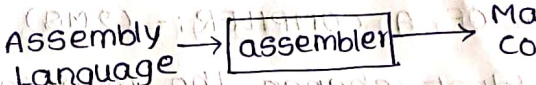
COMPILER FUNCTIONS:

* Check errors and does validation
* construct a model for convertion
* Does memory allocation for intermediate code.
* Programmers can concentrate on effective code to solve a problem.

"COMPILER is a software that contains several lines of code and uses all components of a computer to generate the machine code."

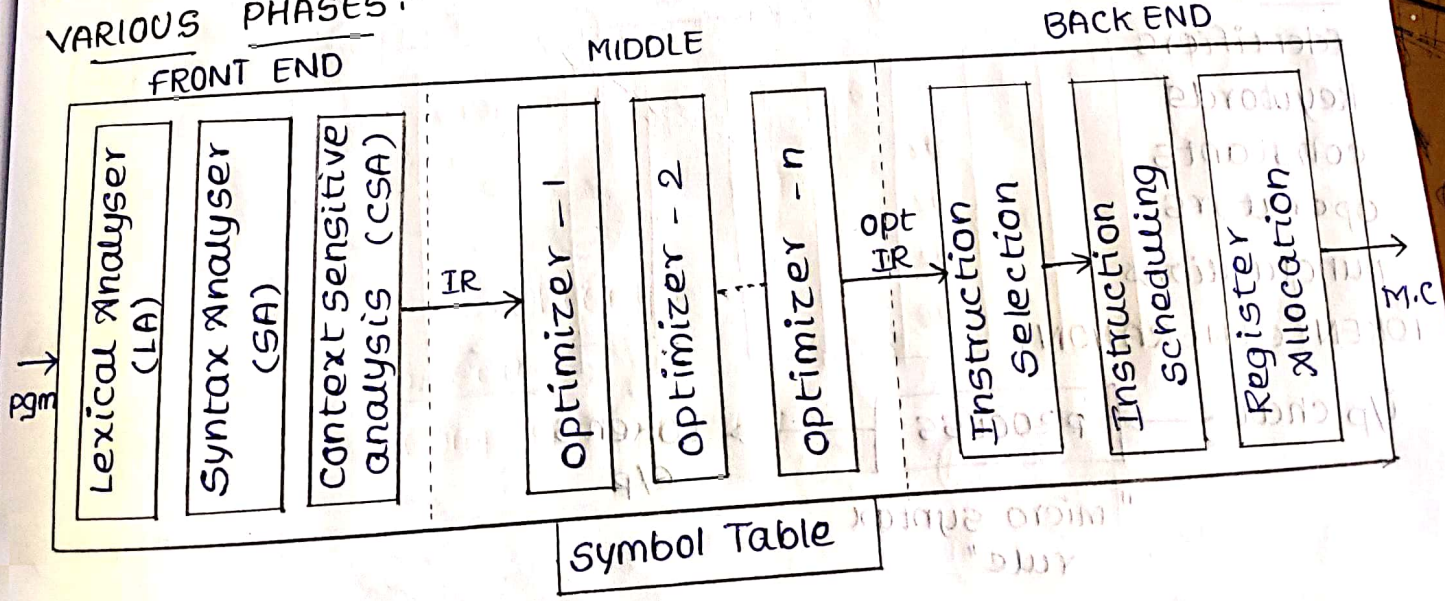
TRANSLATORS OF HLL:

* Assemblers
* Interpreter
* compiler



Interpreter	compiler
Line by Line	full code at the end
at the end result of the program	Machine code

VARIOUS PHASES:-



LA (scanner) $\xrightarrow{\text{generate}}$ tokens : keywords, identifiers, constants, operators.

SA (parser) ⟶ check grammar (syntax) ⟶ generate IR ↓ syntax tree, parse tree, TA

IR → Intermediate Representation

opt → optimize

CSA → Context Sensitive Analysis

Instruction Selection → selects machine instruction for code.

Instruction Scheduling → schedule machine instruction for code.

RA → Register Allocation for IR

Symbol Table → Repository for variables in code.

PRINCIPLES OF A COMPILER:- (2MA)

* It shouldnot change the meaning of the code written by programmar
* It should build a effective machine code.

LEXICAL ANALYSIS [SCANNER]

Stream of characters $\xrightarrow{\text{i/p}}$ [LA] ⟶ tokens (words)

TOKENS

- Identifiers
- keywords
- constants
- operators
- punctuations

TOKEN GENERATION:

i/p char ⟶ [PROCESS] ⟶ tokens o/p

"micro syntax rule"

ANVANTAGES

* parser ge
* LA is fu
* Every ru
Scanner co

1. State tr

(S0)

coding:

→ nextch

New sca

c ← n

if (c

begi

c ←

if

be

els

STATE

1. NE

(S0)

2.

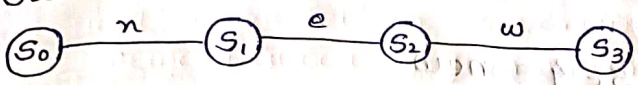
2020.01.20 13:25

ADVANTAGES OF LA OVER PARSER:

* parser gets words as i/p and not letters
* LA is fully automatic than parser
* Every rule added in LA shrinks parser

Scanner construction:

1. State transition



coding:

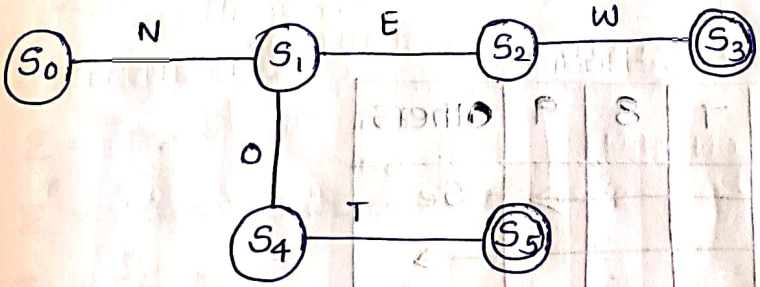
→ nextchar() function to get char i/p.

New scanning:

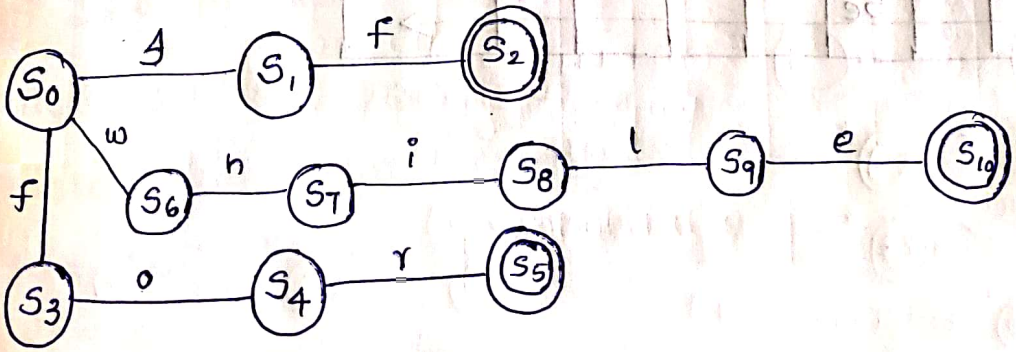
```
c ← nextchar();
 if (c = 'n') then
 begin
  c ← nextchar();
  if (c = 'e') then
  begin
   c ← nextchar();
   if (c = 'w') then
   report 'success'
else
  DO something else
```

STATE TRANSITION FOR:

1. NEW, NOT



2. FOR, WHILE, IF



2020.01.20 13:25

IMPLEMENTING SCANNER:

1. Table driver
2. Hand coded

FINITE AUTOMATA [FA]:

* FA is a collection of states, set of transition between states, alphabets, a start state and one or more final states.
* 5 Tuples : $\{S, \Sigma, \delta, S_0, S_f\}$ new

$S \longrightarrow \{S_0, S_1, S_2, S_3\}$ states

$\Sigma \longrightarrow \{n, e, w\}$ alphabets

$\delta \longrightarrow \{(S_0 \xrightarrow{n} S_1)(S_1 \xrightarrow{e} S_2)(S_2 \xrightarrow{w} S_3)\}$ transition

$S_0 \rightarrow S_0$

$S_F \rightarrow S_3$

Write FA for new, not

$S \longrightarrow \{(S_0 \xrightarrow{n} S_1)(S_1 \xrightarrow{e} S_2)(S_2 \xrightarrow{w} S_3)(S_1 \xrightarrow{o} S_4)(S_4 \xrightarrow{t} S_5)\}$

$S_F \longrightarrow \{S_5, S_3\}$

UNSIGNED INTEGER.

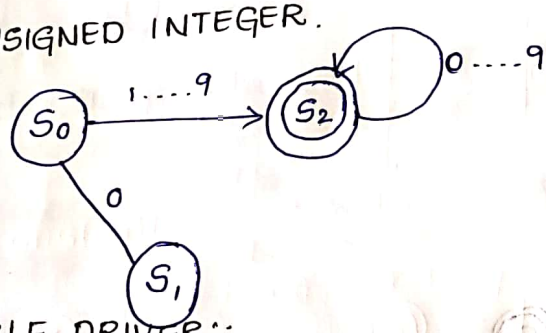
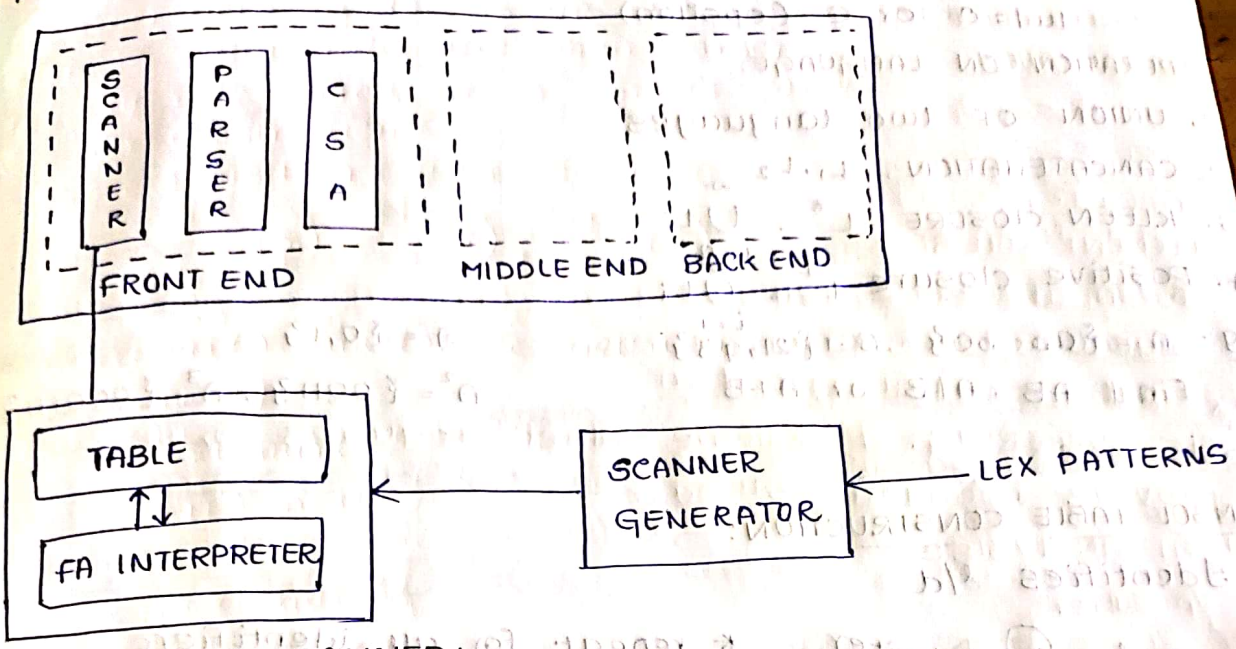


TABLE DRIVER:

	0	1	2	3	4	5	6	7	8	9	Others
S_0	S_1					S_2					Se
S_1						Se					
S_2						S_2					
S_3						Se					

2020.01.20 13:25

AUTOMATIVE SCANNER GENERATION (Table)



DIRECT CODED SCANNER:-

USING RE:- - Regular Expression

- pattern matches a RE then token is recognized.

FEW NOTATION:

1. Alphabet - Set of symbols - Σ

$\Sigma = \{0,1\}$

$\Sigma = \{0,1,2,\ldots 9\}$

$\Sigma = \{A-Z\}$

$\Sigma = \{a-z, A-Z, 0-9\}$

2. String - Σ^* - Sequence of numbers from alphabets.

$\Sigma = \{a,b\}$

$\Sigma^* = \{a, b, ab, aab, abb, \ldots\}$

$\Sigma = \{a,b,c\}$

$\Sigma^* = \{ab, bc, ac, aa, bb, cc, \ldots\}$

3. Length of the string $|w|$

→ Total number of strings

$w = abc$

$|w| = 3.$

2020.01.20 13:25

4. Empty word – without any symbol
$|w| = 0$, or ϵ (epsilon)

OPERATION ON LANGUAGE:

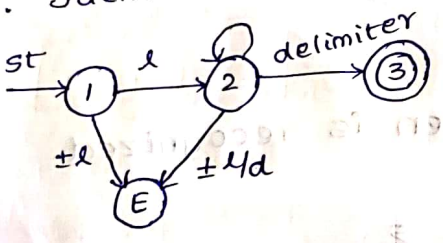
1. UNION OF two languages
2. CANCATENATION $L_1 . L_2$
3. KLEEN CLOSURE $L^* = \bigcup_{i=0}^{\infty} L^i$
4. positive closure $L^+ = \bigcup_{i=1}^{\infty} L^i$

Eg:- $A = \{ab, bc\}$ $B = \{21, 43\}$
Find AB , A/B or A+B

$A = \{0,1\}$
$A^2 = \{00,11\}$, $A^3 = \{000,111\}$

SYMBOL TABLE CONSTRUCTION:

1. Identifies l/d



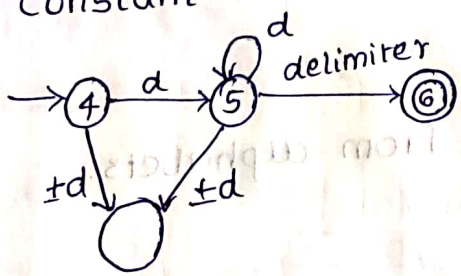
* repeat for all identifiers

return (1, install())
↓
in symbol Table

S.T

TOKEN	code	value
identifier	1	ptr. to ST

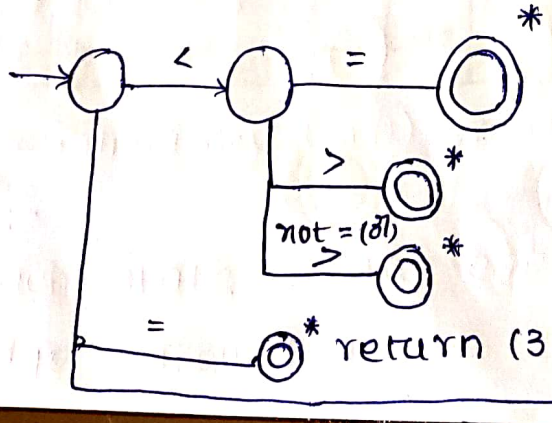
2. Constant



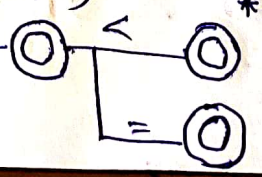
return (2, install())

CONSTANT	2	Ptr to S.T

RELATIONAL OPERATIONS:



return (3,1)
return (3,2)
return (3,3)
return (3,4)



return (3,5)
return (3,6)

S.T | TOKEN

KEYWORDS

WRITE

1. al
2. (a
3. e
4.
5.

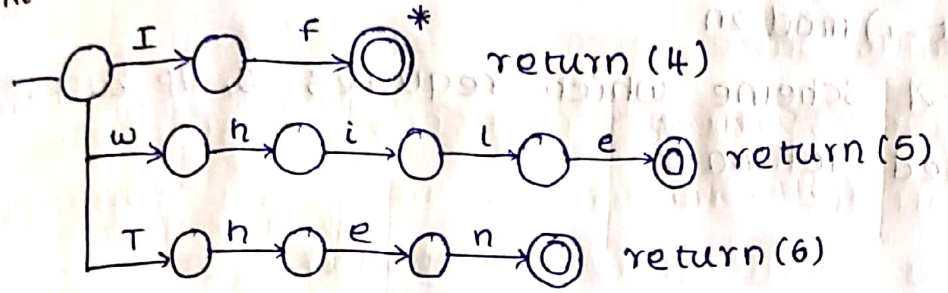
Ins
ca
bu
OF

2020.01.20 13:25

S.T

TOKEN	CODE	VALUE
<=	3	1
<>	3	2
<	3	3
=	3	4
>	3	5
>=	3	6

KEYWORDS:



TOKEN	CODE	V
IF	4	-
WHILE	5	-
THEN	6	-

WRITE STRINGS:

1. $a|ba^*$: {a, b, aa, ab, baaa, ----}
2. $(a|b)(a|b)(a|b)$: {aaa, bbb, abb, bbb, bab, ----}
3. $\epsilon|a|b$: {ϵ, a, b}
4. $(a|b)\ (a|b)\ (a|b)^*$: {aa, bb, aaa----}
5. $\bigcup_{i=0}^{\infty} (a,b)^i$ (Kleene closure) = {ϵ, aa, bb, ----}

HAND CODED SCANNER:

Instead of table driven and direct coded scanners, calling the procedures overhead is reduced by using buffered I/O. i.e., Each read operation returns a string of characters or buffers and the scanner indexes through the buffer. i.e., the scanner maintains a pointer into the buffer and the next character function keeps the buffer filled and traps the current location in the buffer.

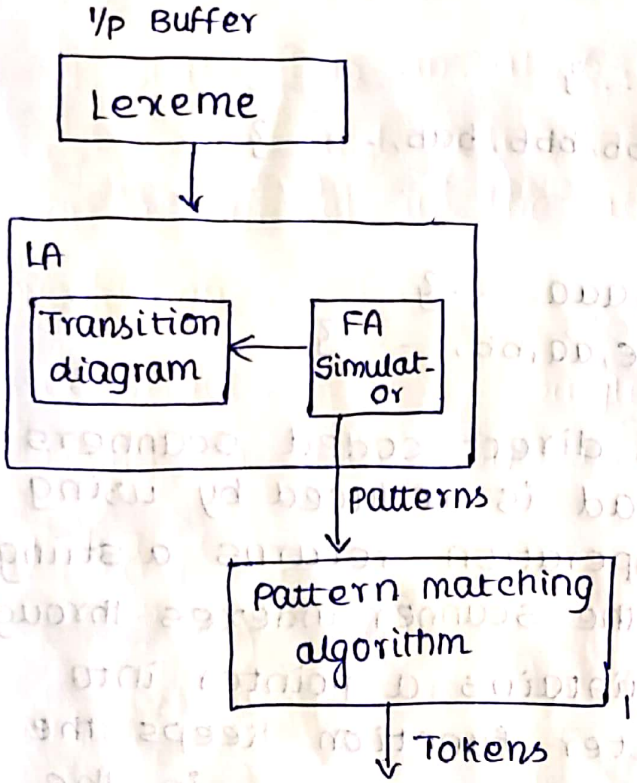
IMPLEMENTING NEXTCHAR & ROLLBACK:

```
Char ← Buffer[input];
input ← (input + 1) mod 2n
IF (input mod n == 0) then
 begin
    fillbuffer[input : input + n - 1]; Fence ← (input + n) mod 2n
```

```
    end
  return char;
INITIALIZATION:
   input ← 0
   fence ← 0 ;
   fillbuffer [0:n];
IMPLEMENT ROLL BACK:
  IF (input = fence) then
    signal rollback error,
    input ← (input-1) mod 2n
```

imp 2M DOUBLE BUFFERING: A scheme which requires 2 i/p buffers to provide bounded roll back.

LEXICAL ANALYSIS:-



i/p buffer → lexemes

LA builds RE for tokens

from RE → FA is built

Lexeme & pattern matched to recognize tokens

DESIGN OF LA INVOLVES 3 TASKS:

1. Representation of tokens as RE
2. representation of RE by FA or FSM
3. Simulation of FA

FA:-

* Deterministic (DFA) or Non Deterministic (NFA).

DFA:-

* Each state s, for every input $a \in \Sigma$, at most one edge labelled a.

FEATURES:-

1. One start state
2. One or more acceptors state
3. No null (∈) Transition
4. One transition on a particular symbol.

Eg: a^*

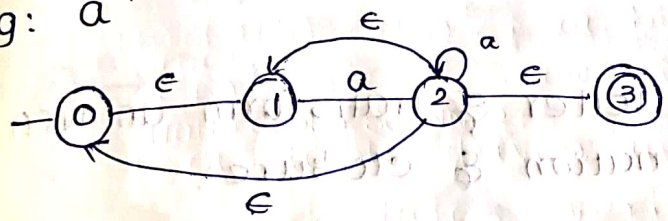
NFA:-

* Start state, few or more Final state and for single input their, can be multiple transition.

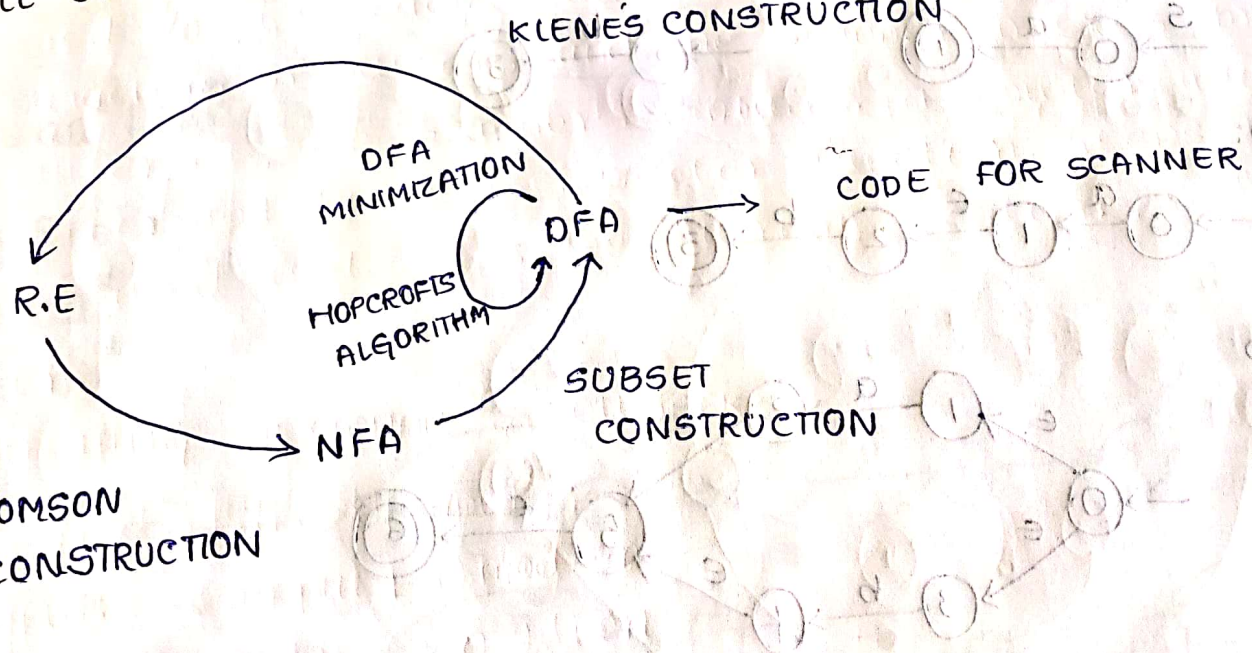
FEATURES:-

1. One start state
2. One or more accepting state
3. can have '∈' transition
4. one or more transition for a input symbol.

Eg: a^*



CYCLE OF CONSTRUCTION FINITE AUTOMATA:-



2020.01.20 13:25

R.E to NFA [Thomson]:-

* Thomson construction moves to connect NFA's for sub expressions.
* NFA derived from this construction which has several useful properties.

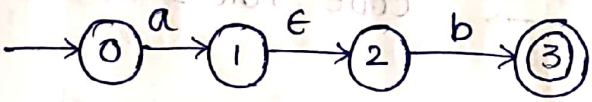
1. Each NFA has a single start state and a single final state. The only transition that enters the start state is the initial transition. it is the one to leave the final state.
2. An epsilon (ϵ) moves always connects 2 states that were earlier in the process i.e., the start state of a process and a final state of another
3. A state has an atmost 2 entering & 2 existing 'ϵ' moves and atleast 1 entering & existing move on a symbol in the alphabet.

