15/03/2022

UNIT-1

Introduction

Structure of Compiler

Translater:-

Translaters convert high level language program to low level language programe (or) machine code.

The different translaters are :- (i) Compiler
(ii) Interpreter

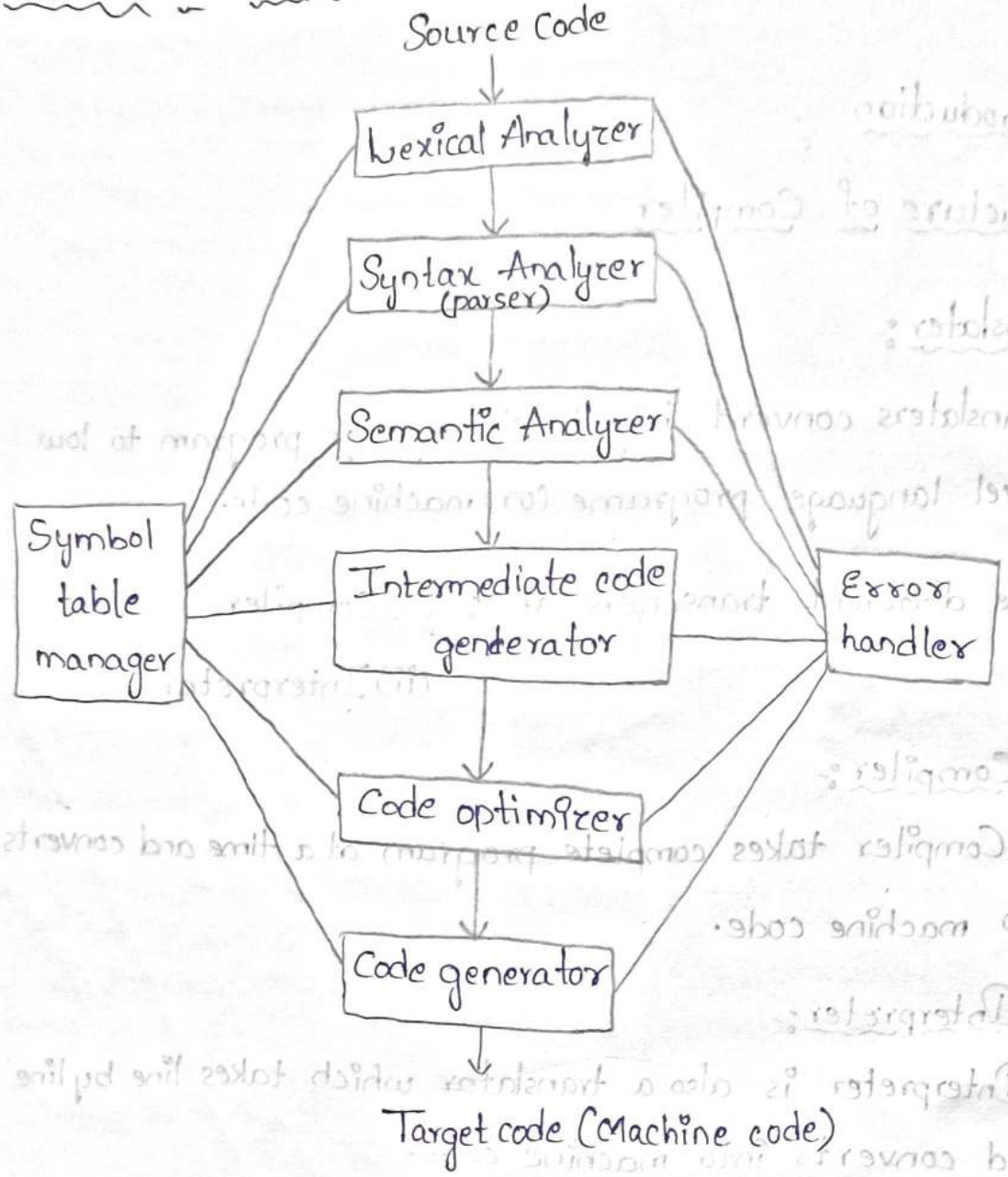
(i) Compiler:-

Compiler takes complete program at a time and converts into machine code.

(ii) Interpreter:-

Interpreter is also a translator which takes line by line and converts into machine code.

Structure of Compiler (or) Phases of a Compiler



(i) Lexical Analyzer:- It takes an instruction and divides it into a small units known as lexemes.(or) tokens.

Given instruction $c = a + b$

$c \longrightarrow$ Identifier

$= \longrightarrow$ assignment Operator

$a \longrightarrow$ identifier

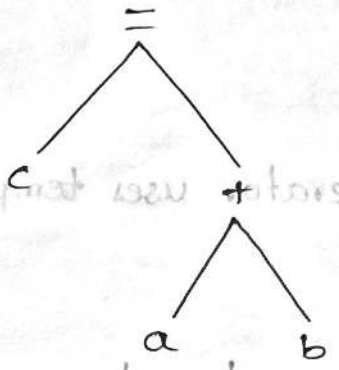
$+ \longrightarrow$ Addition operator

$b \longrightarrow$ Identifier

(ii) Syntax Analyzer :- Syntax analyzer is also known as Parser. It takes the instruction and converts into syntax tree (or) parse tree. With the help of this syntax tree errors can be identified.

Ex :- Given instruction c=a+b

The syntax tree for the given statement is



(iii) Semantic Analyzer :-

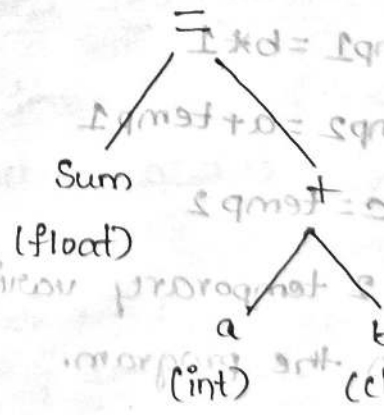
It scans the syntax tree and checks whether rules of a language is followed or not. That means, semantic analyzer checks the errors in the syntax trees. It also verifies the data types of identifiers.

Ex :-

Sum = a+b

Semantic Record

float	sum
int	a
char	b



There is an error in the syntax tree, because character integer cannot be added.

16/03/2022

(iv) Intermediate code generator :- It generates the code using temporary variables.

Ex:- `c = a+b*1;`

temp1 = 1

temp2 = b * temp1

temp3 = a + temp2

c = temp3

Intermediate code generator uses temperary

(v) Code optimizer :-

Code optimizer tries to reduce temporary variables and number of lines in the intermediate code. By doing this execution performance can be increased and memory will also be saved.

Ex:- `c = a+b*1;`

The above intermediate code can be optimized as follows:-

temp1 = b * 1

temp2 = a + temp1

c = temp2

Here only 2 temporary variables are used & only 3 lines are there in the program.

(vi) Code generator :- Code generator takes the given instruction & converts into assembly code (or) Pneumonic code. Th After that; assembly code is converted into machine. This machine code is known as Target code.

Ex:- `C=a+b*1`

assembly code (Pneumonic code)

	Machine code
Mov R1, b	10110
Mul R1, #1	11110
Mov R2, a	00101
Add R2, R1	10111
STR C, R2	11011

19/03/2022

The science of Building a Compiler

A compiler must accept any length of source code which is very small or large and transformation should be performed without changing the meaning of the program.

Modeling in Compiler design and Implementation :-

→ Design :- The compilers are design based on mathematical models like finite state machines and Regular expression.

→ These mathematical models are used to identify

3) Context free grammar:-

The context free grammar which is used to describe syntactical errors in given expression.

4) Science of Tree:-

The science of tree which is used in compiler used to detect errors in given expression.

Science of code optimization

1) Optimization:- The word optimization means increasing the efficiency and performance.

2) Tries to reduce unwanted code:- The science of code optimization tries to reduce unwanted code in given program & improves expression speed.

→ Data abstraction & inheritance concepts can be use to optimize the code in a program.

22/03/2022

Programming Language Basics: (evolution of computers, History of Computers)

The history of programming languages start from mechanical computers to modern tools for software development.

In early 1950's, assembly language is used. Here programming is done using pneumonics codes like add, sub, mul etc.,

→ In later half of 1950's, development of fortan is done for writing engineering & scientific application, and cobal was developed for writing business oriented programming.

→ After 1950's, many more languages were created with innovative features which helps easy programing, more natural & more robost.

Classification of Languages by generation:

→ In first generation, machine language is used for writting Program.

→ In 2nd generation, assembly language is used for writing program.

→ In 3rd generation, high level languages like fortran, cobal, C, C++ were developed for writing programs.

→ In 4th generation, languages design for specific applications like NOMAT, foxpro, sql. All these languages are used to store data in database.

→ In 5th generation, logical constrained based languages like prolog and OPS 5 (Official Production System Version 5). were developed which were used in expert system.

Object Oriented Languages

Object Oriented Languages support object oriented programming. This programming consists of objects which communicate with each other through message passing technique.

Scripting Languages

Scripting languages are developed for writing web applications..

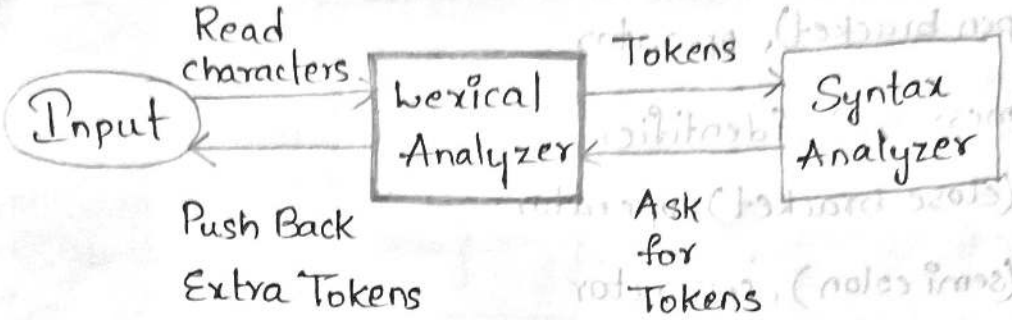
Ex: Java script, perl, php, ruby, TCL.

23/03/2022

Part-2

Lexical Analysis

The Role of Lexical Analyzer



Lexical analysis is the first phase of the compiler. It is also known as Scanner. It reads characters from the input statement and forms a meaningful tokens (or) lexemes and these tokens are given to Syntax analyzer.

function of Lexical Analyzer

i) Divide the program into valid tokens.

ii) Remove white space characters.

iii) Remove comments.

iv) It also provides help in generating error messages by providing row number and column number.

Ex-1:-

Suppose we pass a statement a=b+c, it will generate token sequence as given below:

a → identifier
= → assignment operator
b → identifier
+ → add operator
c → identifier

The above data is given to syntax analyzer.

Ex-2

```
printf("MIST");
```

printf → keyword

(→ (open bracket), operator

MIST → (message), identifier

) → (close bracket), operator

; → (semi colon), separator

24/03/2022

Input Buffering

Generally, lexical analyzer reads character by character of the given program & produces stream of tokens.

To read characters & produce token, two pointers are required they are

(i) Lexim begin pointer

(ii) Forward pointer

Generally, lbp is placed at starting point of the token while reading the characters.

FP reads character by character until it sees white space or comma, or semicolon or EOF. It is considered as end of token.

EOF → End Of File

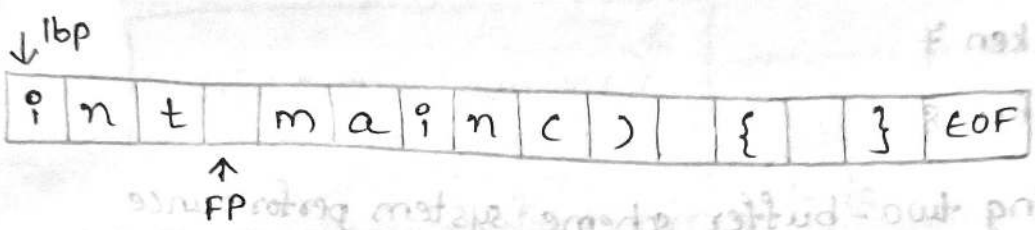
There are two schemes of input buffering.

(i) One buffer Scheme

(ii) Two buffer Scheme

(i) One-buffer Scheme

```
int main( )
{
}
```



Output

Stream of tokens

int – Token 1

main – Token 2

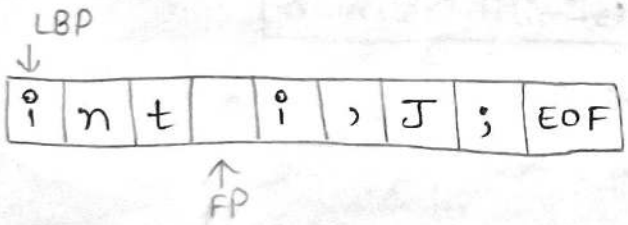
{ – Token 3

} – Token 4

(ii) Two-buffer Scheme

```
int i, j;
i = i+1;
```

Input



i	=	i	+	1	;	EOF			

Output

int – Token 1

i – Token 2

J = Token 3

i = Token 4

= – Token 5

i = Token 6

+ = Token 7

1 = Token 8

By using two-buffer scheme system performance increases compared with one buffer scheme.

25/03/2022

Recognization of Tokens

Tokens are recognized with the help of transition diagrams.

There are 5 types :-

(i) Recognization of Identifier

(ii) Recognization of Delimiter

(iii) Recognization of Relational Operator.

(iv) Recognization of keywords.

(v) Recognization of numbers.

(i) Recognization of Identifier :-

Generally, in C language there are some rules to be followed for identifiers.

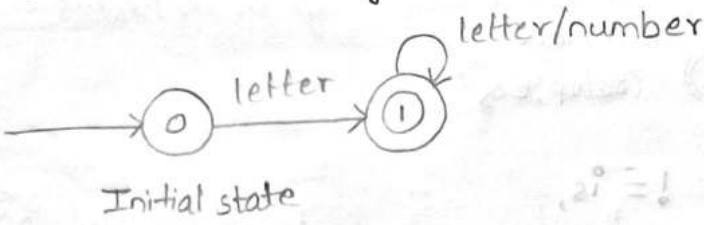
(i) Identifier should start with letter.

(ii) After first character, next character/data/element may be character (or) number.

Now, the production rule for recognizing for the identifier is

letter (letter | number)*

The transition diagram for above production rule is



28/03/2022

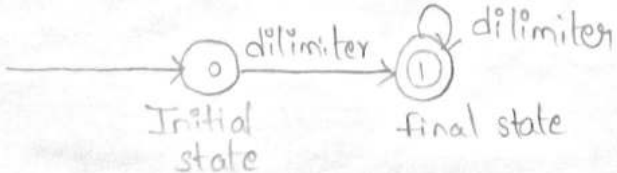
(ii) Recognizer of De-limiter

Blank Space (\b), tab space (\t), new line character (\n) are known as De-limiter. These de-limiter creates white space.

