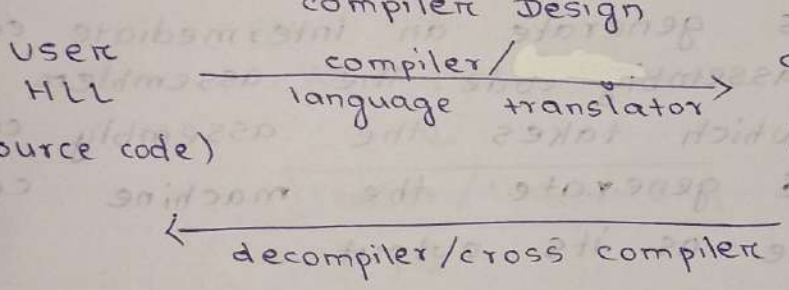
Compiler Design

Module-1

Introduction to compiler

Compiler is a program which takes one language (high level language ie source code) as its input and translates it to an equivalent language known as machine level language ie the object code.

During this process of conversion if some errors are encountered then the compiler displays the error message unless until the errors are not debugged the program will never execute.



- Compiler may be defined as a program or set of programs that translates the text detail in one language to another.
- Hence otherwise called as language translator.
- We can also use the program that translates MLL to a HLL known as decompiler.

Context in which compiler works : The context in which the compiler typically operates are preprocessors, Assemblers, Loaders & Linkers.

1. Preprocessors

The task performed by the preprocessors are

1) The preprocessor allow the user to use the macro in the program. Macro is set up instructions which can be used repeatedly in the program.

For this use the preprocessor directive. #define.

2) It also allow the user to include the header file that defines the library functions to be used in the program. For this purpose we use preprocessor directive #include

3) The role of preprocessors directive in compiler is to generate a complete source code from a skeleton source code.

Skeleton source code → (preprocessor) → complete source code → (compiler) → object code

2. Assembler

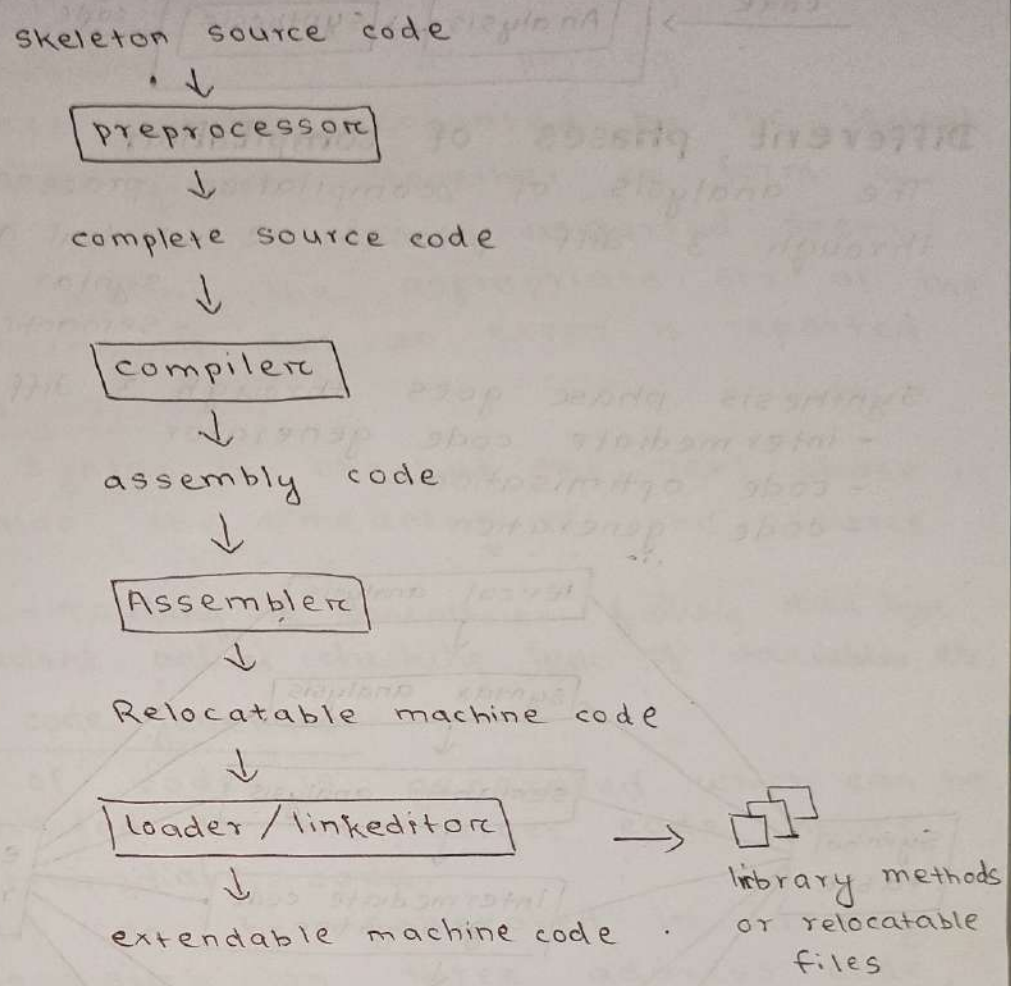
The compilers generate an intermediate code known as assembly code. The assembler is a translator which takes the assembly code as its input & generate the machine code on object code as its output.

source code → (compiler) → Intermediate code/Assembly code → (Assembler) → machine code/ object code

3. Loader & Linker

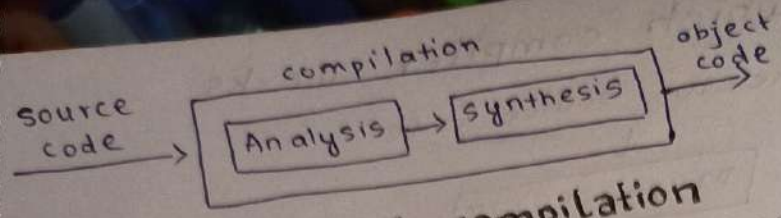
- Loader is a program which performs two functions ie loading and link editing.
- Loading is a process in which the relocatable machine code is read is the relocatable addresses are generated then these codes is corresponding data is placed in memory at proper location.
- The job of a linker is to make a single program from several linked files of the relocatable machine codes.
- If in one file reference to the location of other file is done then it is called external reference which can be resolved by linker editor.

Context in which compiler works



Phases of compiler

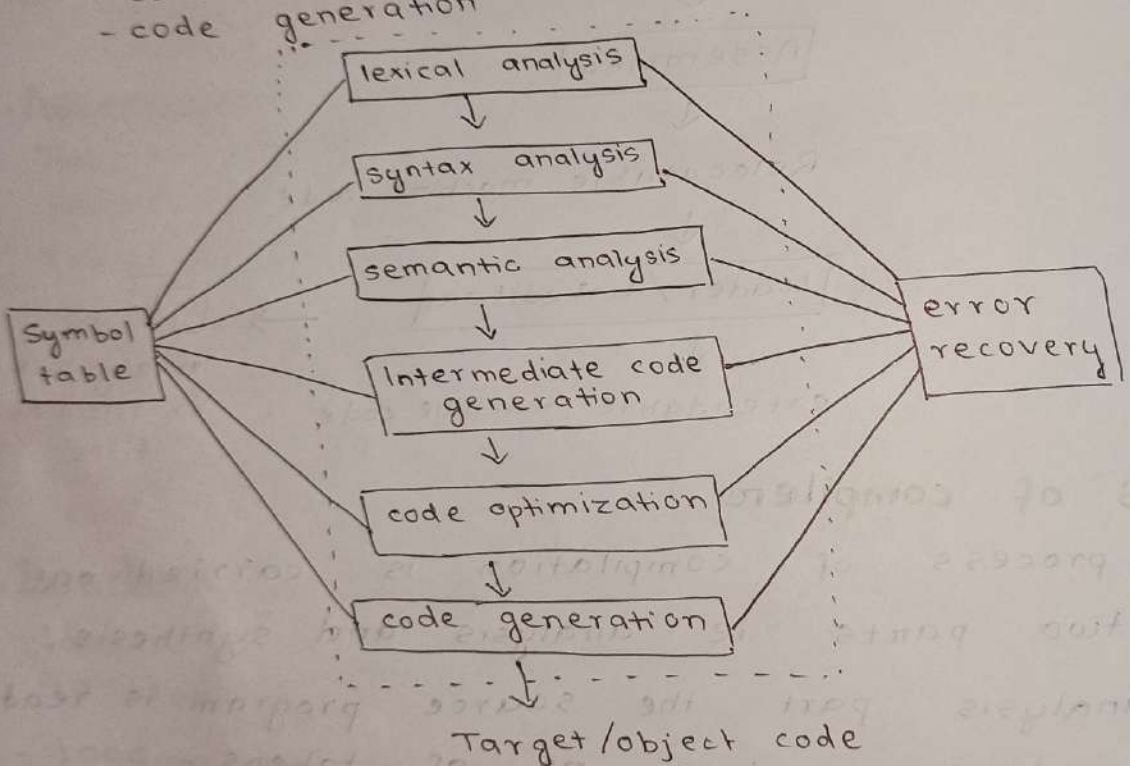
- The process of compilation is carried out in two parts ie - analysis and synthesis.
- In analysis part the source program is read and broken into no. of tokens -
- Then the meaning of each source string is determined and a proper relation is to be established between these strings.
- Then an intermediate code is generated from the input source program.
- In synthesis phase the intermediate form of this code is taken, optimised and converted to the equivalent object code.