CONSTRUCTION:-

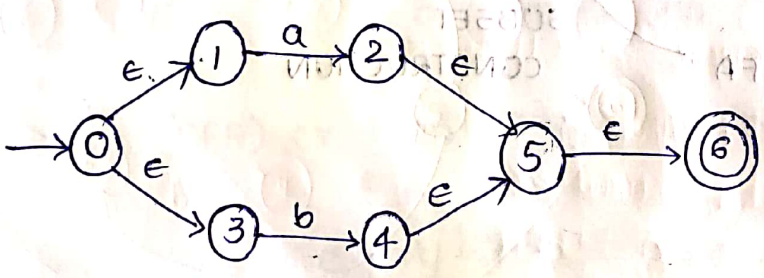
* Construct NFA for single letter & transform an NFA using cancatenation, alternation & closure.

Eg:- 1. a.b

NFA:



2. a/b



1. construct NFA to a(b|c)*

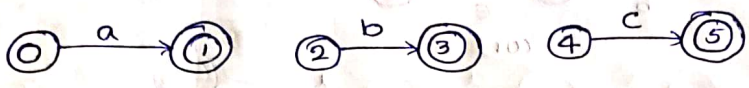
a(b|c)*

R = R1 . R2

R1 = a

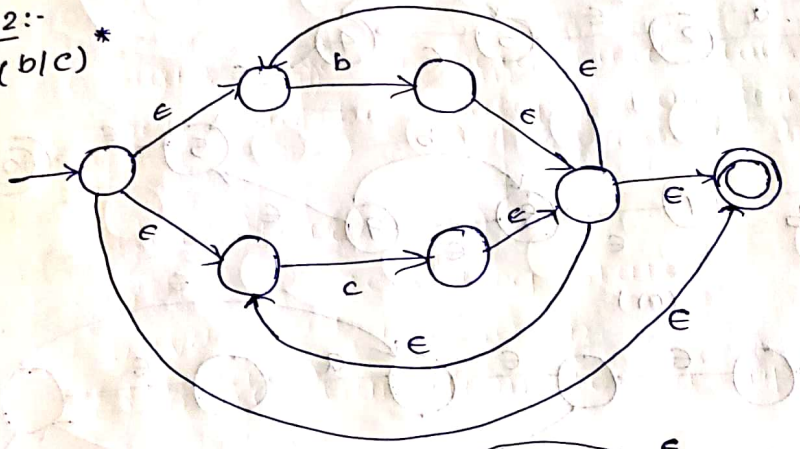
R2 = (b|c)*

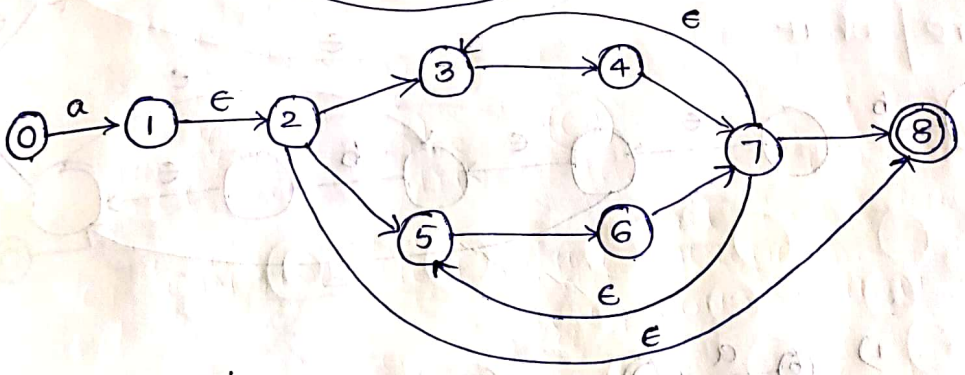
ST:1:



ST:2:-

(b|c)*





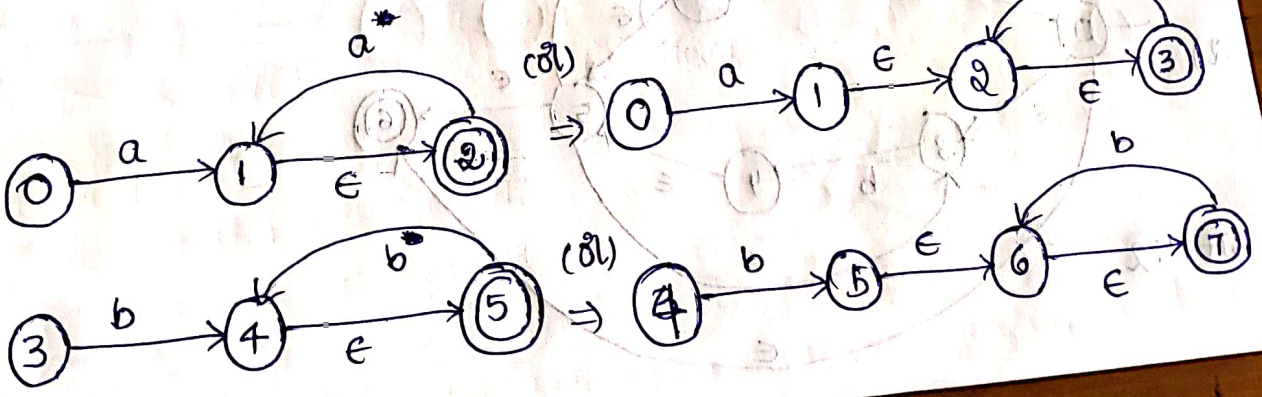
2. R = aa* | bb*

R = R1 | R2

R1 = aa*

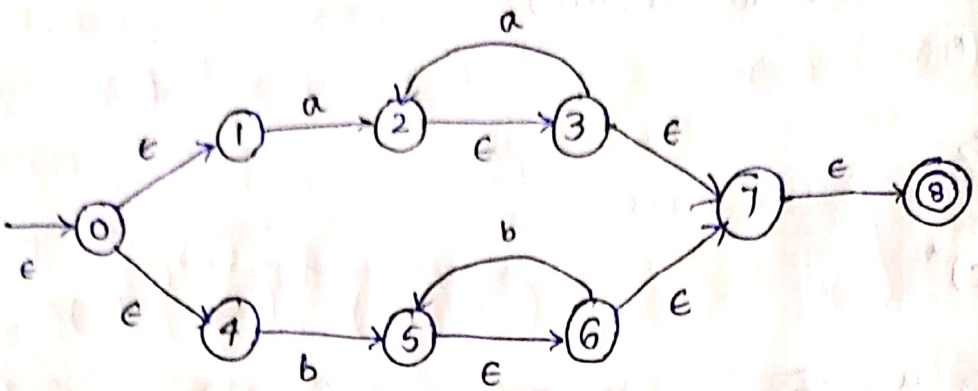
R2 = bb*

ST:1

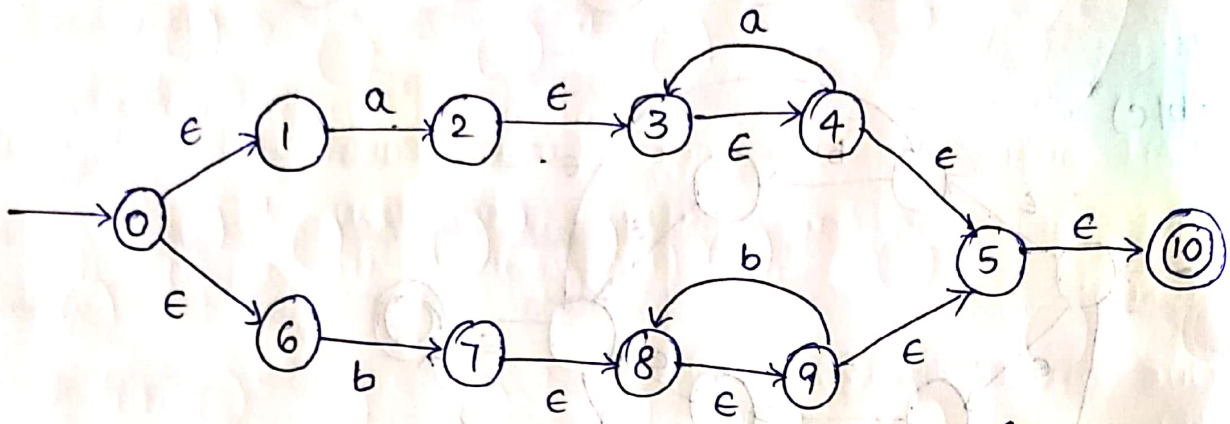


2020.01.20 13:25

ST:2

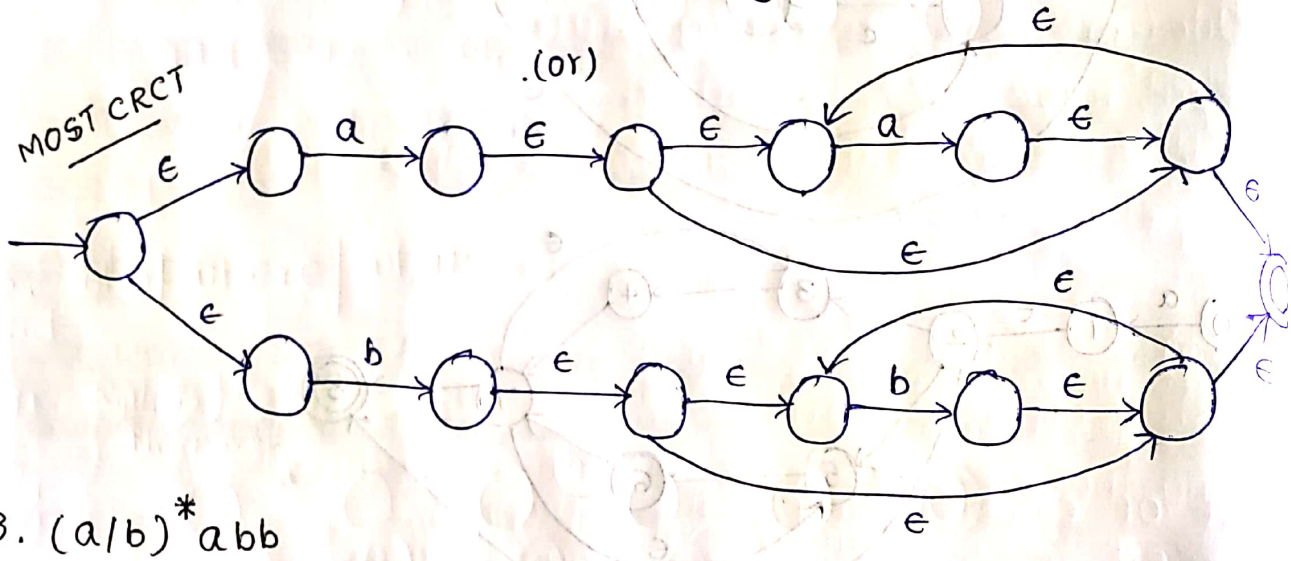


(or)



.(or)

MOST CRCT



3. $(a/b)^*abb$

$R = R_1 . R_2 . R_3 . R_4$

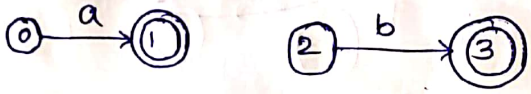
$R_1 = (a/b)^*$

$R_2 = a$

$R_3 = b$

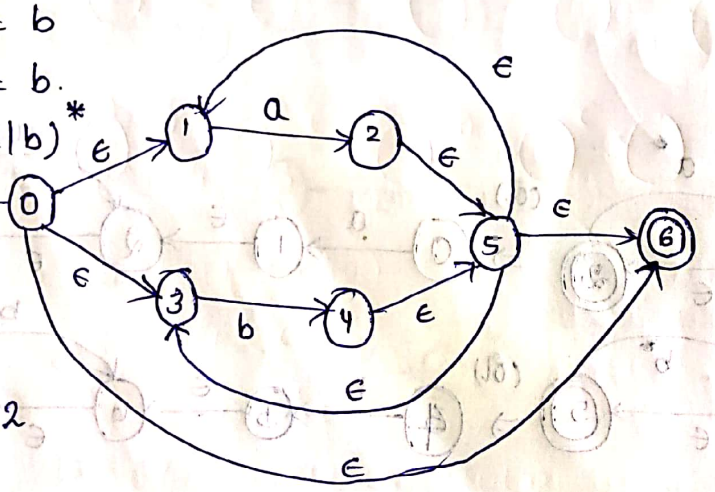
$R_4 = b.$

1)



$R_1 = (a/b)^*$

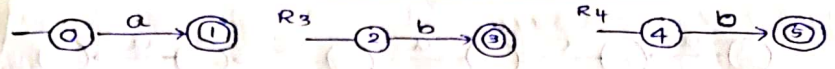
2)



STEP:2

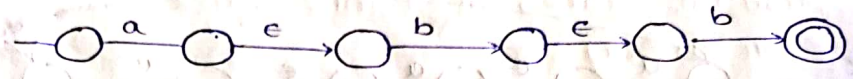
STEP: 1

R_2 =



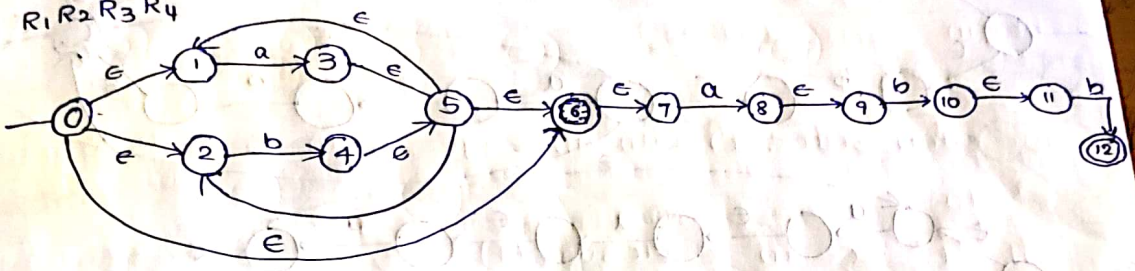
$R = R_1 R_2 R_3 R_4$

$R_2 R_3 R_4$ =



$R = R_1 R_2 R_3 R_4$

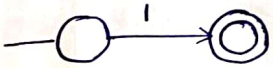
=

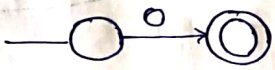


4. $1(0+1)^* 0$

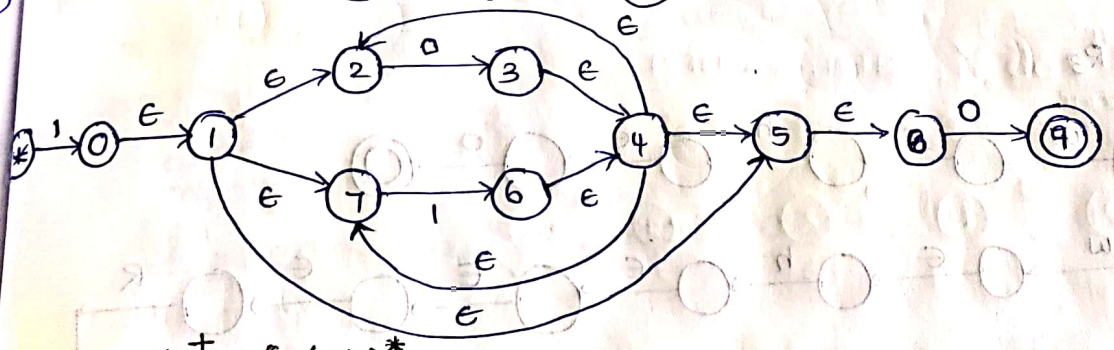
$= 1(0|1)^* 0$

$R = R_1 R_2 R_3$

R_1 = 

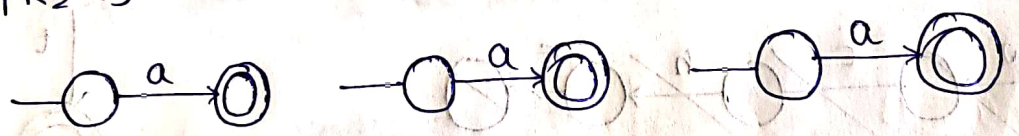
R_3 = 

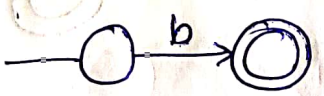
R_2 = 

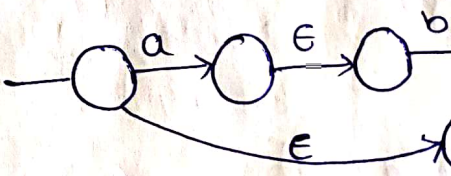


5. $(aaa)^+ \cup b(ab)^*$

$R = R_1 R_2 R_3$

R_1 = 

R_2 = 



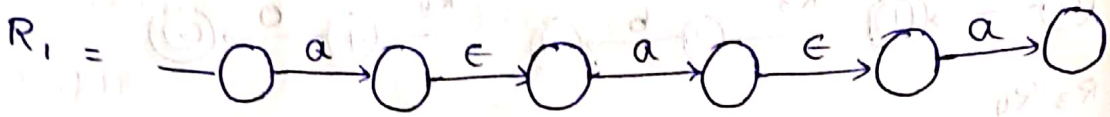
$R_3 = (ab)^*$

= 

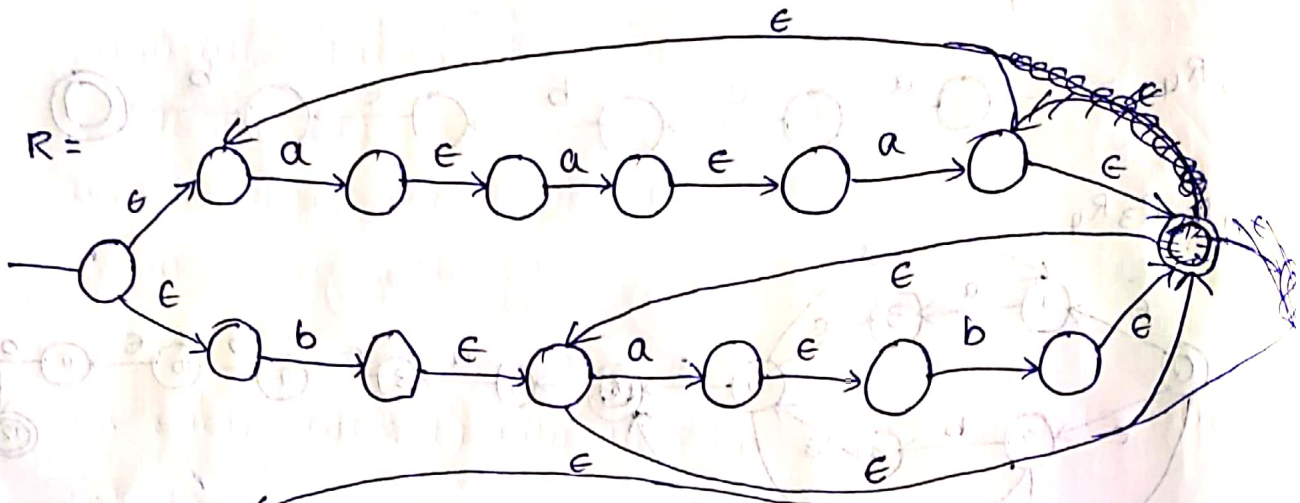
2020.01.20 13:25

+, U, sam

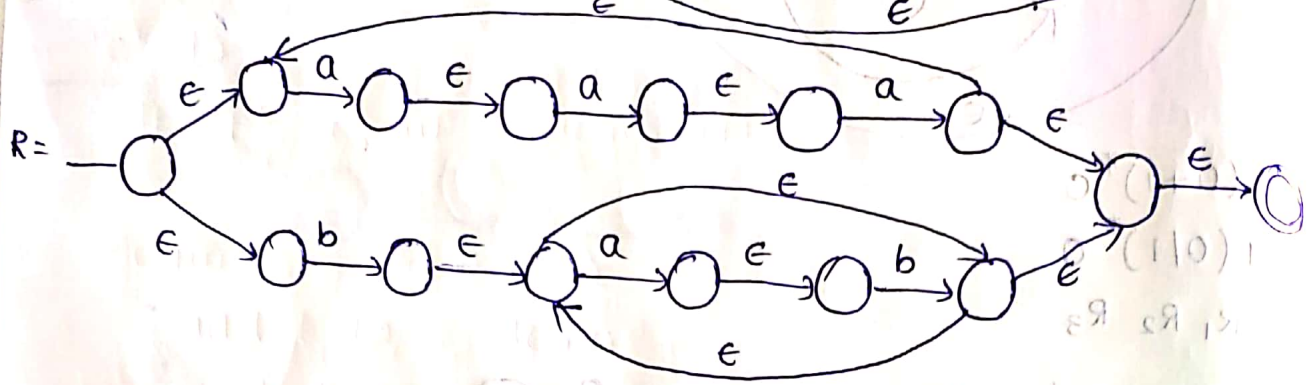
R_1 =



R =



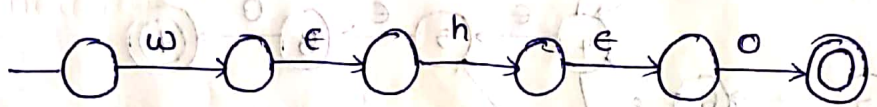
R =



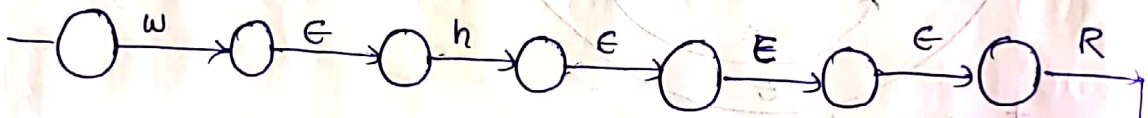
6. who | where | what

$R_* = R_1 \mid R_2 \mid R_3$

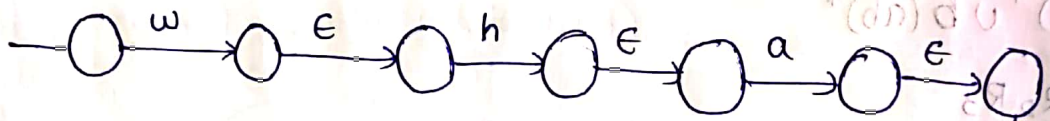
R_1 =



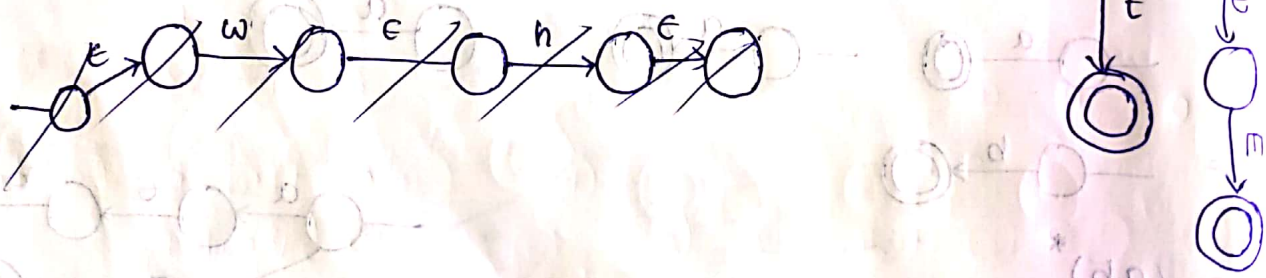
R_2 =

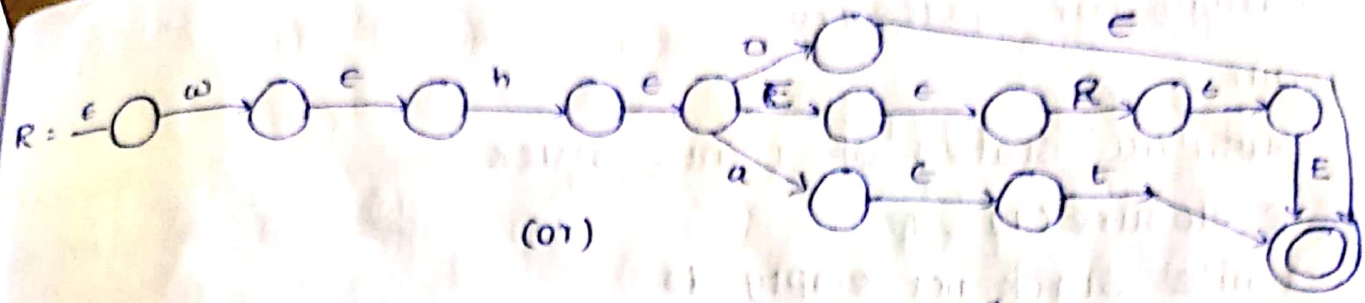


R_3 =

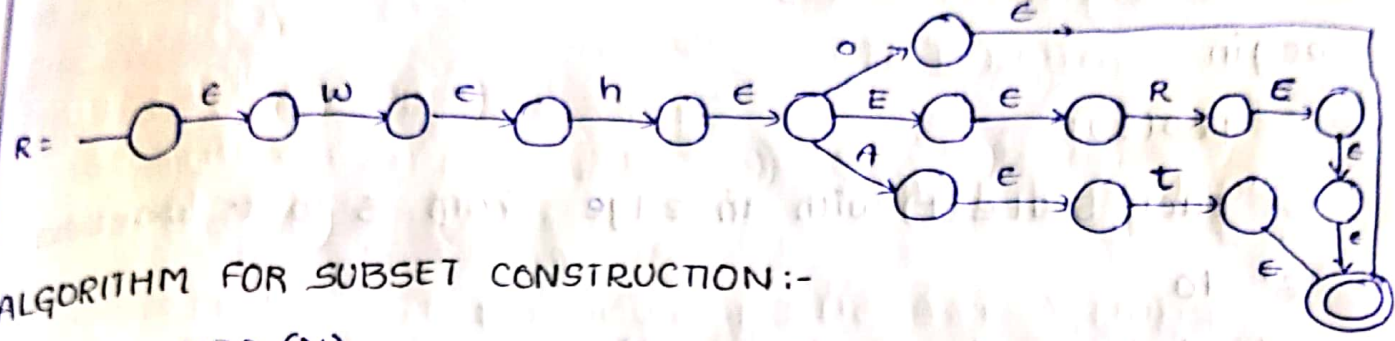


R =





(or)



ALGORITHM FOR SUBSET CONSTRUCTION:-

Input : NFA (N)

output : DFA (D)

Accepting the same language with the initial state of 'D' is the set of ϵ_closure(s_0)

The other states are constructed as follows.

we assume each state of 'N' as unwanted and perform the following.