The production rule for recognization of De-limiter is

dilimiter (dilimiter)*

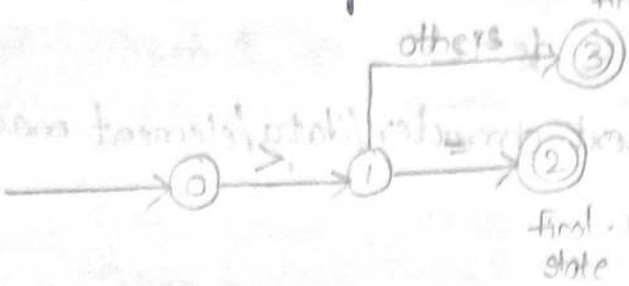
The transition diagram for de-limiters is



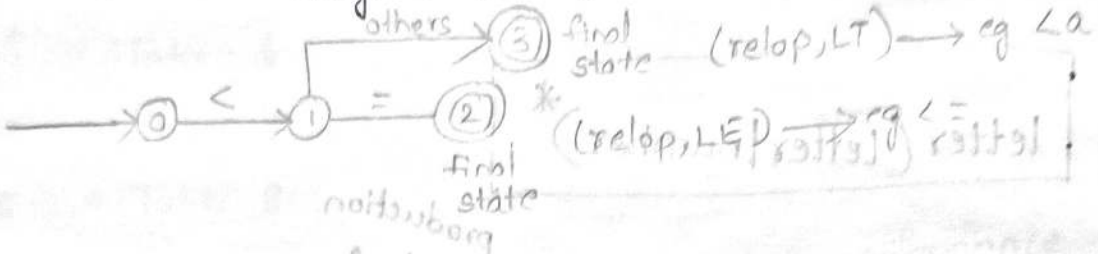
(iii) Recognization of Relational Operator:-

The relational operators are >, >=, <, <=, ==, !=

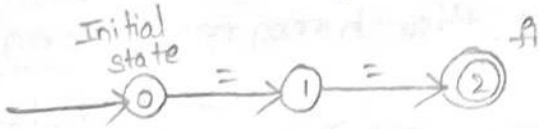
The transition diagram for >, >=, is



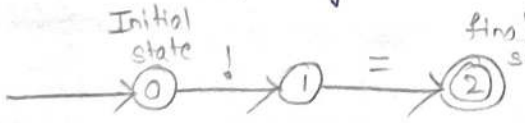
The transition diagram for <, <= is



The transition diagram for == is,



The transition diagram for != is,

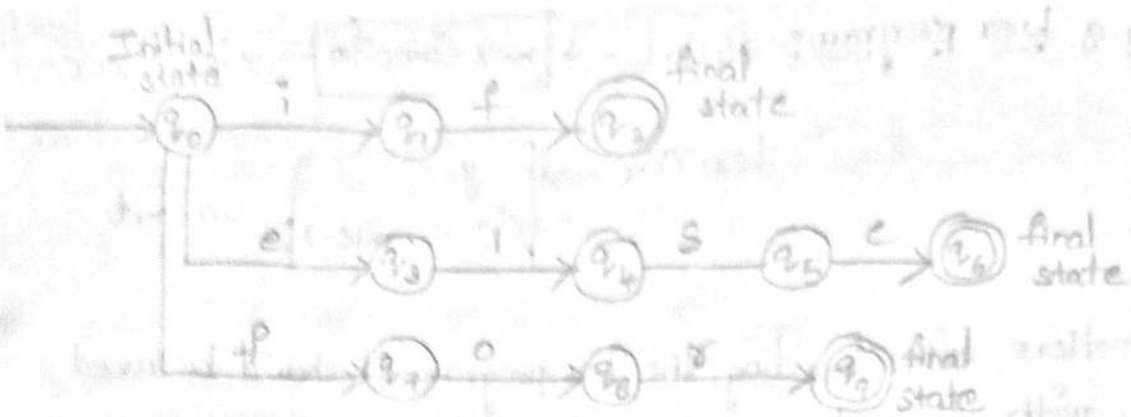


30/03/2022

(iv) Recognization of keywords:-

Keywords are also known as pre-defined words. Some of the keywords are if, else, for.

The transition diagram for above keywords is



(v) Recognization of numbers

Number is defined as a digit or group of digits.

Ex:- 6, 29

Production rule of numbers:

number = digit (digit)*

The transition diagram for numbers is

The lexical Analyser Generator lex:-

Syntax of lex program

header files and Macros	%{ ≡≡≡ %}
rules & defination	%% ≡≡≡ %%

Main function	main()
	{
	yylex();
	}

Running a lex program:- first.l → [lex Compiler] → lex.yy.c

lex.yy.c → [gcc (C-compiler)] → a.out

Step 01:- Here after writing the lex program it should be saved with .l extension.

Step 02:- Lex program should be given to lex compiler. It generates a C program known as lex.yy.C

Step 03:- Now the C program (lex.yy.c) should be given to C-compiler. Now, the C-compiler generates on.

By executing a.out file we get output.

Commands:-

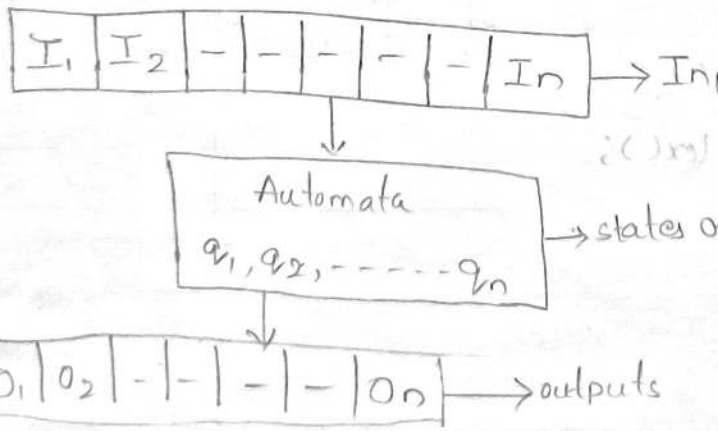
```
>flex first.l
>gcc lex.yy.c
>./a.out
```

This is how we can run lex program with linux.

finite Automata

Finite Automata is used to recognize set of patterns. finite automata has 5 elements or 5 tuples. It has set of states and rules.

The diagram for finite automata is



01/04/2022

Finite Automata consists of the following elements

$$FA = \{Q, \Sigma, q, F, \delta\}$$

$Q \longrightarrow$ Finite set of states.

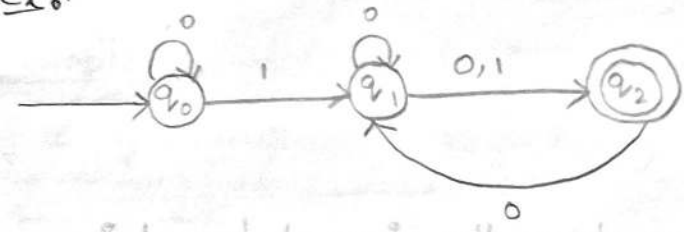
$\Sigma \longrightarrow$ set of input symbols

$q \longrightarrow$ Initial state

$F \longrightarrow$ set of final states.

$\delta \longrightarrow$ Transition function.

Ex:-

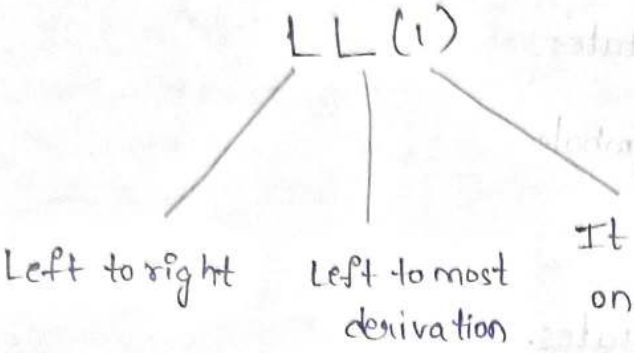


The transition table for above diagram is

Present state	Input (0)	Input (1)
q_0	q_0	q_1
q_1	q_1, q_2	q_2
q_2	q_1	—

Predictive Parsing (LL(1) parser) :-

It compares the statement with the given grammer & tells whether the given statement is correct or not.



Data structure used in LL(1) parser are :-

i) Input buffer

ii) Parsing table

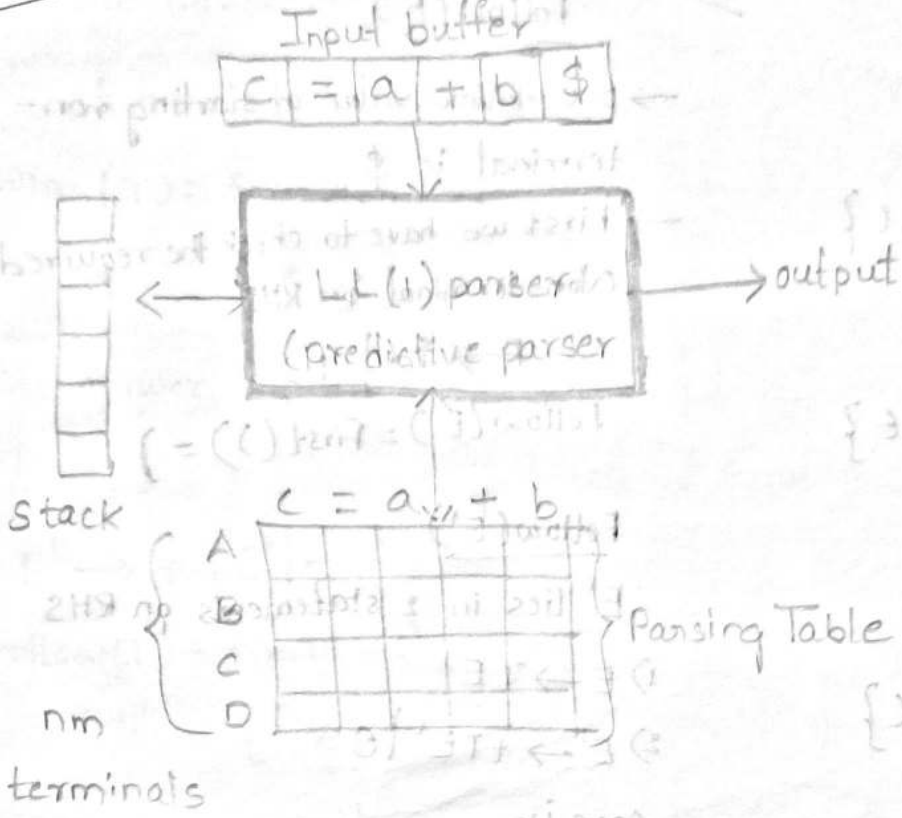
iii) Stack

i) Input buffer :- LL(1) parser stores the given statement in input buffer.

ii) Parsing table :- It converts given grammar into parsing table.

iii) Stack :- With the help of parsing table & stack concept LL 1 parser tells whether the given statements is correct (or) not.

6/04/2022



Example

given grammar	First(NT)	Follow(NT)
$E \rightarrow TE'$	{id, (}	{$,)}
$E' \rightarrow +TE' \mid \epsilon$	{+, ∈}	{$,)}
$T \rightarrow FT'$	{id, (}	{+, $,)}
$T' \rightarrow *FT' \mid \epsilon$	{*, ∈}	{+, $,)}
$F \rightarrow id \mid (E)$	{id, (}	{*, +, $,)}

Follow (NT) — first (T/NT)

To convert given grammar into parsing table we have to find out first(Non-Terminal) and follow(Non-Terminals)

28/04/2022

First(E)

First(E) = T

First(T) = F

First(F) = {id, (}

First(E')

First(E') = {+, ∈}

First(T)

First(T) = F

First(F) = {id, (}

first(T')

First(T') = {*, ∈}

First(F)

First(F) = {id, (}

Follow(E)

→ one follow value of starting non-terminal is $.

→ First we have to check the required Non-terminal on RHS

F → id | (E)

Follow(E') = First( ) =)

Follow(E')

E' lies in 2 statements on RHS

1) E → TE'

2) E' → +TE' | ∈

Case i:-

Follow(E') = Follow(E) = {$,)}

Case ii:-

Follow(E') = Follow(E') = {$,)}

Follow(T)

T lies in 2 statements on RHS

1) E → TE'

2) E' → +TE' | ∈

Case i:-

~~Follow(T) = First(E') = {+, ∈}~~

Case ii:-

~~Follow(T) = Follow(E') = {$,)}~~

Follow(T')

Case (i):- Follow(T) = First(E')

= {+, ∈}

Now substitute ϵ in E' of given statement

$$E \rightarrow T$$

$$\text{Follow}(T) = \text{Follow}(E)$$

$$= \{\$,)\}$$

Final answer in case 1 is $\{+, \$,)\}$

Case (ii):-

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

$$\text{Follow}(T) = \text{First}(E')$$

$$= \{+, \epsilon\}$$

Substitute ϵ in E' of givin statement.

we get

$$E' \rightarrow +T$$

$$\text{Follow}(T) = \text{Follow}(E)$$

$$= \{\$,)\}$$

final Answer $= \{+, \$,)\}$

Follow (T')

T' lies in 2 statements on RHS

1) $T \rightarrow FT'$

2) $T' \rightarrow *FT' | \epsilon$

Case(i):- $\text{Follow}(T') = \text{Follow}(T)$

$$= \{+, \$,)\}$$

Case (ii) :-

Follow(T') = Follow(T')

= {+, $,)}

Final answer is {+, $,)}

Follow(F)

F lies in 2 statements in RHS

1) $T \rightarrow FT'$

2) $T' \rightarrow *FT' | \epsilon$

Case (i) :-

Follow(F) = First(T')

= {*, ϵ}

Now substitute ϵ in T' in given statement

$T \rightarrow F$

Follow(F) = Follow(T)

= {+, $,)}

Final answer is {*, +, $,)}

Case (ii)

follow(F) = First(T')

= {*, ϵ}

Now substitue ϵ in T' in given statement

$T' \rightarrow *F$

Follow(F) = Follow(T')

= {+, $,)}

Final answer in case 2 = {*, +, $,)}

11/04/2022 ↓ continuation

Construction of Predictive Parsing table

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

Rule 1:-

If the production rule is in the form of $A \to \alpha$, then find first(α). We get terminal values as answers.

In the table at terminal cells fill the rule $A \to \alpha$.

Rule 2

If the production rule is $A \Rightarrow$ in the form of $A \to \epsilon$, then find follow(A), we get terminal values as answer.

In the table at terminal cells, fill the rule $A \to \epsilon$.

1) $E \to TE'$

First(T) = {id, (}; Fill $E \to TE'$ at id & (in table

2) $E' \to +TE' | \epsilon$

~~First(E')~~

First(+) = {+}

So, fill $E' \to +TE'$ at + in table

$E' \rightarrow \epsilon$

Follow(E') = {$,)}

So, fill $E' \rightarrow \epsilon$ at $ and at) in table

3) $T \rightarrow FT'$

First(F) = {id, (}

So, fill $T \rightarrow FT'$ at id & at (in table

~~4) $T' \rightarrow FT'$~~

4) $T' \rightarrow *FT' | \epsilon$

$T' \rightarrow *FT'$

First(*) = {*}

So, fill $T' \rightarrow *FT'$ at * in table

$T' \rightarrow \epsilon$

Follow(T') = {+, $,)}

So, fill $T' \rightarrow \epsilon$ at +, $,) in table.

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

$F \rightarrow id$

First(id) = {id}

So, fill $F \rightarrow id$ at id in table

$F \rightarrow (E)$

First(( ) = ((

So, fill $F \rightarrow (E)$ at (in table

Example

w = id * id

Stack
(Initially stack $ symbol) $
$E
$E'T
$E'T'F
$E'T'id
$E'T'
$E'T'F*
$E'T'F
$E'T'id
$E'T'
$E'
$E'T*
$E'T
$E'T'F

Example

w = id * id + id

Stack	Input Symbol	Production rule from table
(Initially stack contain $ symbol) $	id * id + id $	
$ 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 $	
$ E'T'	* id + id $	T' → * FT'
$ E'T'F*	* id + id $	
$ E'T'F	id + id $	F → id
$ E'T'id	id + id $	
$ E'T'	+ id $	T' → ∈
$ E'	+ id $	
$E'	+ id $	E' → +TE'
$E'T+	+ id $	
$ E'T	id $	T → FT'
$ E'T'F	id $	F → id

Reverse

Reverse

Reverse

$E'T'id	id$	
$E'T'	$	$T' \to \epsilon$
$E'\epsilon	$	
$E'	$	$E' \to \epsilon$
$\epsilon	$	
$	$	

- 11/04/2022

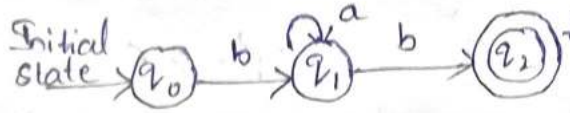
From Regular Expressions to Automata

1) $b\,a^*\,b \to ba^0b, ba^1b, ba^2b, ba^3b, \ldots$

$bb, bab, baab, baaab, baaaab \ldots$

After expanding regular expression is shown above.