Different phases of compilation

The analysis of compilation process again go through 3 diff phases ie - lexical analysis
- syntax analysis
- semantic analysis

Synthesis phase goes through 3 diff phases:
- intermediate code generator
- code optimisation
- code generation



Lexical analysis

- It is also called as scanning.
- It is the phase in which the complete source code is scanned and generates group of string known as tokens.
- A token is a sequence of character having a meaning.

for example - int a ;
- int → Keyword
- a → variable
- ; → symbol delimeter

Syntax anal

- It is oth
- In this ph analyser hierarchic
- This phas source

Semantic an

- once the to deter string. for exam of

Intermediat

- This kin easily called
- This co of fo quadru
- For ex c = a + t1 = a c = t1

- This maxim per

Code opti

- The co or im
- This (ie - th

Code g

- In thi which

source code

Syntax analysis

- It is otherwise called as parsing.
- In this phase tokens generated by the lexical analyser are grouped together to form a hierarchical structure known as parsed tree.
- This phase means the appropriate str of the source string if not, an error is reported.

Semantic analysis

- Once the syntax is checked, the next phase is to determine the meaning of the source string.
  For example - matching of parentheses, finding data type of constant values, checking type of variables etc.

Intermediate code generator

- This kind of code is generated which can be easily converted to the target code hence called intermediate code.
- This code can be generated in variety of forms such as three address code, quadruple and triple.
- For example - If we have a source instruction $c = a + b$, then intermediate code will be

  $t_1 = a + b$
  $c = t_1$

  or

  $c = a + b/d$
  $t_1 = b/d$
  $t_2 = a + t_1$
  $c = t_2$

- This is called as a 3-address code because maximum 3 address location can be referred per instruction.

Code optimization

- The code optimization phase attempts to optimize or improve the intermediate code.
- This is necessary to have a faster executing code (ie - the overall running time of the target program can be improved)

Code generation

- In this phase the target code gets generated for which an equivalent assembly code is to be produced.

Source code:	assembly code:	
$c = a + b$	MOV a, R_1	$R_1 \leftarrow a$
	ADD R_1, b	$R_1 \leftarrow R_1 + b$
	STA c, R_1	

Example of compilation

For the statement $a = b + c * 60$; the compilation phases with the corresponding i/p and o/p act as follows :-

$a = b + c * 60$

↓

lexical analysis (token generated)

↓

Identifiers → a, b, c

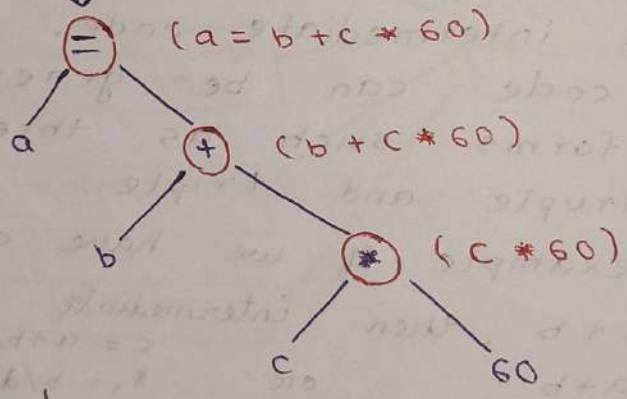
Operator → =, +, *

Constant → 60

delimeter → ;

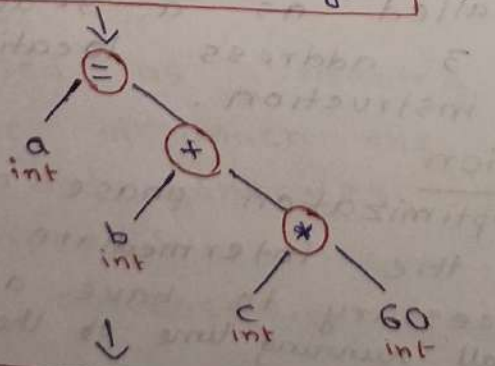
↓

syntax analysis (parse tree/ syntax tree)



↓

semantic analysis (full parse tree)



↓

Intermediate code generation (three adress code)

$t_1 : c * 60$;

$t_2 : b + t_1$;

$a : t_2$;

Grouping

Source code →

- Different
- The fro
- It cons
- Tho ba
-

↓

code optimization (reduce no. of instruction)

$t_1 := c * 60;$

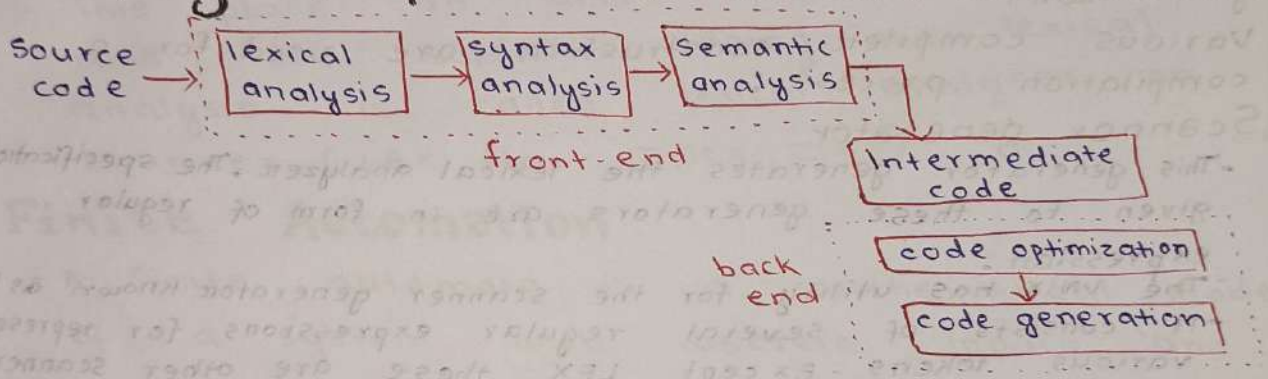
$a := b + t_1;$

↓

code generation (generate equivalent assembly code)

MOV	C, R_1	$C \rightarrow R$
MUL	R_1, #60	$R_1 \rightarrow R_1 * 60$
ADD	R_1, b	$R_1 \rightarrow R_1 + b$
STA	a, R_1	$R_1 \rightarrow a$

Grouping of phases (front-end - back-end model)



- Different phases of compiler can be grouped together to form a end and back-end.
- The front end consists of those phases that primarily depend on the source language and independent on the target code.
- It consists of analysis part typically includes lexical analysis, syntax analysis and semantic analysis.
- The back-end consists of the phases that totally depend upon the target code and independent on source code.
- This end is responsible for translation of intermediate code to equivalent assembly code & target code.

- front end-backend model of the compiler is advantageous because of the following reasons :-
  (i) Keeping the same front end and attaching different back ends, we can produce a compiler for same source language on different machine.
  (ii) Keeping the different front ends & same backends we can compile several different source languages on the same machine.

Tools to form construction of compiler

The compilation process use different tools for different phases is known as compiler or compiler generation.

Various compiler construction are used for a compilation process :

- Scanner generator
  - This generator generates the lexical analyzer. The specification given to these generators are in form of regular expression.
  - The UNIX has utility for the scanner generator known as LE
  - It consists of several regular expressions for represent various tokens - Except LEX these are other scanner generator :- Flex, Zlex, JFlex
- Parser generator
  - This produce the syntax analyzer.
  - The specification given to this generator is CFG (context free grammar) UNIX has a tool YACC (yet another compiler compiler)
- Syntax directed translation engines
  - These engines produce intermediate code with 3 address format from the generated parse tree.
  - In this tool the parse tree is scanned completely to generate an equivalent intermediate code and this translation is done for each node of the parse tree.
- Automatic code generator
  - These generators taken an intermediate code as it's input and convert each code to an equivalent assembly code further to an equivalent machine language.

Lexical

- The l
  into
- In oth
  single
  Ex - A
- The
  langu
  that
- This
  and
  analy
  scan

Finite

- A fin
  of
  outpu
- FSA
  going
  depen
  This
- A fi
  5
  Q = n
  Σ =
  δ =
  q_0 =
  F =
- A f
  tran
  A t
  labe

Lexical Analysis

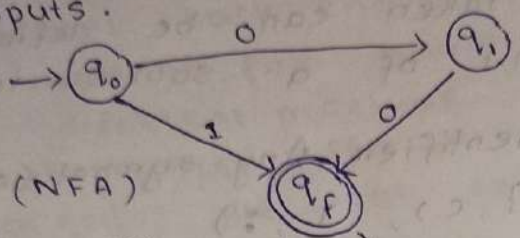
- The lexical analysis phase breaks the source into small pieces called as tokens.
- In other terms token can be defined as a single atomic unit of a source instructions.