$x = \{s_1, s_2, \ldots\ldots, s_n\}$ of D do

begin

~~while~~

mark x

for each i/p symbol 'a' do

begin

Let 't' be the set of symbol to which there is a transition 'a'. for some state s_i in x.

y = ϵ_closure(T);

If 'y' is not added to the set of states of b then make "unmarked state of b"

Add a transition for x to y.

end

end

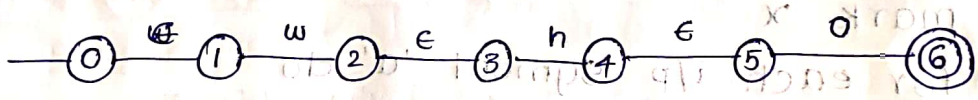
Algorithm for ϵ-closure:

Begin
push all states of T into stack
ϵ-closure (T) = T
while stack not empty do
begin
pop(s);
for each state(t) with an edge from s to t labelled 'ϵ' do
if 't' not in ϵ-closure (T)
do
begin
add t to ϵ-closure
push t on to stack
end
end
end

Eg:- $(a|b)^*abb$ NFA ⟶ DFA

Next page

eg:



STEP:1 : ϵ -

{0,1} = A (initial states)

STEP:2 :

A ⟶ w ⟶ ϵ

{2,3} = B

STEP:3:

B ⟶ H ⟶ ϵ

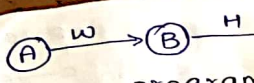
{4,5} = C

STEP:4:

C ⟶ O ⟶ ϵ

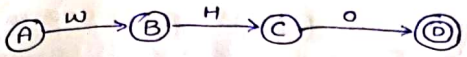
{6} = D (final state

A	w, H	B
B		C
C		D
D		



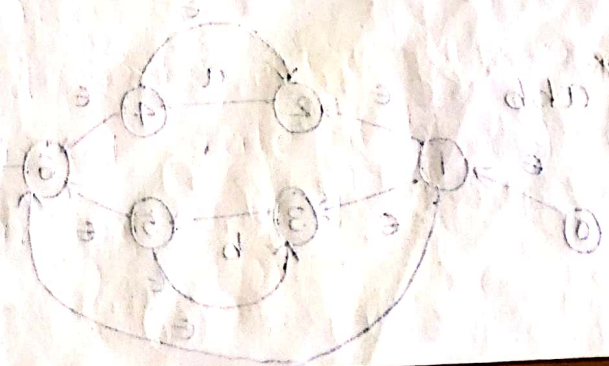
Write a progran
regular expres
include < sta
include < str
int q[30][3]
int i,j,k,x,op
void print_NFA();
void main()
{ char reg
int a,b,
for(i=0
{
for(j=
{
q
}
}
printF
scanf
len =
i=j=0
while
{
if
{

2020.01.20 13:26



Write a program for thomson (NFA) construction for converting a regular expression $(a/b)^*abb$

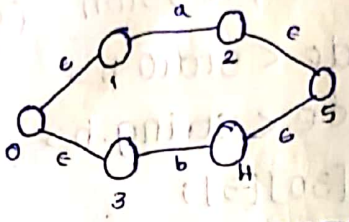
```
# include < stdio.h>
# include < string.h>
int  q[30][3];
int i,j,k,x,open;
void print_NFA();
void main()
{
   char reg[20];
   int a,b,len;
   for(i=0 ; i<30 ; i++)
   {
     for(j=0;j<3 ;j++)
     {
        q[i][j] = -1;
     }
}
printF("ENTER REGULAR EXPRESSION:");
scanF("%s",reg);
len = strlen(reg);
i=j=0;
while (i<len)
{
  if (reg[i] =='a' && reg[i+1]=='|'&& reg[i+2] =='b')
  {
     q[j][2] = ((j+1) *100)+(j+3);
     j++; =1
     q[j][0] = j+1;
     j++;=2
     q[j][2] = j+3;
     j++;=3
```



2020.01.20 13:26

```
        q[j][1] = j+1;
        j++;
        q[j][2] = j+1;
        j++;
        i = i+2;
    }
    print_NFA();
}
void print_NFA()
{
    printf(" TRANSITION FUNCTION ");
    for(i=0; i<=j; i++)
    {
        if(q[i][0] != -1)
            printf("\n q[%d,a] -> %d", i, q[i][0]);
        if(q[i][1] != -1)
            printf("\n q[%d,b] -> %d", i, q[i][1]);
        if(q[i][2] != -1)
        {
            if(q[i][2] < 100)
                printf("\n q[%d,e] -> %d", i, q[i][2]);
            else
                printf("\n q[%d,e] -> %d & %d", i, q[i][2]/100, q[i][2]%100);
        }
    }
}
```

0 → a
1 → b
2 → ε



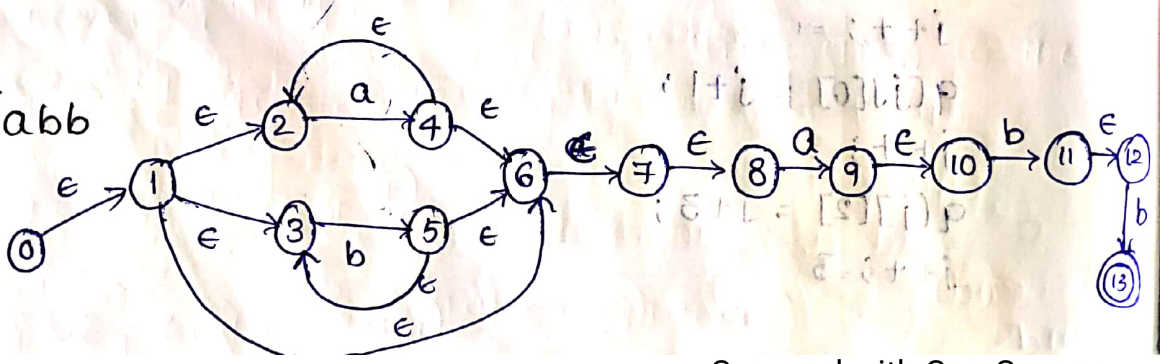
o/p:

ENTER THE REGULAR EXPRESSION: (a|b)

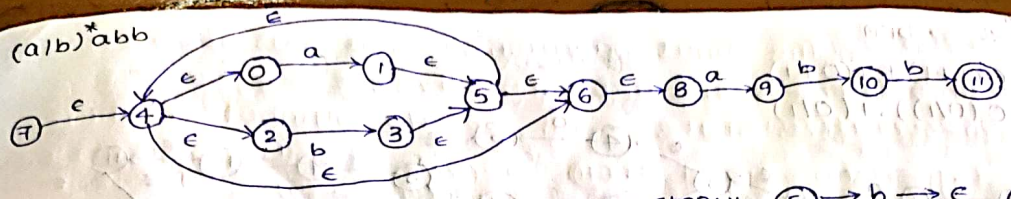
TRANSITION FUNCTION

q[0,e] → 1 & 3
q[1,a] → 2
q[2,e] → 5
q[3,b] → 4
q[4,e] → 5

(a|b)*abb



$(a|b)^*abb$



step:1: T → Start State

T → ε {4,0,2,6,8} (A)

step:2: (A) → a → ε

{0,2,5,6,1,4,8,9} (B)

step:3:-

(A) → b → ε

{3,5,6,8,4,0,2} (C)

step:4:-

(B) → a → ε

{1,5,6,8,4,0,2,9} (B)

step:5:-

(B) → b → ε

{3,5,6,8,4,0,2,10} (D)

step:6:-

(C) → a → ε

{5,0,1,2,6,8,9,4} (B)

step:7:-

(C) → b → ε

{5,0,2,3,6,8,4} (C)

step:8:-

(D) → a → ε

{5,6,8,9,4,0,2,1} (B)

step:9:- (D) → b → ε

{5,6,8,4,0,2,3,11} (F)

step:10:-

(E) → a → ε

{5,4,0,2,1,6,8,9} (B)

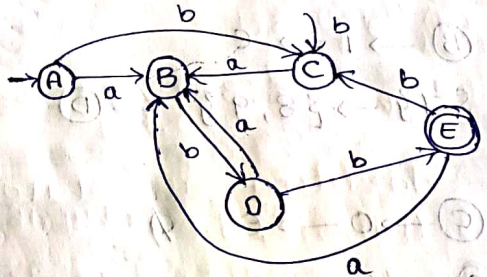
step:11 (E) → b → ε (C)

{5,4,0,2,3,6,8}

'E' is the final state

TRANSITION TABLE

δ	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C



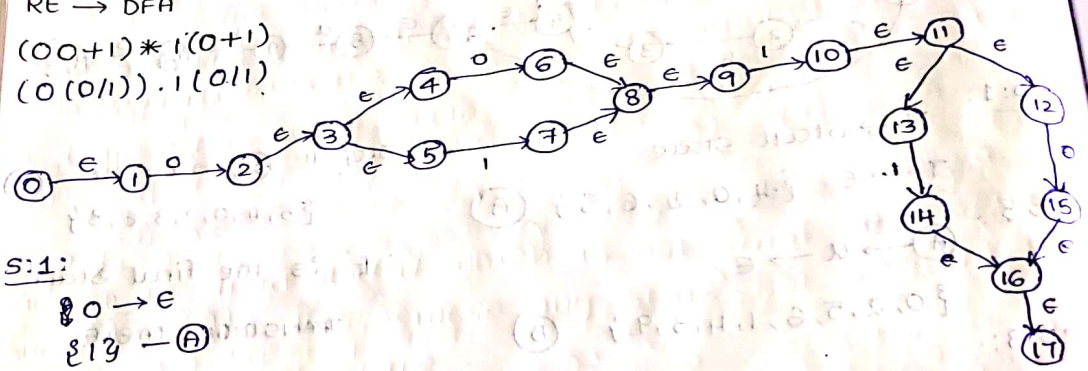
Next problem in last page

2020.01.20 13:26

RE → DFA

(00+1)*1(0+1)

(0(0/1))·1(0/1)



S:1:

{0} → ε

{1} → (A)

S:2:- (A) → 0 → ε

{2} → {3,4,5} → (B)

(A) → 1 → ε ∅

S:3:- (B) → 0 → ε

{6} → {8,9} → (C)

(B) → 1 → ε

{7} → {8,9} → (D)

S:4:

(C) → 0 → ε ∅

(C) → 1 → ε

{10} → {11,12,13} → (E)

S:5:

(D) → 0 → ε ∅

(D) → 1 → ε

{10} → {11,12,13} → (E)

S:6:- (E) → 0 → ε

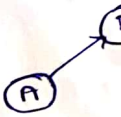
{15} → {16,17} → (F)

(E) → 1 → ε

{14} → {16,17} → (G)

S:7:- (F) → (F)

DFA:-



MINIM

Findi

disti

sta

into

AL

A

S

D

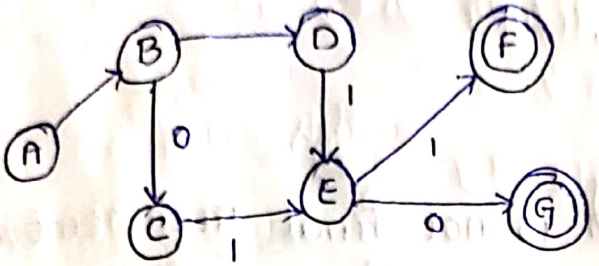
S

2020.01.20 13:26

S:7:- (F) → 0 → E ∅
(F) → 1 → E ∅

S:8:- (G) → 0 → E ∅
(G) → 1 → E ∅.

DFA:-



δ	0	1
A	B	-
B	C	D
C	-	E
D	E	E
E	F	G
F	-	-
G	-	-

MINIMIZATION OF DFA:-

Minimizing the no:of states of DFA works by finding the all the no:of states that can be distinguished by some input string. those group of states that can not be distinguish are then merge into a single state for the entire group.

ALGORITHM:

A DFA 'M' with a set of states 'S' and initial states 'S0', the set of final states Sn the o/p is a DFA minimized accepting the same language but having few states as possible.

S:1:We construct the partition 'Π' of set of states with two groups, i.e., Final state 'F' and non-final state 'S_F'. Now, we construct a new partion Π_{new} by using the following procedure.

for each group G(Π) do
Begin
partition G into subgroups such that 2 states 'S' & 'T' are in the same group iff for all input symbols S&T have same transition place the so formed subgroup as Π_{new}.

STEP:2: In the final partition, π that has been constructed in step:1 pick one representative. Let s be the representative state & for the i/p symbol if there a transition on M from $s \rightarrow T$ then it is replaced by s.

STEP:3

If 'm' is dead state which is not final then remove such a state. Also, remove any state not reachable from the initial state.

Eg:- $(a|b)^*abb$

	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

S:1 → Identify all states

{A, B, C, D, E}

S:2 → Separate final & non-final states

{A, B, C, D} {E}

S:3 → Identify any state that has transition from or to final state, then separate.

{ABC} / {DE}

S:4 → Identify the state that has transition to the final states.

{AC} {BDE}

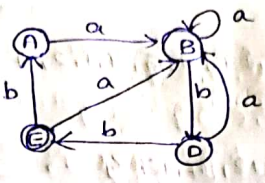
S:5 → C ⇒ A (should be replaced)

been construc
be the
if there
s replaced

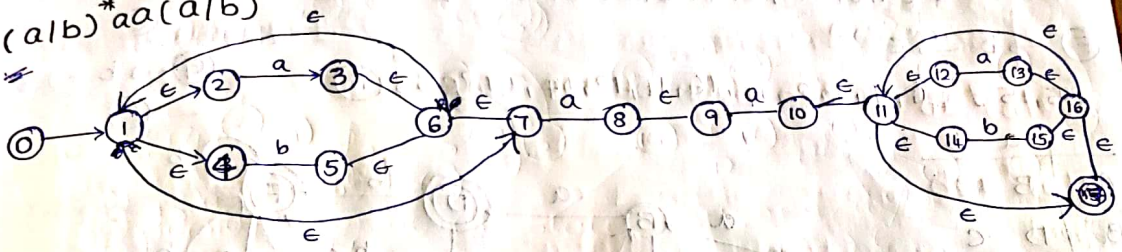
en remove
achable

or to

	a	b
A	B	A
B	B	D
D	B	E
E	B	A



$(a|b)^*aa(a|b)^*$



$0 \rightarrow \epsilon$

$\{0,1,6,7,2,4\}$ — (A)

(A) $\rightarrow a \rightarrow \epsilon$

$\{3,8\}$ $\{6,7,1,2,4,9\}$ — (B)

(A) $\rightarrow b \rightarrow \epsilon$

$\{5\}$ $\{6,7,1,2,4\}$ — (C)

(B) $\rightarrow a \rightarrow \epsilon$

$\{8,3,10\}$ $\{9,6,7,1,2,4,11,12,14,17\}$ — (D)

(B) $\rightarrow b \rightarrow \epsilon$

$\{5\}$ $\{6,7,1,2,4\}$ → (C)

(C) $\rightarrow a \rightarrow \epsilon$

$\{3,8\}$ $\{6,7,1,2,4,7\}$ → (B)

(C) $\rightarrow b \rightarrow \epsilon$

$\{5\}$ $\{6,7,1,2,4\}$ → (C)

(D) $\rightarrow a \rightarrow \epsilon$

$\{10,8,3,13\}$ $\{11,12,14,17,9,6,7,1,2,4,16,17\}$ → (E)

(D) $\rightarrow b \rightarrow \epsilon$

$\{5,15\}$ $\{6,7,1,2,4,16,11,12,14,17\}$ → (F)

2020.01.20 13:26

(E) → a → ϵ

{13,10,8,3} {16,11,12,14,17,9,6,7,1,2,4} → (E)

(E) → b → ϵ

{15,5} {6,7,1,2,4,16,11,12,14,17} → (F)

(F) → a → ϵ

{8,3,13} {9,6,7,1,2,4,16,11,12,14,17} → (G)

(F) → b → ϵ

{5,15} {6,7,1,2,4,16,11,12,14,17} → (F)

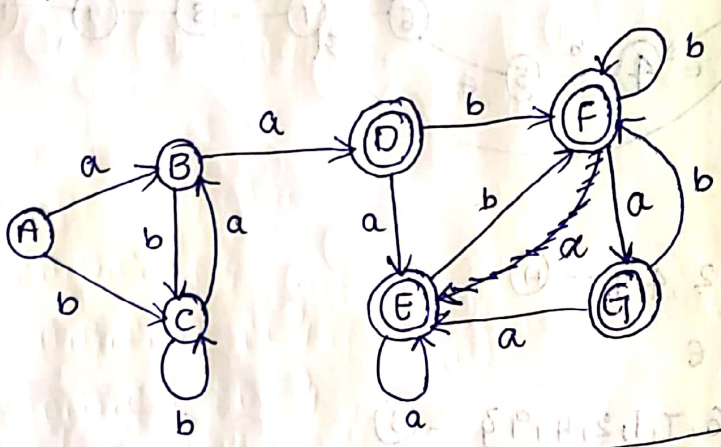
(G) → a → ϵ

(E)

(G) → b → ϵ

(F)

	a	b
A	B	C
B	D	C
C	B	C
D	ϵ	F
E	ϵ	F
F	G	F
G	E	F



MINIMIZATION:

{A,B,C,D,E,F,G}

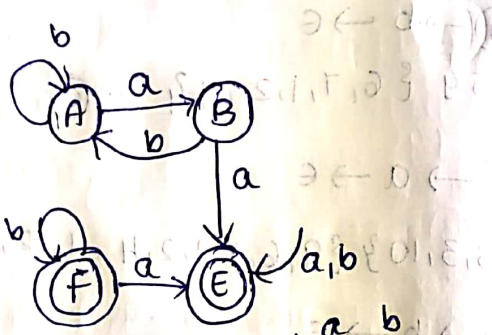
{A,B,C} {D,E,F,G}

{A,C} {B,D,E,F,G} ⇒ {A}{B,D,E,F,G}

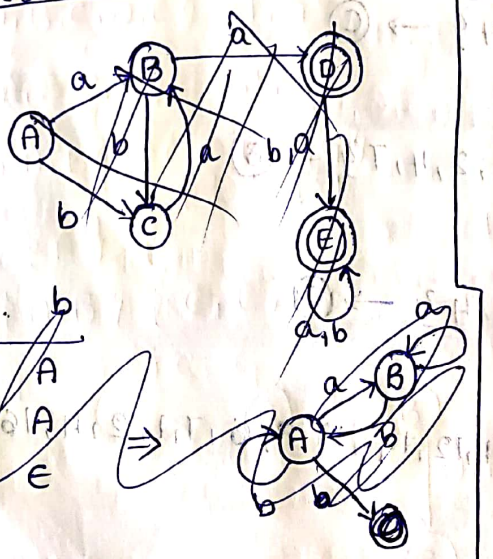
{A}{B,E,F}

A ≡ C

	a	b
A	B	A
B	E	A
E	ϵ	ϵ
f	E	f

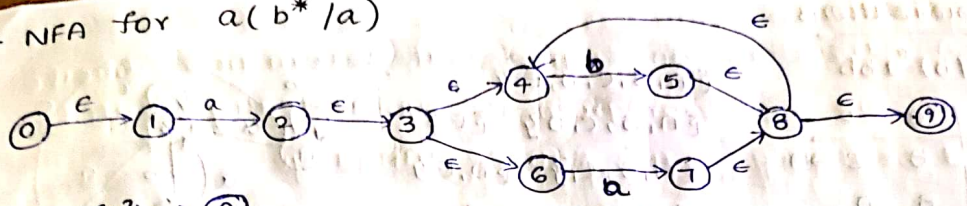


	a	b
A	B	A
B	E	A
E	ϵ	ϵ
f	E	f



	a	b
A	B	A
B	E	A
E	ϵ	ϵ

* NFA for a(b*/a)



0 → {1} → (A)

(A) → a → ε

{2} {3,1,6} → (B)

(A) → b → ε ∅

(B) → a → ε

{7} {8,4,9} → (C)

(B) → b → ε

{5} {8,4,9} → (D)

(C) → a → ε ∅

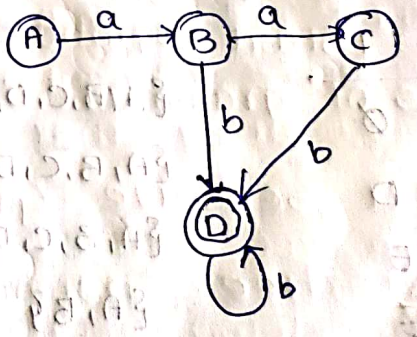
(C) → b → ε

{5} {8,4,9} → (D)

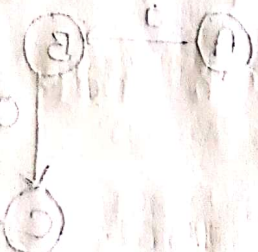
(D) → a → ε ∅

(D) → b → ε

{5} {8,4,9} → (D)



δ	a	b
A	B	–
B	C	D
C	–	D
D	–	D



2020.01.20 13:26

minimizations:

$(a/b)^*abb$

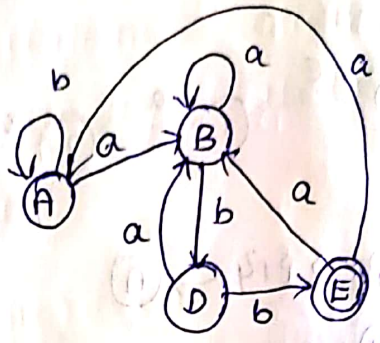
δ	a	b
A	B	C → A
B	B	D
C	B	C
D	B	E
E	B	C → A

{A, B, C, D, E}

{A, B, C, D} {E}

{A, B, C} {D, E}

{A, B} {D, E}



$a(a/b)^*ab$ → See Back of the notes

$(a/b)^*abb$

Repeat

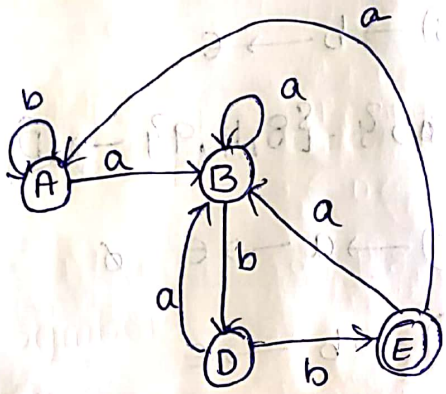
δ	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

{A, B, C, D, E}

{A, B, C, D} {E}

{A, B, C} {D, E}

{A, C} {B, D, E}



$(00+1)^*1(0+1)$

δ	0	1
A	B	∅
B	C	D
C	∅	E
D	∅	E
E	G	F
F	∅	∅
G	∅	∅

{A, B, C, D, E, F, G}

{A, B, C, D, E} {F, G}

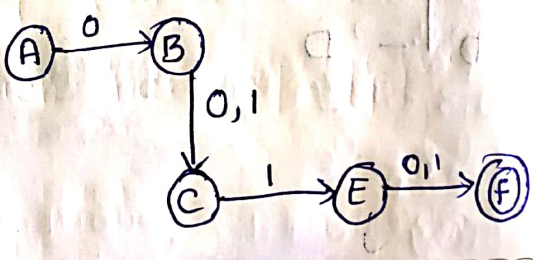
{A, B, C, D} {E, F, G}

{A, B} {C, D, E, F, G}

~~{A, B}~~ {

{A, B} {C, E, F}

δ	0	1
A	B	∅
B	C	C
C	∅	E
E	~~G~~	F
F	∅	∅



DFA to RE

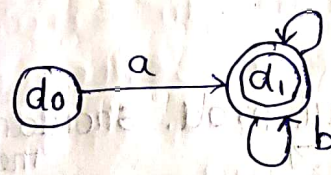
* Kleen's closure construction builds a RE to describe a set of strings generated by the Language.
* corresponds to path problems

KLEEN'S

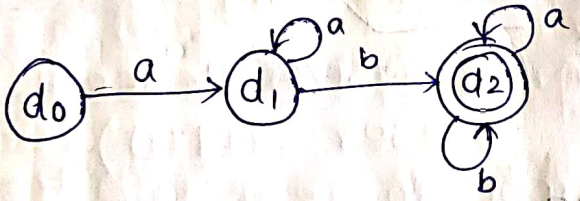
```
for i=0 to |D|-1
  for j=0 to |D|-1
    Rij = { a | δ(di,a) = dj }
      IF (i=j) then
        Rij = Rij ∪ {ε}

for k=1 to |D|-1
  for i=0 to |D|-1
    for j=0 to |D|-1
      R_ij^k = R_ik^(k-1) (R_kk^(k-1))* R_kj^(k-1) R_ij^(k-1)
```

* Algorithm uses states numbered from 0 to |D|-1
* Systamatically, constructs a path expression.