The finite Automata for above regular expression is



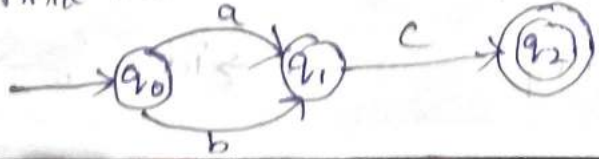
2) $(a+b)c$

After expanding regular expression is as follows

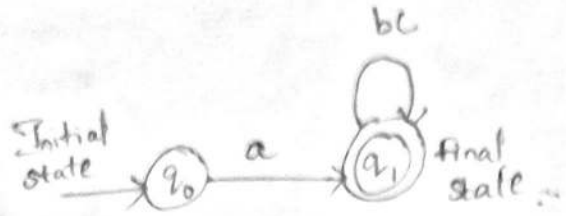
The given expression can also be written as

(a OR b)c

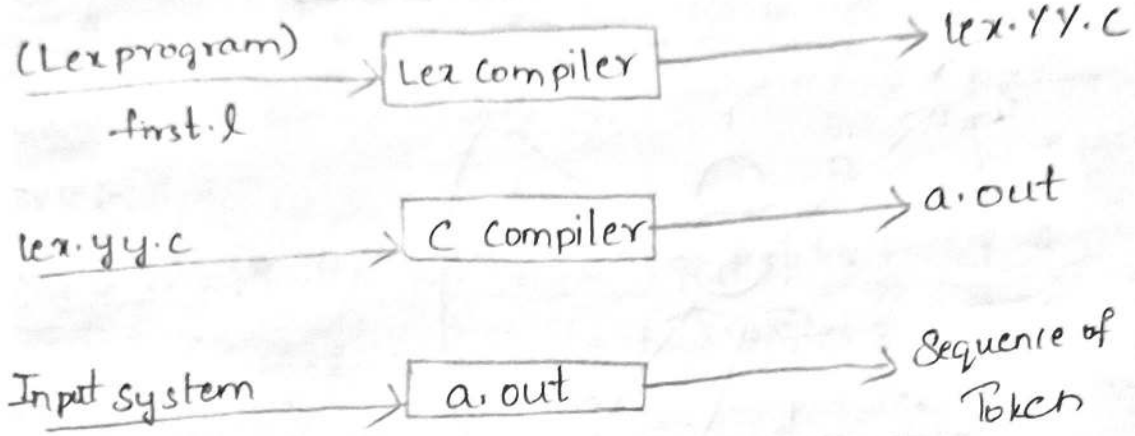
The finite Automata for the above regular expression is



3) $a(bc)^*$



Design of a lexical Analyzer Generator



Step 01 :-

Generally, lex program is save with .l extension. The lex program should be given to lex compiler. Lex compiler converts lex program to C program which is in the form of lex.yy.c

Step 02 :-

In this phase C program is given to C compiler which is converted into an executable file a.output.

Step 03 :-

In this phase an input stream (c=a+b) is given to an executable file a.out.

Here a.out generates a sequence of tokens for the given input stream. So the output is c → id

c - id

= - operator

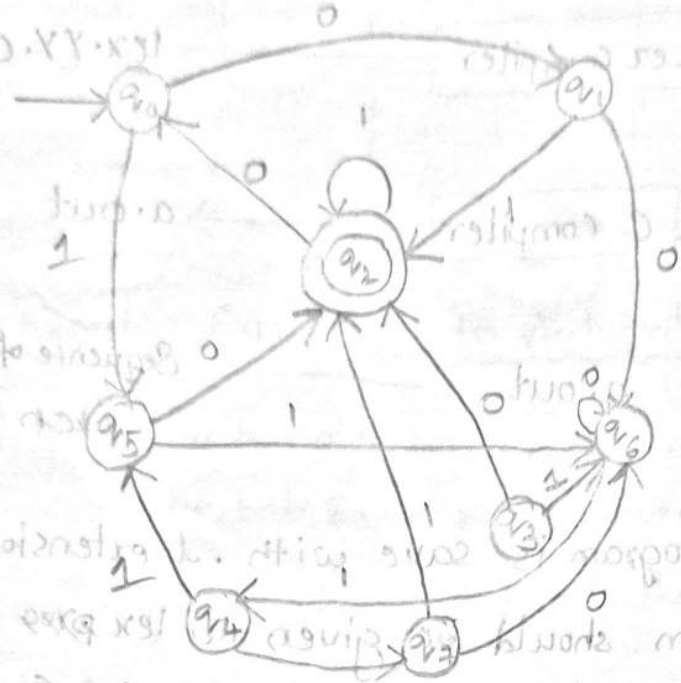
a - id

+ - operator

b - id

12/04/2022

Optimization of DFA based pattern matchers



Compressing the above finite automata is also known as optimization of finite automata.

Transition table for the above DFA is:

State	0	1
$\rightarrow q_0$	q_1	q_5
q_1	q_6	q_2
$*q_2$	q_0	q_2

q_3

q_4

q_5

q_6

q_7

Step 01:-

Set of Q = {q

from above set

states

Final state i

intermediate

Step 02:-

Find out the

state by applyi

{q_2} {q_0,q

Thus we get

{q_2} {q_0 q_2

q_3	q_2	q_6
q_4	q_7	q_5
q_5	q_2	q_6
q_6	q_6	q_4
q_7	q_6	q_2

Step 01:-

Set of Q = $\{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7\}$

from above set of Q seperate final states and intermediate states.

Final state is $\{q_2\}$

intermediate states are $\{q_0, q_1, q_3, q_4, q_5, q_6, q_7\}$

Step 02:-

Find out the intermediate states which are reaching final state by applying any type of input and separate them.

$\{q_2\}$ $\{q_0, q_1, q_3, q_4, q_5, q_6, q_7\}$

↓ ↓ ↓ ↓ ↓

1 0 0 0 1

Thus we get

$\{q_2\}$ $\{q_0\ q_4\ q_6\}$ $\{q_1, q_3, q_5, q_7\}$

Step 03:-

set of intermediate

From the above input states separate the states which are reaching final states by applying same input:

$\{q_2\}$ $\{q_0\ q_4\ q_6\}$ $\{q_1\ q_3\ q_5\ q_7\}$

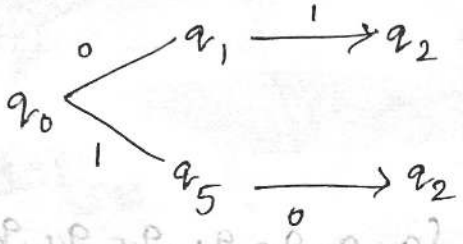
1 ↓ q_2 , 0 ↓ q_2 , 0 ↓ q_2 , 1 ↓ q_2

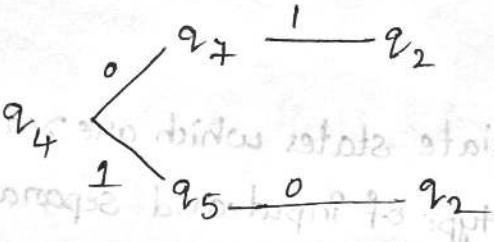
Thus we get

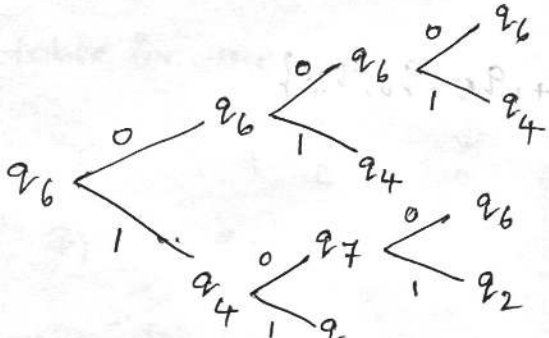
$\{q_2\}$ $\{q_0, q_4\ q_6\}$ $\{q_1, q_7\}$ $\{q_3\ q_5\}$

Step 04:-

Let us take set $\{q_0\ q_4\ q_6\}$ and subdivide it by considering reaching the final state in one step.







So, from the different.

So, the inter

$\{q_2\}$ $\{q_0\ q_4$

The transition is:

States	A	B	C	E	D
→ $\{q_0, q_4\}$					
$\{q_1, q_7\}$					
$\{q_3\ q_5\}$					
* $\{q_2\}$					
$\{q_6\}$					

Optimized DFA

State	A	B	C	* E	D
→ A					
B					
C					
* E					
D					

So, from the above diagram q_0, q_4 are similar and q_6 is different.

So, the intermediate codes we get are

$\{q_2\}\{q_0 q_4\}\{q_6\}\{q_1 q_7\}\{q_3 q_5\}$

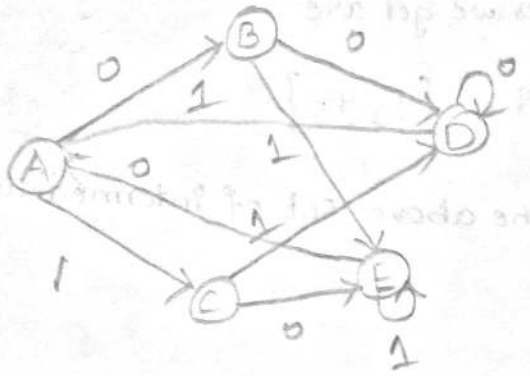
The transition table for the above set of intermediate codes is:

States	0	1
A → $\{q_0 q_4\}$	$\{q_1 q_7\}$	$\{q_5 q_5\}$
B $\{q_1 q_7\}$	$\{q_6\}$	$\{q_2\}$
C $\{q_3 q_5\}$	$\{q_2\}$	$\{q_6\}$
E * $\{q_2\}$	$\{q_0 q_4\}$	$\{q_2\}$
D $\{q_6\}$	$\{q_6\}$	$\{q_0 q_4\}$

Optimized DFA transition table

State	0	1
→ A	B	C
B	D	E
C	E	D
* E	A	E
D	D	A

Now let us draw the optimized transition diagram is



THE END

3/04/2022

UNIT-2

Part-1

Syntax Analysis

Introduction

* Syntax analysis is the second phase of the compiler design. It checks syntactical structure of given input. i.e., it checks whether given input is correct or not.

* Syntax analysis generates a syntax tree or parse tree to find the given input is correct or not.

* The syntax is generated from the given grammer.

Ex:-

Given grammar

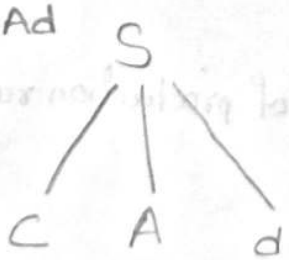
$S \rightarrow CAd$

$A \rightarrow bc/a$

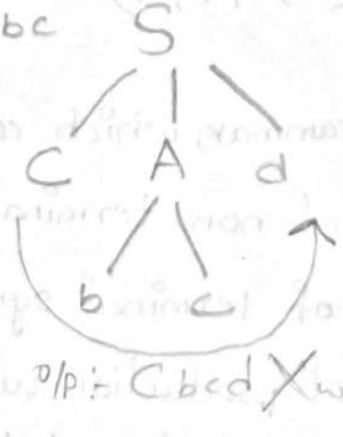
Given Input string is Cad

Check whether Input string is correct or not for given grammar.

$S \rightarrow CAd$

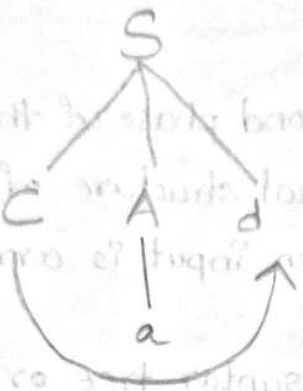


$A \rightarrow bc$



As per the obtained syntax tree the given input is not matching with the leaf node.

So, backtracking and substitute A→a



o/p :- Cad

In the above tree the output is matching with the given input. So, the given input is syntactically correct input.

19/04/2022

Context-free Grammar

A Context-free grammar is a formal grammar, which is used to generate all the possible patterns of strings.

It is defined as a 4 tuples.

$$G = (V, T, P, S)$$

where

G is grammar, which consists of set of production rules.

V is set of non-terminal symbols.

T is set of terminal symbols.

P is set of production rules which are used to replace non-terminals

S is starting symbol used to derive the string.

Example:- Construct CFG for the language having any no. of a's over the set $\Sigma = \{a\}$.

Sol:- $RE \Rightarrow a^*$

production rules for above RE is

$S \rightarrow aS \longrightarrow$ rule 1

$S \rightarrow \epsilon \longrightarrow$ rule 2.

The series of a^* is

a, aa, aaa, aaaa, aaaaa

Let us derive aaaaa is correct for above grammar.

S $\longrightarrow$ start symbol

as $\longrightarrow$ rule 1

aas $\longrightarrow$ rule 1

aaas $\longrightarrow$ rule 1

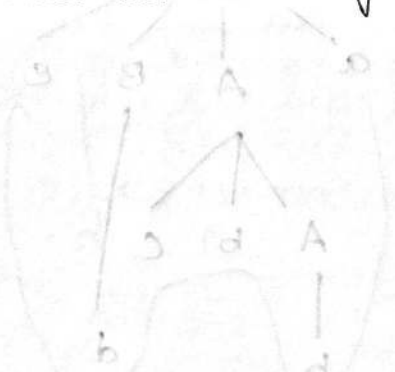
aaaas $\longrightarrow$ rule 1

aaaaas $\longrightarrow$ rule 1

aaaaaϵ $\rightarrow$ rule 2

$\therefore$ aaaaa is correct.

The string aaaaa is correct for above given grammar.



Top-down parsing

In Top-down parsing technique parse-tree is constructed from top to bottom. Here the input is read from left to right.

Here parse tree is generated from starting symbol.

Let us consider an example where grammar is given and you need to construct a parse tree by using Top-down parsing technique to obtain given input string.

Given grammar

→ S → aABe

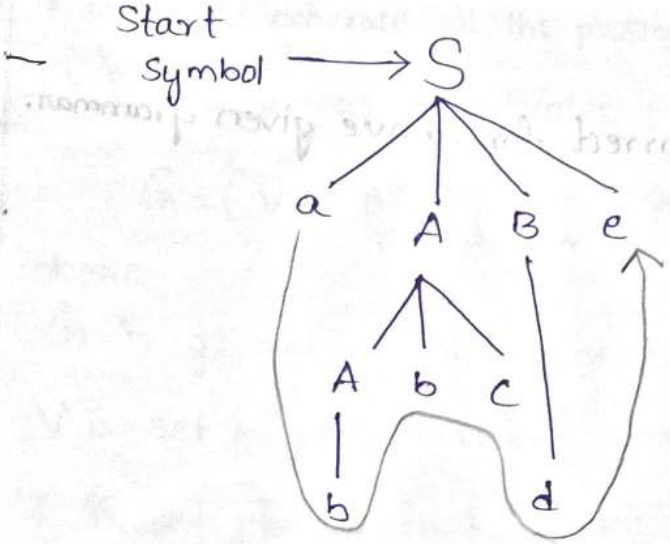
A → Abc | b

B → d

Given input string

abbcde

Top down parsing to obtain Input string.



Bottom-Up parsing

→ Bottom-Up parsing will start from Input string and move to the start symbol of the grammar.

→ It will follow rightmost derivation in reverse order.

Let us consider an example where grammar is given and you need to construct a parse tree by using Bottom up parsing technique.

Given grammar

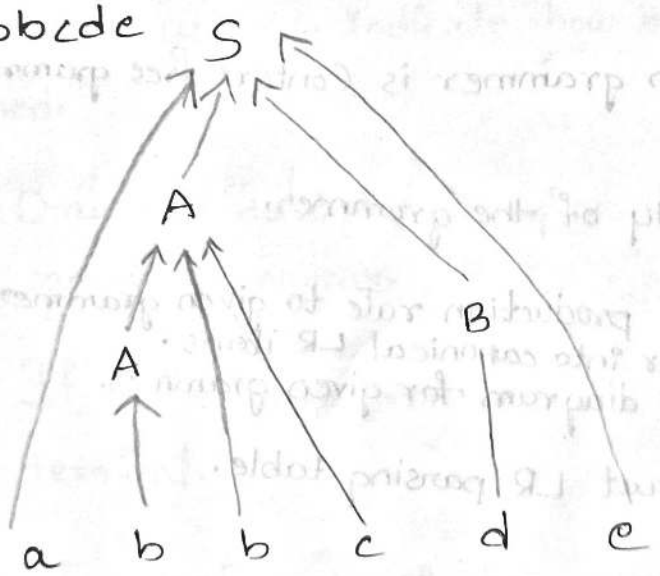
$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

Given Input string

abbcde



20/04/2022 Part-2

Introduction to LR Parsing

→ It is also known as shift Reduce bottom up parsing.