  Ex - A keyword, Identifier, Any symbol (operators)
  Delimeter ({}, (), [], ;)
- The token syntax is typically a regular language hence a finite state automation that can be used to recognize the token.
- This phase can also called as lexing, scanning and the software doing the lexical analysis is called lexical analyser or scanner (LEX, flex, Jflex, Zlex)

Finite Automation

- A finite automata is a mathematical model of a system with discrete inputs and outputs.
- FSA has a set of states and rules for going from one state to another state depending on the input symbol.
  This state change is called as transition.
- A finite automation can be defined by 5
  $M = (Q, \Sigma, \delta, q_0, F)$
  Q = number / set of states
  Σ = set of input symbols
  δ = Transition function / mapping function $\quad \delta: Q \times \Sigma \to Q$
  q_0 = initial state $\quad q_0 \subseteq Q$
  F = final state $\quad F \subseteq Q$
- A finite automation can be represented using transition system.
  A transition system is a finite directed labelled graph in which each vertex

represents a state and the directed edge indicates the transition labelled with the given inputs.



$Q = \{q_0, q_1, q_2\}$

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

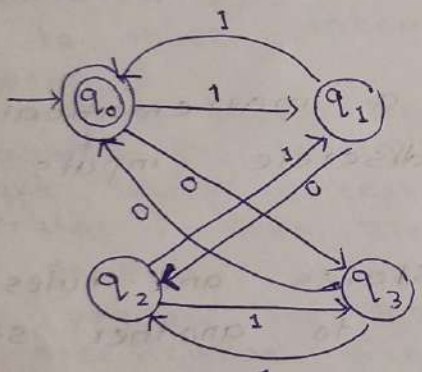
$q_0 : \rightarrow 0 \quad \{q_0\}$

$F : ⓪ \quad \{q_f\}$

(Transition system)

States	Input	
	0	1
$\rightarrow q_0$	q_3	q_1
q_1	q_2	q_0
q_2	q_1	q_3
q_3	q_0	q_2

Start	Input	
	0	1
$\rightarrow q_0$	$\{q_0\}$	$\{q_f\}$
q_1	$\{q_f\}$	-
$*q_f$	-	-



The finite automation further classified into

i) DFA (deterministic finite automation)

ii) NFA (non-deterministic

- A finite automation can be deterministic if
  i) It has no ϵ-transition.
  ii) for each state and each input symbol there is atmost one edge or transition

- A finite automation is non-deterministic if

i) It has ϵ-transition

ii) one input symbol can cause multiple transition ie a state may have more than one transition with the same input symbol.

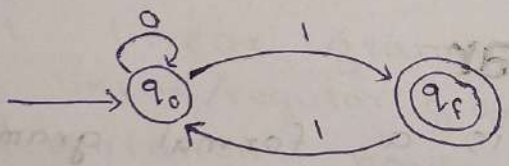
iii) It is not compulsory that all the states have to consume all the input symbols.

Regular language

- A language is called as a regular language if some finite automation recognizes it.

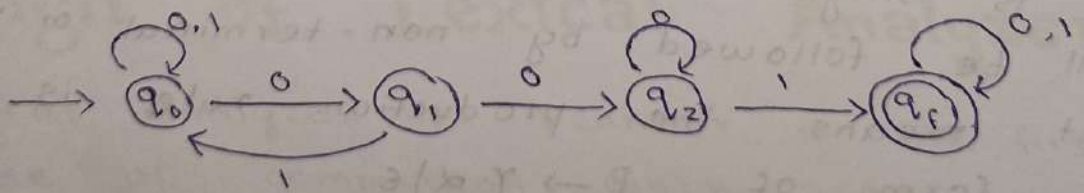
$L(M) = \{w \mid w$ is a string with odd no. of 1's$\}$

$L(M) = \{01, 00111, 0111, 111, 111110\}$



$L(M) = \{w \mid w$ is a string containing 001 as a substring $\}$

$L(M) = \{001, 000010, 111001111\}$



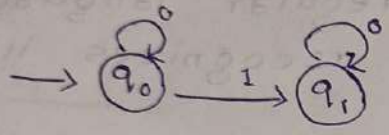
Regular expression

The expressions that are generated by sing regular operations are called regular expressions.

Union (∪)
Concatenation (•)
Closure { * (star closure)(klene closure)
+ (+ve closure) }

R.E = $0^* \perp 0^*$ = {1, 01, 10, 001, 0010, 00010, 00100...

$\approx 0^* \circ 1 \circ 0^*$



$0^* = \{\not\epsilon, 0, 00, 000...\}$

L(m) = {w|w is a string with only one 1 having any no. of zeroes at beginning or end}

Regular Grammar

- A regular grammar is a formal grammar that describes a regular language.
- A grammar is said to be regular if and only if 'γ' is single non-terminal and 'α' is single terminal & the terminal will be followed by non-terminal or vice-versa
- That means the production rule is in the form of $P \rightarrow \gamma\alpha | \epsilon$
  or $P \rightarrow \alpha\gamma | \epsilon$
- Generally the regular grammar is of two types
  - Right linear regular grammar/
  - Left linear regular grammar

- Right regular grammar :-
  A right regular grammar is a formal grammar with 4 tuples.
  i) N/V : non-terminal & variable
     derivation impossible
  ii) T/Σ : terminal & non-variable
     derivation not possible
  iii) P :
  (iv) S :

N: non
T: terminal
P: set of production rules
S: start symbol

Having the production rule is in form of
$P \longrightarrow \alpha (\gamma) | \in$

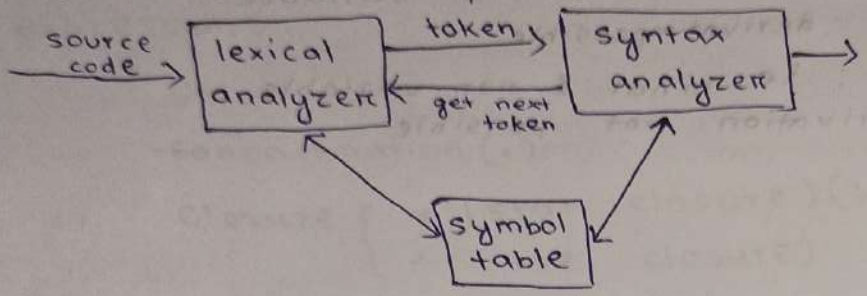
$P \longrightarrow \alpha$

- Left linear grammar :-
  Left linear/regular gramar is also a 4 tuple representation having the production rule
  $P \longrightarrow \gamma\alpha \mid \in$
  $P \longrightarrow \alpha$

Lexical Analysis Process or Design of Lexical Analyzer

- Lexical analyzer or scanner is the first phase of compilation which scans the input program & break it up into no. of smallest units called as tokens.

- Each token is a sequence of characters having a specific mean.
- It can be represented as



- It also performs some task at user interface one of it is - removing the comment lines and white spaces (blank space, /t, /n, ' ')
- Another process is to copy the error messages from the compiler to the source program.
- Each error message is also be associated with a line number because the lexical analyser keeps the track of new line characters.

Basic units of lexical analyzer

01. Lexemes

- These are the smallest logical units of a program.
- It is a sequence of character in a source program for which tokens are produced.

For example - int , a , b , 10 , + , =
(keyword , variables , operators)

02. Tokens

- classes of similar lexemes are identified by the same tokens such as keyword, operators etc.

03. Patterns

- It is the rule which describes a token.

For example - An identifier is a string consisting of digits or letters but should start with a letter or alphabet.

For example - Consider the following function in C code

Source code

```
float sum_two(float n1, float n2)
{
 float sum;
 sum = n1 + n2;
 return sum;
}
```

Lexemes	Tokens
float, return	keyword
sum_two	Identifier
n_1	Identifier
n_2	Identifier
{, }, ;	Delimeter
sum	Identifier
=, +	Operator

* The lexical analyzer parse additional information along with the tokens.

These informations are called as attributes.

$X = Y + 2 * Z$

Lexemes	<token, attributes>
x	<identifier, x>
y	<identifier, y>
z	<identifier, z>
2	<constant, 2>
=	<assignment operator, =>
+	<arithmatic operator, +>
*	<arithmatic operator, *>
;	<delimeter>

Construction of Lexical Analyzer

- The lexical analyser can be constructed for a language in which keywords, operators, delimeters, identifiers & white spaces are allowed.
- From the input stream it recognises the tokens and their corresponding attributes and returns them to syntax analyzer and also stores the newly generated tokens to symbol table.
- Again it has to keep track of line numbers for purpose of reporting errors & debugging.

Scanning Process of Lexical Analysis

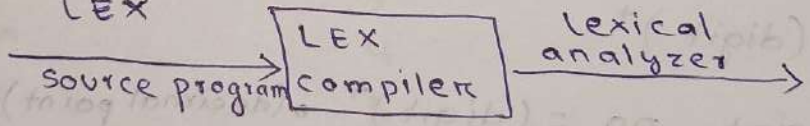
- Lexical analyzer reads the source program character by character and put it into a buffer.
- It may use one buffer scheme or two buffer scheme.
- In one buffer scheme single buffer is used having 8 character long.
- If a lexing crosses over the boundary of the buffer it has to again refilled in order to scan the rest of the lexeme.
(That means the first part of the lexeme is overwritten each time in this process)
- To avoid this difficulty two buffer scheme is implemented where two buffers were used alternatively if second buffer is filled, we may overwrite the first buffer which reduces the frequency of re-writing.

