^* a b b$

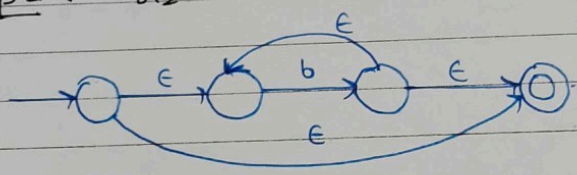


Q. $r = (a^* | b^*)^*$

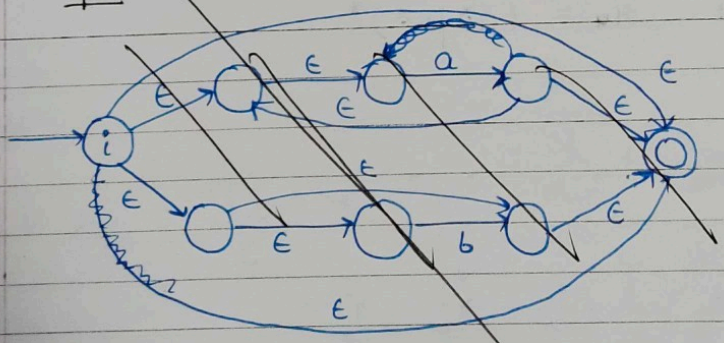
Step-1 : $r_1 = a^{**}$ ~~Step-2~~

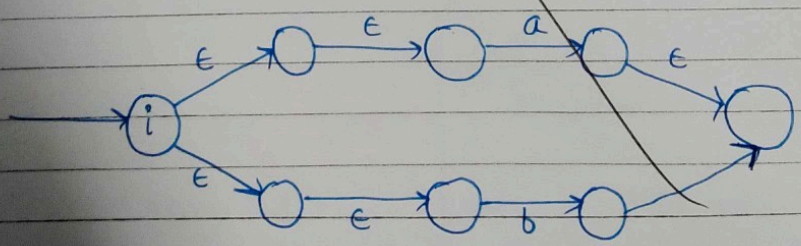


Step-2 : $r_2 = b^*$



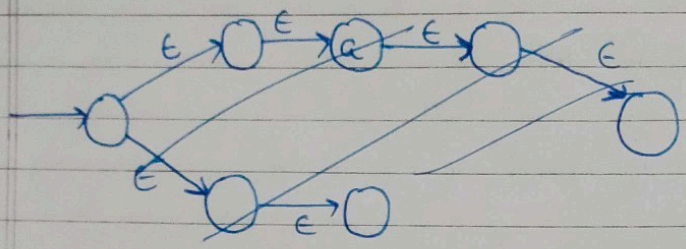
Step-3 : $r_3 = (a^* | b^*)$

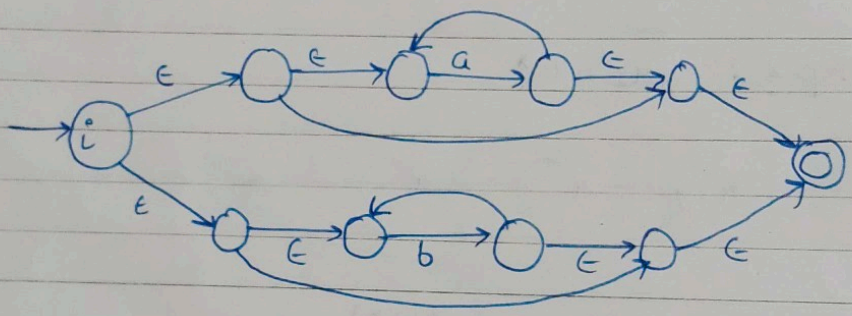




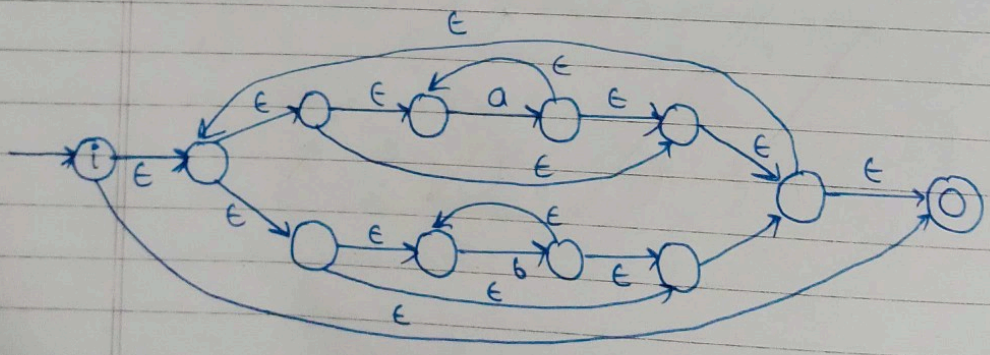
Date: __/__/__
Page:

Step-3 $r_3 = (a^* | b^*)$





Step-4 : $r_4 = (a^* | b^*)^*$



Conversion of NFA to DFA :

(Subset construction algorithm)

Input : An NFA N.

Output : A DFA D accepting the same language as N.

Method : Our algorithm constructs a transition table Dtran for D. Each state of D is a set of NFA states and we construct Dtran.

'e' is a single state of N while 'T' is the set of states of N.

25th Jan'17
Lecture-12

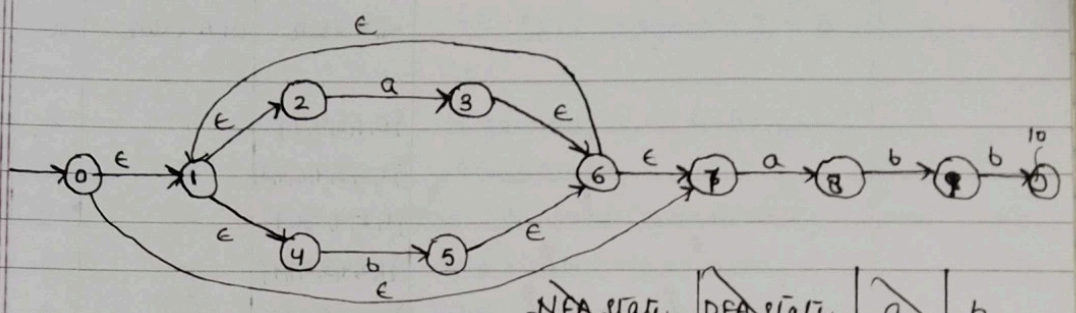
Operation	Description
• ε-closure(s)	Set of NFA states reachable from NFA state s on ε transitions alone.
• ε-closure(T)	Set of NFA states reachable from some NFA state s in set T on ε transitions alone.
• move(T, a)	Set of NFA states to which there is a transition on i/p symbol a from some state s in T.

Operation	Description
ϵ closure(s_0)	The start state of D DFA is the ϵ closure s_0. The accepting state of D are all those set of N NFA's state that include atleast one accepting state of N. Initially ϵ closure s_0 is the only state in Dstate. & it is unmarked.

* **Algorithm :**

```
while ( there is an unmarked state T in Dstates)
{
    mark T;
    for(each input symbol 'a')
    {
        U = ε-closure(move(T,a));
        if (U is not in Dstates)
            add U as an unmarked state of Dstates;
        Dtran[T, a] = U;
    }
}
```

Q. $r = (a/b)^*\ abb$



NFA state	DFA state	a	b
{0,1,2,4,7}	A	B	C
{1,2,3,4,6,7,8}	B		
{1,2,4,5,6,7}	C		

A = ϵ-closure(0)

Dstates = A = {0,1,2,4,7}

Dtran[A,a] = ϵ-closure(move(A,a))

= ϵ-closure(move(0,a), move(1,a), move(2,a), move(4,a), move(7,a))