→ In LR(K) → L means left to right scanning the input.
R means construction of right most derivation in Reverse order
K means Look-a-head value.

→ LR parsing is classified into 4 types.

1) LR(0)
2) Simple LR (SLR) } LR(0)
3) Look-a-head LR (LALR)
4) Canonical LR (CLR) } LR(1)

Powerfull ↓

1) LR(0) parsing

→ Check the given grammer is Context free grammer (or) not.

→ Check ambiguity of the grammer.

→ Add augument production rule to given grammer.
→ Convert grammer into canonical LR items.
→ Draw Data flow diagram for given grammer.

→ from DFD construct LR parsing table.

Example,

Given grammer

$S \rightarrow AA$

$A \rightarrow aA|b$

ns:-

Step 01:- Add agument production rule to given grammer.

$S' \longrightarrow S \longrightarrow$ augument production rule

$S \longrightarrow AA$

$A \longrightarrow aA|b$

Step 02:- Convert grammar into canonical LR items.

$S' \longrightarrow \cdot S$

$S \longrightarrow \cdot AA$

$A \longrightarrow \cdot aA | \cdot b$

LHS RHS

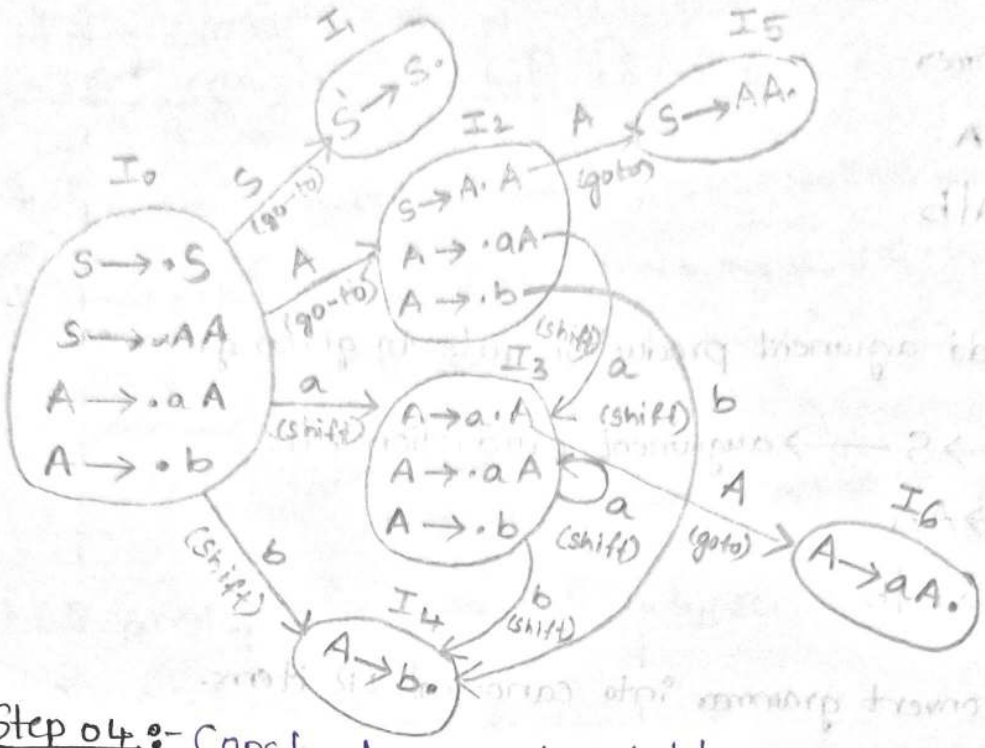
→ Here • is used to indicate how much of Input is scanned.

Step 03:- Draw the data flow diagram for the above canonical grammar.

Rule 1 :- If • is before non-terminal expand the non-terminal.

Rule 2 :- If • is before terminal no need to expand.

Rule 3:- If • arrives at end it indicates scanning is completed.



Step 04 :- Construct a parsing table.

Parsing Table :-

Rule 1 :- If a state is going to another state by appling a non-terminal then it is called as go-to.

Rule 2 :- If a state is going to another state by applying terminal then it is called as shift.

Rule 3 :- If a state is having final item write reduce node completed.

r=reduce

States	
I_0	
I_1	
I_2	
I_3	
I_4	
I_5	
I_6	

22/04/20

Simple

→ It is

→ It u

→ It

→ SLR tabl

→ In pro

r = reduced

States	Shift (Terminals)			goto (non-terminals)	
	a	b	$	S	A
I_0	I_3	I_4		1	2
I_1			accept		
I_2	I_3	I_4			5
I_3	I_3	I_4			6
I_4	r_3	r_3	r_3		
I_5	r_1	r_1	r_1		
I_6	r_2	r_2	r_2		

22/04/2022

Simple LR (SLR)

→ It is also known as Simple LR parser.

→ It works on Simplest class of grammar.

→ It contains few no. of states.

→ SLR parser technique is simple and fast to construct parsing table.

→ In SLR we place the reduce only in the follow of LHS production not in entire row.

Given grammar

$A \rightarrow (A) | a$

Step 01:- Add augument production rule to the given gramm

$A' \rightarrow A$ → augument production rule

$A \rightarrow (A)$

$A \rightarrow a$

Step 02:- Convert grammar into canonical LR items.

$A' \rightarrow \cdot A$

$A \rightarrow \cdot (A)$

$A \rightarrow \cdot a$

→ Here $\cdot$ is used to indicate how much of Input is scanned.

Step 03:- Draw the data-flow diagram for the above canonical grammar.

Rule 1 :- If $\cdot$ is before non-terminal expand the non-terminal.

Rule 2 :- If $\cdot$ is before terminal no need to expand.

Rule 3 :- If $\cdot$ arrives at end it indicates scanning is completed.

I_0

$A' \rightarrow \cdot$

$A \rightarrow \cdot$

$A \rightarrow \cdot$

Step 04

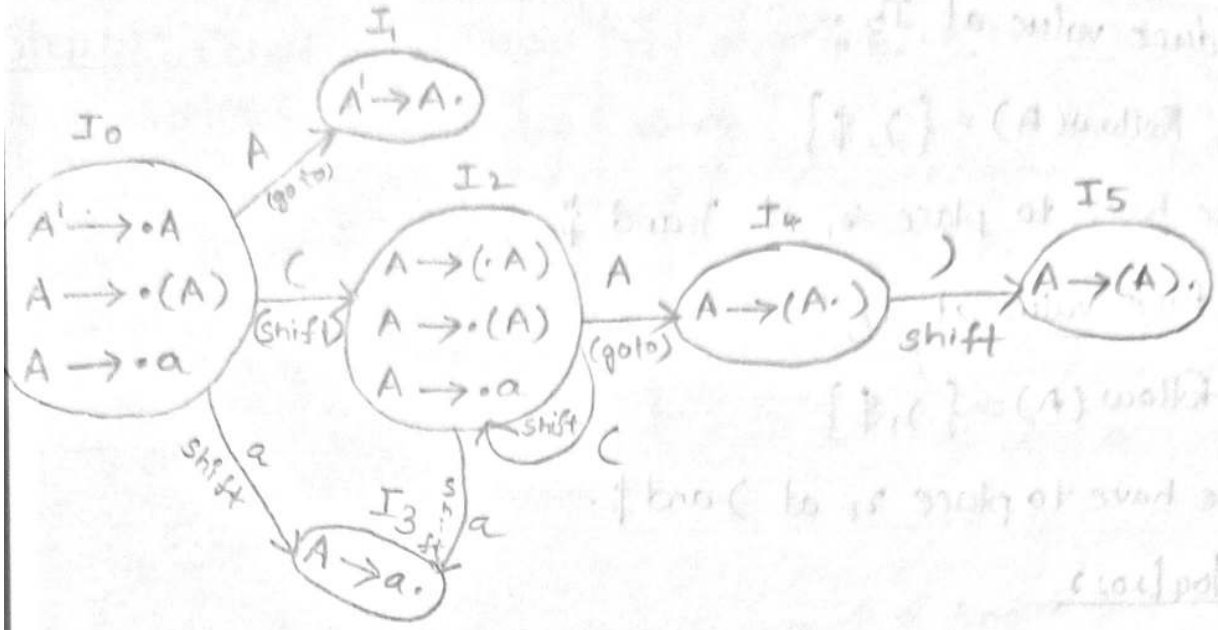
Parsing

Rule 1

Rule 2

Rule 3

State
I_0
I_1
I_2
I_3
I_4
I_5



Step 04 :- Construct a parsing table.

Parsing table

Rule 1 :- If a state is going to another state by applying a non-terminal then it is called as go-to.

Rule 2 :- If a state is going to another state by applying terminal then it is called as shift.

Rule 3 :- If a state is having final item write reduce note completed.

State	Shift (Action				goto
	a	(	)	$	A
I_0	I_3	I_2			1
I_1				accept	
I_2	I_3	I_2			4
I_3			r_2	r_2	
I_4			I_5		
I_5			r_1	r_1	

reduce value at I_3 :-

Follow(A) = {), $}

We have to place r_1 at) and $

reduce value at I_5 :-

follow (A) = {), $}

we have to place r_1 at) and $.

23/04/2022

Canonical LR (CLR) :-

→ CLR is a powerful parsing technique.

→ It is also known as LR(1) parsing technique.

→ LR(1) = LR(0) + Look-a-head value.

→ Here, we place reduce item at Look-a-head symbol in the parsing table.

Given grammar

$E \rightarrow BB$

$B \rightarrow cB \mid d$

Step 01 :- Add augument production rule to the given grammar.

$E' \rightarrow E$ → augument production rule.

$E \rightarrow BB$

$B \rightarrow cB$

$B \rightarrow d$

Step 02:- Convert grammar into canonical LR items along with Look-a-head values.

$E' \longrightarrow \cdot E, \$$ → Look-a-head value

$E \longrightarrow \cdot B\ B, \$$

$B \longrightarrow \cdot cB, c|d$

$B \longrightarrow \cdot d, c|d$

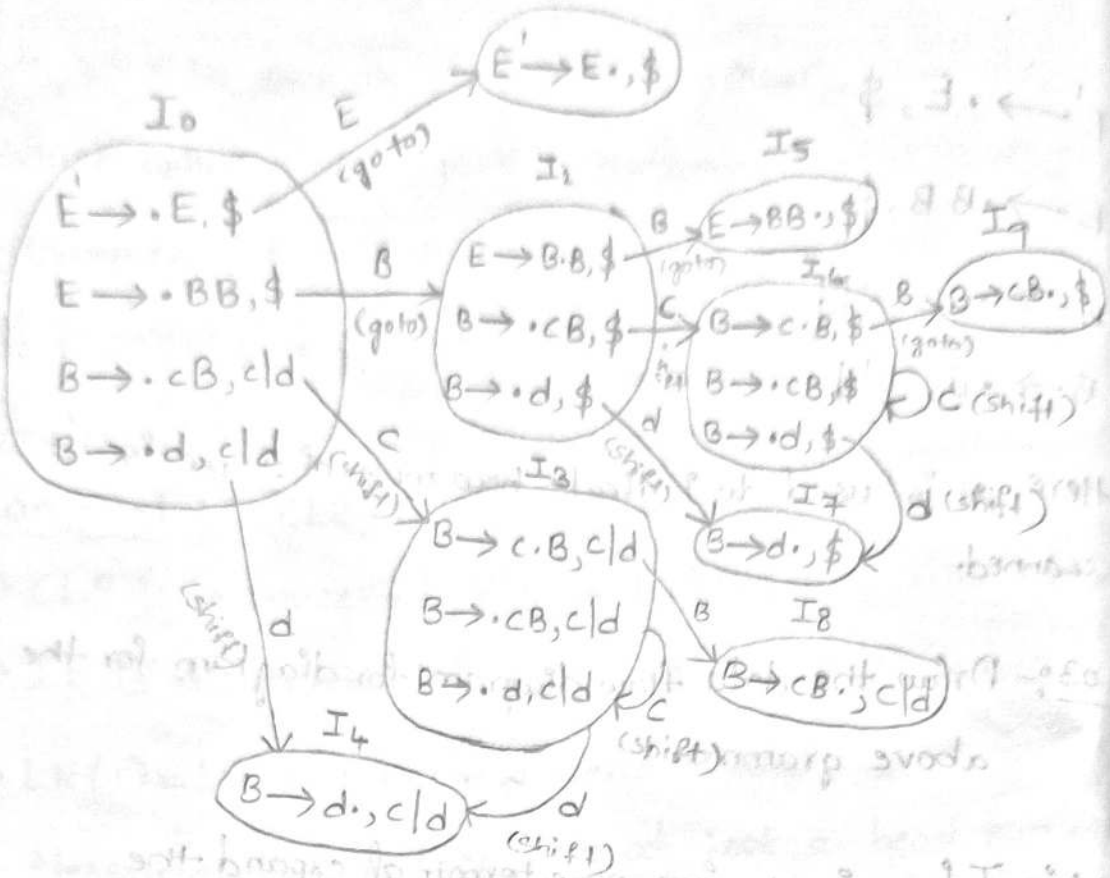
→ Here • is used to indicate how much of Input is scanned.

Step 03:- Draw the data flow diagram for diagram for the above grammar.

Rule 1:- If • is before non-terminal expand the non-terminal.

Rule 2:- If • is before terminal no need to expand.

Rule 3:- If • arrives at end it indicates scanning is completed.



Step 04:- Construct a parsing table:

Parsing table

Rule 1 :- If a state is going to another state by applying a non-terminal then it is called as go-to.

Rule 2 :- If a state is going to another state by applying terminal then it is called as shift.

Rule 3 :- If a state is having final item write reduce rule completed.

State	Shift			go-to	
	c	d	$	E	B
I_0	I_3	I_4		1	2
I_1			accept		
I_2	I_6	I_7			5
I_3	I_3	I_4			8
I_4	R_3	R_3			
I_5			R_1		
I_6	I_6	I_7			9
I_7			R_3		
I_8	R_2	R_2			
I_9			R_2		

26/04/2022

Look-a-head LR (LALR) parsing technique (LR(1))

LR(1) = LR(0) + Look-a-head value.

→ LALR is powerful parsing technique than other parsing techniques.

→ When LALR is compared with CLR, LALR contains few no.of states.

→ As a result, execution speed will be increased.

Given grammar

$E \rightarrow BB$

$B \rightarrow cB \mid d$

Step 01:- Add augumentation production rule to the given grammar.

$E' \rightarrow E$

$E \rightarrow BB$

$B \rightarrow cB$

$B \rightarrow d$

Step 02:- Convert grammar into canonical LR items along with look-a-head values.

$E' \rightarrow .E, \$$

$E \rightarrow .BB, \$$

$B \rightarrow .cB, c/d$

$B \rightarrow .d, c/d$

How to calculate look-a-head values

1) $E' \rightarrow .E, \$$ → look-a-head value

The above rule is augument production rule.

So, by default look-a-head value is $.

2) $E' \rightarrow .E, \$$

We get second rule as given below.

After . non-terminal E is present, so write production rule of E

$E \rightarrow .BB$

We get look-a-head value in second rule as follows.