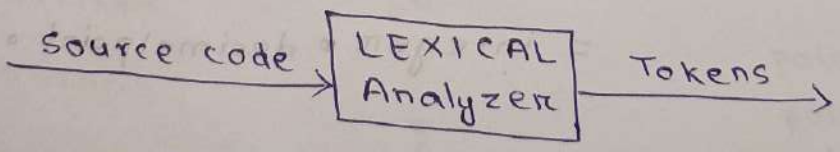
x : = : y : + : 2 : * : z : eof

Each buffer adds a sentinel which is a special character and not the part of the program.

A Language for specifying Lexical Analyzer

- A LEX is a tool for automatically generating lexical analyser.
- A LEX source program is a specification of a lexical analyzer consisting of a set of regular expressions together with an action for each regular expression.
- The action is a piece of code which is to be executed whenever a token specified by the corresponding regular expression is recognized.





(The role of LEX)

- A LEX source program consists of 2 parts, a sequence of auxillary defination followed by a sequence of translation rule.

Auxiliary definition

- These are the statements in form of

$D_1 = R_1$

$D_2 = R_2$

$D_n = R_n$

where D_i is a distinct name, R_i is a regular expression whose symbols are choosen from $\Sigma \cup \{D_1, D_2 \dots D_n\}$

- For example -

generally the auxillary definition for a programming language is represented as

letter = A|B|C|....|z|a|b|...|z

D_1 R_1

digit = 0|1|2|....|9

identifier = (letter) ∘ (Letter ∪ digit)*

D R

integer = (digit)+

floating_point_no. = (digit)+ ∘ (decimal point) ∘ (digit)+

= integer ∘ decimalpoint ∘ integer

sign = +/-

signedinteger = (sign)(integer)

Translation rule

- These are the rules of a LEX program in form of

$P_1 \{A_1\}$

$P_2 \{A_2\}$

:

$P_n \{A_n\}$

where P_i is a regular expression or pattern and A_i defines the action the Lexical analyzer should take when a pattern is found.

- For example -

Let the tokens are

Token	Code	Value
begin	1	-
end	2	-
if	3	-
then	4	-
else	5	-
identifier	6	pointer to symbol table
constant	7	pointer to symbol table
<	8	1
<=	8	2
=	8	3
< >	8	4
>	8	5
>=	8	6

- The auxillary definition is represented as

letter = a|b|c .... |z|A|B .... |Z.

identifier = (letter) (letter ∪ digit)*

digit = 0|1| .... |9

constant = (digit)+

- Translation rules (is to be written for each token provided here)

begin {return 1}

end {return 2}

if {return 3}

then {return 4}

else {return 5}

(letter) (letter|digit)* {return 6; LEX VAL := Install()}

(digit) {return 7; LEX VAL := Install()}

< {return 8; LEX VAL := 1;}

<= {return 8; LEX VAL := 2;}

= {return 8; LEX VAL := 3;}

<> {return 8; LEX VAL := 4;}

> {return 8; LEX VAL := 5;}

>= {return 8; LEX VAL := 6;}

Q For the following lexical analyzer construct an NFA.

Auxiliary defn

{none}

Translation rule

a { } /* Action omitted */

abb { }

a* b+ { }

Q1. Constr

Q2. Constr

Q3. Const

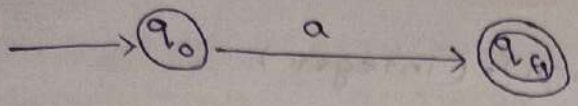
Q4. Const

Q5. Const

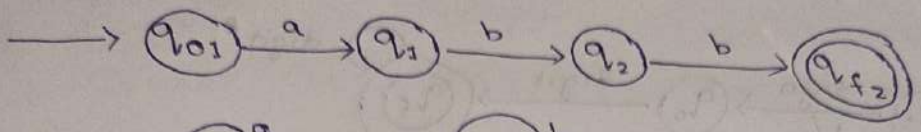
Ans

1. identif
letter
intege
digit

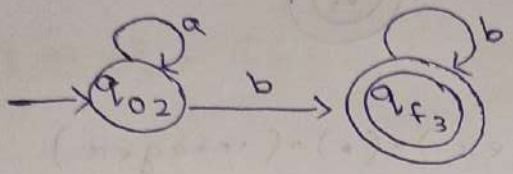
2. integ



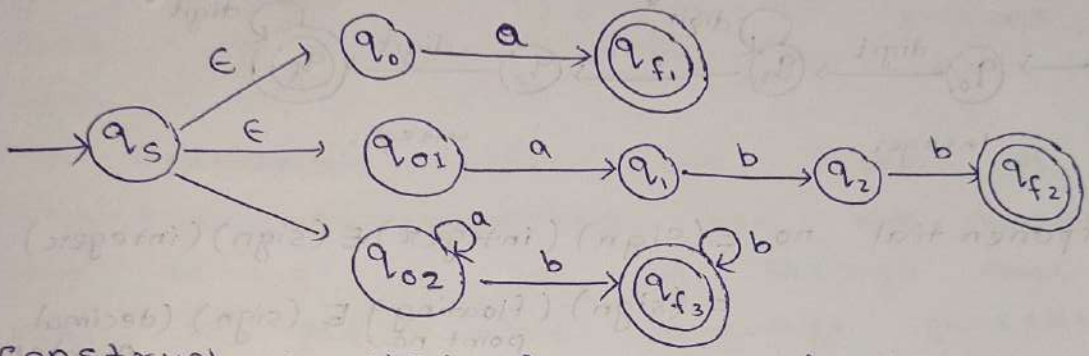
{a}



{abb}



{$a^* b^+$}



Q1. Construct a NFA for the identifiers

Q2. Construct the NFA for integers

Q3. Construct NFA for signed integer

Q4. Construct NFA for decimal no.s

Q5. Construct NFA for exponential no.s

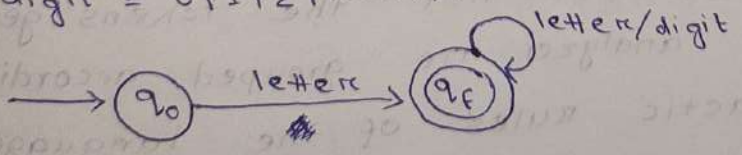
Ans

1. identifier = (letter) (letter/digit)*

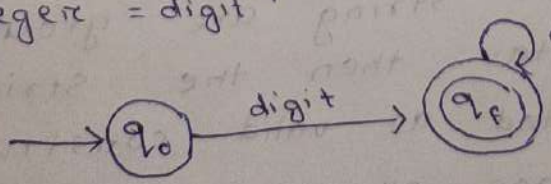
letter = A|B| .... z|a|b| ....z

integer = digit +

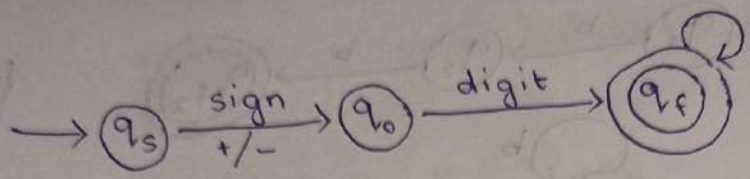
digit = 0|1|2| ..... |9



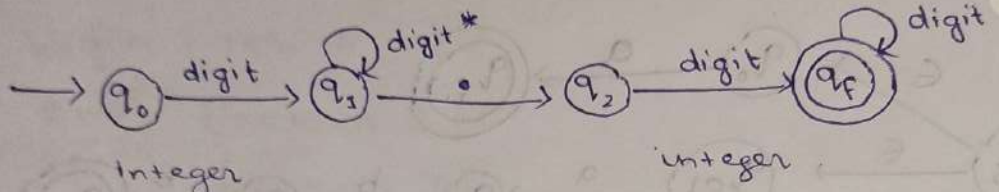
2. integer = digit +



3. Signed integer = (sign) . (integer)

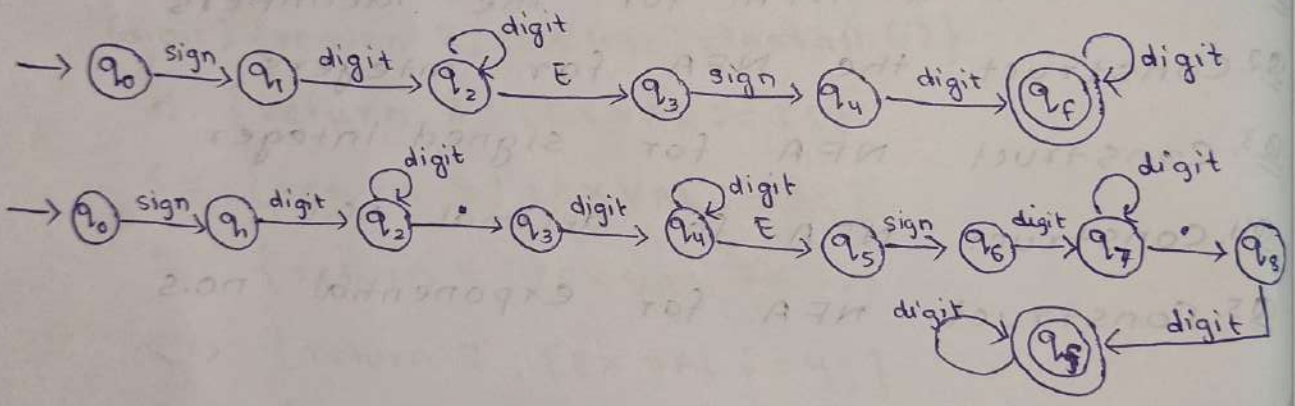


4. Decimal point = (integer) . (.) . (integer)



5. Exponential no. = (Sign) (integer) E (sign) (integer)

= (sign) (floating point no.) E (sign) (decimal number)