= ϵ-closure(3,8)

= ϵ-closure(3) ∪ ϵ-closure(8)

= {3,6,7,1,2,4} ∪ {8}

Dtran[A,a] = {1,2,3,4,6,7,8} = B

Dtran[A,b] = ϵ-closure(move(A,b))

= ϵ-closure(move(0,b), move(1,b), move(2,b), move(4,b), move(7,b))

= ϵ-closure(5)

= {5,6,7,1,2,4}

Dtran[A,b] = {1,2,4,5,6,7} = C

B = {1,2,3,4,6,7,8}

Dtran[B,a] = ε-closure(move(B,a))
= ε-closure(move(1,a), move(2,a), move(3,a), move(4,a), move(6,a), move(7,a), move(8,a))
= ε-closure(3) ∪ ε-closure(8)
= {3, 8, 3, 4, 6, 7, 8}

Dtran[B,b] = ε-closure(move(B,b))
= ε-closure(move(1,b), move(2,b), move(3,b), move(4,b), move(6,b), move(7,b), move(8,b))
= ε-closure(5, 9)
= ε-closure(5) ∪ ε-closure(9)
= {5,6,7,1,2,4} ∪ {9}
Dtran[B,b] = {1, 2, 4, 5, 6, 7, 9} = D

C = {1,2,4,5,6,7}

Dtran[C,a] = ε-closure(move(C,a))
= ε-closure(move(1,a), move(2,a), move(4,a), move(5,a), move(6,a), move(7,a))
= ε-closure(3,8)
= {1, 2, 3, 4, 6, 7, 8}

NFA state	DFA state	a	b
{0,1,2,4,7}	A	B	C
{1,2,3,4,6,7,8}	B	B	D
{1,2,4,5,6,7}	C	B	C
{1,2,4,5,6,7,9}	D	B	E
{1,2,4,5,6,7,10}	(E)	B	C

Dtran[C,b] = ε-closure(move(C,b))
= ε-closure(move(1,b), move(2,b), move(4,b), move(5,b), move(6,b), move(7,b))
= ε-closure(∅)
= {1,2,4,5,6,7}

D = {1,2,4,5,6,7,9}

Dtran[D,a] = ε-closure(move(D,a))
= ε-closure(move(1,a), move(2,a), move(4,a), move(5,a), move(6,a), move(7,a), move(9,a))

Dtran [D, a] = ε closure (3, 8)

= {1, 2, 3, 4, 6, 7, 8}

Dtran [D, b] = ε-closure (move (D, b))

= ε closure (move(1,b), move(2,b), move(4,b), move(5,b), move(6,b), move(7,b), move(9,b))

= ε closure (5, 10)

= ε closure(5) ∪ ε closure (10).

= {5, 6, 7, 1, 2, 4} ∪ {10}

= {1, 2, 4, 5, 6, 7, 10} = E

E = {1, 2, 4, 5, 6, 7, 10}

Dtran [E, a] = ε closure (move E, a))

= ε closure (move(1,a), move(2,a), move(4,a), move(5,a), move(6,a), move(7,a), move(10,a))

= {1, 2, 3, 4, 6, 7, 8}

= B

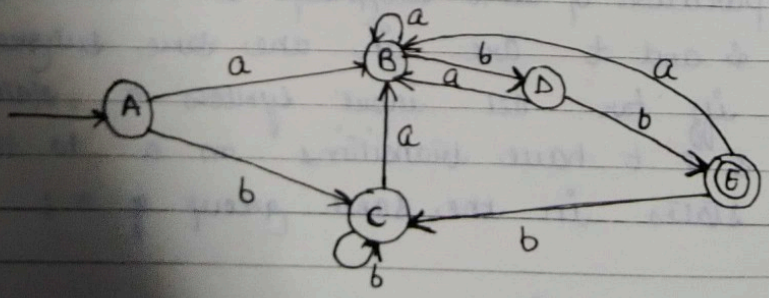
Dtran [E, b] = ε closure (move(E, b))

= ε closure (move(1,b), move(2,b), move(4,b), move(5,b), move(6,b), move(7,b), move(10,b))

= ε closure (5)

= {1, 2, 4, 5, 6, 7}

= C



28th Jan '17
Lecture-13

Minimization of DFA (States):

Input : A DFA D with set of states S, input alphabet Σ, start state s_0 & set of accepting states F.