After E, we have to find first (non-terminal) If there is no non-terminal, look-a-head value is $, so

$$E \rightarrow \cdot BB, \$$$

3) We get 3rd rule from 2nd production rule.

$$E \rightarrow \cdot BB, \$$$

After . non-terminal B is present, so write B production rule.

$$B \rightarrow \cdot cB$$

$$B \rightarrow \cdot d$$

Now we get look-a-head value as follows:

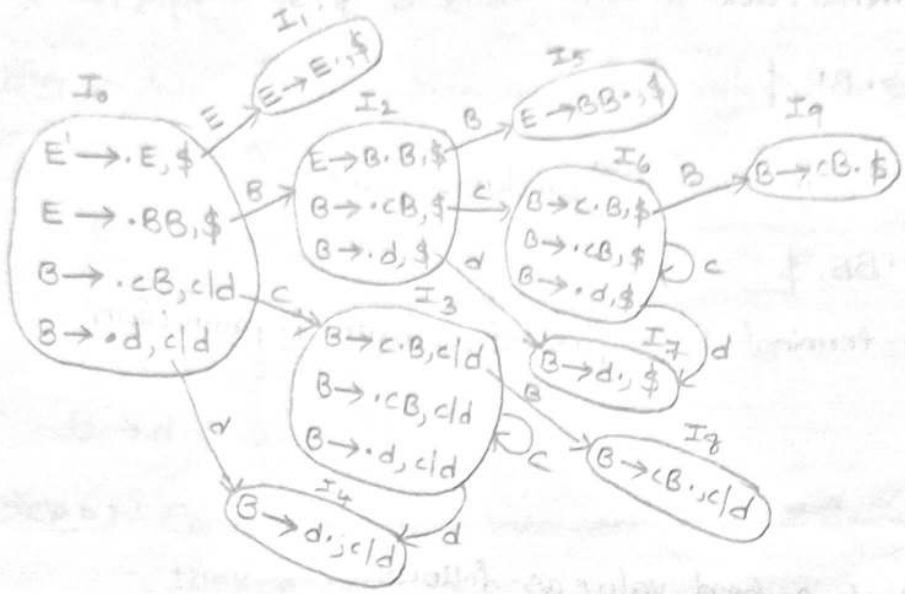
In second rule after B non-terminal is present, so to get look-a-head value we have to find first(B).

First (B) = c|d

So c|d is the look-a-head value in 3rd & 4th rule.

$$B \rightarrow \cdot cB, c|d$$

$$B \rightarrow \cdot d, c|d$$



29/04/2022

LALR parsing table before merging

States	Shift			goto	
	c	d	$	E	B
I_0	I_3	I_4		1	2
I_1			Accept		
I_2	I_6	I_7			5
~~I_3~~	I_3	I_4			8
~~I_4~~	R_3	R_3			
I_5			R_1		
~~I_6~~	I_6	I_7			9
~~I_7~~			R_3		
~~I_8~~	R_2	R_2			
~~I_9~~			R_2		

In DFD, the following s
different look-a-hea
They are:-

1) I_3 and I_6 =
2) I_4 and I_7 =
3) I_8 and I_9 =

Now, merge above
rules, different proc

LALR parsing tab

States	Sh	
	C	
I_0	I_3	I
I_1		
I_2	I_6	
I_{36}	I_{36}	
I_{47}	R_3	
I_5		
I_{89}	R_2	

Before
After merg
efficiency
easy.

In DFD, the following states have same rules having different look-a-head values.

They are:-

1) I_3 and I_6 = I_{36}

2) I_4 and I_7 = I_{47}

3) I_8 and I_9 = I_{89}

Now, merge above same states, having same production rules, different production rules.

LALR parsing table after merging

States	Shift			goto	
	C	d	$	E	B
I_0	I_3	I_4		1	2
I_1			Accept		
I_2	I_6	I_7			5
I_{36}	I_{36}	I_{47}			89
I_{47}	R_3	R_3	R_3		
I_5			R_1		
I_{89}	R_2	R_2	R_2		

Before merging there are 10 states in the parsing table. After merging there are only 7 states. So, by doing this, efficiency & Performance of the LALR algorithm will be easy.

30/04/2012

Ambiguous Grammars

A grammar is said to be ambiguous if there exists many

(i) more than one left most derivation

(ii) more than one right most derivation

(iii) more than one parse trees for the given strings

If a grammar has ambiguity then it is not good for compiler construction.

Ex:- Let us consider a grammar with production rules.

$E \rightarrow E+E$

$E \rightarrow E * E$

$E \rightarrow (E)$

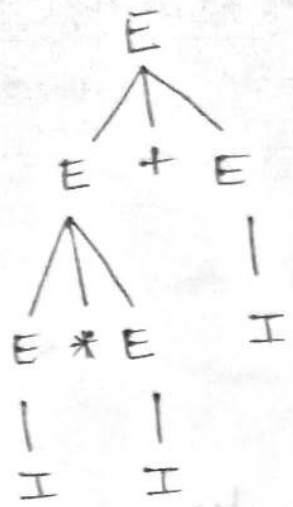
$E \rightarrow I$

$I \rightarrow E|0|1|2---9$

Given string

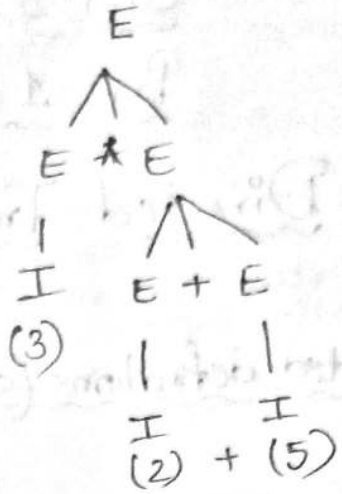
3 * 2+5

for the given string the above grammar generates 2 parse trees as given below.



(3) * (2)

3*2+5



3* 2+5

Since there are 2 parse trees for given string the above grammar is said to be ambiguity grammar. This grammar is not good for compiler constructions.

Unit-3

Part-1

Syntax-Directed Translation

Syntax-Directed definations (SDD):

SDD = Context free grammar + Semantic rules.

An SDD is a context free grammar together with semantic rules

- Production Rule:-

Production Rule	Semantic Rules
$E \rightarrow E+T$ } CFG	$E.value \rightarrow E.value + T.value$
$E \rightarrow T$	$E.value \rightarrow T.value$

Grammar Symbols — attributes

→ Attributes may be numbers, strings, reference, datatypes etc---

Types of attributes:-

There are 2 types of attributes. They are

1) Synthesized attributes

2) Inherited attributes.

1) Synthesized

Parent node chi

A → BC

If a parent node is known

Ex:-

A.valu

A.value

A.value

In the above attribute. Bec node B.value

Inherited at

Parent node chil

A → BC

If a child no siblings nodes attributes

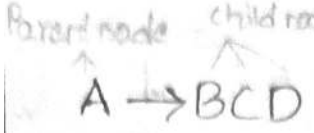
Ex:- B.value =

B.value =

B.value =

In the above attribute. Bec A.value with si

Synthesized attributes:-



If a parent node gets value from child node then that node is known as Synthesized attributes.

Ex:-

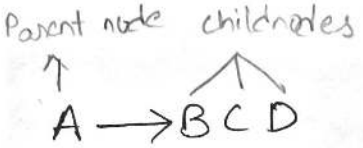
A.value = B.value

A.value = C.value

A.value = D.value

In the above example, A.value is known as Synthesized attribute. Because, A.value is obtained from child node B.value, C.value, D.value.

Inherited attribute:-



If a child node gets values from parent nodes (or) it's siblings nodes. then that node is known as Inherited attributes

Ex:- B.value = A.value

B.value = C.value

B.value = D.value

In the above example, B.value is known as Inherited attribute. Because, B.value is obtained from it parent node A.value & it's sibling nodes C.value & D.value.

Evaluation Orders for SDD's

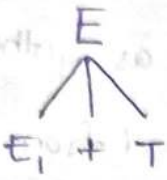
Dependency graph determine the evaluation order of attributes in a parse tree.

Dependency graph:

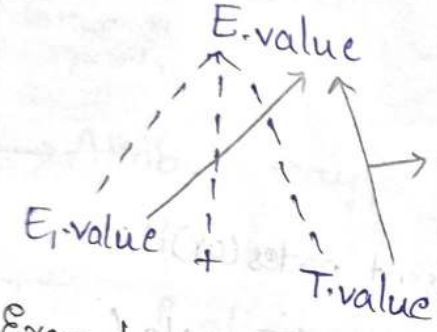
Consider an example

Production Rule	Semantic Rule
$E \rightarrow E_1 + T$	$E.value \rightarrow E_1.value + T.value$

Parse tree for production rule



Annotated parse tree for semantic Rule



Example:

Given grammar

$T \rightarrow FT'$

$T' \rightarrow * FT_1'$

$T_1' \rightarrow \epsilon$

$F \rightarrow digit$

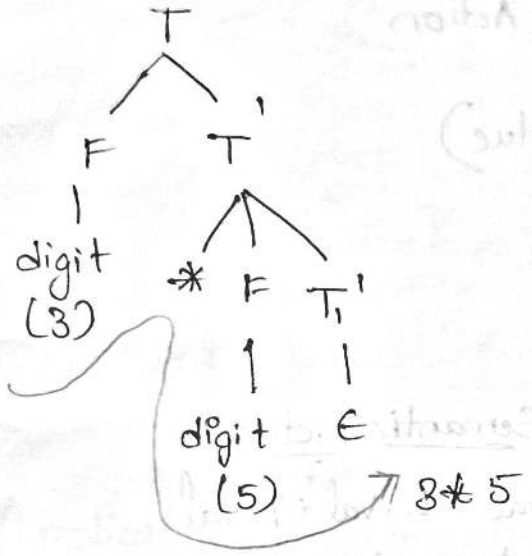
ven Input string: 3*5.

ow find out the Evaluation order of given Input string.

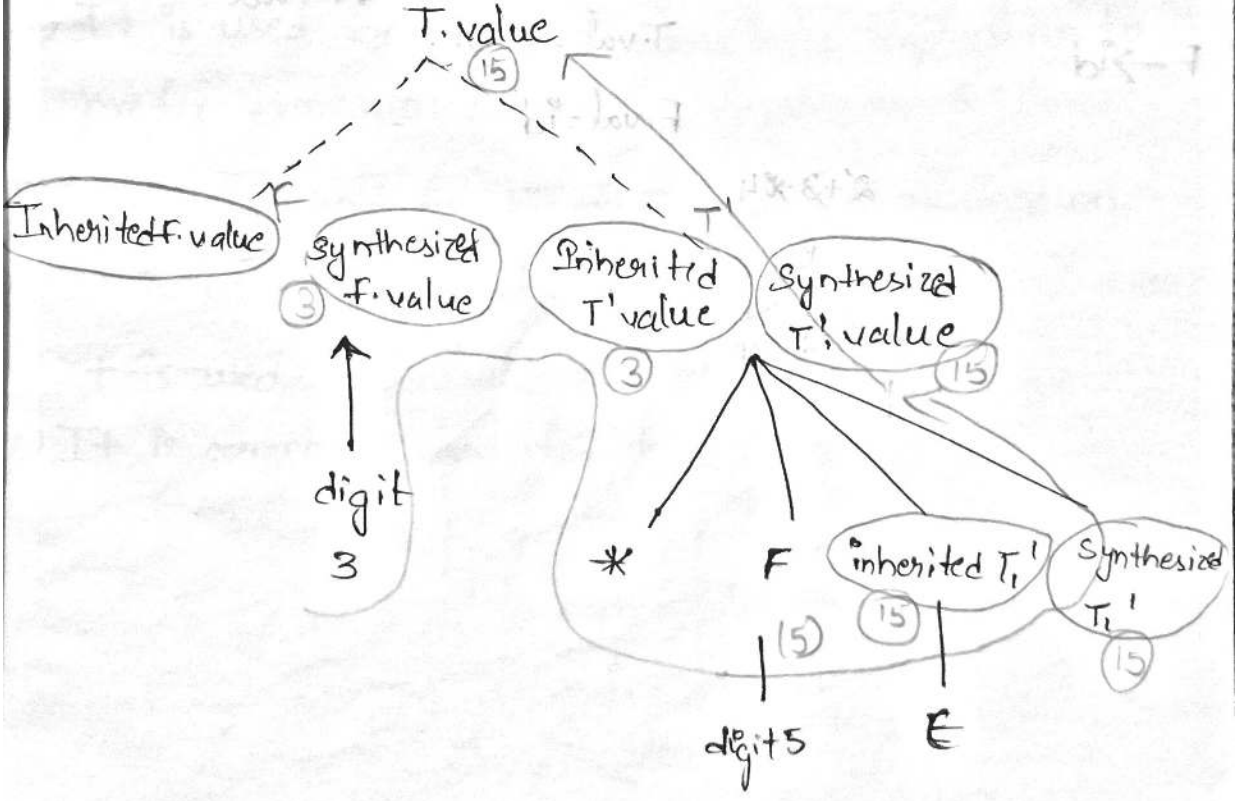
or given grammar.

Parse tree for given Input string:-

)*5



notated parse tree for given Input string:-



redu 22/06/2022

SDT (Syntax Directed Translation)

We SDT is used for semantic analysis. SDT is basically
redi used to construct the parse tree with grammar and
fr semantic action. So, we can say that

we SDT = Grammar + Semantic Action

23/0 Ex:- (E = E+T + E.value + T.value)

Ca

→ Ex:- Input → 2+3*4

→ output → 14

→ Grammar

→ E → E+T | T

T → T * F | F

Gli F → id

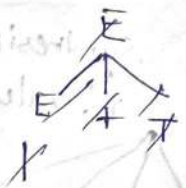
E

B

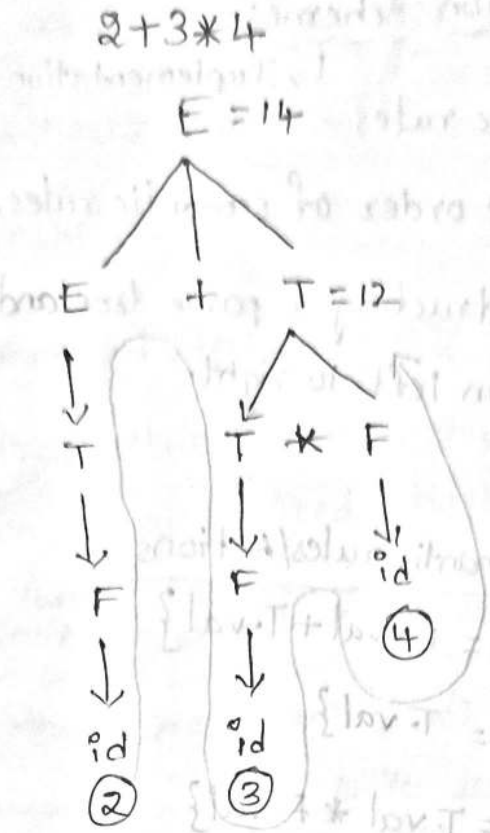
Ste

Semantic Action
E.val = E.val + T.val
E.val = T.val
T.val = T.val * F.val
T.val = F.val
F.val = id

2+3*4



Parse tree



Applications

1) SDT is used for executing arithmetic expressions.
2) It is used for conversion from infix expression to prefix expression or postfix expression.
3) It is also used for binary to decimal conversion.
4) It is used for creating a syntax tree.
5) It is used to generate intermediate code.
6) It is commonly used for type checking.

07/06/2022

Syntax directed translation schemes

↳ Implementation

SDT = Grammar + Semantic rules.

It is used to evaluate the order of semantic rules.

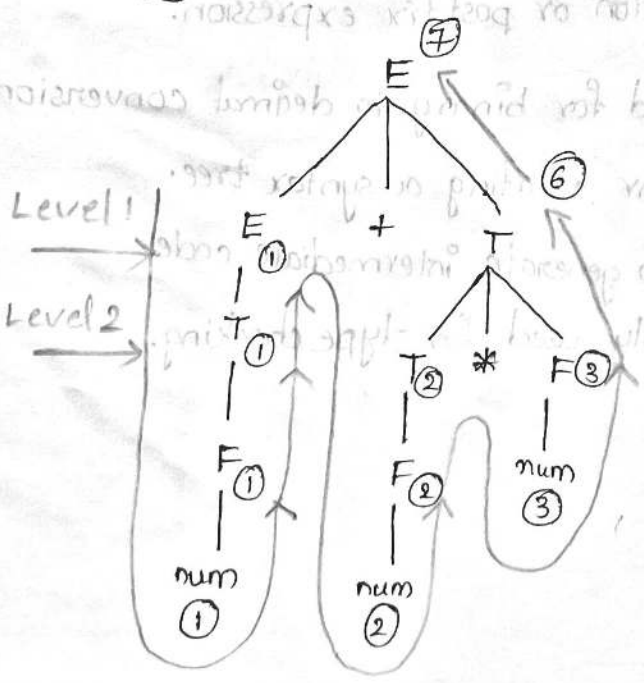
SDT is implemented by constructing a parse tree and performing the actions from left to right.