Given DFA denoted by

1. 

→ $a.(a|b)^*$

2. 

→ $aa^*b(a|b)^*$

PARSERS:

A parser is an engine that determines whether or not the input program is syntactically (valid) valid sentence in the source language.

* For one programming languages, there can be many compilers.
* To write efficient parsers, we need a formal mechanism for specifying the syntax of the input language.
* The systematic method of the membership in this formally specified language.

GRAMMAR:-

A grammar is a four tupled one. G, where

$G = \{ V_n, V_T, P, S \}$

$V_n \rightarrow$ set of non terminal symbols
$V_T \rightarrow$ set of Terminal symbols
$P \rightarrow$ set of productions
$S \rightarrow$ start symbol.

* 3 types of grammar

1. Context Sensitive Grammar (Null prod. shouldn't be there)
2. Context free Grammar
3. Regular Grammar

CONTEXT SENSITIVE GRAMMARS:-

A CSG given by G, where $G = \{ V_n, V_T, P, S \}$, if for every production, $\alpha \rightarrow \beta$ such that $|\alpha| \leq |\beta|$

CONTEXT FREE GRAMMARS:-

In CFG, where $G = \{V_n, V_T, P, S\}$ for every production rule $\alpha \rightarrow \beta$, it should follow the below instructions.

1. 'α' is a single non terminal.
2. β shouldn't have (ϵ) transitions. ($\beta \neq \epsilon$)

REGULAR GRAMMARS:-

A Regular Grammar $G = \{V_n, V_T, P, S\}$, if every rule P is of the form $A \Leftrightarrow aA$, $A \in a$

'A' is a set of terminal symbols.

* If a language has a set of strings given by $\{a, aa, \ldots\}$ then grammar is given by $G = \{V_n, V_T, P, S\}$

P →

$S \rightarrow A$

$\{A \rightarrow aA$
$\quad | a$

$\}$

$V_n \rightarrow \{A\}, \{S\} \Rightarrow \{S, A\}$

$V_T \rightarrow \{a\}$

* Generate a grammar for a language given by

$\{a^n b^n \mid n \geq 1\}$

$\{ab, aabb, \ldots\}$

$G = \{V_n, V_T, P, S\}$

P ⇒ $S \xrightarrow{A; A \rightarrow} aAb \mid ab$

$V_n \Rightarrow \{S, A\}$

$V_T \Rightarrow \{a, b\}$

* Generate a grammar for a language given by

$\{a^n b^m \mid n \geq 1, m \geq 1\}$

$V_n = \{S, A, B\}$

$V_T = \{a, b\}$

$S \rightarrow A$

$A \rightarrow aAb \mid B$

$B \rightarrow bB \mid b$

2020.01.20 13:26

* Generate a grammar for PALINDROME (wcw^R)

wcw^R

$G = \{V_n, V_T, P, S\}$

$S \rightarrow aSa / bSb / cSc / a / b / c$

$V_n = \{S\}$

$V_T = \{a, b, c\}$

* Generate a grammar for (((---)))

$G = \{V_n, V_T, P, S\}$

~~$S \rightarrow (S) / ()$~~

$S \rightarrow A$

$A \rightarrow (A) / ()$

$V_n = \{S, A\}$

$V_T = \{(,)\}$

* Generate a grammar for arithmetic operations.

$S \rightarrow S+S \mid S-S \mid S*S \mid S/S$

$V_n = \{S\}$

$V_T = \{+, -, *, /\}$

DEF

* A parser for Grammar G, is a program that accepts an i/p string w, & produces either the parse tree for w, if 'w' is a sentence of G, (or) an error message indicating that 'w' is not a sentence of G.

* There are two approaches for parsing

1. Top down approach (Recursive descent parser) → (LL(1)) Left to right lookup
   - ↳ checking from root to leaf.
2. Bottom up approach:
   * Shift reduce parser, left recursive parser (LR parser)

* The pro
a specific
NOTE:
Most
ed with
TOP DOW
LL(1):
large
backtr
In
conve
gra
* H
PAR
*

2020.01.20 13:26

* The process of constructing a derivation from a specific input sentence is called PARSING.

NOTE:

Most programming languages can be constructed with LR(1) parser (or) LL(1) parser.

TOP DOWN APPROACH: ↓

LL(1): The key to top down approach is from a large set of CFG, it can be passed without backtracking. Hence,

In top down approaches, it transforms (or) convert the grammar into the backtrack free grammar.

* Hence, it is otherwise known as PREDECTIVE PARSER. (or) BACKTRACKFREE PARSER.

* There are 2 common topdown process

1. LL(1) - Left Lookup
2. Hand Coded Recursive Descent parser.

LL(1):

STEPS FOR CONSTRUCTION:

1. Eliminate ambiguity in the grammar.
2. Eliminate Left recursion in the grammar.
3. Eliminate Left factorization in the grammar.
4. calculate 'FIRST' for every production and 'follow' for every non-terminal in the grammar.
5. compute parsing table
6. Finally, check the grammar is LL(1).

2020.01.20 13:26

ELIMINATING AMBIGUITY:-

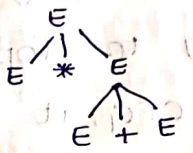
* Consider the following Grammar G,

$E \rightarrow E * E$
$\quad | \; E + E$
$\quad | \; id$

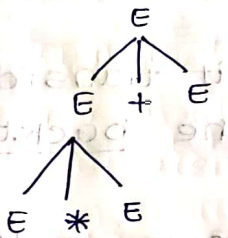
A sentential form $E * E + E$ has 2 derivations

$E \rightarrow E * E \rightarrow E * E + E$

$E \rightarrow E + E \rightarrow E * E + E$



// PARSE TREE



REASONS FOR AMBIGUITY:

No precedence followed.
No associativity

TO REMOVE AMBIGUITY:

Ambiguity can be removed by introducing several different variable each representing a binding string from the grammar.

for eg: For the above grammar, 2 variables can be introduced. i.e., 'F' and 'T', A 'F' is a factor, is an identifier and an paranthesized function and cannot be broken by an operator.

't' is a term by ~~is a~~ one (or) more factor that cannot be broken down by the parser.

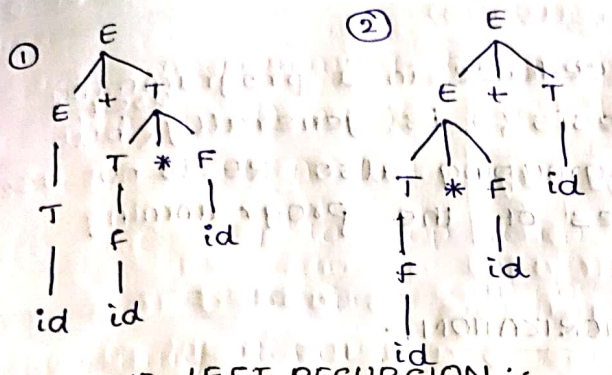
The above grammar can be rewritten as

$E \rightarrow E + T$
$\quad | \; T$

$T \rightarrow T * F$
$\quad | \; F$

$F \rightarrow id$
$\quad | \; (E)$

Given, id+id*id ①
id*id+id ②



TO REMOVE LEFT RECURSION :-
The product of 'T' is left recursive if it has the form $A \overset{*}{\Rightarrow} A\alpha/\beta$, where 'α' is a grammar symbol and A is the non-terminal to eliminate left recursive the following rules are followed.

1. $A \longrightarrow \beta A'$
2. $A' \longrightarrow \alpha A' / \epsilon$

2M

The production can also be of the form $A \longrightarrow A\alpha/\beta$

$E \longrightarrow E + T$
$A \quad A \alpha \beta$

Now, we have to rewrite the expression in terms of above rules,

$E \longrightarrow E + T$
$A \quad A \alpha/\beta$

$E \longrightarrow TE'$
$E' \longrightarrow +E'/\epsilon$

$T \longrightarrow T * F$
$A \quad A \alpha \beta$

$T \longrightarrow FT'$
$T' \longrightarrow *T'/\epsilon$

* If there are several such forms

$A \longrightarrow A\alpha_1 / A\alpha_2 / A\alpha_3 \text{ ------ } / \beta_n$ and

replace by

$A \longrightarrow \beta_1 A' / \beta_2 A' \text{ -------- } / \beta_n A'$

$A' \longrightarrow \alpha_1 A' / \alpha_2 A' \text{ ------ } / \alpha_m A' / \epsilon$

Eg:- $A \longrightarrow AC / Aad / bd / \epsilon$

NOTE: Why not RE in parsing?

* RE lacks the power to derive the full syntax of programming languages.

for eg:- To resolve precedence in a programming languages, the paranthesis () is generally used, but in RE, it is used for grouping. also RE is not used to describe the features of the programming language.

ELIMINATION OF LEFT FACTORIZATION:-

To eliminate, common left factors that appears in two productions of the same non-terminal. it is converted to equivalent form by removing the common left factors.

for instance, $A \rightarrow qb \mid qc$: The common left factors are removed as the factor is confused to choose one of the above two productions and if a wrong production is chosen, it leads to backtracking.

Hence, the grammar is converted by introducing a variable that takes out the common ~~production~~ prefixes in the production rule.

FIRST:

For each grammar symbol, α FIRST(α) is the set OF terminal symbols that can appear as the FIRST word in some string derived from α. The domain of FIRST is the set of grammar symbols including Terminals U Non-Terminal U {ϵ, eof}

* If 'α' is either a terminal, ϵ or eOF then FIRST(α) has exactly one member α,

* For a non-terminal A, FIRST(A) contains the complete set of terminal symbols that can appear as a leading symbol in a sentential form derived from A.

LL(1) (unambiguous) ⟶

Given,

$E \rightarrow E+E \mid E*E \mid id$

$E \rightarrow E+T \mid T$

$T \rightarrow T*E \mid F$

$F \rightarrow id \mid (E)$

Remove left recursion:

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \epsilon$

$F \rightarrow id$

$F \rightarrow |(E)$

FIRST (T & NT)

TERMINALS:

+	*	ϵ	id	(	)
+	*	ϵ	id	(	)

NON-TERMINALS:

E	E'	T	T'	F
id, (	+, ϵ	id, (	*, ϵ	id, (

$E \rightarrow TE'$

$T \rightarrow FT'$

$F \rightarrow id$

FOLLOW FOR NON-TERMIALS:-

for a Non-Terminal α, FOLLOW(α) contains a set of words that occur immediately after α in a sentence.

Steps:

1. Insert Follow(start) ⟵ $ (eof)
2. If there is a production $A \rightarrow \alpha B \beta$ & $\beta \neq \epsilon$ then everything in FIRST(β) but ϵ, in FOLLOW(B).

2020.01.20 13:27

e. If there is a production rule, $A \rightarrow \alpha B$, or $A \rightarrow \alpha B\beta$ whose FIRST(β) contains ϵ then everything in FOLLOW(A) will be in FOLLOW(B).

FOLLOW

E	E'	T	T'	F
$,)	$,)	$,), +	$,) +	$, +,), *

* By rule 1, add $ to FOLLOW(E)
* By rule 2, The right paranthesis is added to E,
* By rule 3, on $E \rightarrow TE'$, the $ and the right paranth are in FOLLOW(E')
* FOLLOW(E) = FOLLOW(E') = $,)
* Similarly, FOLLOW(T), FOLLOW(T') and FOLLOW(F) are computed where FOLLOW(T) = FOLLOW(T') = $,), +

PREDICTIVE PARSING TABLE:

Input: Grammar G; without ambiguity, left recursion and left factorization

OUTPUT: parsing table for LL(1).

METHOD 1: For each production, $A \rightarrow \alpha$ for the gramm do the following (2&3) steps.

cont'd:
find FIRST & FOLLOW for $S \rightarrow Bb|Cd$
$B \rightarrow aB|\epsilon$
$C \rightarrow cC|\epsilon$

	FIRST	FOLLOW
$S \rightarrow Bb\|Cd$	a, c, b, d	$, b
$B \rightarrow aB\|\epsilon$	a, ϵ	b
$C \rightarrow cC\|\epsilon$	c, ϵ	d

2. $S \to ABCDE$ — FIRST: a, b, c, ε — FOLLOW: $
   $A \to a \mid \varepsilon$ — FIRST: a, ε — FOLLOW: b, c
   $B \to b \mid \varepsilon$ — FIRST: b, ε — FOLLOW: c
   $C \to c$ — FIRST: c — FOLLOW: d, e, $
   $D \to d \mid \varepsilon$ — FIRST: d, ε — FOLLOW: e, $
   $E \to e \mid \varepsilon$ — FIRST: e, ε — FOLLOW: $

CONT'D:

STEP:2: For each terminal, a in the FIRST(α), add $A \to \alpha$ to M[A,a]

STEP:3: If 'ε' is in the ~~FIRST~~ FIRST(α), add $A \to \alpha$ to M[A,b] for each terminal in the FOLLOW(A)

STEP:4: If 'ε' in the FIRST(α) and the '$' in the FOLLOW(A), add $A \to \alpha$ to M[A,$]

STEP:5: each undefined entity of M is error

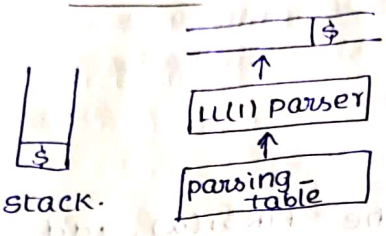
PREDICTIVE TABLE:

	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to id$			$F \to (E)$		

	FIRST	FOLLOW
$E \to TE'$	id, (	$,)
$E' \to +TE' \mid \varepsilon$	+, ε	$,)
$T \to FT'$	id, (	+, $,)
$T' \to *FT' \mid \varepsilon$	*, ε	+, $,)
$F \to id \mid (E)$	id, (	*, +, $,)

2020.01.20 13:27

LL(1) PARSER: (WORKING PRINCIPLE)



* The parser takes 4 possible actions

1. Shift
2. reduce
3. accept
4. error

SHIFT: In this action, the nxt i/p symbol is shifted to the top of the stack.

REDUCE: In this action, the parser knows the right end of the handle is at the top of the stack and within stack must then locate the left end of the handle and decide what non-terminal to replace with handle

ACCEPT: parser successfully complete parsing.

ERROR: The parser announces error.

check id

STACK

$ E
$ E' T
$ E' T' F
$ E' T' id
$ E' T
$ E'
$ E' T
$ E' T
$ E'
$ E
$ E
$ E
$ E
$
$

2020.01.20 13:27

check id + id * id

STACK	i/p BUFFER	ACTION
$ E	id + id * id $	E → TE'
$ E' T	id + id * id $	T → FT'
$ E' T' F	id + id * id $	f → id
$ E' T' id	id + id * id $	reduce
$ E' T'	+ id * id $	T' → ϵ
$ E'	+ id * id $	E' → +TE'
$ E' T +	+ id * id $	Reduce
$ E' T	id * id $	T → FT'
$ E' T' F	id * id $	f → id
$ E' T' id	id * id $	reduce
$ E' T'	* id $	~~reduce~~ T' → *FT'
$ E' T' F *	* id $	reduce
$ E' T' F	id $	f → id
$ E' T' id	id $	reduce
$ E' T'	$	T' → ϵ
$ E'	$	E' → ϵ
$	$	reduce.

RECURSIVE DESCENT PARSER:-

A parser that uses the set of recursive procedures to recognize its inputs with no backtracking is caued as RECURSIVE DESCENT PARSER.

The two cases that causes backtracking are

1. Left factoring (non_deterministic)
2. left recursion (indefinite looping)

The above two problems are overcome in a top down parsers in suitable forms.

LEFT RECURSIVE ($A \rightarrow A\alpha | \beta$)
then 1. $A \rightarrow \beta A'$
2. $A' \rightarrow \alpha A' | \epsilon$

LEFT FACTORING
$A \rightarrow qb$
qa
$A \rightarrow qA'$
$A' \rightarrow a|b$

BACKTRACK FREE FORM:

$E \rightarrow TE'$
$E' \rightarrow +TE'$
$T \rightarrow FT'$
$T' \rightarrow *FT | \epsilon$
$F \rightarrow id | (E)$

$E \rightarrow TE'$

```
procedure E()
begin
    T();
    EPRIME();
end
```

$E' \rightarrow +TE'$

```
procedure EPRIME()
begin
  IF (inputsymbol='+'
  then
  Advance();
  T();
  EPRIME();
end.
```

$T \rightarrow FT'$

```
procedure T()
begin
    F();
   TPRIME();
end
```

$T' \rightarrow *FT|\epsilon$

```
procedure TPRIME()
begin
    if (inputsymbol = '*') then
        Advance();
    F();
    TPRIME();
end.
```

$F \rightarrow id | (E)$

```
procedure F();
begin
[illegible]
end
if ( inputsymbol = 'id') then
    Advance();
if (inputsymbol = '(') then
    Advance();
E();
if (inputsymbol = ')') then
    Advance();
end
```

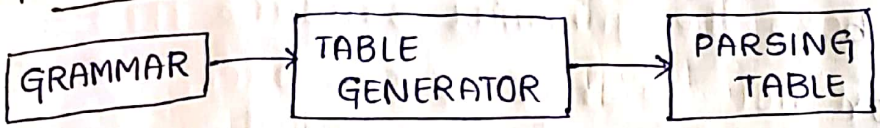
2020.01.20 13:27

LEFT RECURSIVE PARSER:- Our aim is to construct efficient bottom up parser for a large class of CFG. The LR parser scans the i/p from left to right and constructs right most derivation in reverse.

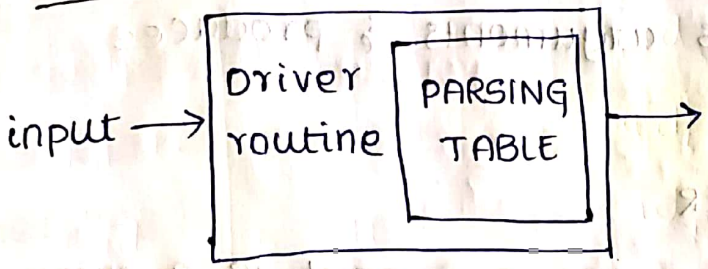
FEATURES:

* LR parser can be constructed to recognize all programming language constructs for which CFG can be written.
* LR parser method is most general than other parsing techniques also, it dominates the common forms of top down parsing without backtracking.
* LR parser can detect the syntactic error at the earliest as it scans the input from left to right.

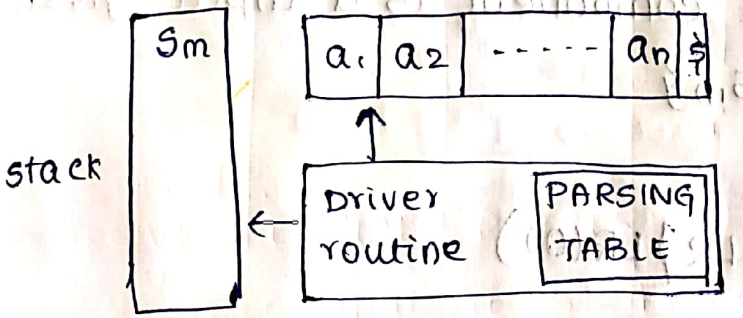
GENERATING LR PARSER:



Operation:



LR Parser:



* this parser has the input as stack & parsing table.
* the i/p is read one symbol at a time from left to right.

* the stack contains the strings of the form. $S_0 X_1 S_1 X_2 \ldots X_m S_m$ where S_m is the top element in the stack. each X_i is a grammar symbol and each S_i is called a state. each state symbol summarizes the information summarized in the stack and is used to guide shift reduce decision.

The parsing table consists of two parts. The parsing action function and 'GOTO' function represented by ACTION' & 'GOTO' respectively.

the program driving the LRParsers behaves as follows.

1. It determines s_m, the state currently on the top of stack and a_i, the current input symbol.
2. then it consults ACTION OF $[s_m, a_i]$ the parsing ACTION table entry for the state s_m and the input entry a_i.
3. The ACTION $[s_m, a_i]$ can have one of the 4 values
   1. Shift the state S.
   2. Reduce a to b
   3. Accept code & error
   4. The function GOTO state takes the state & grammar symbol as arguments & produces another state.

CONFIGURATION OF LR PARSER

It is a pair whose first component is a stack content and the second component is a input given by $\{ s_0 X_1, s_1 X_2, \_ \_ \_ \_ \_ X_m s_m \}$

TYPES:-

1. Simple LR parser (SLR / LR(0))
2. LOOK Ahead left Recursive parser (LALR / LR(1))
3. Canonical left recursive parser (CLR / LR(2))

SIMPLE LR PARSER:

LR(0) is a item of grammar G, is a production of G with a . (dot) at some position @ right side

of the production.

$E \rightarrow .E+T$
$E \rightarrow E.+T$
$E \rightarrow E+.T$
$E \rightarrow E+T.$

* For a given production $S \rightarrow xyz$ Generate LR(0) itemsets

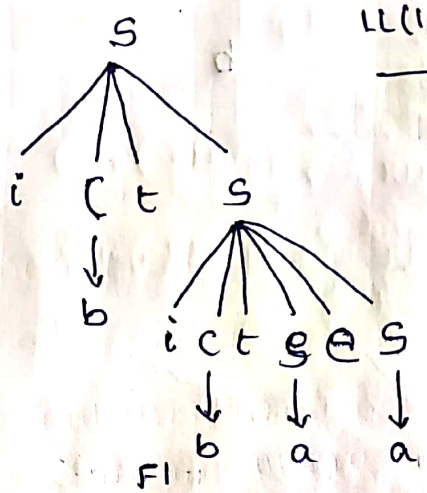
$S \rightarrow .XYZ$
$S \rightarrow X.YZ$
$S \rightarrow XY.Z$
$S \rightarrow XYZ.$

NOTE: for a production rule $A \rightarrow \epsilon$, the LR(0) itemsets is only one $A \rightarrow .$

AUGMENTED GRAMMAR; IF G is a grammar with a start symbol S, then G' is the augmented grammar for G. S' is the start symbol given by the production $S' \rightarrow .S$

* The purpose is to indicate when it should stop and announce acceptance this gives $S' \rightarrow .S$ if parser reduces.