Output : A DFA D' accepting the same language as D and having as few states as possible.

Method :

Step 1 : Start with an initial partition π with 2 groups F & S-F, the accepting & non-accepting states of D.

Step-2 : Apply the following procedure to construct a new partition π_{new}.

Initially let, $\pi_{new} = \pi$
for (each group G of π)
{
partition G into subgroups such that 2 states s and t are in the same subgroup iff for all input symbols a, states s & t have transitions on a to the states in the same group of π;

Replace G in π_{new} by the set of

all subgroups formed ;
}

Step-3 :

if ($\pi_{new} = \pi$), let $\pi_{final} = \pi$ and continue with step-4. otherwise repeat step-2 with π_{new} in place of π.

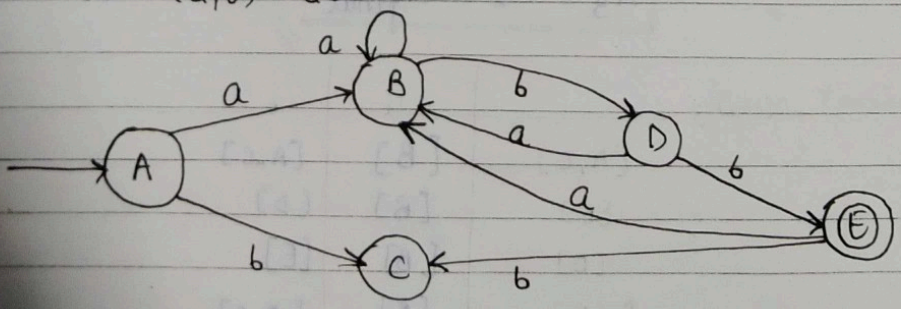
Step-4 :

Step-4 : Choose one state in each group of π_{final} as the representative for that group. The representatives will be the states of the minimum state DFA D'. The other components of D' are constructed as follows :

a.) the start state of D' is the representative of the group containing the start state of D.

b.) the accepting states of D' are the representatives of those groups that contain an accepting state of D.

Q $r = (a|b)^* abb$



Date: / /
Page:

1st Feb'1
Lecture-1

$F = \{E\}$

$S - F = \{A, B, C, D\}$ $\quad \pi_0 = \{A, B, C, D\}^{q_1}, \{E\}^{q_2}$

A $\xrightarrow{a} B(q_1)$, A $\xrightarrow{b} C(q_1)$

B $\xrightarrow{a} B(q_1)$, B $\xrightarrow{b} D(q_1)$

C $\xrightarrow{a} B(q_1)$, C $\xrightarrow{b} C(q_1)$

D $\xrightarrow{a} B(q_1)$, D $\xrightarrow{b} E(q_2)$ $\}$ → new group.

$\pi_1 = \{A, B, C\}, \{D\}, \{E\}$

$q_1 \quad q_3 \quad q_2$

A $\xrightarrow{a} B(q_1)$, A $\xrightarrow{b} C(q_1)$

B $\xrightarrow{a} B(q_1)$, B $\xrightarrow{b} D(q_3)$ → new group.