Ex:-

Grammar	Semantic rules/Actions
$E \to E+T$	$\{E.val = E.val + T.val\}$
$E \to T$	$\{E.val = T.val\}$
$T \to T*F$	$\{T.val = T.val * F.val\}$
$T \to F$	$\{T.val = F.val\}$
$F \to num$	$\{F.val = num.lexvalue\}$

Ex:- Given string

1+2*3



Implementing S-attribute and L-attributes

There are 2 types of attributes. They are:

1) Synthesized attributed

2) Inherited attributes.

1) Synthesized attribute:- If parent attribute depends on children attributes then that parent attribute is known as synthesized attribute.

Ex:- Parent children $A \longrightarrow BCD \Rightarrow A.val = B.val$, $A.val = C.val$, $A.val = D.val$

In the above example, A is known as synthesized attributes, because A is depending on B (or) C (or) D with values.

This parent attribute is also known as S-attribute.

2) Inherited attribute :-

If an attribute depends on parent attribute or sibling attributes then that attribute is known as inherited attribute.

Ex:- $A \longrightarrow BCD$

Child $\longrightarrow$ parent or sibling

$B.val = A.val$

$C.val = B.val$

$D.val = B.val$

L-attribute

If an attribute depends on left side synthesized (or) Inherited attributes then it is known as L-attribute.

Ex:- Grammer

$A \rightarrow BCD$

1) C.valu = D.val
2) D.val = B.val
3) C.val = A.val
4) A.val = B.val

- In the above example 2 and 3 statements belongs to L-attributes, remaing statements will not depend on L-attributes.

3/06/2022

Part-2

Intermediate Code Generator

Variants of Syntax Trees

There are 2 types of Syntax trees. They are:-

1) Directed Acyclic graphs (DAG) method.
2) The value-Number Method for constructing DAG.

1) Directed Acyclic graph (DAG):-

A Directed Acyclic graph identifies the common sub expression in a given expression.

A DAG tree contains operands in the leaves and operators in the interior nodes of the tree.

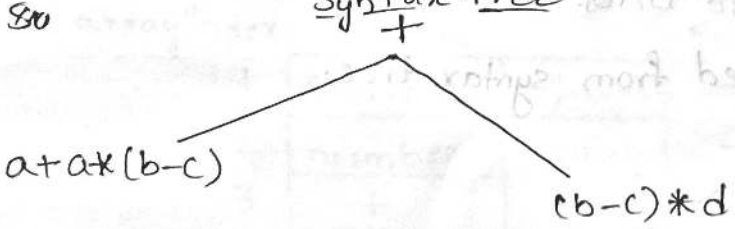
Here a node can have more than one parent.

Ex:-

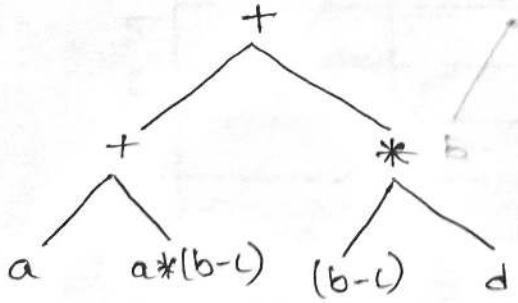
a+a*(b-c)+(b-c)*d

1) Soo

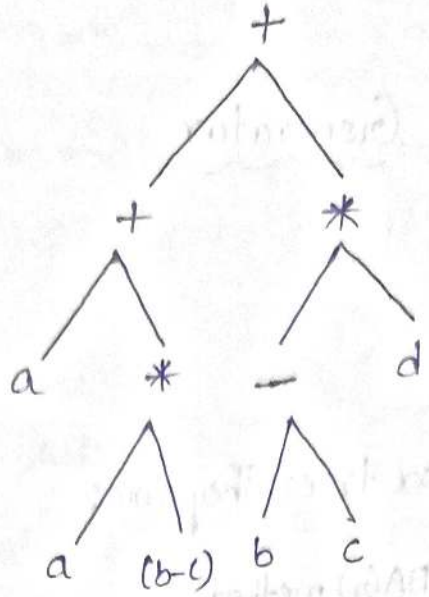
Syntax tree



2)



red 3)



Gl

rec

-

we

23|

Cc 4)

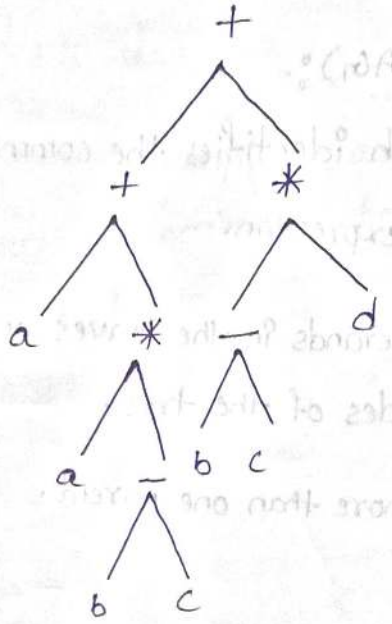
"

→

→

→

→

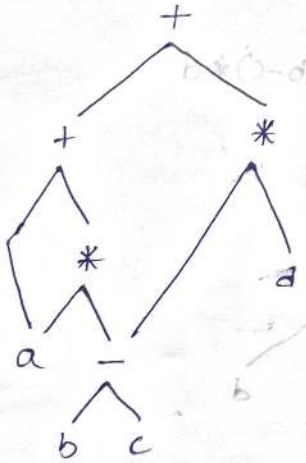


Gl

E

B Let us convert into DAG.

St DAG :- It is derived from syntax tree.



here node can have more than one parent.

The value number method for constructing DAG :-

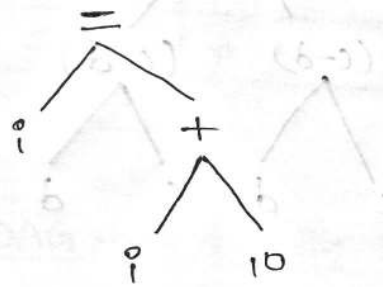
In value number method the nodes of syntax tree (or) DAG tree are stored in an array of record.

Here we may use hash table to store the data of syntax tree.

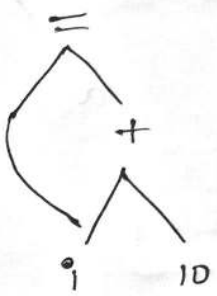
Here array index is known as node value number.

Ex :- (i) L = L + 10

Syntax tree



DAG Tree



Nodes of DAG for i = i + 10 is allocated in an array.

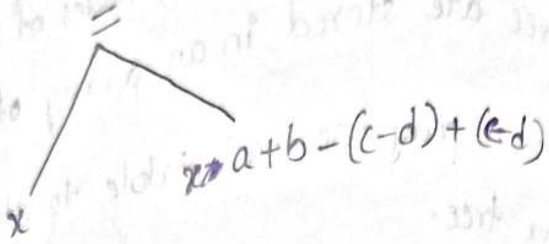
array index	Meaning	
1	identify	i
2	number	10
3	+	1 : 2
4	=	1 : 3
5		
6		
7		
8		

(ii)

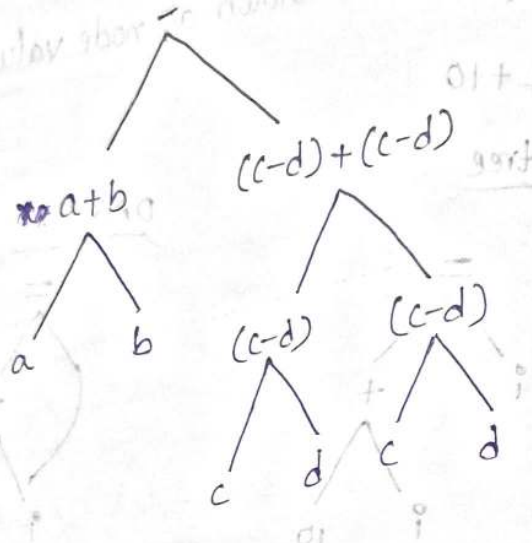
Ex:- $x = a + b - (c-d) + (c-d)$

Syntax tree

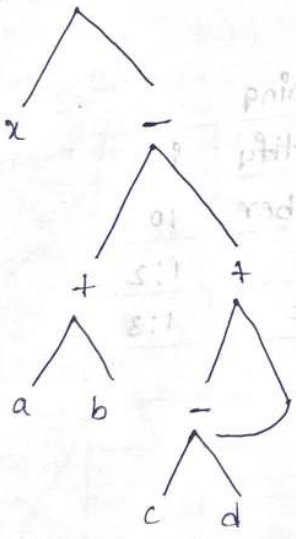
(i)



(ii)



DAG



Nodes of DAG for $x = a+b-((c-d)+(c-d))$ is allocated in an array.

arrayindex	Meaning	
1	identify	c
2	identify	d
3	−	1:2
4	identify	a
5	identify	b
6	+	3:3
7	+	4:5
8	−	7:6
9	identify	x
10	=	9:8

9/06/2022

Generate the DAG tree for the following expression using stack:-

Step 01 :- Convert the given expression into postfix expression.

Step 02 :- Generate syntax tree using stack.

Step 03 :- Convert syntax tree into DAG.

Ex:- $(a+b)*(c-d)+((e/f)*(a+b))$

Step 01 :-

$((ab+)*(cd-))+((ef/)*(ab+))$

$\Rightarrow (ab+cd-*)+(ef/)*(ab+))$

$\Rightarrow (ab+cd-*ef/ab+*+$ (Postfix expression)

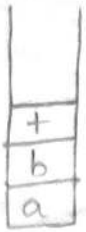
Step 02 :- Draw syntax using stack.

ab+cd-*ef/ab+**+

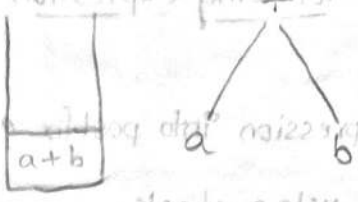
1) Push 'a' into stack.

2) Push 'b' into stack.

3) Push '+' into stack



Perform addition operation and draw the syntax tree.

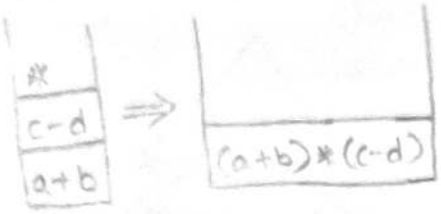


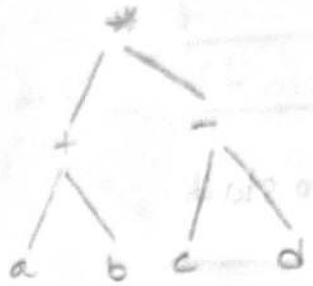
4) Push 'c' into stack.

5) Push 'd' into stack.

ish '-' into stack.

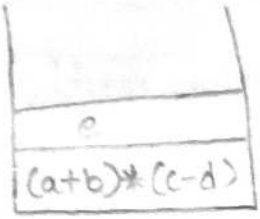
Push * into stack.



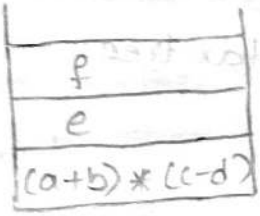


nform multiplication operation and draw syntax tree.

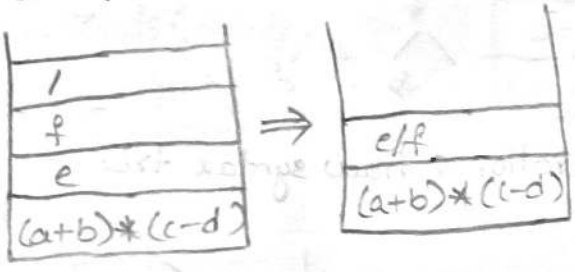
Push 'e' into stack.

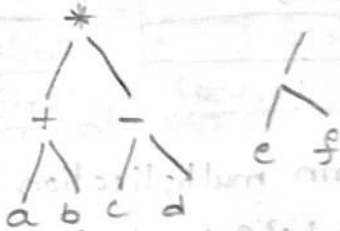


)Push 'f' into stack.



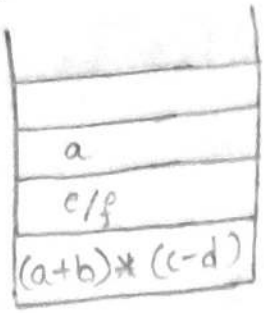
)Push '/' into stack.



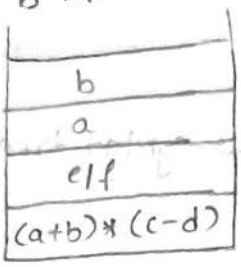


Perform division operation & draw syntax tree.

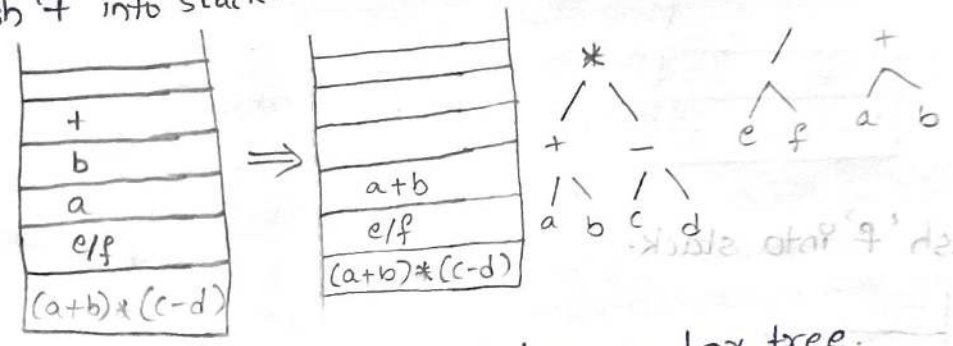
11) Push 'a' into stack.



12) Push 'b' into stack.

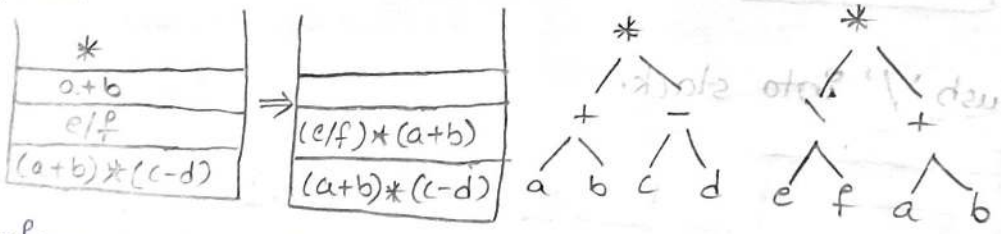


13) Push '+' into stack.



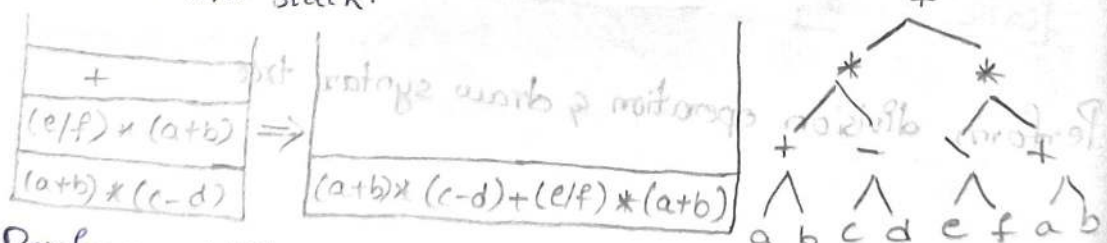
Perform addition operation and draw syntax tree.