* Draw parse tree for "IF-THEN-ELSE STATEMENT"



LL(1) ⟶

$S \rightarrow ictsA' | a$
$A' \rightarrow es | \epsilon$
$C \rightarrow b$

FI

$S \rightarrow ictsA' | a$
$A' \rightarrow es | \epsilon$
$C \rightarrow b$

	FIRST	FOLLOW
S	i, a	$, e
A'	e	$, e
C	b	t

PREDICTIVE TABLE:

	i	t	a	b	e	$
S	S→ictsA'		S→a			
A'					A'→eS A'→ε	
C				C→b		

ambiguity

LL(1)

∴ So, we can't construct the LL(1)

CONSIDER THE GRAMMAR.

A → Bq

B → dab

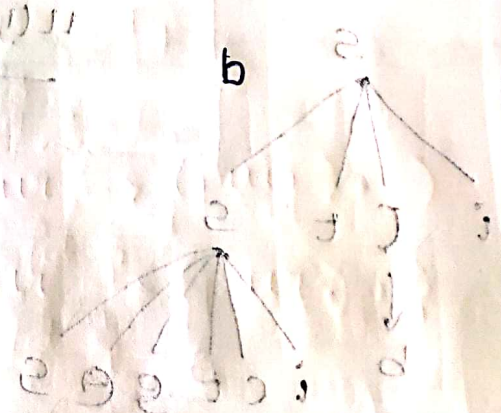
| Cb

C → cB

| AC

Does the grammar satisfies LL(1), if not rewrite & construct.

	FIRST	FOLLOW
A → Bq	d, c	$, c
B → dab	c, d	a, b
| Cb		
C → cB	c, d	b
| AC		



2020.01.20 13:27

CONSIDER THE GRAMMAR

$S \to E$
$E \to T$
$E \to (E)$
$T \to n$
$T \to +T$
$T \to T+n$

construct LR(1) item sets

LR(1) = LR(0) + LOOK ahead

	FIRST	FOLLOW
S	(, n, +	$
E	(, +, n	$,)
T	+, n	$,), +

LR(1) (S_0)

$S \to .E$, $
$E \to .T$, $
$E \to .T$,)
$E \to .(E)$, $
$E \to .(E)$,)
$T \to .n$, $
$T \to .n$,)
$T \to .n$, +
$T \to .+T$, $
$T \to .+T$,)
$T \to .+T$, +
$T \to .T+n$, $
$T \to .T+n$,)
$T \to .T+n$, +

(S_1)

$S \to E.$, $ x
$E \to T.$, $ x
$E \to T.$, $) x
$E \to (.E)$, $
$E \to (.E)$,)
$T \to n.$, $ x
$T \to n.$,) x
$T \to n.$, + x
$T \to +.T$, $
$T \to +.T$, $)
$T \to +.T$, +
$T \to T.+n$, $
$T \to T.+n$,)
$T \to T.+n$, +

(S_2)

$E \to (E.)$, $
$E \to (E.)$,)
$T \to +T.$, $ x
$T \to +T.$,) x
$T \to +T.$, + x
$T \to T+.n$, $
$T \to T+.n$,)
$T \to T+.n$, +

(S_3)

$E \to (E).$, $ x
$E \to (E).$,) x
$T \to T+n.$, $ x
$T \to T+n.$,) x
$T \to T+n.$, + x

(S_4)

NO states checking completed.

CANONICAL FORM OF ITEM SETS

Consider the paranthesis grammar.

Goal → list
list → list pair
| pair
pair → (pair)
| ( )

What are the LR(1) items for this grammar? construct canonical collections of set of LR(1)

Goal → list
list → list pair
| pair
pair → (pair)
| ( )

	first	follow
Goal	(	$
list	(	$
pair	(	$,), (

LR(1)

S_0

Goal → .list, $
list → .list pair, $
list → .pair, $
pair → .(pair),)
pair → .(pair), $

S_1

Goal → list., $
list → list.pair, $
list → pair., $
pair → (.pair),)
pair → (.pair), $

S_2

list → list pair., $
pair → (pair.),)
pair → (pair.), $

S_3

pair → (pair).,)
pair → (pair)., $

Bottom-up
→ all programming constructs.

2020.01.20 13:27

LR(1)

```
Procedure closure (I)
begin
Repeat
    for each item [A → α.Bβ,a] in I
        for each production B → ϒ & each
        terminal b in FIRST(βa) such
        that [B → .ϒb] is not in I do add
        [B → .ϒ.b] to I. until no more items to
        be added to I
    return I
end
procedure GOTO(I, X)
begin
    Let I be the set of items [A → αX.β,a] s.t
    [A → α.Xβ,a] is in I
    return closure [I]
end
begin
    C = {closure ((S' → .S,$))}
repeat
    for each set of items
        I in C & grammar symbol X s.t
        GOTO(I,X) is not empty & already in C
        add GOTO(I,X) to C
        until no more sets of items
        added to C
end.
```

2020.01.20 13:28

AUGMENTED GRAMMAR:

		FIRST	FOLLOW
1.	$S' \to S$	c, d	$
	$S \to CC$	c, d	$
	$C \to cC \mid d$	c, d	c/d

ITEMSETS:

I_0 : $S' \to .S, \$$
$S \to .CC, \$$
$C \to .cC, c/d$
$C \to .d, c/d$

GOTO (I_0, S) : $S' \to S., \$ \longrightarrow I_1$

GOTO (I_0, C) : $S \to C.C, \$$
$C \to .cC, c/d/\$$
$C \to .d, c/d/\$$
$\} I_2$

GOTO (I_0, c) : $C \to c.C, c/d$
$C \to .d, c/d$
$C \to .cC, c/d$
$\} I_3$

GOTO (I_0, d) : $C \to d., c/d \quad I_4$

GOTO (I_2, C) : $S \to CC., \$ \quad I_5$

GOTO (I_2, c) : $C \to c.C, c/d/\$$
$C \to .d, c/d/\$ \; I_6$
$C \to .cC, c/d/\$$

GOTO (I_2, d) : $C \to d., \$ \; I_7$

GOTO (I_3, C) : $C \to cC., c/d \quad I_8$

2020.01.20 13:28

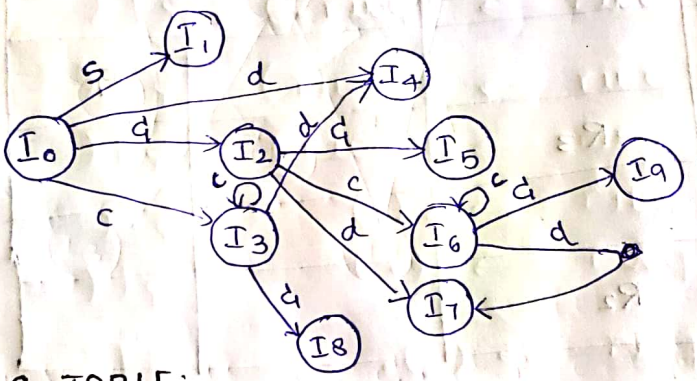
GOTO (I_3, c) : C → c.C , c/d
C → .cC , c/d I_3
C → .d , c/d

GOTO (I_3, d) : C → d. , c/d I_4

GOTO (I_6, C) : C → cC. , $ I_9

GOTO (I_6, c) : C → c.C , $
C → .cC , $ I_6
C → .d , $

GOTO (I_6, d) : C → d. , $ I_7



LR(1) PARSING TABLE:

1. If [A → α.aβ, b] is in I_i & GOTO (I, a) = I_j then set ACTION [i, a] to shift j

2. If [A → α., a] is in I_i then set ACTION [i, a] to Reduce A → α

3. If [S' → S., $] is in I_i, then set ACTION [i, $] to Accept

4. If GOTO [I_i, A] = I_j then GOTO (i, A) = j [Non terminal]

Action : Terminal

GOTO : nonTerminal → non-Terminal

	ACTION			GOTO	
	c	d	$	S	C
0	S_3	S_4		1	2
1			Accept		
2	S_6	S_7			5
3	S_3	S_4			8
4	R_3	R_3			
5			R_1		
6	S_6	S_7			9
7			R_3		
8	R_2	R_2			
9			R_2		

$S \rightarrow CC$
$C \rightarrow cC$
$C \rightarrow d$

* check if the string cdd can be parsed

word	stack	Action
c	$\$_0$	S_3
d	$\$_0 c_3$	S_4
d	$\$_0 c_3 d_4$	R_3 $C \rightarrow d$
d	$\$_0 c_3 C_8$	R_2 $C \rightarrow cC$
d	$\$_0 C_2$	GOTO 2
$	$\$_0 C_2 d_7$	R_3
$	$\$_0 C_2 C_5$	R_1
$	$\$_0 S_1$	Accept

2020.01.20 13:28

Construct the paranthesis grammar & canonical collection of set of items. derive the table and check it is LR(1). and.

	FIRST	FOLLOW
Goal → list	(	$
list → list pair	(	$, (
| pair		
pair → (pair)	(	$, (,)
| ( )		

I_0 : Goal → .List , $
List → .List pair , $/(
List → .pair , $/(
pair → .(pair) , $/(/)
pair → .( ) , $/(/)

GOTO(I_0, List): Goal → List . , $
List → List . pair , $/(I_1
list → .pair , $/(
pair → .(pair) , $/(
pair → .( ) , $/(

GOTO (I_0, pair): List → pair . , $/(I_2

GOTO (I_0, ( ): pair → (.pair) , $/(/)
pair → .(pair) , $/(/) I_3
pair → .( ) , $/(/)
pair → (.) , $/(/)

GOTO (I_1, pair): List → List pair . , $/(I_4

GOTO (I_1, ( ): pair → (.pair) , $/(
pair → .(pair) , $/(I_5
pair → .( ) , $/(

2020.01.20 13:28

pair → (.) , $/(I_6

GOTO(I_3, pair): pair → (pair.) , $/(/)

GOTO(I_3, ( ) : pair → (.pair) , $/(/) I_3
pair → .(pair) , $/(/)
pair → .() , $/(/)
pair → (.) , $/(/)

GOTO(I_3,)) : pair → (). , $/(/) I_7

GOTO(I_5, pair) : pair → (pair.) , $/(I_8

GOTO(I_5, ( ) : pair → (.pair) , $/(
pair → .(pair) , $/(I_5
pair → .() , $/(
pair → (.) , $/((

GOTO(I_5,)) : pair → (). , $/(I_9

GOTO(I_6,)) : pair → (pair). , $/(/) I_{10}

GOTO(I_8,)) : pair → (pair). , $/(/) I_{11}

Goal → List

R_1 - List → List pair

R_2 | pair

R_3 - pair → (pair)

R_4 | ( )

	(	)	$
0	S_3		
1	S_5		
2	R_2		R_2
3	S_3		
4	R		R
5	S		
6			
7	R		R
8			
9			
10			
11			

che

2020.01.20 13:28

	ACTION			GO TO	
	(	)	$	list	pair
0	S_3			1	2
1	S_5		Accept		4
2	R_2		R_2		
3	S_3	S_7			6
4	R_1		R_1		
5	S_5	S_9			8
6		S_{10}			
7	R_4	R_4	R_4		
8		S_{11}			
9	R_4		R_4		
10	R_3	R_3	R_3		
11	R_3	R_3	R_3		

check if the string () can be parsed.

word	stack	action	
(	$\$_0$	S_3	
)	$\$_0(_3$	S_3	
)	$\$_0(_3(_3$	S_7	
)	$\$_0(_3(_3(_7$	R_4	pair → ( )
)	$\$_0(_3pair_6$	S_{10}	
$	$\$_0(_3pair_6)_{10}$	R_3	pair → (pair)
$	$\$_0 pair_2$	R_2	list → pair
$	$\$_0 list_1$	Accept	

UNIT-II

2M CONTEXT SENSITIVE ANALYSIS:

→ context of
1) Variable (data type)
2) expression (type compatability)
3) procedure calls

compiler builds a large knowledge base from programs.

↓

about details in the program

↓ to do it, it should know

1. How values are represented?
2. How values flow between variable?
3. understand structure
4. program interaction with external files.

Knowledge base:

ex: if var 'x' is used in program, it checks

1. what kind of values are stored.
2. memory requirements ⇒ (overflow can be avoided)
3. If 'x' is a procedure, checks args & return type
4. memory allocation < compile (or) runtime.
5. How long it has to be stored

Type system:-

→ The set of types in programming language along with rules.

Need for type system.

1. ensure runtime safety & avoid compile time errors.

ex: a+b

+	int	r	d	c
int	i	r	d	c
real	r	r	d	c
double	d	d	d	not possible
complex	c	c	n.p	c

two types

- strongly typed (C, C++, Java)
- weak typed (Python)

2. Expressiveness

↳ precise meaning

ex: + (or) addition / overloading

3. Generate bettercode

a	b	a+b
int	int	int
int	real	real

2M

TYPE CHECKING:

↳ compiler analyse program and assign types to each expression also ensure context - this is called type checking.

Type checking:- 2 ways

↳ attribute grammar Framework

↳ ASDT [Adhoc Syntax Directed Translation]

2020.03.02 14:38

Attribute grammar frameworks:

→ attribute grammar on CFG.

↓

Set of rules on CFG

↓

each rule specifies an attribute in terms of another.

ex: CFG for Signed Binary no's (SBN)

{ S, T, NT, P }

S = { Number }

T = { −, +, 1, 0 }

NT = { Number, list, Bit, sign }

P = { S ⟶ Number

Number ⟶ sign List

Sign ⟶ − | +

List ⟶ List bit | bit

bit ⟶ 0 | 1

Attribute grammar to each non-Terminal

eg:

Symbol	attributes
Number	value
Sign	negative
List	positional value
bit	position value.

Attributes grammar for SBN

productions	Attribute rules

1. Number ⟶ Sign list	list.position ⟵ 0
	IF sign.negative then
	number.val ⟵ −list.val
	else
	number.val ⟵ list.val

2. Sign ← +
3. Sign ← −
4. list ← bit

sign.neg ← false
sign.neg ← true
Bit.position ← list.position
list.value ← Bit.value
[right to left movement]

5. list ← list bit

list.pos ← list.pos + 1
Bit.pos ← list.pos [rt→lf
list.val ← list.val + Bit.val

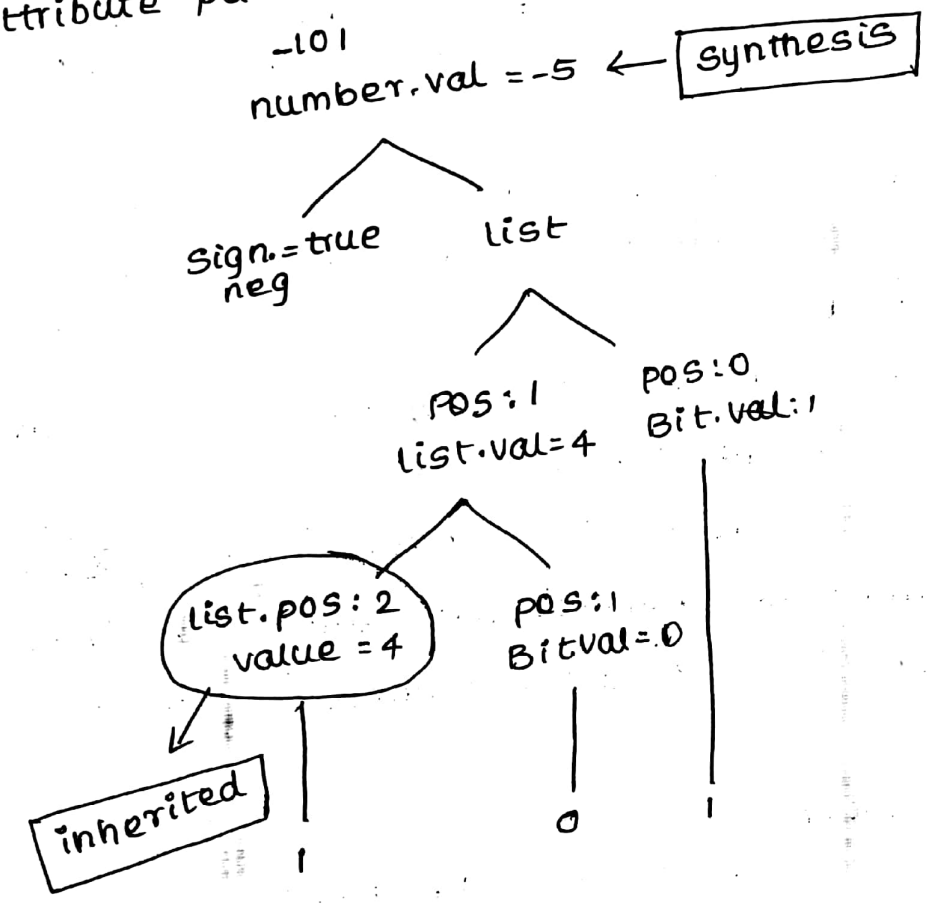
6. Bit ← 0
7. Bit ← 1

Bit.value ← 0
Bit.value ← 1

2M TWO TYPES OF ATTRIBUTES

* Synthesized → only on the own node & from children
* inherited → depended on its own node, parent, sibling children.

ex:- Attribute parse tree for −101.



T. cost

To evaluate the attributed parse tree. the sentence of the grammar attributes are instanted. for each node in the parse tree. each rule defines the set of dependancies and depends on each argument in the rule.

Write the attribute grammar for signed decimal (SDN)

// FORMALITIES IN PRE. PAGE

Number ⟶ Sign List

Sign ⟶ + / −

List ⟶ bit / List bit.bit

bit ⟶ 0|1|2|3|4|5|6|7|8|9

* A compiler writer may create attribute grammar & design set of rules ~~can be created to do it.~~ to compute the value. An evaluator can be created to do it:-

2M

EVALUATION METHODS:

1. Dynamic
   - depends on structure of attributed parse tree
   - rule fixed as soon as operands are available
2. Oblivious
   - independent on order
   - designer selects method for evaluation
3. Rule based
   - It depends on static analysis and on grammar.

PROBLEMS WITH THE RULE BASED: CIRCULARITY:

- refer to its own results either directly or indirectly
- Such attribute grammar are called CIRCULAR ATTRIBUTE GRAMMAR. (⟶ fail to get evaluated ⟶ so removed.)

METHOD to overcome:-

1. Avoidance
   → restrict on CFG to avoid.

use evaluation → to identify circularity.

Ex:2 EXPRESSION TYPES.

Inter expression type

Function Attribute rules

$E_0 \rightarrow E_1 + T_1$ | $E_1 - T_1$ | T

$E_0.type \rightarrow F+(E_1.Type, T_1.type)$ | $F-(\ "\ )$ | $T.type$

$T_0 \rightarrow T_1 * F$ | $T_1 \% F$ | F

$T_0.type \rightarrow F*(T_1.type, F.type)$ | $F\%(\ "\ )$ | $F.type$

$F \rightarrow (E)$ | id | num

$F.type = (E.type)$ | id | num } already def.

F+, F-, F*, F% → for evaluating.

Eg: Execution time estimator

* To estimate execution time of sequence of assignment statements
* we generate by adding 3 new productions

Black → Black Assignment | Assignment

Assignment → id = Expr is added to classical expr grammar:

production

Attribute rules

$Black_0 \rightarrow Black, Assign$ | $Assign$

$Black_0.cost \leftarrow Black_1.cost + Assign.cost$

$\leftarrow Assign.cost$

$Assign \rightarrow id = Expr$

$Assign.cost \rightarrow cost(store) + E.cost$

$E_0 \rightarrow E+T$ | $E-T$ | T

$E_0.cost = E_1.cost + T_1.cost +$ cost.(add)

$E_1.cost + T_1.cost + cost(sub)$

$\leftarrow T.cost$

$T_0 \rightarrow T_1 * F$ $\quad T_0.cost \leftarrow T_1.cost + F.cost + cost(mul)$

$\mid T_1 \div F$ $\quad T_0.cost \leftarrow T_1.cost + F.cost + cost(Div)$

$\mid F$ $\quad \leftarrow F.cost$

$F \rightarrow (E)$ $\quad F.cost \leftarrow E.cost$

$\mid id$ $\quad cost(load)id$

$\mid num$ $\quad cost(load)num$

consider $x = y + y$ for the above

the load operation, $\rightarrow$ activation record

load AI $\quad r_{arp} @ y \Rightarrow r_y$

add $\quad r_y, r_y \Rightarrow r_x$

store AI $\quad r_x \Rightarrow r_{arp}, @x$

To appropriate above behaviour, for y only one load operation inside a block then to include

If (ident has not been loaded) then

f.cost $\leftarrow$ cost(load)

else

f.cost $\leftarrow$ 0

To implement the above we include before & after to each node.

Before $\rightarrow$ contains names of id in the block-

After $\rightarrow$ before + any id that would be loaded.

After modification :-

$F \rightarrow (E)$

$F \rightarrow num$

$F \rightarrow id$

Attribute

f.cost $\leftarrow$ E.cost

E.before $\leftarrow$ F.before

f.before $\leftarrow$ E.after

f.cost $\leftarrow$ cost(load num)

Fafter $\leftarrow$ f.before

IF (idname $\notin$ f.before) then

f.cost $\leftarrow$ cost(load)

f.after $\leftarrow$ f.before $\cup$ id.nam

else

f.cost $\leftarrow$ 0

f.after $\leftarrow$ f.before.

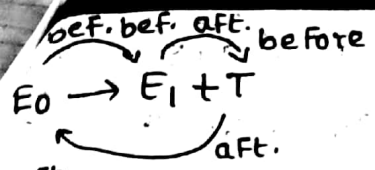
pattern

DB sepcod

T_{ID}	L
T_{100}	I_1
T_{200}	I_2
T_{300}	I_2
T_{400}	I_1
T_{500}	I
T_{600}	I
T_{700}	I
T_{800}	I
T_{900}	

min-supp

2020.03.02 14:39



E_0 cost ← E_1 cost + T cost + cost(add)
E_1. before ← E_0. before
T. before ← E_1. after
E_0. after ← T. after

$E_0 \rightarrow E_1 - T$

$E_0 \rightarrow T$.

PROBLEMS OF ATTRIBUTE GRAMMAR (AGF) ⇒ CSA knowledge building

* Handling Non local infor. Hence, rules are included as requirement.
* It constructs parse tree for evaluation but it is a known fact that compiler never builds a parse tree as it constructs only a abstract syntax.
* To locate the rules it needs to traverse through the parse tree to find the appropriate one.

Adhoc Syntax Directed Translation (ASDT) (or) (ADST):

The problems with AGF is the increase in complexity & space requirement to overcome the above problem ASDT does the following:

1. Associates a snippet of code with each production.
2. At each reduction of the rule the corresponding snippet runs.
3. Allowing arbitrary code provides complete flexibility.

IMPLEMENTING ASDT: To implement ASDT, the parser must include the mechanisms to parse value from their definitions in one action to the users & in another to provide convinient meaning allowing for the actions for the parser. Some of the mechanisms for handling are

1. COMMUNICATE BIN THE ACTION: In ASDT, the parser doesn't construct a parse tree so, the values are to be stored in an internal stack. usually, the parse tree stores as a pair of info. given by <symbol, state>, for each grammar rule & now for implementing ASDT, the parser includes values & hence the triple info. is given by <value, symbol, state> is stored for every grammar symbol. To manage the stack the values are pushed or popped up.

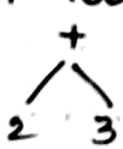
for instance a reduction given by $A \rightarrow \beta$ leads to locations popped from the top of stack. To solve this storage the parser may omit actual grammar symbol but the grammar symbol provides better error reporting & debugging.