Syntax Analysis

- In the syntax analysis phase the compiler verifies whether or not the tokens generated by the lexical analyzer are grouped according to the syntactic rules of the language.
- If the tokens in a string are grouped accordingly, the syntax then the string of tokens is accepted as valid construct of the given language otherwise an error handler is called.

- The syntax structure specification for the programming language uses context-free grammar.
- This phase is otherwise called as parser.

Context-free grammar

- CFG notation specifies a context-free language that consists of terminals, non-termials, a start symbol and set of production rules.
  (N, T, P, S)
- The terminals are the tokens of the language, non-terminals are the variables that denote a set of strings that are used to define the language generated by the grammar.
- Formally we can represent a CFG
  $G = (N/V, T/\Sigma, P/R, S)$ where
  - N/V: a finite set of symbols called as variables or non-terminals
  - T/Σ: set of symbols called terminal or input symbols
  - P/R: set of production rules
  - S: start symbol where $S \in N/V$

Eg: If $P/R : \{ S \to asa, S \to bSb, S \to \epsilon \}$

N/V : {S}

T : {a, b, ∈}

P : {S → aSa | bSb | ∈

S : {s}

Derivation

- This refers to replacing an instance of a non-terminal by the equivalent right hand side of the production rule whose left hand side contains the non-terminal to be replaced.
- If the string obtained as the result of derivation contains only the terminal symbols then no further derivation is possible.

Ex - consider the production rule

P/R : { S → aSa , S → bSb , S → ∈ }

derive the string w : abba

Ans. S → a S a

→ a bSb a

→ a b ∈ b a

→ a b b a

Types of derivation

1. Left most derivation
2. Right most derivation

(1) A derivation in which the left most non-terminal is replaced at every step is said to be left most derivation.

- Represented as $S \xRightarrow[lm]{*} w$

S = start symbol
w = string to be derived

(2) When a right most non-terminal is replaced at every step then it is referred as right most derivation or canonical derivation.

- Represented as $S \xRightarrow[rm]{*} w$

Q1. $S \rightarrow iCtS \mid iCtSeS \mid a$
$C \rightarrow b$
generate the string - ibtibtaea using LMD and RMD.

Q2. $E \rightarrow E+E \mid E*E \mid id$. Derive the string $w = id + id * id$ using both LMD & RMD.

Ans- 1. $S \rightarrow iCtS \rightarrow ibtS \rightarrow ibtiCtSeS$

$\rightarrow ibtibtSeS \rightarrow ibtibtaeS \rightarrow ibtibtaea$

$S \rightarrow iCtSeS \rightarrow iCtSea \rightarrow$ [illegible]

$\rightarrow iCtiCtSea \rightarrow iCtiCtaea \rightarrow iCtibtaea \rightarrow ibtibtaea$

2. $E \rightarrow E+E \rightarrow id+E \rightarrow id+E*E \rightarrow id+id*E$

$\rightarrow id+id*id$

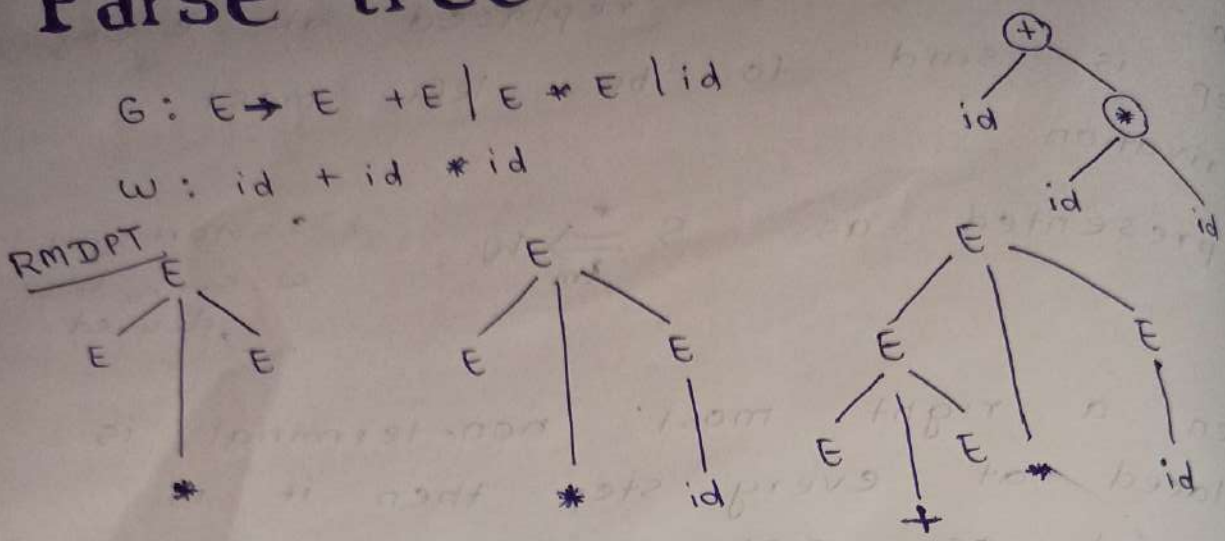
$E \rightarrow E+E \rightarrow E+E*E \rightarrow E+E*id \rightarrow E+id*id$

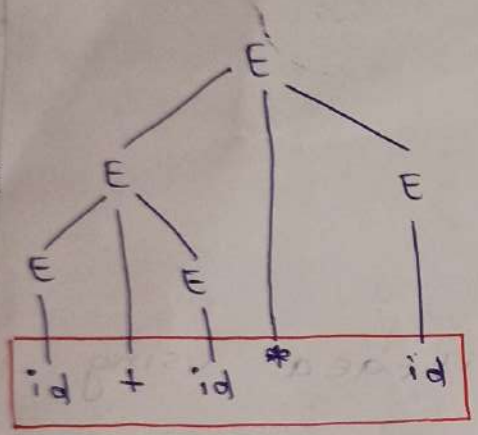
$\rightarrow id+id*id$

Parse tree

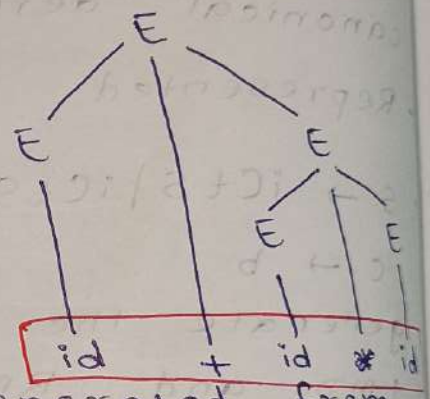
G: E → E + E | E * E | id

w: id + id * id





Similarly LMDPT



Parse tree is a tree generated from steps of derivation to construct a string w from a start symbol S.

- A parse tree can be constructed by using following rules
  - → All leaf nodes of the tree are labelled by the terminals of the grammar
  - → The root node of the tree is labelled with the start symbol of the grammar.
  - → The interior nodes are always the non-terminals

→ If any non-terminal derives n number of symbols then its corresponding parse tree has that many branches

Ambiguous grammar

- The grammar that consist of multiple ways of derivation to derive a string w having multiple parse trees is known as ambiguous grammar.

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

$w : id + id * id$

$E \rightarrow E * E$

$\rightarrow E * (E) \rightarrow (E) * id \rightarrow E + (E) * id$

$\rightarrow (E) + id * id \rightarrow id + id * id$

As the above grammar produces multiple right most derivation for the given string hence it is an ambiguous grammar.