14) Push '*' into stack.



Perform multiplication operation & draw syntax tree.

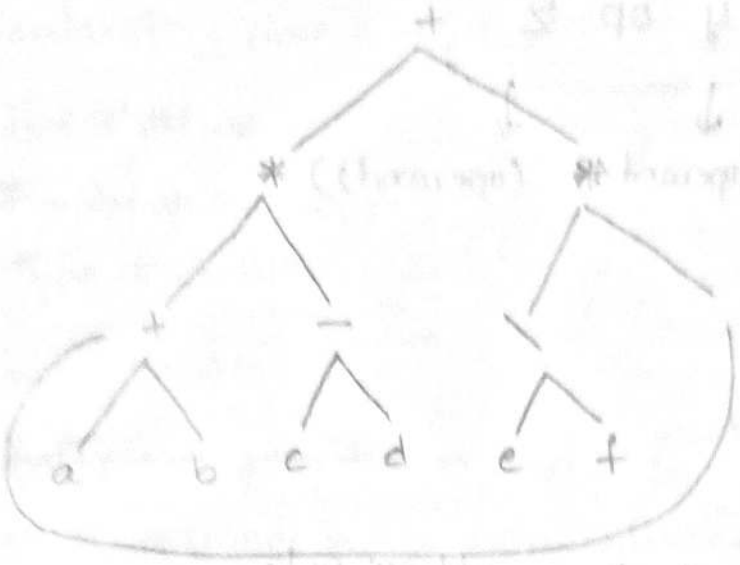
15) Push '+' into stack.



Perform addition operation and draw syntax tree.

tep 03 :-

To draw DAG tree. We have to reduce repeated sub expression.



10/06/2022

Three address code

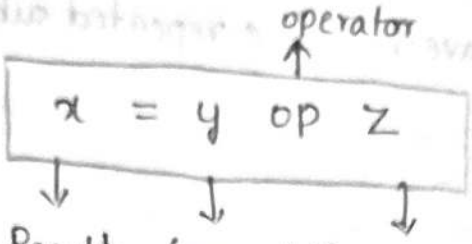
Three address code is a intermediate code. It is used by optimizing compilers.

In three address code the given expression is broken down into several separate instructions. Now these instructions can be easily translated into machine code.

Each three address code instructions are maximum 3 operands.

It is combination of assignment operator and arithmetic operator.

General form of three address code:-



Example:-

1) Given Expression

$a = b + c + d$

Convert the above to three address code.

$temp\ 1 = b + c$

$temp\ 2 = temp\ 1 + d$

$a = temp\ 2$

∴ This is in the form of three address code.

2) $a = (-c * b) + (-c * d)$

Convert into 3 address code.

$temp\ 1 = -c$

$temp\ 2 = temp\ 1 * b$

$temp\ 3 = temp\ 1 * d$

$a = temp\ 2 + temp\ 3$

Types and declarations

While evaluating the expression the need of types and declarations place a major role.

1) Type checking
2) Translation application
3) Type expression

1) Type checking :-

Before evaluating the expression type checker checks the type of operands in the expression whether to find out it is possible to execute the expression or not.

Ex:- a * b execution is possible
int int

a * b execution is not possible.
int char

2) Translation application :-

From the type of the variable it allocates storage space to the operands.

Ex:- int a; → It allocates 16 bits of storage space.

char a; → " " 8 bits " " "

11/06/2022

3) Type Expression:- Type expressions are formed from basic types like int, float, double, char.

Generally, type checking depends on particular language we are using.

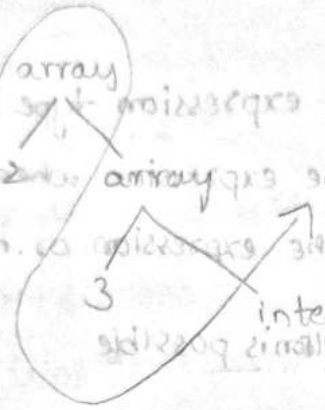
Ex:- int[2][3];

The above declaration can be read as "array of 2 array of 3 Integers each".

The above declaration is written as

array(2, array(3, integers)) in intermediate code.

The type expression of int [2][3]; in tree structure is shown below:



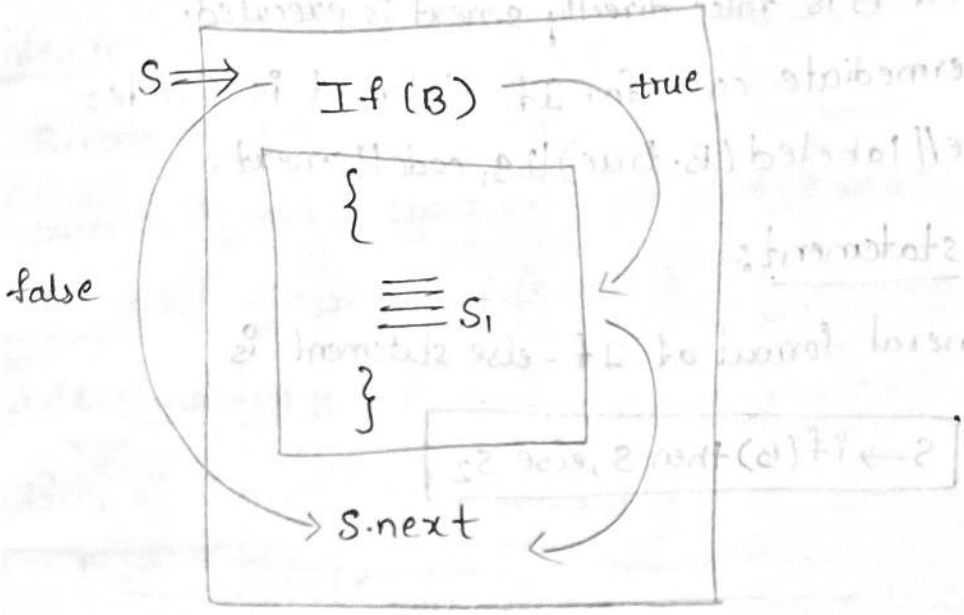
Control Flow:

Generally we use if statement or if-else statement or while statement or switch case-statement to jump the curser from one location to another location in the program. Therefore, these statements are also known as Control Flow statements.

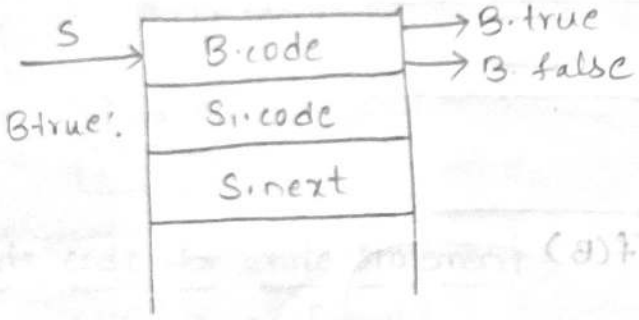
If statement:-

$S \longrightarrow if\ (B)\ S_1$

General format of If statement



Representation of execution of If statement :-



→ Here B.code is also known as condition.

→ If B.code is true S_1.code is executed and then s.next is executed.

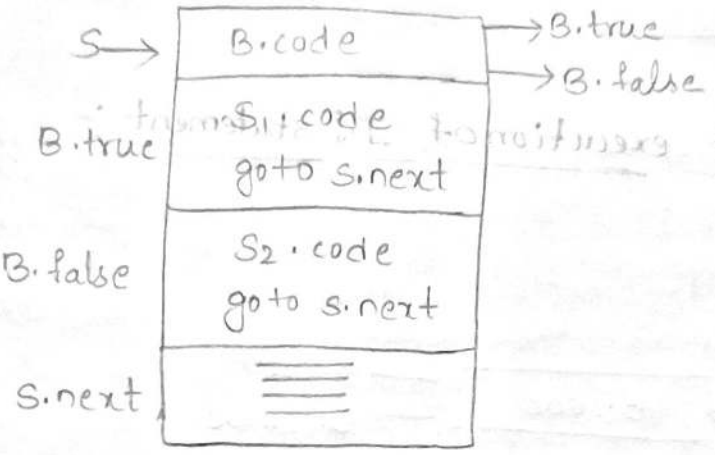
→ Suppose if B is false directly s.next is executed.

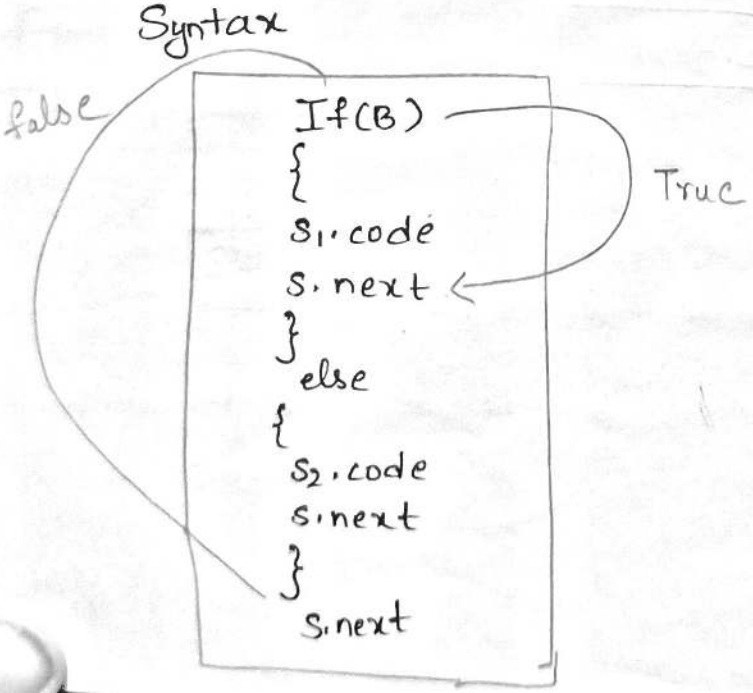
→ The intermediate code for If statement is s.code = B.code || labeled (B.true) || S_1 code || s.next.

If-else statement:-

The general format of If-else statement is

$$S \rightarrow \text{if}(B) \text{ then } S_1 \text{ else } S_2$$





Here if B.code is true S_1.code is executed and then S.next is code executed.

Suppose if B.code is false S_2.code is executed then S.next is executed.

Intermediate code for If-else statement:-

S.code = B.code || label (B.true) ||

S_1.code || generate (goto S.next) || label (B.false)

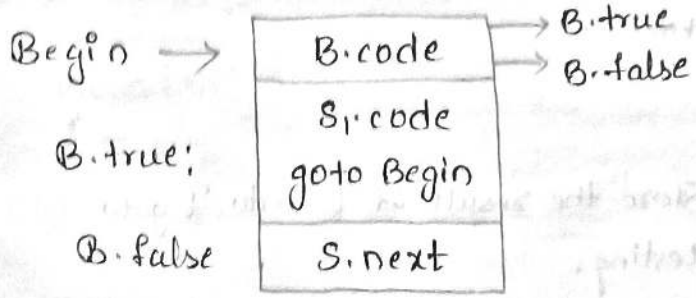
S_2.code || generate goto(S.next)

While statement :-

General format (Production rule)

$S \rightarrow$ while (B) then S_1

If (B.condition) is true S_1.code is executed this process is continued until B.code is false.



Intermediate code for while statement :-

S.code = B.code || label (B.True) || S_1.code || generate (goto.Begin) || label (B.false) || S.next .

16/06/2022

Switch Case statement:- Generally, we use switch where their are multiple choices & we have to sel among them in that situation we use switch cas statement.

Syntax of Switch Case statement:-

```
Switch(E)
{
 case V1 : S1 ;
        break;
 case V2 : S2 ;
        break;
 .
 .
 .
case n=1 : Sn ;
        break;
default : Sn
        break.
}
```

Code to Evaluate E:- Store the result of 'E' into 't' goto testing.

Testing:-

```
If t=V1
   goto L1
   If t=V2
      goto L2
if t=Vn-1
      goto Ln-1
      goto Ln
next: break;
L1: code for S1;
      goto next.
L2 ; code for S2,
goto next
```

```
Ln-1 : code for Sn-1
       goto next
Ln : code for Sn
     goto next.
```

06/2022

Unit-4

Part-1

Run-Time Environments

Generally, memory is allocated in 2 ways.

They are:-

1) Static memory allocation
2) Dynamic memory allocation.

1) Static memory allocation:-

Here memory is allocated only once to the variables in static area.

Memory is deallocated after program is complete.

Drawbacks:-

1) int a[5];

Here, fixed memory cells are allocated. In the above example we can store only 5 values.

Here we can't store more than 5 values.

If we store less than 5 values in the array memory is wasted.

2) Dynamic memory allocation:-

In Dynamic memory allocation memory is allocated the variables during the run time, that means, memory is alloted only if we give values to the program.

Advantages:-

1) Memory wastage will not be there.

Ex:- node = (struct node *) malloc (size of (struct node))

malloc and calloc functions are used for dynamic memory allocation.

17/06/2022

Stack Allocation of Space

In stack LIFO concept is used to store data. A the variables in main function and sub function are as active records.

Generally, code of main function is stored in the section 1 of the stack. Tly, code of sub program/fun is stored in the section 2 of the stack.

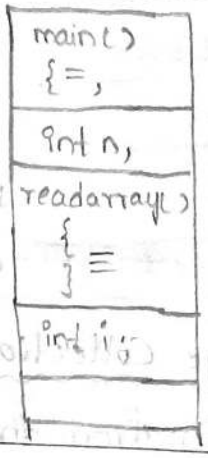
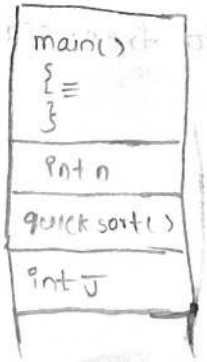
When the execution of main function starts the operation pushes all the active records of main fu into the section 3 of the stack. Tly, when the exe of subprogram starts the push operation pushes a active records of subfunction into the section 4 stack.

rally, when main program execution completed, pop operation eletes all the active records of section 3 of the stack.

Tly, when the execution of subprogram is completed pop operation deletes all the active records of section 4.

In this way compiler executes a C program sing a stack.

Ex:-

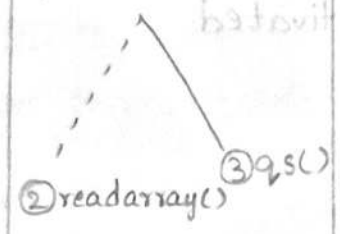
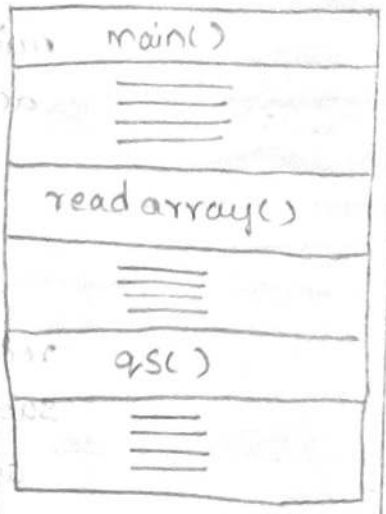
Quick sort Activation tree	Activation Records in stack	Remarks
main() { int n; ≡ }		main() is activated.
main() / readarray()		read array subprogram is activated.
main() pop / Push readarray() quicksort()		readarray() is deactivated and quicksort is activated.