NAMING VALUES: To simplify the use of stack based values a convinient notation is required for naming them. The symbol $$ refers to the current location for the current production of the right hand symbol. Hence, the symbol $1, $2, ..., $n refers to 1st, 2nd & nth location in RHS for eg:

production	code snippet
Number → signlist	$$ ← $1 * $2
Sign → +	$$ ← 1
Sign → -	$$ ← -1
list → bit	$$ ← $1
$list_0$ → list bit	$$ ← 2 * $1 + $2
Bit → 0	$$ ← 0
Bit → 1	$$ ← 1

ACTIONS: To perform the actions in the middle of the compiler writer can transform the grammar so that it performs a reduction at each point when required. performs an abstract syntax tree. The compiler front end must build an IR of the program for the use in middle & back end part of compilation. AST is a common form of tree structured IR. A compiler uses the series of routines to construct AST & one such routine called MakeNode where $0 \le i \le 3$. The routine takes the 1st argument a constant that uniquely identifies a grammar symbol that node will represent the remaining arguments represent a subtree for eg: $MakeNode_0$(Num), constructs a leaf node. $MakeNode_2$(plus, $MakeNode_0$(num), $MakeNode_0$(num)). this builds an AST rooted with "+" & with 2 children.

i.e., 2+3



The general principle in AST is the operators are created as nodes and operands as children.

PRODUCTION	(Graphical form of IR) ASDT
$E \rightarrow E+T$	$\$\$ \leftarrow$ makenode$_2$ (plus, $\$1, \$2$)
	$\$\$$type $\leftarrow F+$ ($\$1$.type, $\$2$.type)
$\mid E-T$	$\$\$ \leftarrow$ makenode$_2$ (sub, $\$1, \$2$)
	$\$\$$type $\leftarrow F-$ ($\$1$.type, $\$2$.type)
$\mid T$	$\$\$ \leftarrow \$2$
$T \rightarrow T * F$	"
$\mid T \div F$	
$\mid F$	$\$\$ \leftarrow \$3$
$F \rightarrow (E)$	$\$\$ \leftarrow (\$1)$
$\mid$ id	$\$\$ \leftarrow$ makenode$_0$ (id)
$\mid$ num	$\$\$ \leftarrow$ makenode$_0$ (num)

ILOC : Intermediate Language OF Compiler:

- for an optimized compiler
- IR
- uses 4 supporting routines

AENV

Address → takes var names as its arguments.
Emit →
Next register →
value →

ADDRESS: Takes a variable name as its argument and returns a register number. (temporary memory in processor). It ensures that the register contains the variable address (memory address)

EMIT: It handles the details of creating a concrete representation for the various ILOC operations. It can even format and print to the file.

NEXTREGISTER: Returns a new register number

Simple implementation to increment a global counter

VALUE: Takes the number as its argument and returns the register that contains the numbers value passed as its argument. If required, it generates the code to move that number into a register.

PRODUCTION	SDT → Linear form of IR
$E \rightarrow E+T$	$$ ← NextRegister Emit (add, $1, $2, $3)
| $E-T$	$$ ← Next Register Emit (sub, $1, $2, $3)
| T	$$ ← $2
$T \rightarrow T*F$	$$ ← NextRegister Emit(mul, $1, $2, $3)
| T/F	$$ ← NextRegister Emit (div, $1, $2, $3)
| F	$$ ← $3
$F \rightarrow (E)$	$$ ← ($1)
| id	$$ ← Address (id)
| num	$$ ← value (num)

INTERMEDIATE REPRESENTATION:

IR is required to represent the code for further optimization across various phases of the compiler. Hence, IR must be.

1. expressive to record all useful facts in a program
2. In expensive
3. concise
4. easily examinable

CLASSIFICATION:-

IR has a structural organization and can be classified into three.

1. Graphical IR
2. Linear IR
3. Hybrid IR.

GRAPHICAL IR: Encodes the compiler knowledge in the form of graph. The algorithms express in terms of nodes and edges (or) list and trees.

Eg:- parse tree, Abstract syntax tree.

LINEAR IR:- resembles pseudo code for some machine languages. The algorithms iterate over simple linear sequences of operators.

Eg:- ILOC, 3 address code

HYBRID IR:- this combines the elements of both linear and graphic IR in an attempt to minimize drawbacks of both the IR's.

GRAPHICAL IR:-

- consists of nodes & edges
- difference lies in structure & in graph.

1. Syntax Related trees

   1. parse tree
   2. AST - Abstract Syntax tree
   3. DAG - Directed acyclic graph

imp

2. Graph

* control flow graph [CFG]
* Dependancy graph

Tree

→ correspond to syntax of source code

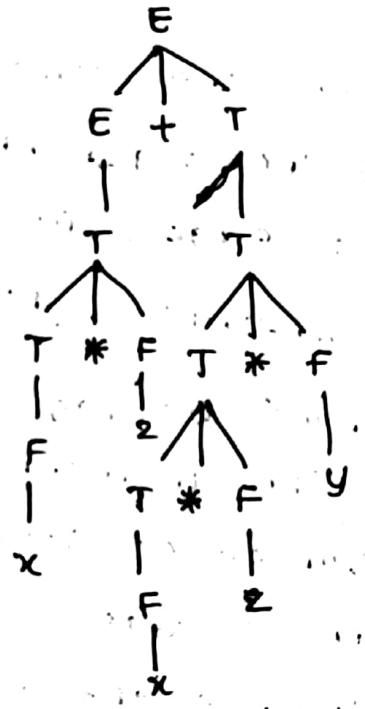
1. parse tree
   (syntax tree)
   corresponds to derivation from i/p program.

Ex: Draw parse tree for x * 2 + x * 2 * y

$E \longrightarrow E+T \mid E-T \mid T$

$T \longrightarrow T*F \mid T \div F \mid F$

$F \longrightarrow id \mid num \mid (E)$

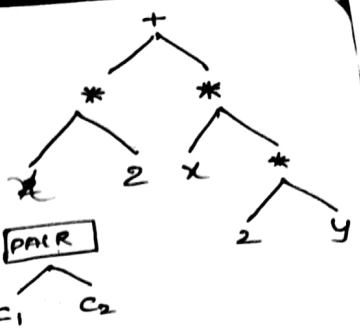


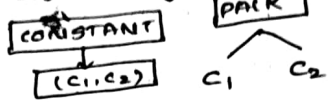
AST:

* Retains basis of parse tree but eliminates extra nodes
* precedence & meaning of expression remains (no extra nodes whereas in parse tree, we have extra nodes)

near source level representation
- used in many compilers

ex: in FORTRAN
a complex constant (C_1, C_2).
2 foll AST may be used.

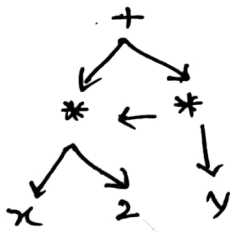




DAG:
- contradiction of AST
- Avoids duplicates
- In DAG, a node can have multiple parents but identical subtrees reused.
- hence a more compact representation (reused)

x * 2 + x * 2 * y

DAG

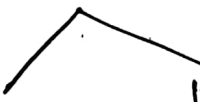


→ better code
→ But compiler has to prove that x*2 doesn't change. (with directions more importantly)

ADVANTAGES:-
1. Smaller memory requirements
2. To expose potential redundancy.
3. Better for the optimization.

LEVELS OF DISTRACTION:-

Trees can be used to represent both

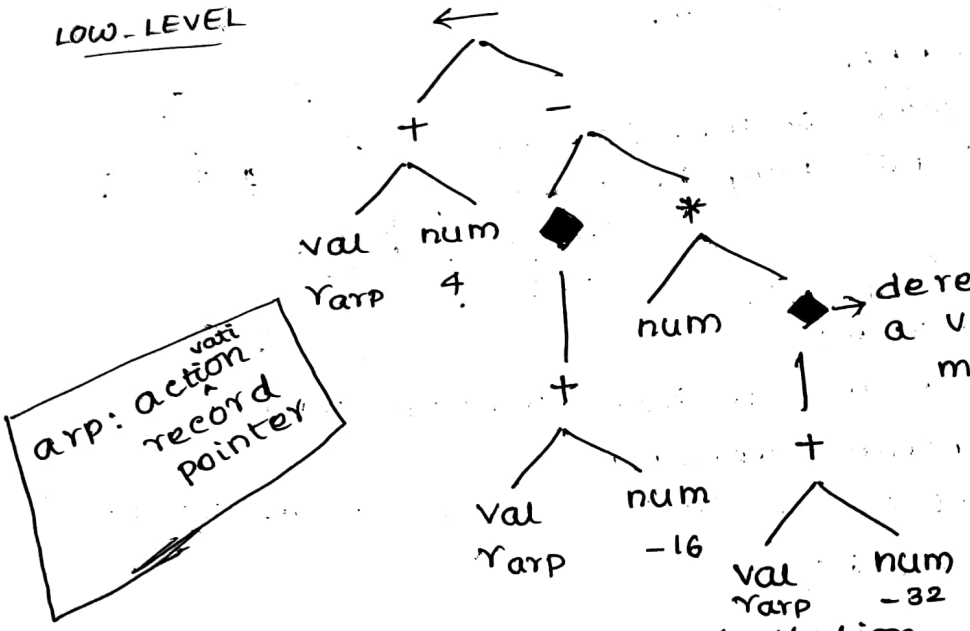


source level low level
detail in code

eg: $w \leftarrow x - 2 * y$

SOURCE LEVEL

LOW-LEVEL



address calculation for w, x & y

1. Draw AST for

1. $2 * 7 + 3$
2. $2 * (7+3)$
3. $(((1+2)+3+4)+5$
4. $7+3*(10|(12|(3+1)-1)$
5. Low level AST

$c = a + b * d$

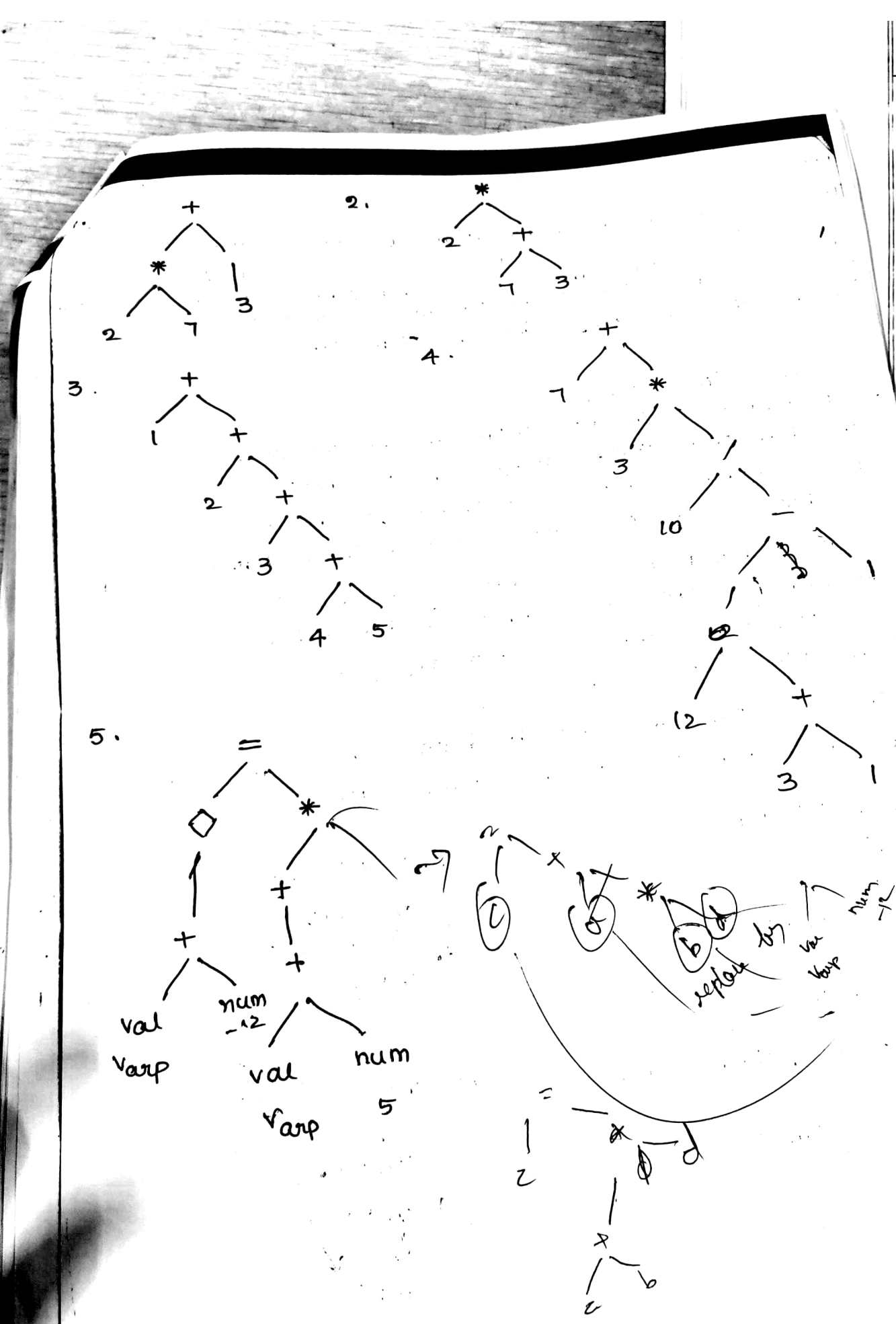
$a \rightarrow 5$ from r_{arp}

$b \rightarrow$ ptr from r_{arp}, @6

$d \rightarrow 7$ from r_{arp}

$c \rightarrow$ ptr from r_{arp} @ -12

2020.03.02 14:40



NOTE:

Trees provide Natural representation for the grammatical structure of course code, rigid structure makes them less useful for representing other properties of a. Hence, to model the other properties compiler uses graphs that are more general than trees.

CFG:

- Control Flow Graph
- It models flow of control in the programs, it is the directed graph given by G= (N,E) where N= is a node that corresponds to the each basic block. (set of operations with in a loop)

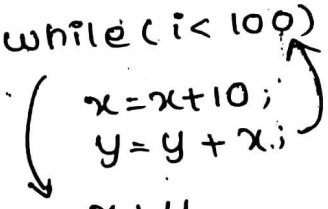
* A basic block is a sequence of operations that are always executed together. each edge 'E' corresponds to the possible transfer of control from block Ni to Nj.

- Thus, CFG provides the graphical representation of possible runtime control flow path.

CODE:

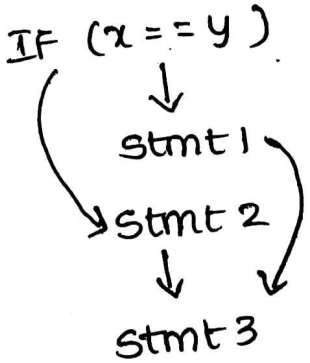
1. while (i< 100)
   begin
   x = x+10;
   y = y+x;
   end
   z = x+y

CFG:



2. if (x==y) then stmt1
   else
   stmt2

   stmt3



A CFG for conditional statement is acyclic & for looping construct, it is cyclic

* compiler uses CFG along with other IR, such as AST, DAG etc.
* CFG may be useful as it supports
    1. Code optimization
    2. Instruction scheduling
    3. Global register allocation

DEPENDANCY GRAPH:

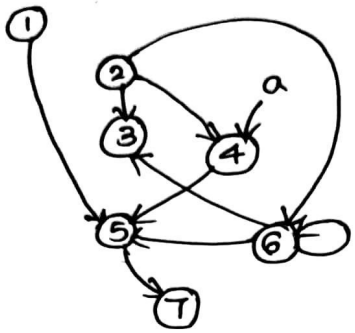
- These are useful for determining an evaluation model for attribute instances in a given parse tree, while unallocated parse trees shows the attribute values. The dependancy graph emphasizes the flow of information among the attribute instances. An edge from one attribute instance to another means the value of the 1st is required to compute the second.

Edges express the constraints implied by the semantic use. compilers use the Data dependancy graph to encode the flow of values, from the point where the values created (definition points) to the points where it is used (used points). Through the dependancy graph, the real constraints are represented as sequence that it doesn't capture the program control flow fully.

pgm:

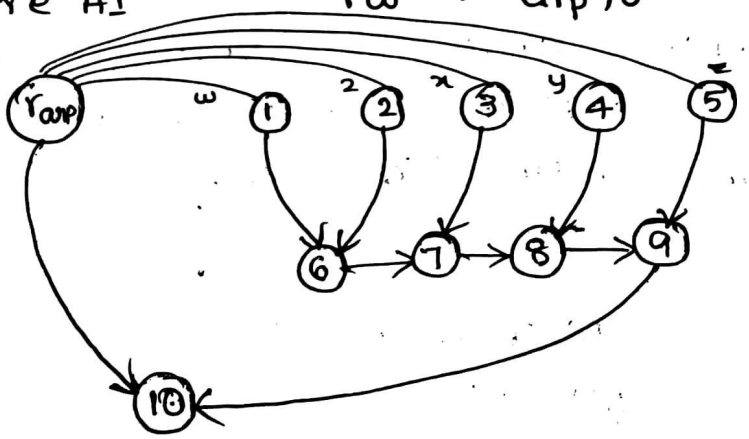
```
① x ← 0
② i ← 1
③ while (i<100)
④   IF (a[i]>0)
⑤ then x ← x + a[i]
⑥   i ← i+1
⑦ print(x)
```

DEPENDANCY TREE



ILOC BLOCK:

1. LOAD AI r_{arp} , @w $\Rightarrow r_w$
2. LOAD AI 2 $\Rightarrow r_2$
3. LOAD AI r_{arp} , @x $\rightarrow r_x$
4. LOAD AI r_{arp} , @y $\rightarrow r_y$
5. LOAD AI r_{arp} , @z $\rightarrow r_z$
6. MULT r_w , r_2 $\rightarrow r_w$
7. MULT r_w , r_x $\rightarrow r_w$
8. MULT r_w , r_y $\rightarrow r_w$
9. MULT r_w , r_z $\rightarrow r_w$
10. STORE AI r_w $\rightarrow r_{arp}$, 0



LINEAR IR:-

Requirements of instructions that execute in their order of appearance.

sourcecode $\xrightarrow{I/P}$ compiler $\xrightarrow{O/P}$ Linear IR

1. One addresscode → Accumulator
2. two addresscode } Obselete
3. three addresscode
4. Stack machine code

* ILOC is called linear IR.

STACK MACHINE CODE:-

⇒ compact representation (size)

→ Operations take operands from stack & push results on to the stack.

└ creates implicit namespace.

Ex: ~~(x - (z * y))~~

~~LOAD y~~

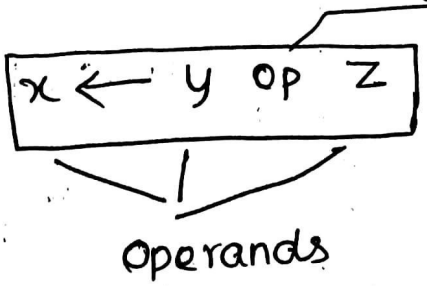
L

PUSH y
PUSH 2
MULTIPLY
PUSH x
SUBTRACT

TAC:-

THREE ADDRESSCODE:

Operations of form



ADVANTAGES:-

* Reuse names & values

...s & 1 operator ...implement 3 address operator ...well for execution.

...optimize as it contains low level operation ...implemented as a set of quadraples.

x = * a;
a = & b;

$t_1 \leftarrow a$
$t_2 \leftarrow * t_1$
$t_3 \leftarrow x$
$t_3 \leftarrow t_2$
↳ registers

$t_4 \leftarrow b$
$t_5 \leftarrow \& t_4$
$t_1 \leftarrow t_5$

$t_1 \leftarrow 2$
$t_2 \leftarrow y$
$t_3 \leftarrow t_1 * t_2$
$t_4 \leftarrow x$
$t_5 \leftarrow t_4 - t_3$

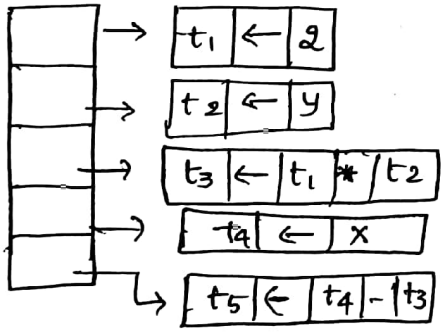
TAC IMPLEMENTATION:

1. Simple array

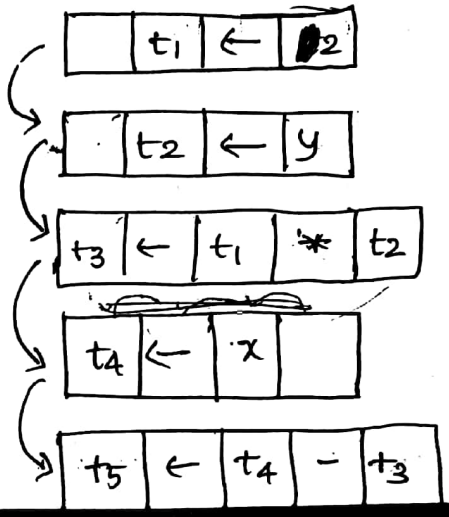
target	op	arg1	arg2
t_1	←	2	
t_2	←	y	
t_3	*	t_1	t_2
t_4	←	x	
t_5	−	t_4	t_3

} most of space is nt utilised & can overcome by following methods

2. Array of pointer



3. Linked List (usually followed)



T_{100}	I_1, I_2 I_5
T_{200}	I_2 I_4
	I_3
	I_4

* compact 3 names & 1 operator
* modern processors implement 3 address operator model · Fixes well for execution.
* easier to optimize as it contains low level operations
* represented as a set of quadraples.

$x - 2 * y$

TAC:

$t_1 \leftarrow 2$
$t_2 \leftarrow y$
$t_3 \leftarrow t_1 * t_2$
$t_4 \leftarrow x$
$t_5 \leftarrow t_4 - t_3$

```
x = *a;
a = &b;

t1 ← a
t2 ← *t1
t3 ← x
t3 ← t2
  ↳ registers

t4 ← b
t5 ← &t4
t1 ← t5
```

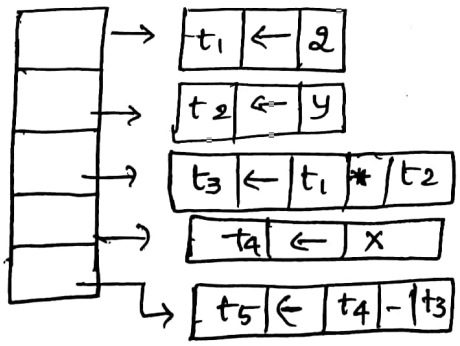
TAC IMPLEMENTATION:

1. Simple array

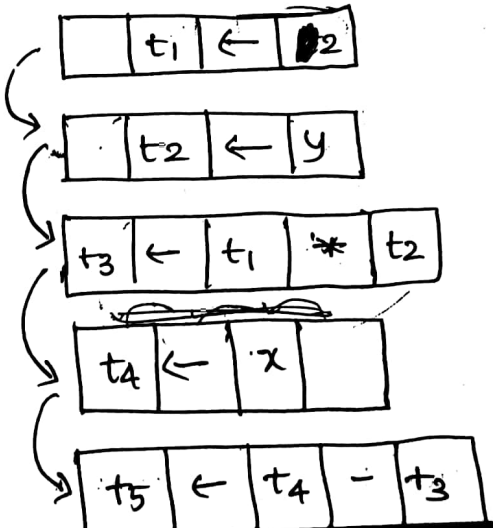
target	op	arg1	arg2
t_1	←	2	
t_2	←	y	
t_3	*	t_1	t_2
t_4	←	x	
t_5	−	t_4	t_3

} most of space is nt utilised & can overcome by following methods

2. Array of pointer



3. Linked List (usually followed)



2020.03.02 14:41

MAPPING VALUES TO NAMES:

Source code

```
a ← b+c
b ← a-d
c ← b+c
d ← a-d
```

2 methods:

1. Using source names
2. Using value names

Using source names

```
t1 ← b
t2 ← c
t3 ← t1+t2
t4 ← d
(t1 ← t3-t4
 b ← t1)
t2 ← t1+t2
c ← t2
t4 ← t3-t4
d ← t4
```

using value names

```
t1 ← b
t2 ← c
t3 ← t1+t2
a ← t3
t4 ← d
(t5 ← t3-t4
 b ← t5)
t6 ← t5+t2
c ← t6
t5 ← t3-t6
d ← t5
```

diff

diff

diff

2MQ

MEMORY MODELS:-

1. Register to Register
2. Memory to Memory

SINGLE STATIC ASSIGNMENT FORM (SSA FORM):

In this form, values are assigned a static name, both data flow and control flow can be represented by ϕ (Psi) function. this function corresponds uniquely to specific determined points in the code as each name has one operation. it is called as SINGLE STATIC ASSIGNMENT FORM.