Ex $G : S \rightarrow aSbS \mid bSaS \mid \epsilon$

$w = abab$

$S \rightarrow a(S)bS \rightarrow a\,bSaS\,bS \rightarrow ab\epsilon a(S)bS$

$\rightarrow aba\epsilon b(S) \rightarrow ababc \rightarrow abab$

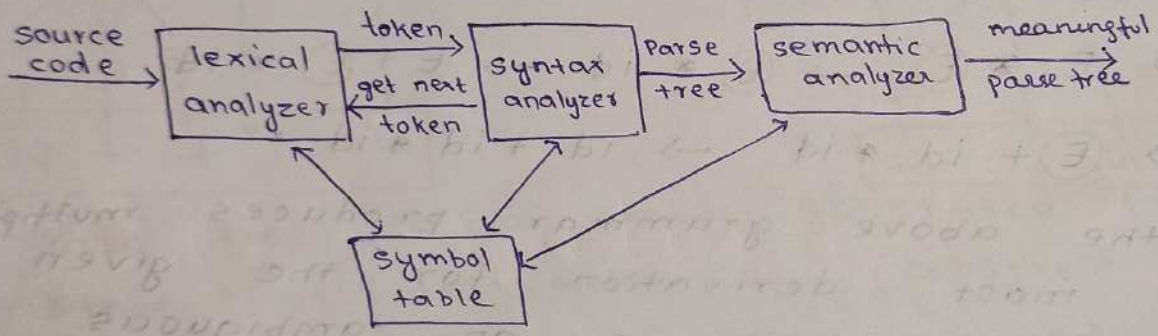
$S \rightarrow a(S)bS \rightarrow a\epsilon b(S) \rightarrow ab\,a(S)bS \rightarrow aba\epsilon b(S)$

$\rightarrow ababc \rightarrow abab$

Syntax analysis phase of compilation (basic parsing technique)

The parsing otherwise called as syntax analysis where grammar G is used to derive a string w starting from the start symbol S and generates a parse tree.

If the parsing is not proper one (ie w is not part of G) then an error message is generated.



- Two basic types of parsers are there
  1) Top down parser (It starts from the start symbol S & generates w)
  2) Bottom up parser (starts from the string w & generate start symbol S)

1) # Top-down

In top-down parsing the technique used is recursive descent parsing (LL(1) grammar) → left to right scanning with leftmost derivation

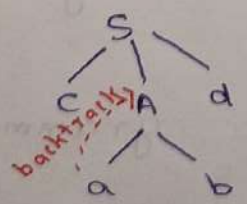
2) # Bottom-up

Bottom-up parsing is also called as shift-reduce parsing that use LR(1) grammar → Left to right scanning with rightmost derivation

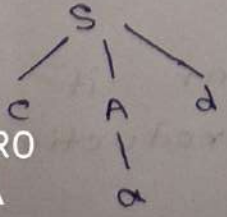
1) Top-down

- In general the top-down parsing involves backtracking.
- This mechanism can be viewed as an attempt to find a proper ~~left~~ derivation for string w.
- If the substitution is not appropriate one then backtrack again to find another alternate.

Ex – S → CAd w = cad A → ab|a



1st
S → CAd
→ cabd [A → ab]
≠ w



2nd
cad ≈ w (successful)

- The top down parsing is also called as recursive descent parser which is built from a set of mutually recursive procedures where each pradure implements one of the production rules of the grammar.
- If parsing does not generate the given string then alternative production rule is choosen.
- To eliminate the limitations we can design a predictive parser that does not require backtracking.
  It choose the appropriate substitution from the number of given production to generate the string 'w'.

(a) Left factoring

- Generally a recursive descent parser is not deterministic.
- Hence a predictive parser must be generated that is generally deterministic in nature.
- For example - consider the grammar
  S → if a
  S → if a then b
  In the above grammar it is not clear exactly which production rule

is to be used to recognize a conditional statement.

- The solution to this problem is left factoring.
- It is a method for manipulating grammars into a form suitable for recursive descent parsing.
- It is the process of factoring out the common prefixes of each alternatives
- The above grammar after left factoring is represented as

common prefix

$S \rightarrow$ if a S'

$S' \rightarrow \epsilon \mid$ then b ← remaining symbols

- In general if we have the production rules $\alpha\beta \mid \alpha\gamma$ then

$A \rightarrow \alpha A'$

$A' \rightarrow \beta \mid \gamma$

w = cata.

s → data.

- $S \rightarrow iCtS \mid iCtSeS \mid a$

$C \rightarrow a$

After left factoring, $S \rightarrow iCtSS' \mid a$

$S' \rightarrow \epsilon \mid eS$

1

Elimination of left recursive

- If we have a left recursive pair of productions such as $A \rightarrow A\alpha \mid \beta$ where β does not begin with A, then we can eliminate ~~with~~ this left recursion by replacing these productions as

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid \epsilon$$

- consider the following grammar

$$E \rightarrow E+T \mid T$$
$$T \rightarrow T*F \mid F$$
$$F \rightarrow (E) \mid id$$

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

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

$$E \rightarrow idE'$$
$$E' \rightarrow +EE' \mid \epsilon$$
$$T \rightarrow idT'$$
$$T' \rightarrow *TT' \mid \epsilon$$

$A \to A\alpha \mid \beta$

$A \to A\alpha\alpha$

$A \to A\alpha\alpha\alpha \ldots$

$\Rightarrow \beta\alpha\alpha\alpha$.

$A' \to \alpha A'$

$\to \alpha\alpha\alpha A'$

$\beta\alpha\alpha\alpha\epsilon \simeq \beta\alpha\alpha\alpha \to \alpha\alpha\alpha\epsilon$.

Design of predictive top-down parser

- A backtracking parser is generally a non-deterministic recogniser but it can function as deterministic recogniser if it is capable of predicting or detecting the appropriate alternative for expansion of non-terminal during parsing of the string 'ω'.

- If $A \rightarrow \alpha_1 | \alpha_2 | \alpha_3 \ldots \alpha_n$ (n. no. of production rules) are production in the grammar then the predictive parser can decide by using which alternative the non-terminal A is to be expanded.

FIRST ( )

- The function FIRST (α) for $\alpha \in (V \cup T)^*$ where V is a variable & T is a terminal.

- We can determine the FIRST() as the set of those terminals which will derive the string with the substitution of α.

- if $\alpha \rightarrow XYZ$

  FIRST (α) = FIRST (XYZ) = {X} if X is a terminal.

& i

- Otherwise FIRST (α) = FIRST (XYZ) = FIRST (X) provided that X does not derive an empty string ∈ that is FIRST (X) does not contain ∈.

- If FIRST (X) contains ∈ then,

  FIRST (α) = FIRST (XYZ) = FIRST (X) − {∈} ∪ FIRST{YZ}

- FIRST (YZ) = {Y} if Y is a terminal.
- FIRST (YZ) = FIRST (Y) if Y does not derive ∈
- If ~~FIRST (YZ)~~ Y derives ∈ then,

  FIRST (Y) − {∈} ∪ FIRST (Z)

Consider the following set of grammar & generate FIRST ()

S → AcB | cbB | Ba
A → da | BC
B → g | ∈
C → h | ∈

FIRST (S) = FIRST (AcB) ∪ FIRST (cbB) ∪ FIRST (Ba)
FIRST (A) = FIRST (da) ∪ FIRST (BC) = {d} ∪ FIRST (BC)
FIRST (B) = FIRST (g) ∪ FIRST (∈) = {g, ∈}
FIRST (C) = FIRST (h) ∪ FIRST (∈) = {h, ∈}
Now FIRST (A) = {d} ∪ FIRST (BC)
= {d} ∪ { FIRST (B) − ∈ ∪ FIRST (C)}
= {d} ∪ {g ∪ {h, ∈}}
= {d, g, h, ∈}

Now,

FIRST (ACB)

= FIRST (A) - ϵ ∪ FIRST (CB)

= {d, g, h} ∪ {FIRST (c) - ϵ ∪ FIRST (B)}

= {d, g, h} ∪ {h ∪ {g, ϵ}}

= {d, g, h, ϵ}

FIRST (cbB)

= FIRST (c) - ϵ ∪ FIRST (bB)

= {h} - ϵ ∪ FIRST (bB)

= {h} ∪ {b}

= {h, b}

FIRST (Ba)

= FIRST (B) - ϵ ∪ FIRST (a)

= {g, ϵ} - ϵ ∪ {a}

= {g, a}

Now, FIRST (S) = FIRST (ACB) ∪ FIRST (cbB) ∪ FIRST (Ba)

= {d, g, h, ϵ} ∪ {h, b} ∪ {g, a}

= {a, b, d, g, h, ϵ}

Q) E →
T →
F →

E → T
E' →
T →
T' →
F →

FIRST (E
FIRST (T
FIRST (
FIRST (
FIRST (
FIRST (T

FIRST (E

Q) $E \to \underbrace{E}_{A} \underbrace{+T}_{\alpha} \mid \underbrace{T}_{\beta}$

$T \to T * F \mid F$

$F \to (E) \mid id$

$A \to A\alpha \mid \beta$

$A \to \beta A'$

$A' \to \alpha A' \mid \epsilon$

$E \to TE'$

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

$T \to FT'$

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

$F \to (E) \mid id$

FIRST (E) = FIRST (TE')

FIRST (T) = FIRST (FT')

FIRST (F) = FIRST ((E)) ∪ FIRST (id) = { (, id }

FIRST (T) = FIRST (F) = { (, id }

FIRST (E) = FIRST (T) = { (, id }

FIRST (T') = FIRST (*FT') ∪ FIRST (ϵ)

FIRST (E') = FIRST (+TE') ∪ FIRST (ϵ)

∪

Ba)

FOLLOW()

FOLLOW(A) can be obtained as if there is a production $A \to \epsilon$, then 'b' may be the follow of A, otherwise follow(A) = set of terminals that immediately follows A in any string occuring on the right hand side of the production rule in the given grammar.