C $\xrightarrow{a} B(q_1)$, C $\xrightarrow{b} C(q_1)$

$\pi_2 = \{A, C\}, \{B\}, \{D\}, \{E\}$

$q_1 \quad q_4 \quad q_3 \quad q_2$

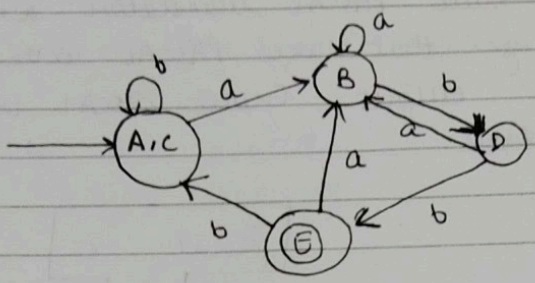
A $\xrightarrow{a} B(q_4)$, A $\xrightarrow{b} C(q_1)$

C $\xrightarrow{a} B(q_4)$, C $\xrightarrow{b} C(q_1)$

$\pi_3 = \pi_2 = \pi_{final}$ //

Now;

	a	b
[A,C]	[B]	[A,C]
[B]	[B]	[D]
[D]	[B]	[E]
* [E]	[B]	[A,C]



1st Feb'17
Lecture-14

Bootstrapping :

- The process by which simple language is used to translate more complicated program which inturn may handle an even more complicated program is known as Bootstrapping.

- The compiler is characterized by three languages:
  - Source language (S)
  - Target language (T)
  - Implementation language (I).

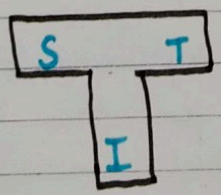


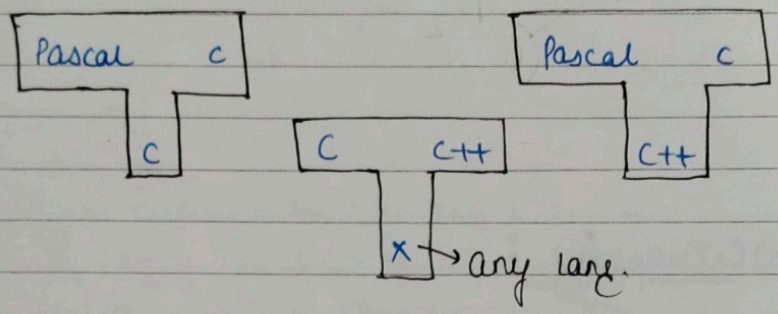
fig: Basic T-Diagram

S T
I

fig: Textual form

Date: / /
Page:

Example : We have PASCAL translator written in C language that takes PASCAL code & produces C as output. Create PASCAL translator in C++ for same.



- Pascal C (C) + C (J) C++ = Pascal C (C++)

- compilers are of 2 types :

1. Native compiler :

They are written in same language as the target language.

Example : SMM (Source S, Implementation M, Target M) is a compiler for the language S ; i.e; in the language that runs on M/c M. and generates o/p code that runs on M/c M.

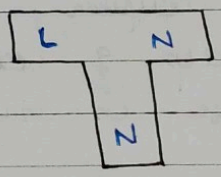
II. Cross compiler : (Retargeting compiler)

They are written in different languages as the target language.

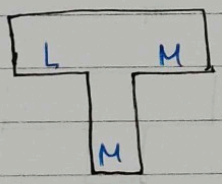
Example : SNM is a compiler for the language S that is in a language that runs on m/c N & generate o/p code that runs on m/c M.

Q. Write a compiler of language L for machine N with the help of an existing compiler of language L for machine M.

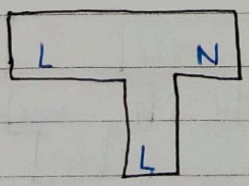
Required compiler :



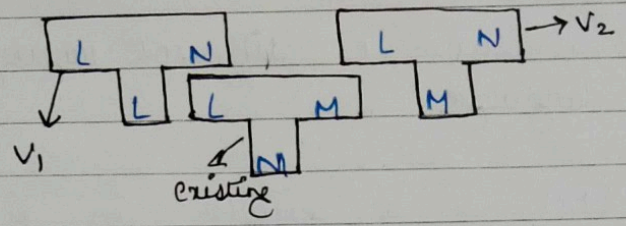
Existing compiler :



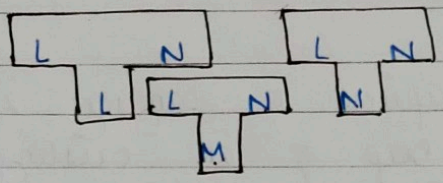
Step-1 : Create a compiler V_1



Step-2 : Compile the compiler V_1 with the existing compiler.