CODE

```
x ←
y ←
while (x<100)
    x ← x+1
    y ← y+x
```

SSA FORM

```
x0 ←
y0 ←
if (x0 ≥ 100) go to next
loop: x1 ← φ(x0, x2)
      y1 ← φ(y0, y2)
      x2 ← x1+1
      y2 ← x2+y1
IF (x2 < 100) go to loop
next: x3 ← φ(x0, x2)
      y3 ← φ(y0, y2)
```

* Two considerations for SSA:
1. each definition should be distinguishable.
2. each use refers to single definition. so, compiler inserts a ϕ function. when different controller merge & it renames variables to create SSA.

ϕ functions can be inserted where multiple distinct values reach for the block

ex:1

$y = x - a$
$k = y - t$
$y = m - k$
$k = y - k$

$y_1 = x_1 - a,$
$k_1 = y_1 - t,$
$y_2 = m_1 - k,$
$k_2 = y_2 - k_1$

imp

SYMBOL TABLE:- It is central repository of the facts to provide efficient access to it, compiler discovers and stores a wide variety of information from the program as given below.

1. Variables - stores datatype, name & scope
2. Arrays - Name, dimension, datatype, lower bound & upperbound.
3. functions & procedures - Argument list, function name, return type
4. records & structures - Fields & associated types

USES:

- provides centralized uniform access with the singl implementation
- localizes information from the source code.
- avoids the expense of searching from IR to find fac

2020.03.02 14:41

IMPLEMENTATION: Better method - Hash Table.

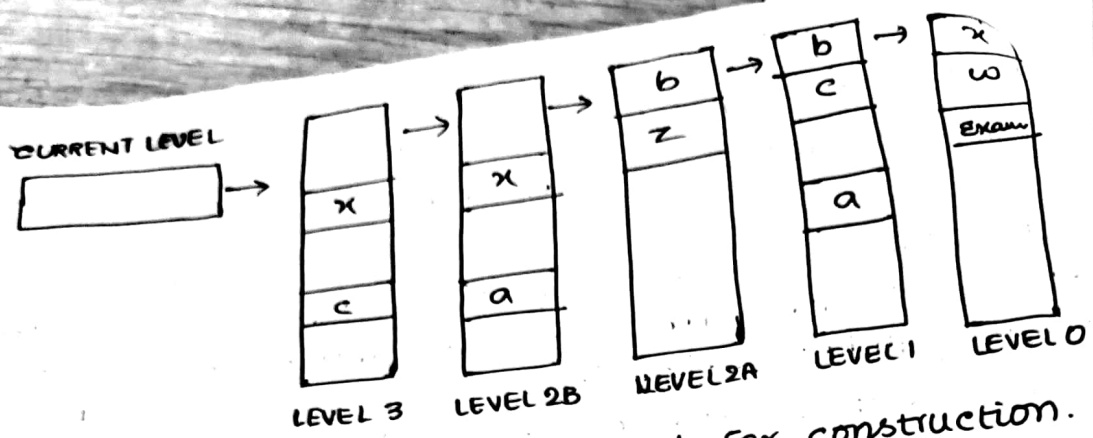
HASH TABLE: The key issue efficiency as a compiler access symbol table very frequently. It provides a constant time for storing and retrieval using hash.
- The hash function maps the facts to small integers and these integers are used to index the table, with hash-symbol table; the compiler stores all information that it derives about the name 'n' into the tableslot H(n).
- Building up a single table uses two routines,

1. LOOKUP(name): returns the record stored at H(name), if 1 exists and it returns a value indicating that the name was found (or) not

2. INSERT(name, record): stores the information in the hash table @H(name), it may expand the table to accommodate new class

SCOPE (Level)

```
static int w; } # LEVEL 0
int x;
void example (int a, int b) } # LEVEL 1
int c;
{
    int b,z;          } # LEVEL 2a
    :
}
{
    int a,x;          } # LEVEL 2b
    :
}
int c,x;              } # LEVEL 3
b = a+b+c+w;
```



Two more routines are used for construction.

1. Initialize scope
2. Finalize scope

MANAGEMENT OF SPACE:-

* Separate table for each record
* Selector table for field names that are identified.
* Unified table

NOTE:-

MEMO FUNCTION:- A function that avoid recomputation of prior results by storing in a hash table under a key built with its arguments.

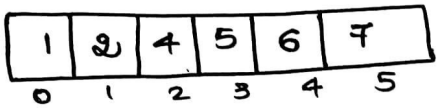
PURE FUNCTION:- A function always returning the same results.

3. OPTIMIZATION

to optimize code compiler uses various strategies & knowledge.

STORING & ACCESSING ARRAYS:

- collection of elements of same data type.
- requirements contigious memory location
- One dimension
  ↳ vector (6)



- array element ~~required~~ referred by ~~address~~/ index
- third element V[2] = 4.
- requirement offset of i^{th} element

FORMULA:

$(i - low) * w \rightarrow$ length in bytes

for int,
low = 0
w = 2

offset of 3
(3-0) * 2 = 6.

3 ADDRESS CODE FOR ACCESSING ELEMENT:

load I @V $\rightarrow r_{@V}$
sub I $r_{i,3} \rightarrow r_1$
Mult r_1 2 $\rightarrow r_2$
add $r_{@V}$, $r_2 \rightarrow r_3$
load $r_3 \rightarrow r_4$

MULTIDIMENSION ARRAY:

3 schemes
1. Row major order
2. column major order
3. Indirection vectors

ROW MAJOR ORDER:

- adjacent elements of single row occupies contigu location.

eg:

1,1	1,2	1,3	1,4
2,1	2,2	2,3	2,4

row major order is given by

1,1	1,2	1,3	1,4	2,1	2,2	2,3	2,4

loop for above,

```
for i ← 1 to 2
  For j ← 1 to 4
    A[i,j] ← A[i,j] + 1
```

ADDRESS CALCULATION:

$A[1,\ldots.2, 1,\ldots.4]$

Where $low_1 = 1$ & $high_2 = 4$

A[i,j]

row i & follow that offset for element j

each row 4 element

$len_K = high_K - low_K + 1$

$= 4 - 1 + 1 = 4$

For row base address $A[\overset{i}{2}, \overset{j}{3}]$

$(i - low_1) * len * w \rightarrow$ float (least index)

$(2-1) * 4 * 4 = 16.$

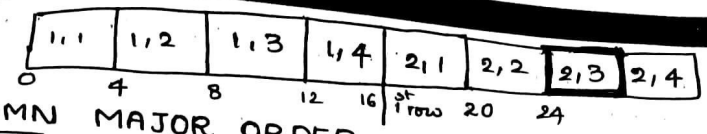
Address computation (A[2, 3])

row = @A + 16

col row

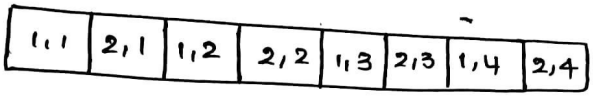
@A + 16 + (j − low₂) * w

⇒ @A + 16 + (3−1) * 4 ⇒ @A + 16 + 8 ⇒ @A + 24.



COLUMN MAJOR ORDER:

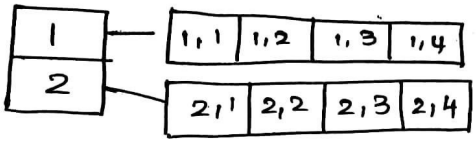
→ elements OF array mapped into adjacent locations OF single column.



→ non sequential access.

INDIRECTION VECTORS:

→ multi dimensional arrays into vectors.



elements address as vectors.

1. extra storage
2. Initialization code representation

CODE OPTIMIZATION: It is to discover information about the runtime behaviours at compile time. (forecasting)

Two types of compilers: 1. Debugging compilers
2. Optimizing compilers

OPTIMIZING CONSIDERATION:

1. Results should be same / safety concern (meaning shouldn't change)
2. profitability (NO:of iterations in pgm. should be reduced) & (Removing duplicates)
3. Identifies RISK

[Registered Demand should be minimum Address calculations]

OPPURTUNITY:

1. Reduce the override OF abstraction.
   eg: array address calculations.

2. matching the code to the system resources decrease memory allocation.
3. Takes advantage of special functions.

SCOPE OF OPTIMIZATION OR CATEGORIES:

1. Local methods
   → Basic blocks
2. Regional methods
   → Extended basic blocks larger than single block
3. Intraprocedural optimization
   → procedures
4. Interprocedural optimization
   → whole program

LOCAL METHODS:-

→ Optimization within basic blocks
→ local value numbering [LVN]
   ↳ Algorithm uses hashing to map names, constants & expressions into distinct value numbers.
→ IF "hash key" already found then
   replace
   else
   insert a new value

Ex: *

$a^3 \leftarrow b^1 + c^2$
$b^5 \leftarrow a^3 - d^4$
$c^6 \leftarrow b^5 + c^2$
$d^7 \leftarrow a^3 - d^4$

REWRITTEN CODE:- *

$a \leftarrow b+c$
$b \leftarrow a-d$
$c \leftarrow b+c$
$d \leftarrow b$

*

$a^3 \leftarrow x^1 + y^2$
$b^4 \leftarrow x^1 + y^2$
$a^5 \leftarrow 17$
$c^6 \leftarrow x^1 + y^2$

*

$a^3 \leftarrow x + y$
$b^4 \leftarrow a$
$a^5 \leftarrow 17$
$c^6 \leftarrow b$

* execution time is reduced

// LOCAL VALUE NUMBERING TECHNIQUE //

NE SUBSTITUTION :

→ code substituted

```
inline void sum (int a, int b)
{
    cout << a+b;
}
```

3. CONSTANT FOLDING:

① x = 3
y = x+2

② return (2+3)
optimized as
return (5);

compiler only optimizes the code.

Folding:

x = 3
y = 5

EXTENSIONS: Algebraic identities ⇒ for constant folding

1. a * b = b * a
2. a + b = b + a
3. A - 0 = A ← (constant folding)
4. A - A = 0
5. a * 1 = a
6. a * 0 = 0
7. a and a = a
8. a or a = a

LOCAL METHODS:

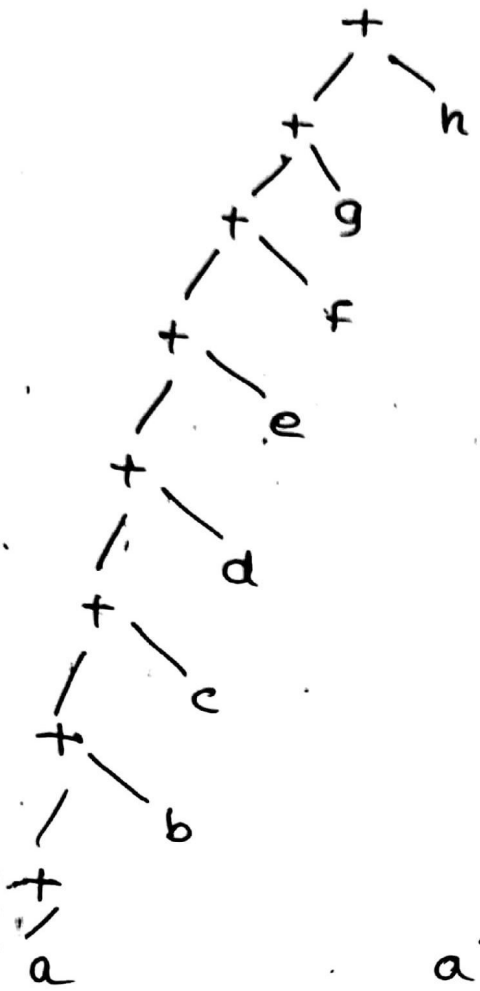
1. LVN
2. Tree height balancing

TREE HEIGHT BALANCING:

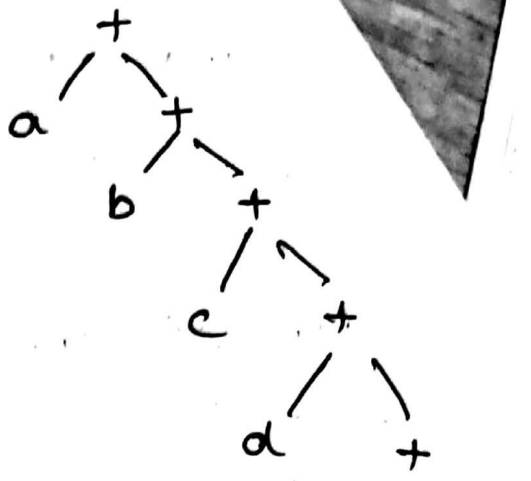
Eg:- a + b + c + d + e + f + g + h.

2020.03.02 14:42

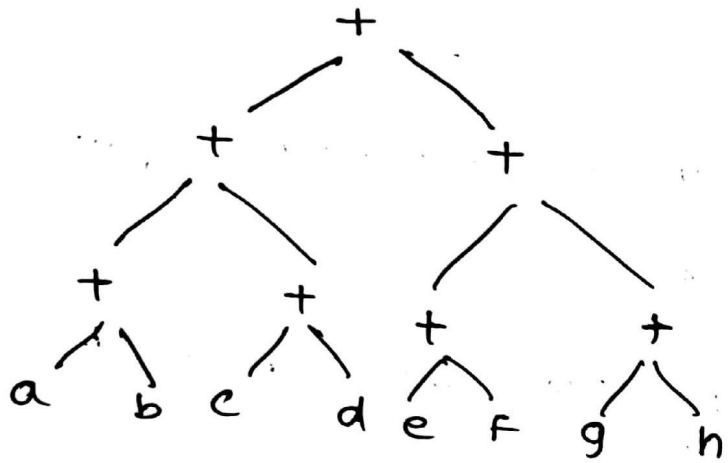
Left associative tree



right associate



BALANCED TREE:



Balanced tree

unit-0

1) $t_1 \leftarrow a+b$
2) $t_3 \leftarrow e+f$
3) $t_5 \leftarrow t_1+t_2$
4) $t_7 \leftarrow t_5+t_6$

unit-1 → processors

$t_2 \leftarrow c+d$
$t_4 \leftarrow g+h$
$t_6 \leftarrow t_3+t_4$

4) units both processors used

left Association

unit 0

$t_1 \leftarrow a*b$
$t_2 \leftarrow t_1+c$
$t_3 \leftarrow t_2+d$
$t_4 \leftarrow t_3+e$
$t_5 \leftarrow t_4+f$
$t_6 \leftarrow t_5+g$

unit 1

$t_7 \leftarrow t_6+h$

7 units
1 processor idle

ompiler identifies candidate expression in locks. all the expression in the candidate tree must be commutative and ~~Associative~~ Identitative (Associative)

* each name that labels the interior node must be used exactly once.
* for each candidate tree, the algorithm finds all its operands assigns each a rank and enters them into a priority queue ordered in ascending order. from this queue, the algorithm builds a balance tree
* several factors for identifying the candidate tree

1. Trees cannot be larger than the block
2. The Rewritten code cannot change the Observable values
3. Trees cannot extend backwards.

* The algorithm uses the following methods for finding the tree phase:
* The set uses(T_i) contains the index in the block for each use of T_i. If T_i is used after the block then uses(T_i) should contain '2' additional entries.

i.e., Integer > no:of Operations in the block.

TREE HEIGHT BALANCING :-

* The algorithm constructs maximum signed candidate tree using DG (Dependancy Graph) (for utilizing resources) ↑ usage

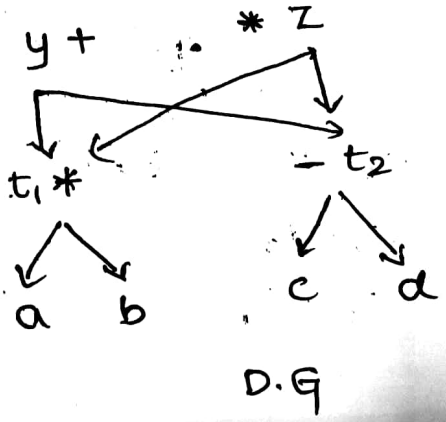
eg:

$t_1 \leftarrow a * b$

$t_2 \leftarrow c - d$

$y \leftarrow t_1 + t_2$

$z \leftarrow t_1 * t_2$



D.G

code

2020.03.02 14:42

SEVERAL FACTORS FOR TREE ID:

1. trees cannot be larger than block.
2. rewritten code cannot change its value
3. trees cannot extend backward.

ALGORITHM TREE FOR FINDING PHASE:

Each T_i def in the block.

(i) uses(T_i) contains the index in the block for each use(T_i).

If T_i used after a block, it should contain 2 additional entries greater than no: of operations in the block.

II REGIONAL OPTIMIZATION:

Larger than single block.

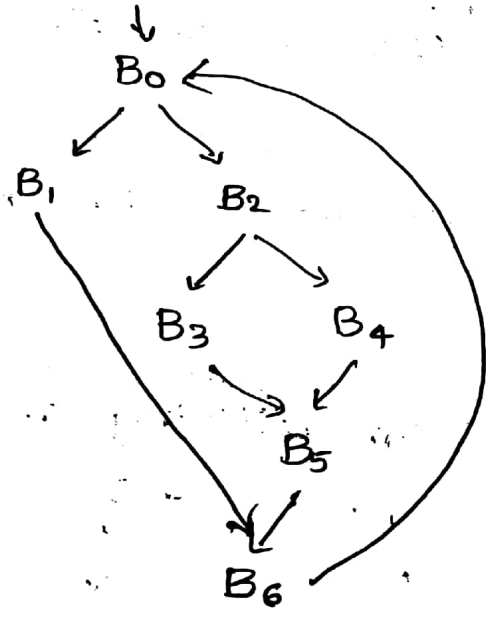
2 methods

(1) Super local value no (larger region) → used for redundant (or) duplicate identificati equation elimination in more than one block.

(2). loop unrolling.

CFG imp

CONTROL FLOW GRAPH



B_0 : $m_0 \leftarrow a_0 + b_0$
$n_0 \leftarrow a_0 + b_0$
$(a_0 > b_0) \rightarrow B_1, B_2$

B_1 : $P_0 \leftarrow c_0 + d_0$
$r_0 \leftarrow c_0 + d_0$
$\rightarrow B_6$

B_2 : $q_0 \leftarrow a_0 + b_0$
$r_1 \leftarrow c_0 + d_0$
$(a_0 > b_0) \rightarrow B_3, B_4$

B_3 : $i_0 \leftarrow b_0 + 18$
$s_0 \leftarrow a_0 + b_0$
$u_0 \leftarrow c_0 + t_0$
$\rightarrow B_5$

B_4 : $e_1 \leftarrow e_0 + 17$
$t_0 \leftarrow e_0 + d_0$
$v_1 \leftarrow e_1 + f_0$

B_5 : $e_2 \leftarrow \phi(e_0, e_1)$
$v_2 \leftarrow \phi(v_0, v_1)$
$v_0 \leftarrow a_0 + b_0$
$w_0 \leftarrow l_0 + d_0$
$x_0 \leftarrow e_2 + f_0$
$\rightarrow B_6$

B_6 : $r_2 \leftarrow \phi(r_0, r_1)$
$y_0 \leftarrow a_0 + b_0$
$z_0 \leftarrow c_0 + d_0$
$\rightarrow B_0$

} # for, loop functions we use the ϕ fn's in code loop

Algorithm include

1. B_5, B_6 :→ not important
2. B_0, B_1, B_2, B_3, B_4 → important

In B_0 : no ~~no~~. is redundant
B_1 : r_0 is redundant
B_2 : q_0, r_1 is redundant

2) LOOP UNROLLING

→ unroll inner loop → " loop fusion".
→ unroll outer loop → " Jamming".

UNROLL INNER LOOP:

```
do 60 j=1, n2
   do 50 i=1, n1
      y[i] = y[i] + x[j] * m(i,j)
50 continue
60 continue
```

UNROLL BY 3:

```
do 60 j=1, n2
   do 50 i=1,n
      nex = i mod 3
      IF (nex .ge. 1) then
         do 49 i=1, nex
            y[i] = y[i] * x[j] * m[i,j]
49 continue
         do 50 i = nex+1, n, 3
            y[i] = y[i] + x[j] + m[i,j]
            y[i+1] = y[i+1] + x[j] + m[i+1,j]
```

T100
T200
T300
T400
T500
T600
T700
T800
T900

min-su

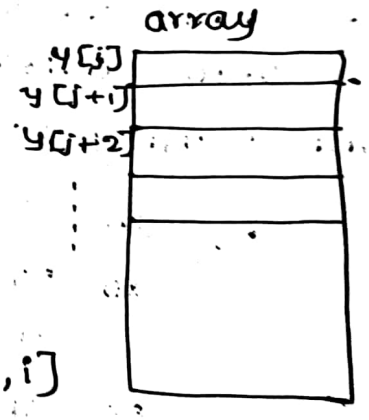
```
        y[i+2] = y[i+2] * x[j] + m[i+2, j]
    50 continue
60 continue
```

EQUIVALENT CODE:

```
for (i=1; i<n2; i++)
{
   for (j=1; j<i; j++)
   {
      y[j] = y[j] + x[i] * m[j,i]
   }
}
```

UNROLL BY 3:

```
for (i=1; i<n2; i++)
{
   for (j=1; j<i; j+=3)
   {
      y[j]   = y[j]   + x[i] + m[j,i]
      y[j+1] = y[j+1] + x[i] + m[j+1,i]
      y[j+2] = y[j+2] + x[i] + m[j+2,i]
   }
}
```



↳ Reduce test & branch sequence....

LOOP UNROLLING:

Outer loop (fusion)

eg:

```
DO I=1,N
   DO J=1,N
      DO K=1,N
         A[I,J] = A[I,J] + B[I,K] * C[K,J]
      end DO
   end DO
end DO.
```

ROLL OUTER LOOP BY THE FACTOR OF 2:

```
I=1,N,2
DO J=1,N
  DO K=1,N
     A[I,J]=A[I,J]*B .....
  end DO
end DO
DO j=1,N
  DO K=1,N
     A[I+1,J]=A[i+1,J]+B[i+1,K] * c[k,J]
  end DO
end DO
end DO.
```

USING INNERLOOP (JAMMING) UNROLL BY A FACTOR OF 3:-

GLOBAL OPTIMIZATION:

- an entire procedure
- As loops are included requires Global analysis before modifying.

STEPS:

1. finding uninitialized var with live information
2. Global code placement.

FINDING UNINITIALIZED VARIABLE WITH LIVE INFORMATION:

In a procedure B until a variable V is assigned a value, it is uninitialized.

uninitialized variable indicates a logical error. Hence, a programmer should be informed about its existence.

uninitialized variables can be identified by using liveliness.

LIVEVAR USE-

A variable 'v' is live at a point p iff th
exist a path in CFG from p to use(v) along
which v is not redefined.
We encode a set of liveout(b) that contains all
the variables that are live on emit from B

$$liveout(n) = \bigcup_{m \in succ(m)} (v \in VAR(m) \cup (liveout(m) \cap \overline{VARKILL(m)}))$$

(VEVAR) ⇒ Upward exposed variable in m, i.e., those
(UEVAR) ⇒ variables that are used in the m' before
any redefinition in m.

VARKILL(m) ⇒ set of variables not defined in m.

To complete the liveout sets the compiler uses 3
step algorithm:

(1) Build a CFG (control Flow Graph)
(2) Gather initial information by computing UEVAR and
VARKILL set for each block B
(3) Solve the equation for LIVEOUT(B) for each block

ITERATIVE LIVE ANALYSIS:-

// block b & Operations x ← y op z

```
for each block b
    init(b)
init(b)
    UEVAR(b) ← ϕ
    VARKILL(b) ← ϕ
    for i ← 1 to k
        if y ∉ VARKILL(b)
            then add y to UEVAR(b)
```

2020:03:02 14:43

LIVEVAR

A variable 'v' is live at a point p iff exist a path in CFG from p to use(v) along which v is not redefined.

We encode a set of liveout(b) that contains the variables that are live on emit from B.

$$liveout(n) = \bigcup_{m \in succ(m)} (UEVAR(m) \cup (liveout(m) \cap \overline{VARKILL(m)}))$$

(UEVAR) ⇒ Upward exposed variable in m, i.e., those (UEVAR) ⇒ variables that are used in the 'm' before anyredefinition in m.

VARKILL(m) ⇒ set of variables not defined in m.

To complete the liveout sets the compiler uses 3 step algorithm:

(1) Build a CFG (control Flow Graph)

(2) Gather initial information by computing UEVAR and VARKILL set for each block B.