Never write ϵ in follow.

For example - consider a production rule

$A \longrightarrow \alpha B \mid \beta$

At the right hand side of production rule B is a non-terminal present followed by Symbol β.

Now we can obtain

i) FOLLOW(B) = FIRST(β) if FIRST(β) doesn't contain ϵ

ii) FOLLOW(B) = {FIRST(β) - ϵ} $\cup$ FOLLOW(A) when FIRST(β) contains ϵ as when β derives to ϵ then the terminal Symbol following A will be considered as FOLLOW(B)

- We can generate a predictive parsing table as follows

Step 1> compute FIRST() and FOLLOW() for every non-terminal of the given grammar.

Step 2> Now for every production $A \to \alpha$, do the following.

- for every non-ϵ non-terminal FIRST(α) ~~first of α~~ is entered as,

  TABLE $[A, a] = A \to \alpha$

  where a is a terminal in first(A).
- if FIRST(α) contains ϵ then for every 'b' the entry for FOLLOW(A) is TABLE $[A, b] = A \to \epsilon$

In other words, ~~i) FIRST~~

i) $A \not\to \epsilon$ if FIRST(A) = a, $[A, a] = A \to \alpha$

ii) $A \to \epsilon$ if FOLLOW(A) = b, $[A, b] = A \to \epsilon$

Q $S \to aABb$

$A \to c \mid \epsilon$

$B \to d \mid \epsilon$

Consider the grammar & construct a predictive parsing table.

FIRST(S) = FIRST(aABb) = {a}

FIRST(A) = FIRST(c) ∪ FIRST(ϵ) = {c, ϵ}

FIRST(B) = FIRST(d) ∪ FIRST(ϵ) = {d, ϵ}

As the end marker of the start is dollar $ present at the bottom of the stack, it is immediately following the start symbol S. Hence for every start symbol, follow is '$'.

FOLLOW(S) = {$}

FOLLOW(A) can be obtained S → aABb

FOLLOW(A) = FIRST(Bb) = FIRST(B) − ϵ ∪ FIRST(b)

= {d, ϵ} − ϵ ∪ {b} = {d, b}

FOLLOW (B) can be obtained from the production rule $S \rightarrow aABb$

FOLLOW(B) = FIRST (b) = {b}

FIRST (S) = {a}

FIRST (A) = {c, ∈}

FIRST (B) = {d, ∈}

~~FIRST (A) = {d, b}~~

~~FIRST (~~

FOLLOW (A) = {d, b}

FOLLOW (B) = {b}

terminal / non terminal	a	b	c	d	$
S	S→aABb				
A		A→∈	A→c	A→∈	
B		B→∈		B→d	

Consider the string w = aab, then the top down parsing steps are shown as below.

Stack	Input	moves
$	aab$	move start symbol S to the stack
$S	aab$	put S→aABb
$bBAa	aab$	pop 'a' from the stack and symbol 'a' from the input.
$bBA	ab$	It calls an error handling routine because A does not derive a. $A \not\rightarrow a$

let us consider the input string w = acdb

Stack	Input	moves
$	a c d b $	move start symbol to the stack
$S	a c d b $	put S → aABb
$ bBAa	a c d b $	pop 'a' from stack & remove symbol 'a' from input
$ bBA	c d b $	put A → C
$ bBc	c d b $	pop 'c' from stack & symbol 'c' from input
$ bB	d b $	put B → d
$ bd	d b $	pop 'd' from stack and input
$ b	b $	pop 'b' from stack and input
$	$	Announce successful completion of top down parsing

2. Consider the following grammar.

E → E + T | T

T → T * F | F

F → (E) | id

Construct a predictive parsing table & show the moves by the predictive parser for the string w = id + id * id.

LL(1) grammar

- If the parsing table contains multiple entries then the parser is non-deterministic.
- The parser will be deterministic if and only if there is no multiple entry in the parsing table.
  Such grammars are called LL(1) grammar.
- A grammar is said to be LL(1) if and only if following conditions satisfy for the production Rule

i) $FIRST(\alpha) \cap FIRST(\beta) = \phi$

FIRST(α) and FIRST(β) should produce disjoint sets of elements for every pair of production.

ii) At most one of α and β can derive empty string ϵ

iii) If FIRST(β) contains ϵ ie $\beta \overset{*}{\Rightarrow} \epsilon$, then α does not derive any string beginning with a terminal in FOLLOW(A).
ie $FIRST(\alpha) \cap FOLLOW(A) = \phi$

Q. Consider the following grammar & check whether it's LL(1) or not.

$S \rightarrow iEtSS' \mid a$
$S' \rightarrow eS \mid \epsilon$
$E \rightarrow b$

i) $FIRST(S) = FIRST(iEtSS') \cup FIRST(a) = \{i\} \cup \{a\} = \{i, a\}$

ii) $FIRST(S') = FIRST(eS) \cup FIRST(\epsilon) = \{e, \epsilon\}$

iii) $FIRST(E) = \{b\}$

i) FOLLOW (S) = {$}

find follow of S from $S \rightarrow iEtSS'$

$S' \rightarrow eS$

Now FOLLOW(S) from $S \rightarrow iEtSS'$

$FOLLOW(S) = FIRST(S') - \epsilon \cup FOLLOW(S)$

$= \{e\} \cup \{\$\} = \{e, \$\}$

Again, FOLLOW(S) from $S' \rightarrow eS$

$FOLLOW(S) = FOLLOW(S')$

ii) Now, FOLLOW(S') can be obtained from

$S \rightarrow iEtSS'$

↑

Now, $FOLLOW(S') = FOLLOW(S) = \{e, \$\}$

iii) FOLLOW(E) can be obtained from

$S \rightarrow iEtSS'$

$FOLLOW(E) = \{t\}$

$FIRST(S) = \{i, a\}$

$FIRST(S') = \{e, \epsilon\}$

$FIRST(E) = \{b\}$

$FOLLOW(S) = \{e, \$\}$

$FOLLOW(S') = \{e, \$\}$

$FOLLOW(E) = \{t\}$

NT \ T	i	a	e	b	t	$
S	$S \rightarrow iEtSS'$	$S \rightarrow a$				
S'			$S' \rightarrow eS$ $S' \rightarrow \epsilon$			$S' \rightarrow \epsilon$
E				$E \rightarrow b$		

From the above table the entry for S',e contains 2 production rules : $S' \rightarrow eS$, $S' \rightarrow \epsilon$ hence the grammar is ambiguous. So, not LL(1).

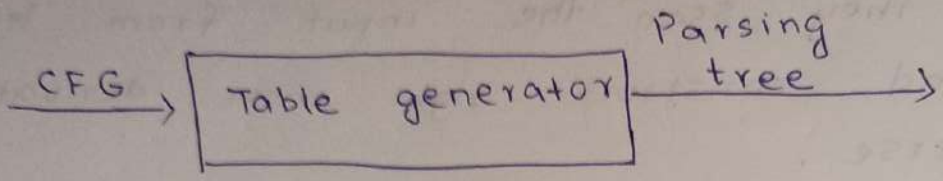
2) Bottom-up

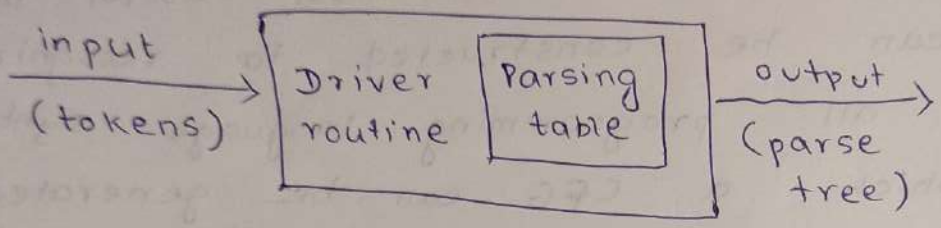
- The bottom-up parsers are called LR parsers because they scan the input from left to right and construct right most derivation in reverse.
- The LR parsers are most useful because they can be constructed to recognize almost all programming language syntax for which a CFG can be generated.
- LR parser can detect syntactic errors as soon as possible, to do so on a left to right scan of the input

* LR parser consist of two parts
  1) A driver routine & a parsing table

  - The driver routine is same for all LR parsers only the parsing table changes from one parser to another.
  - There are many different parsing tables that can be used in an LR parser for a given grammar
  - LR parser can use 3 different techniques for producing LR parsing tables
    - -> Simple LR (SLR)
    - -> Cannonical LR (CLR)
    - -> Loop ahead LR (LALR)

Diagrametically this can be represented as -





- Bottom-up parsing is called shift-reducing parsing because it reduce the given string to generate a single symbol ie start symbol.

- This parsing method is called bottom-up as it attempts to construct a parse tree for an input string beginning from the bottom that is leaf node & working towards top that is the root node.