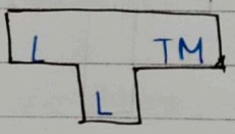


Step-3 : Compile V_1 on V_2

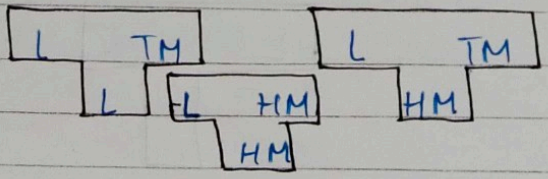


Q. Write a compiler L for machine TM with the help of existing compiler for the same language on machine HM.

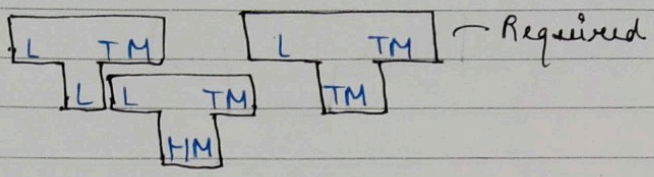
Step 1 & : V_1



Step-2 : V_1 with existing.



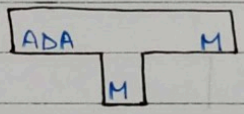
Step-3 : V_1 & V_2



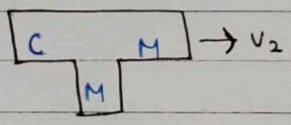
2nd Feb'17
Lecture-15

Q. Implement an ADA compiler for machine M and no existing compiler is given.

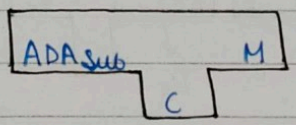
Required :



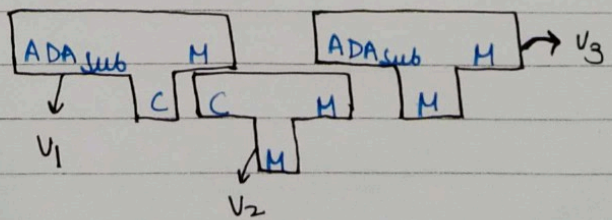
Let, existing :



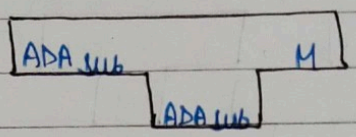
Step-1 : Create a compiler V_1.



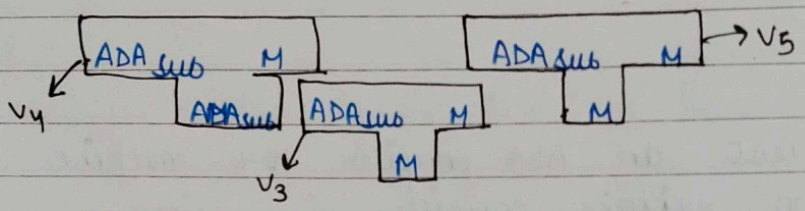
Step-2 : Compile compiler V_1 on V_2



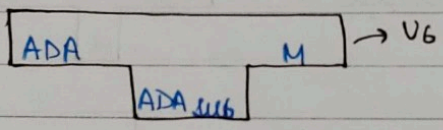
Step-3 : Create a compiler V_4.



Step-4 : Compile V_4 on V_3.



Step-5 : Create a compiler V_6.



Step-6 : Compile V_6 on V_5