22/06/2022

Heap allocation

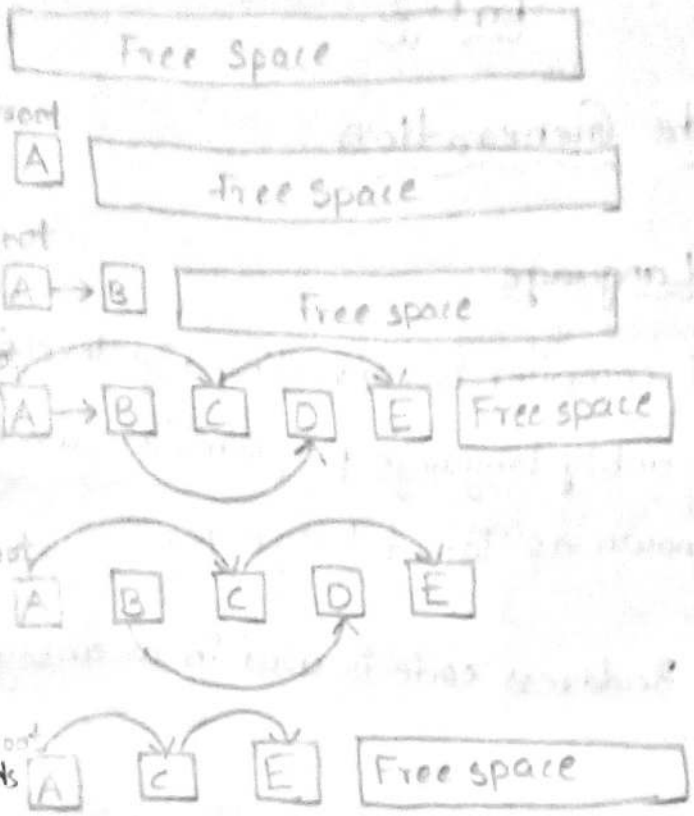
The drawback of stack allocation is it removes the activation records from the memory of its usage is completed. So, retain of activation records is not possible in stack allocation.

Retain of activation records is possible in heap allocation because activation records are not deleted from the memory even though there usage is completed in heap allocation.

Position in activation tree	Activation records in heap	Remarks
		Retain of activation records readarray() is possible because after readarray() procedure ends, it is still in the heap, it is not deleted.

Introduction to Garbage Collection

Removing unused data from the memory and adding that memory to free space is known as garbage collection.



link from A to B is remained, So B and D records all deleted as there is no connection with root.

Introduction to trace-based collection

Garbage collector runs continuously to detect unused objects and make memory free whereas trace-based collector runs whenever free space is exhausted or it's amount drops below some threshold value, and time only trace-based collector executes automatically and detects unused objects and makes memory free

By using Garbage collectors system performance will be reduced whereas system performace increases by using trace-based collector.

01/07/2022

Part-2

Code Generation

The Target Language

The aim of compiler is converting high level language program to assembly language program. This assembly language is known as Target Language.

Generally, 3address code is used in assembly language.

The 3address code contains load, store, computation, and condition, un-conditional statement.

1) Load:-

Load is used to move value from variables to registers

General format of load is

Ld dst, address

Ex:-

Ld r, x

Here, the value of variable x is loaded in register r.

2) Store:-

This instruction moves values from registers to variables.

Ex:- ST x, r

The value of register r is stored in variable x.

3) Computation statement:-

Computation statements are used to perform addition, subtraction, multiple, division etc.,

General format is

op code destination, src 1, src 2

Here op code is known as operation code. It may be add (or) sub (or) mul etc.,

In the above format, the values of src1 & scr2 are computated & result is stored in destination.

Ex:- Add z, x, y

Add r_1, r_2, r_3

In the above instruction, r_2 data & r_3 data are added & result is stored in r_1.

4) Un-conditional Jump:-

The general format of unconditional statement is

BR L

Br - Branch

L - Label

In the above instruction, control jumps from current position to the label placed in the program.

5) Conditional Jump:-

The general format is

BR (condition) L

Here, control jumps from one location to another location based in condition checking.

02/07/2022

Addresses in the Target Code

1) Static Allocation

2) Dynamic allocation.

1) Stack allocation:-

In static, fixed memory is allocated to the variable. Here, memory cannot be changed.

Ex:- int a[5];

2) Dynamic allocation :- Here memory is allocated at runtime of the program

Ex:- stack, heap, linked list etc.

Stack:-

In stack, activation records are inserted into stack using push operation.

After execution is completed, activation records are deleted from stack using pop operation.

In stack, activation record data is not available after execution of the program, because activation records are deleted from stack after execution.

Heap:-

In heap, activation records data is available forever. because activation records are not deleted even after execution of the program.

Basic blocks

The basic block is a sequence of consecutive statements which are always executed in sequence of without halt or possibility of branching.

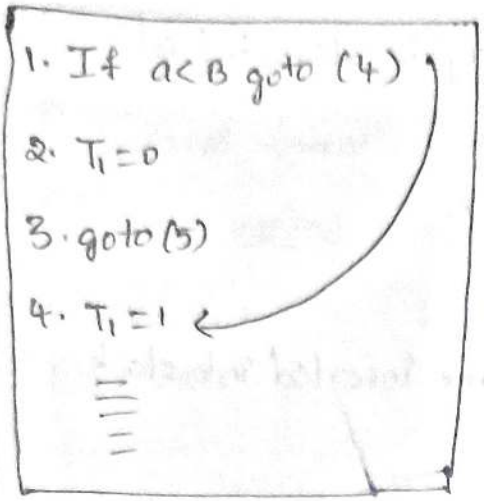
The basic blocks does not contain jump statements.

Ex:-

Three address code

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

Basic block



non-Basic block

Rules for partitioning program into blocks

When an intermediate code is generated we can use the following rules to partition the program into basic blocks.

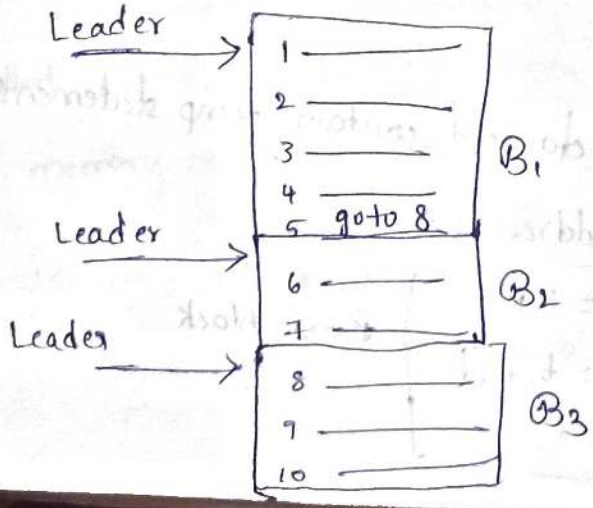
Rules :-

1) The first state is a leader.

2) Any target statement of conditional or unconditional statement is a leader.

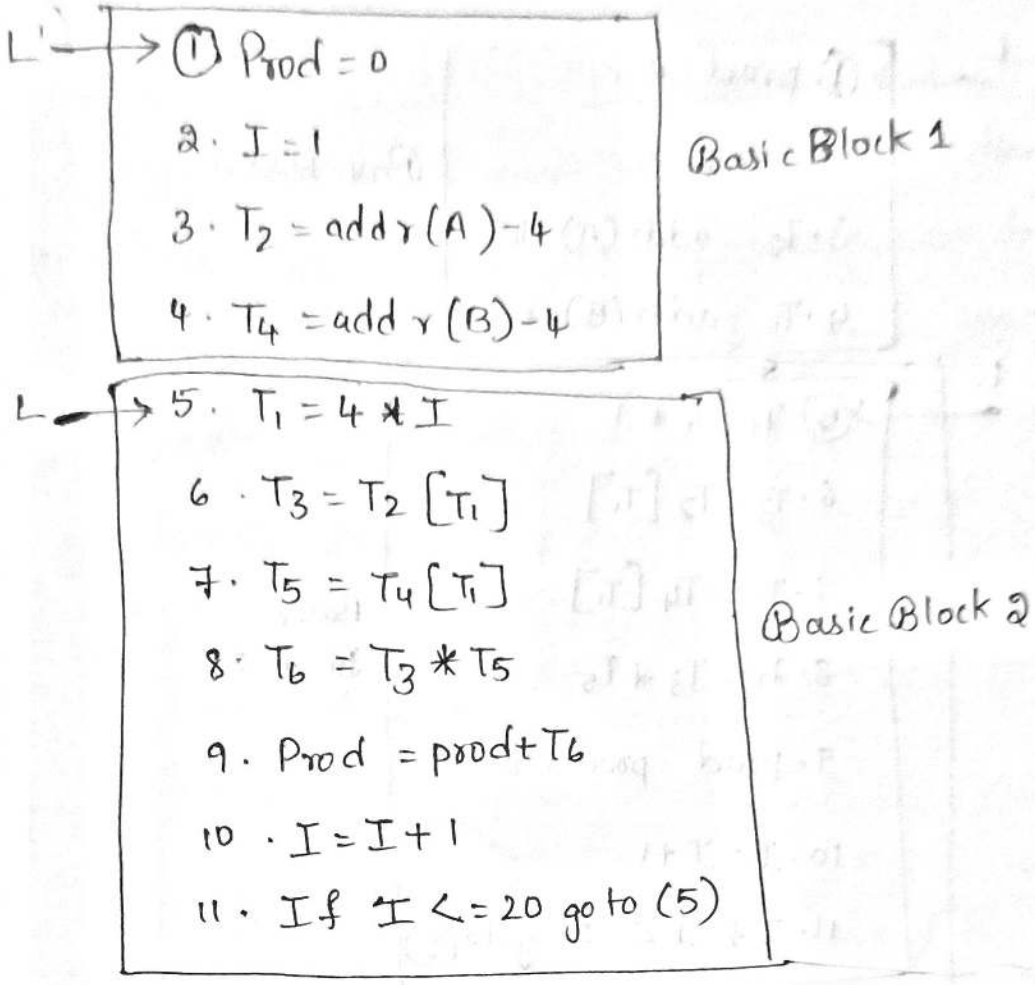
3) Any statement that immediately follows goto statement is a leader.

4) The basic block starts with leader statement & ends with the line before the line of next leader statement



Problem

Convert the three address code into Basic blocks



Flow Graphs

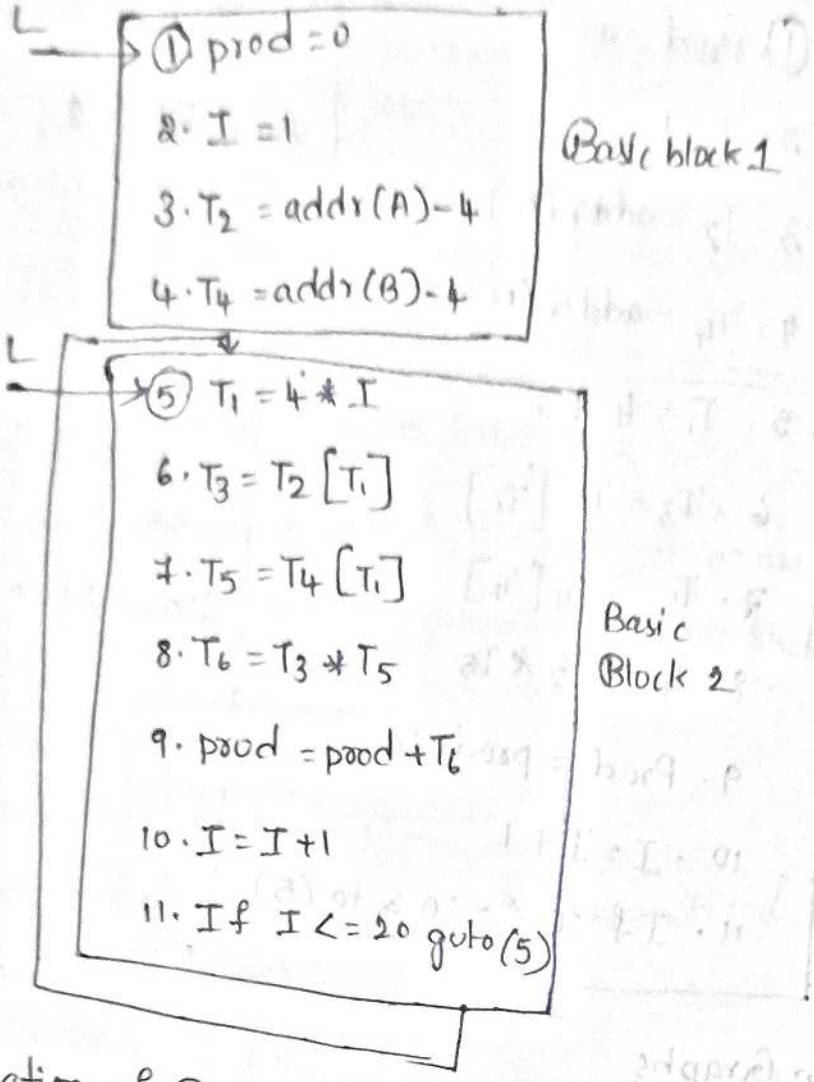
A flowgraph is a directed graph in which the flow of control information is added to the Basic blocks.

Rules for flow graph :-

1) The directed edges are placed b/w basic blocks.

2) If Basic block 2 is to be executed immediately after basic block 1, there should be a directed edge from basic block 1 to basic block 2.

Let us convert the above 3-address code into flow graph.



Optimization of Basic Blocks

Optimization techniques increases the performance of the code by saving memory and execution time.

Optimization code techniques are :-

1. **Dead code elimination** :- Dead code means the code which never executes is known as dead code.

Ex:-
```
main()
{
    x=10;   → Dead code remove it →
    y=20;
    z=30;
```
```
main()
{
    y=20;
    z=20;
    p=2*y+z
}
```
} optimized code

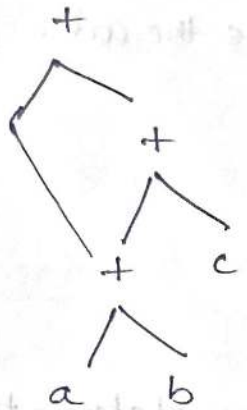
```
p = 2* y+z
}
```

2) Common Suberxpression Elimination:-

By removing common suberxpression execution speed will be increased memory space is saved.

By using DAG technique we can remove common suberxpression.

Ex:- x=(a+b)+(a+b)+c



05/07/2022

3) Constant folding:-

If we allaborate constant expressions memory space will increase & execution time/space increases.

So, by constant folding we can optimize the code.

Ex:- x=2*3+y → original code

x=6+y → optimized code.

4) Copy propagation:-

Copy propagation of two types. They are:-

i) Variable propagation

ii) Constant Propagation.

i) Variable propagation & Constant Propagation

If variable (or) constant propagates through a no. of s execution time will be increase and also memory spa will also increase.

So, we should try to eliminate variable propagation constant propagation to optimize the code.

Ex:- $x = y$

$z = y + 3$

$z = x + 3$

5) Strength Reduction :- (Expensive statement).

Cole should try to replace expensive statement with cheeper statement to optimize the code.

Ex:- Expensive statement → Cheaper statement

$x = z * y$ → $x = y + y$

10/7/2022

A Simple Code Generator

Code generator is used to produce the target code for three address statements. It uses register to store address statements. Here target code is also called as Assembly language Code.

Statement	Code generated	Register description

t = a - b	Mov R0, a
	Sub R0, b
	STR t, R0
		R0 contain t
		R0 value store in t

u = a - c	Mov R1, a
	Sub R1, c
	STR u, R1
		R1 contain u
		R1 value store in u

Peephole Optimization

It is a type of code optimization.

Objectives:-

1) To improve performance.
2) To reduce memory size.
3) To reduce code size.

Techniques:-

In this technique

1) Redundant Load to store elimination.
2) Constant folding.
3) Strength Reduction.

1) Redundant load and store elimination

In this technique redundancy is eliminated.

Initial code	Optimized Code
$y = x + 5$	$y = x + 5;$
$i = y$	$w = y * 3$
$z = i$	
$w = z * 3$	

2) Constant folding:-

Here the code can be simplified by folding the constant values in the expression.

Initial code	Optimized code
$x = 2 * 3;$	$x = 6;$

3) Strength Reduction:-

The operators that consume higher execution time are replaced by the operators consuming less execution time.

Initial code	Optimized Code
$y = x * 2;$	$y = x + x;$