(3) Solve the equation for LIVEOUT(B) for each block.

ITERATIVE LIVE ANALYSIS:-

// block b & operations $x \leftarrow y$ op z

```
for each block b
    init(b)
init(b)
    UEVAR(b) ← ϕ
    VARKILL(b) ← ϕ
    for i ← 1 to k
        if y ∉ VARKILL(b)
            then add y to UEVAR(b)
```

2020.03.02 14:43

```
if   z ∉ VARKILL(b)
     then add z to UEVAR(b)
Add x to VARKILL(b)
```

a) GATHERING INITIAL INFORMATION

SOLVING THE INITIAL EQUATION :-

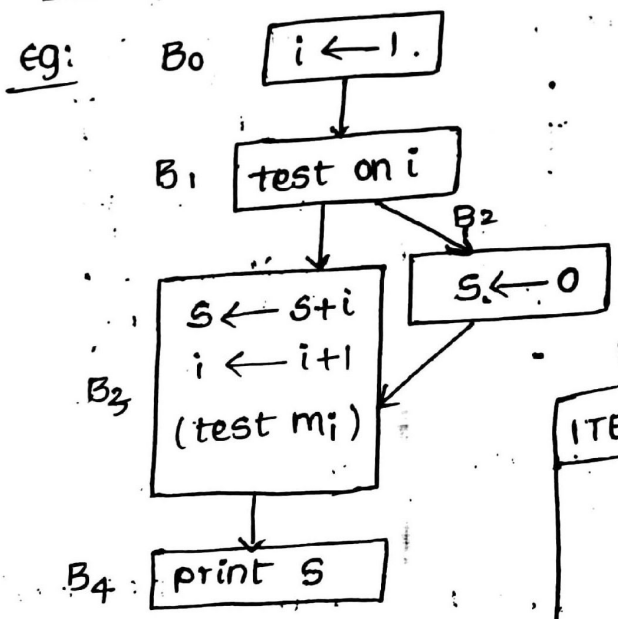
```
// CFG with V blocks 0 to n-1.
for i ← 0 to n-1
    liveout(i) ← ∅
changed ← true
while(changed)
    changed ← false
    for i ← 0 to n-1
        recompute LIVEOUT(i)
        IF (liveout(i) changed) then
        changed ← true.
```

UEVAR - used
VARKILL - assigned

Initial var

BLOCK	UEVAR	VARKILL
B_0	∅	i
B_1	i	∅
B_2	∅	S
B_3	S,i	S,i
B_4	S	∅

eg:



ITERATION	LIVEOUT B_0	B_1	B_2	B_3	B_4
	COMPARE SUCCESSOR BLOCK				
0	∅	∅	∅	∅	∅
1	i	i,S	S,i	S,i	∅
2	S,i	i,S	i,S	i,S	∅
3	S,i	i,S	i,S	i,S	∅

LIVEVAR USES:-

1. used for global register allocation
2. improve SSA construction
3. can discover useless store operation.

GLOBAL OPTIMIZATION:

2 techniques

i) → finally uninitialized var using Liveout

ii) → Global code placement.

→ Branching cost considered

→ fall through branch - either taken falls through or next instruction.

→ taken.

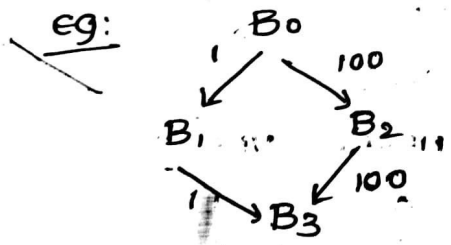
$x = y + 2 \rightarrow x = y$ or $x = 1$, $y = 1$, $z = 1$
$x = 2$

1. CRG
2. id VEVA & VARKIL
3. comput Liveout formul

→ each branch has either falls through (or) taken.

→ compiler can choose which block lives on it.

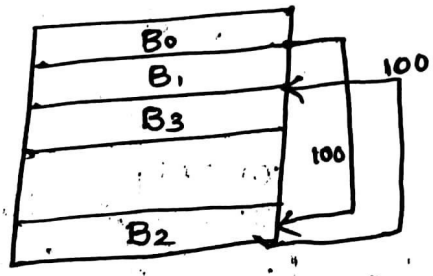
eg:



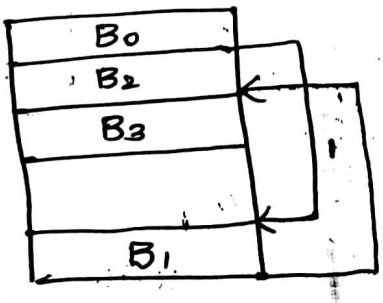
(B_0, B_2) → 100 times more often than (B_0, B_1)

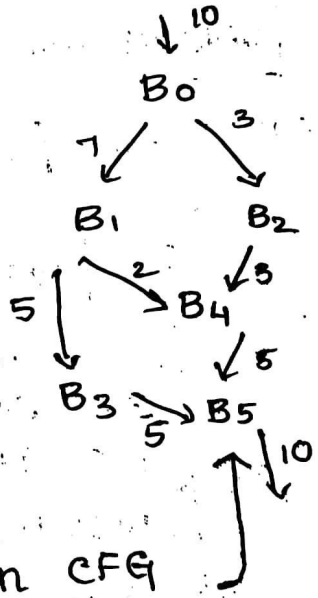
Two layouts:

1. Slow layout



Fast layout





Placement of code:-

→ constructing chains as hotpaths in CFG

→ layout of code

" COMPILER USES CFG PATHS THAT ARE MOST FREQUEN EXECUTED EDGES CALLED HOTPATHS".

...ying hot paths:

(edges)

for each block b

make generate chain d_i for b

P ← 0

for each CFG edge $\langle x, y \rangle$, $x \neq y$, in decreasing order.

IF 'x' is tail of chain a & 'y' is head of chain b then

t ← priority(a)

append b to a

priority(a) ← min(t, priority(b), P++)

(B_0,B_1) (B_3,B_5) (B_4,B_5) (B_1,B_2)
6 5 5

(B_0,B_2) (B_2,B_4) (B_1,B_4)
3 3 2

S	set of chains	P
	$(B_0)_\varepsilon (B_1)_\varepsilon (B_2)_\varepsilon (B_3)_\varepsilon (B_4)_\varepsilon (B_5)_\varepsilon$	0
(B_0,B_1)		
(B_3,B_5)		
(B_4,B_5)		
(B_1,B_2)		
(B_0,B_2)		
(B_2,B_4)		
(B_1,B_4)		

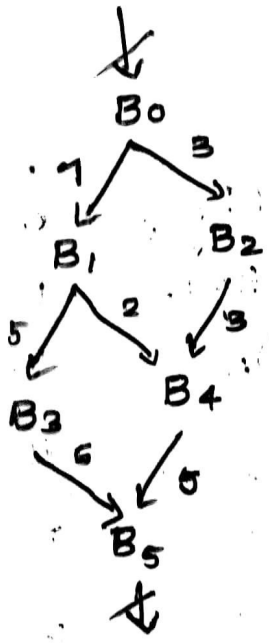
GLOBAL OPTIMIZATION:-

2 tools → Finding unintialized using Liveout
→ code placement using notpaths ⇒ (to minimize the jumps)

↓

(most frequent executable blocks)

1) degenerate block
2) find chains from decreasing order.



edges	chains	P
(B_0,B_1)		0
	$(B_0B_1B_3B_5)_0$ (B_2,B_4) 3	

CODE LAYOUT ALGORITHM:-

t ← chain headed by CFG entry node.

worklist ← {t, priority(t)}

while (worklist ≠ 0)

2020.03.02 14:43

remove a chain c & t
lowest priority from worklist
for each block x in c in chain order
place x at the end of execu. code
for each edge {x,y} where y is unplaced.
t ← chain Containing (x,y)
IF (t, priority(t) ∉ worklist then
worklist ← worklist ∪ {t, priority(t)}

CODEPLACEMENT: (utilizes branching controls)

Step	worklist	codelayout
-	$\{B_0, B_1, B_3, B_5\}_0$	
1	$\{B_2, B_4\}_3$	$B_0 B_1 B_3 B_5$
2	ϕ	$B_0 B_1 B_3 B_5 B_2 B_4$

*

Edges	chains						P
	$(B_0)_E$	$(B_1)_E$	$(B_2)_E$	$(B_3)_E$	$(B_4)_E$	$(B_5)_E$	0
(B_0, B_1)	$(B_0, B_1)_0$		$(B_2)_E$	$(B_3)_E$	$(B_4)_E$	$(B_5)_E$	1
(B_3, B_5)	$(B_0, B_1)_0$		$(B_2)_E$	$(B_3, B_5)_1$	$(B_4)_E$		2
(B_1, B_3)	$(B_0, B_1, B_3, B_5)_2$		$(B_2)_E$		$(B_4)_E$		3
(B_4, B_5)	$(B_0, B_1, B_3, B_5)_2$		$(B_2)_E$		$(B_4)_E$		3
(B_0, B_2)	$(B_0, B_1, B_3, B_5)_2$		$(B_2)_E$		$(B_4)_E$		3
(B_1, B_4)	$(B_0, B_1, B_3, B_5)_2$		$(B_2)_E$		$(B_4)_E$		3
(B_2, B_4)	$(B_0, B_1, B_3, B_5)_2$		$(B_2, B_4)_3$				4

Edges	chains	P
(B_0, B_1)	$(B_0)_E$ $(B_1)_E$ $(B_2)_E$ $(B_3)_E$ $(B_4)_E$ $(B_5)_E$	0
(B_4, B_5)	$(B_0, B_1)_D$ $(B_2)_E$ $(B_3)_E$ $(B_4)_E$ $(B_5)_E$	1
(B_1, B_3)	$(B_0, B_1)_0$ $(B_2)_E$ $(B_3)_E$ $(B_4, B_5)_E$	2
(B_3, B_5)	$(B_0, B_1, B_3)_0$ $(B_2)_E$ $(B_4, B_5)_E$	3
(B_0, B_2)	$(B_0, B_1, B_3)_0$ $(B_2)_E$ $(B_4, B_5)_E$	3
(B_2, B_4)	$(B_0, B_1, B_3, B_5)_0$ $(B_2)_E$ $(B_4, B_5)_E$	3
(B_2, B_4)	$(B_0, B_1, B_3, B_5)_0$ $(B_2, B_4, B_5)_E$	4
(B_2, B_4)	$(B_0, B_1, B_3)_0$ $(B_2, B_4, B_5)_E$	4

INTERPROCEDURAL OPTIMIZATION:-

* Having more no:of procedure in the program has the advantage where it limits the length of the code (decreases in size).

Disadvantages.

* limits the compiler's ability to understand, what happens inside a call.

Eg:- call by reference.

* As call by reference changes the actual arguments, it cannot use it for further Optimization.

* each call requires a pre & post overheads (content should be stored in stack & PC is in different location)

Two interprocedural Techniques are:

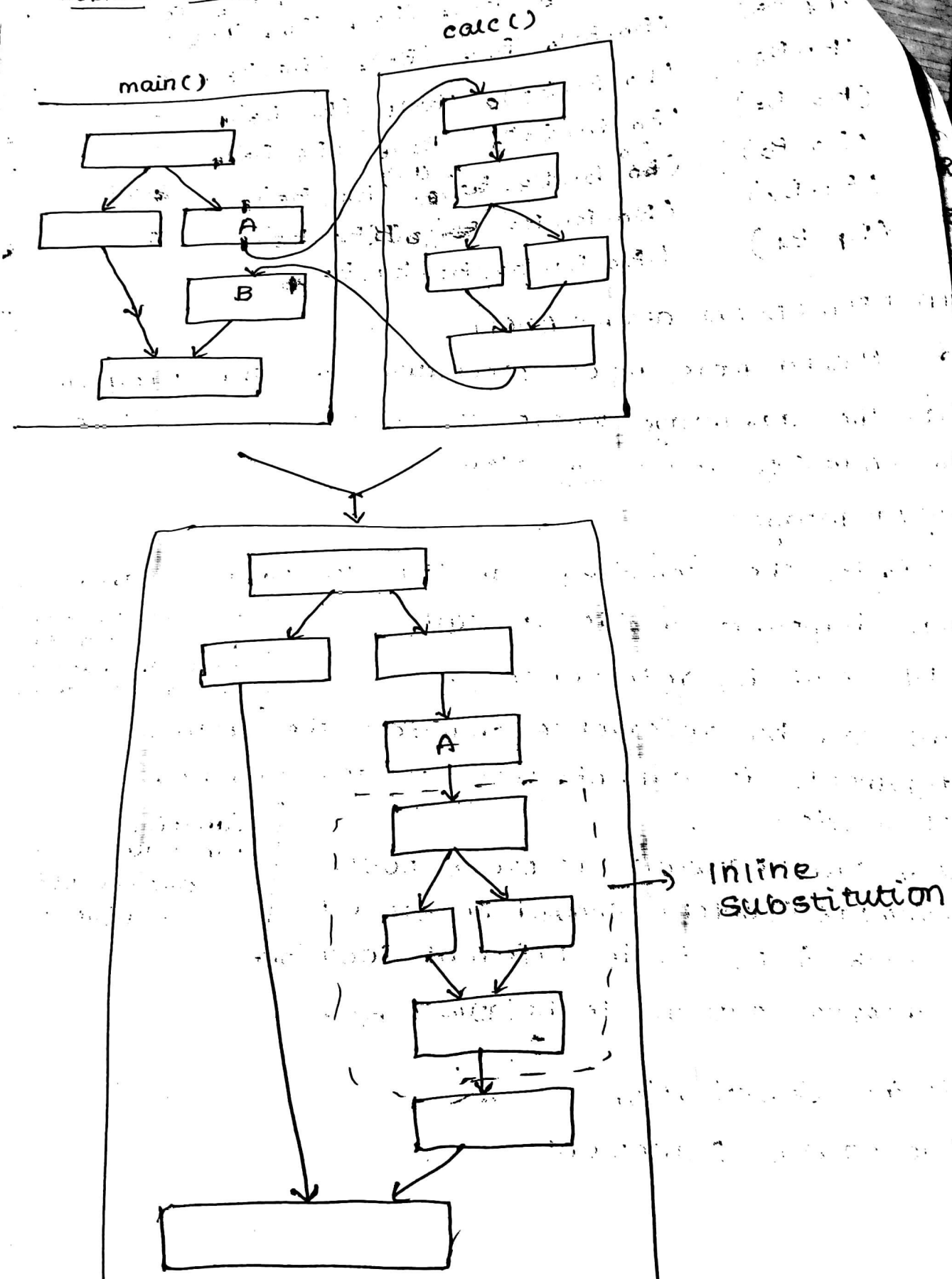
1. Inline Substitution
2. procedural placement.

In order to reduce the execution time, PC is involved & it's content changes

function procedure → overheads involved

INLINE SUBSTITUTION :-

* size is small
* less no: of calls to functio



callee and the call site, the criteria's include

1. The callee size, if the callee size is smaller to the procedure linkage (precall, postcall, epilog & prolog) Inlining the callee should reduce the code size and also executes with fewer operation
2. If the overall size of any procedure Be lesser
3. Dynamic call count: Improve on frequently executed call site than infrequently executed call site
4. The constant valued actual parameters and will definetly improve
5. Other parameter values includes a static call count. The parameters count looping & meeting depth & fraction of execution time.

PROCEDURE PLACEMENT:- Rearranging procedures using callgraph. A callgraph as a node for each procedure and an edge (x,y) for each call from x to y. procedure placement algorithm has more freedom than block layout and depends on cache

* The two phases are. 1. Analysis - Operates on call graph
2. Transformation

ANALYSIS:- Repeatedly selects two nodes, from callgraph and combines them. The order of the combination is driven by execution frequency data either measured or estimated.

TRANSFORMATION:- Dependent on the order of combination (or) layout.

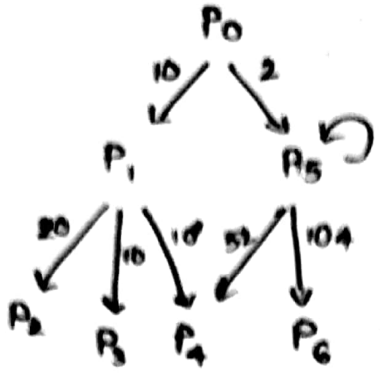
ALGORITHM: It operates over the programs caugra
iteratively by considering edges in the order of
their estimated execution frequency.

1. Builds a caugraph
2. Assigns each edges an execution frequency.
3. combines all edges between two nodes into
   a single edge.

INITIALIZE: // Build the call multigraph G.

```
for each edge (x,y) ∈ G        // add weight
    If (x=y) then
        delete (x,y) from G
    else
        wt((x,y))              // execute frequently
for each node x∈G              // init list
    List(x) ← {x}
If multiple edges exists from x to y then
    combine them & their actions
for each edge (x,y) ∈ G        } put each edge
    Enqueue (0,(x,y),wt(x,y)) } into Q
// iterative reduction of graph.
while Q is not empty // Fake highest
    (x,y) ← Dequeue(Q) - // priority else
for each edge (y,z) ∈ G.
    Resource ((y,z),x)         // move source, from y→x
for each edge (y,z) ∈ G // move target
    Retangent((z,z),x)         // from y to x
append list(y) to list // update ; delete y & its edges from
```

eg:-

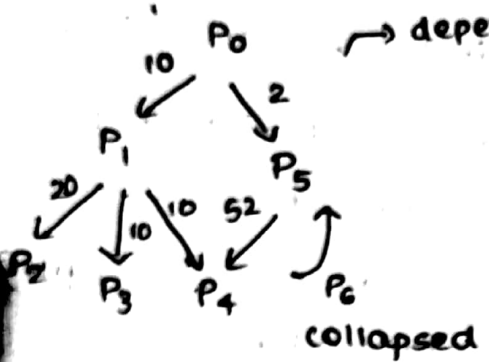


(self loops are not considered)

	P0	P1	P2	P3	P4	P5	P6
List(x)	P0	P1	P2	P3	P4	P5	P6

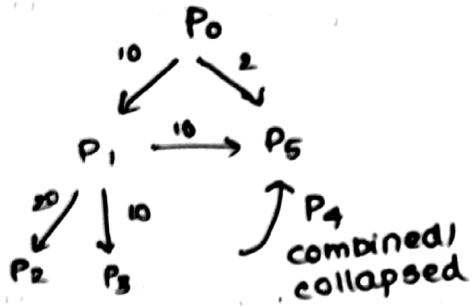
{ (P5 P6) (P5 P4) (P1 P2) (P1 P4) (P1 P3) (P0 P1) (P0 P5) }
104 52 20 10 10 10 2

→ depends on the execution frequency.



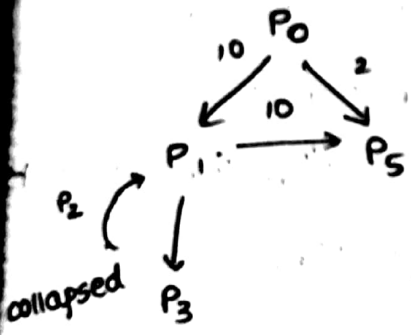
	P0	P1	P2	P3	P4	P5
List(x)	P0	P1	P2	P3	P4	{P5,P6}

{ (P5 P4) (P1 P2) (P1 P4) (P1 P3)
52 20 10 10
(P0 P1) (P0 P5) }
10 2



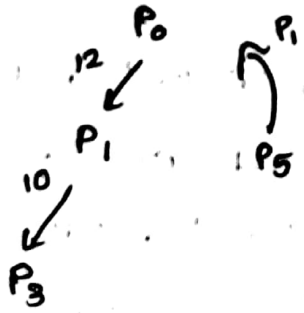
	P0	P1	P2	P3	P5
list(x)	P0	P1	P2	P3	{P5 P6 P4}

{ (P1 P2) (P1 P5) (P1 P3) (P0 P1)
20 10 10 10
(P0 P5) }
2



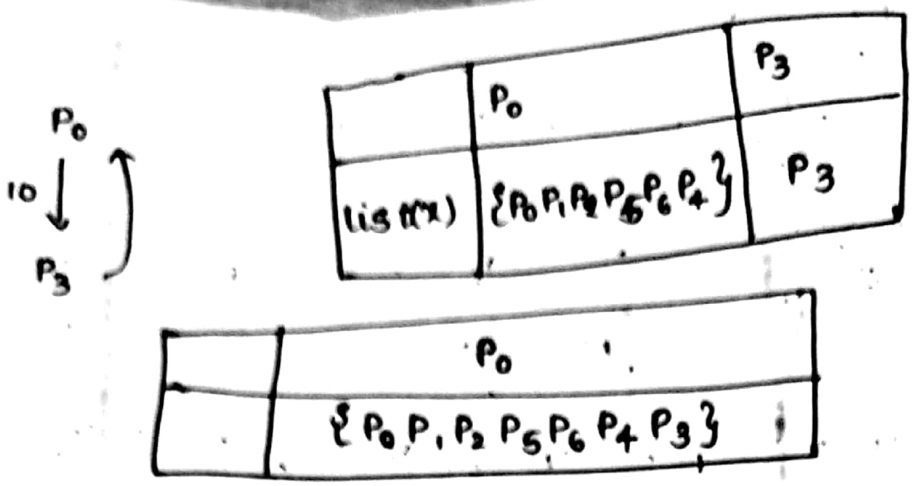
	P0	P1	P3	P5
list(x)	P0	(P1 P2)	P3	{P5 P6 P4}

{ (P1 P5) (P1 P3) (P0 P1) (P0 P5) }
10 10 10 2



	P0	P1	P3
list(x)	P0	{P1 P2 P5 P6 P4}	P3

(P1 P3) (P0 P1)
10 12



- DATA FLOW ANALYSIS: It is a classical technique for compile time program analysis and it allows to reason the run time flow of values in the program
  - * Before improving the code, the compiler must locate points in the program where modifying the code is likely to improve and safe
  - * compiler uses static analysis methods for run time flow of values.
  - * Two steps in static analysis are:
    1. control flow analysis
    2. Data flow analysis followed by SSA form.

By performing static analysis, the compiler will know exactly what will happen for instance,.

* either immediately display results (or) if it gets inputs from users and takes modest time to display results.

ITERATIVE DATA FLOW ANALYSIS: It is a collection of techniques for compile time reasoning on the run time flow of values to locate oppurtunities for optimization and to prove the safety of specific

T_{ID}	List of
T_{100}	I_1, I_2 I_4
T_{200}	I_2 I_4
T_{300}	I_2 I_3
T_{400}	I_1 I_2 I_4
T_{500}	T_3

...sformation.

...E STEPS: FOR ITERATIVE ANALYSIS:

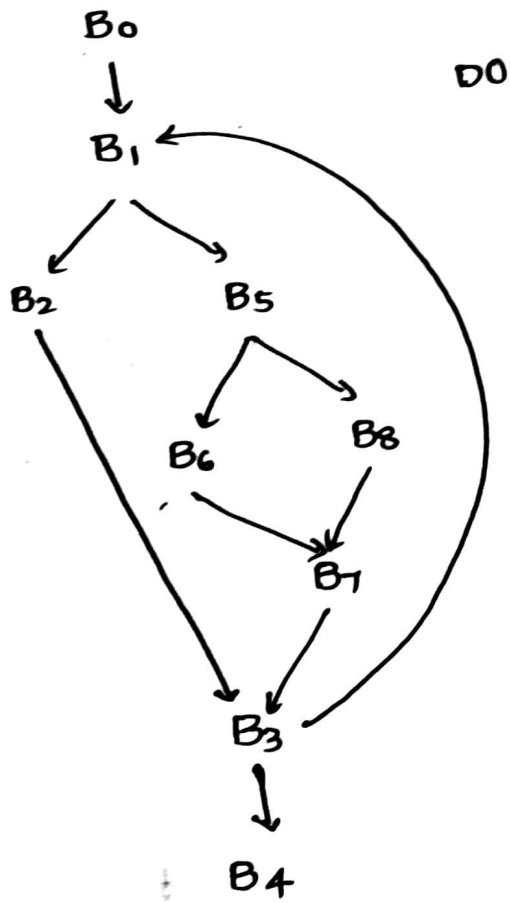
Build CFG

Gather initial information about blocks.

Solve equation to produce DOM (Dominance)

...minance is the shape and structure of CFG is ...e idea behind [dominance class].dominators.

...x:



$$DOM(n) = \{n\} \cup \left(\bigcap_{m \in pred_s(n)} DOM(m) \right)$$