For ex - consider the grammar

$S \rightarrow aAcBe$

$A \rightarrow Ab \mid b$

$B \rightarrow d$

Consider the string $w = abbcde$

Ans - Step 1 - Scan the string w from left to right.

we have $S \rightarrow aAcBe$

Hence, the substring b and d is to be replaced by corresponding productions.

step-2 - Let the leftmost b is choosen and replace it by A that is the left side of the production must contain A.

Hence $A \rightarrow b$

Now the resultant string is

$abbcde \rightarrow aAbcde$

step-3 - Again we have to replace Ab and d.

For Ab the replacement will be with A and for d the ~~capit~~ replacement will be B.

$A \rightarrow Ab$

$B \rightarrow d$

$aAbcde \longrightarrow aAcBe$

step-4 - Again the right hand side of the production rule that contains aAcBe is produced by the Symbol S.

Again reduce the string by applying the production rule

$S \rightarrow aAcBe$

$w = abbcde$

↓

$aAbcde$

↓

$aAcBe$

↓

S

Q What do you mean by Handle pruning?

Ans - Each replacement of the right side of a production by the left side is called the reduction.

In the above example, the sequence of 4 productions to reduce the string to s are

a<u>b</u> b c d e	A → b
↓	
a <u>A b</u> c <u>d</u> e	A → Ab
↓	
<u>a A c B e</u>	B → d
↓	
S	S → a A c B e

Handle

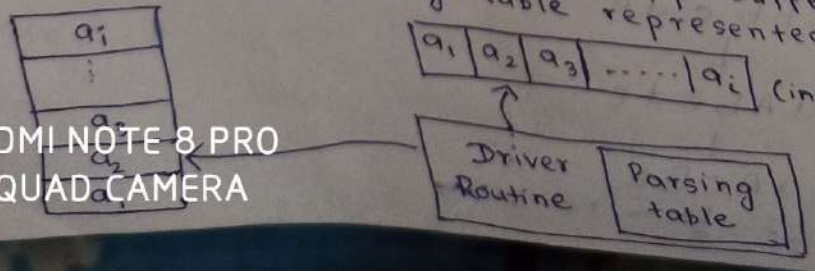
A substring which is the right side of the production by ~~the~~ its left side non-terminal is called handle.

In the above example the handles are b, Ab, d, aAcBe

Handle pruning

The process of identifying the handles and replacing them is known as handle pruning

LR parser - LR parsers have i/p buffer, a stack & a parsing table represented as



SLR
CLR
LALR

Q consider the following grammar.

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

$w \rightarrow id + id * id$

Ans -

Stack	Input	Action
$	id + id * id $	Shift
$id	+ id * id $	Reduce by E → id
$E	+ id * id $	Shift
$E +	id * id $	Shift
$E + id	* id $	Reduce by E → id
$E + E	* id $	Shift
$E + E *	id $	Shift
$E + E * id	$	reduce by E → id
$E + E * E	$	reduce by E → E*E
$E + E	$	reduce by E → E+E
$ E	$	Accepted

Q $w = (id + id) * id$

Stack	Input	Action
$	(id + id) * id $	Shift
$(	id + id) * id $	Reduce by E → id
$(E	+ id) * id $	shift
$(E +	id) * id $	Shift
$(E + id)	* id $	Reduce by E → id
$(E + E)	* id $	Reduce by E → E+E
$ E	* id $	shift
$ E *	id $	shift
$ E * id	$	Reduce by E → id
$ E * E	$	Reduce by E → E*E
$ E	$	Accepted

The input is read from left to right one symbol at a time & macrted with content of stack based on which LR parsers defines two behaviours,

1. ACTION
   either shift or reduce or accept or error
2. GO TO
   specifies a state generation from another state and symbol

→ SLR parsing

For SLR first we have to generate LR(0) items of the given grammar G* with a dot (.) are a position of right side of the production unless untill all the symbols aren't shifted.

Ex - Consider a grammar

$A \rightarrow XYZ$

Then the list of LR(0) items are:

$I_0 : A \rightarrow .XYZ$

$I_1 : A \rightarrow X.YZ$

$I_2 : A \rightarrow XY.Z$

$I_3 : A \rightarrow XYZ.$

One collection of the set of items for a production rule is called as cannonical LR(0) collection.

This are provided as the basics for constructing a class of LR parsers called simple LR.

Ex - consider the following grammar

$E \rightarrow E+T / T$

$T \rightarrow T*F / F$

$F \rightarrow (E) / id$

Find out the parsing table and show the moves of the input for id + id * id for SLR parser.

Step 1: In this step number the individual production of the grammar.

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

Step 2: Now find out the augmented grammar of the given grammar by adding a new start symbol to the previous grammar. Hence, the production rule will be;

new start symbol $\longrightarrow$ old start symbol

1. $E' \rightarrow E$
2. $E \rightarrow E + T$
3. $E \rightarrow T$
4. $T \rightarrow T * F$
5. $T \rightarrow F$
6. $F \rightarrow (E)$
7. $F \rightarrow id$

Step 3: The LR(0) items for the obtained grammar are as follows :-

I_0 : $E' \rightarrow .E$
$E \rightarrow .E + T$
$E \rightarrow .T$
$T \rightarrow .T * F$
$T \rightarrow .F$
$F \rightarrow .(E)$
$F \rightarrow .id$

I_1 : $E' \rightarrow E.$
$E \rightarrow E. + T$

I_2 : $E \rightarrow T.$
$T \rightarrow T. * F$

I_3 : $T \rightarrow F.$

I_4 : $F \rightarrow (.E)$
$E \rightarrow .E + T$
$E \rightarrow .T$
$T \rightarrow .T * F$
$T \rightarrow .F$
$F \rightarrow .(E)$
$F \rightarrow .id$

*NOTE

Now the above item set we have to obtain the closure of each non-terminal to be shifted / replaced with their corresponding production rules.

Ex - The closure of (.E) will be

$E \rightarrow .E+T$

$E \rightarrow .T$

Similar for T and F.

I_5 : $F \rightarrow id.$

I_6 : $F \rightarrow E+.T$

$T \rightarrow .T*F$

$T \rightarrow .F$

$F \rightarrow .(E)$

$F \rightarrow .id$

I_7 : $T \rightarrow T*.F$

$F \rightarrow .(E)$

$E \rightarrow .id$

I_8 : $F \rightarrow (E.)$, $E \rightarrow E.+T$

I_9 : $E \rightarrow E+T.$, $T \rightarrow T.*F$

I_{10} : $T \rightarrow T*F.$

I_{11} : $F \rightarrow (E).$

Hence the cannonical collection of the given grammar :

$E' \rightarrow E.$ (Item I_1)

$E \rightarrow E+T.$ (Item I_9)

$E \rightarrow T.$ (Item I_2)

$T \rightarrow T*F.$ (Item I_{10})

$T \rightarrow F.$ (Item I_3)

$F \rightarrow (E).$ (Item I_{11})

$F \rightarrow id.$ (Item I_5)

step 4:

construct a DFA for GOTO graph. from the generated cannonical LR(0) collections

Step-5 : Generation of parsing table.

The list of ~~parsing table~~ FIRST and FOLLOW.

FIRST (E) = {(, id}

FIRST (E') = {+, ∈}

FIRST (T) = {(, id}

FIRST (T') = {*, ∈}

FIRST (F) = {(, id}

FOLLOW (E) = {+,), $}

FOLLOW (E') = {$,)}

FOLLOW (T) = {*, +,), $}

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

FOLLOW (F) = {*, +,), $}
(FOLLOW (T))

we can construct a parsing table that is combination of state, action and GOTO

The action table has entries
S_i where i denotes the item set.
and π_i where i denotes the corresponding production rule which one is reduced.

* For shifting we can generate two separate tables one for terminal another for non-terminal.

For each reduction $A \rightarrow \alpha$. we have to write π (rule no.) for each entry in FOLLOW (A)

Now can

State (Items)
0
1
2
3
4
5
6
7
8
9
10
11

step-6 : Sho

Now the parsing table for the above grammar can be constructed as

State (Items)	Action						GOTO		
	id	+	*	(	)	$	E	T	F
0	S_5			S_4			1	2	3
1		S_6				Accepted			
2		π_2	S_7		π_2	π_2			
3		π_4	π_4		π_4	π_4			
4	S_5			S_4			8	2	3
5		π_6	π_6		π_6	π_6			
6	S_5			S_4				9	3
7	S_5			S_4					10
8		S_6			S_{11}				
9		π_1	S_7		π_1	π_1			
10		π_3	π_3		π_3	π_3			
11		π_5	π_5		π_5	π_5			

Step-6 :
Show the move for input ~~id id id id id~~

ω = id + id * id

stack	input	move/action
$	id + id * id $	shift
$0 id 5	+ id * id $	reduce by F → id
$0 F 3	+ id * id $	reduce by T → F
$0 T 2	+ id * id $	reduce by E → T
$0 E 1	+ id * id $	shift
$0 E 1 + 6	id * id $	Shift.
$0 E 1 + 6 id 5	* id $	reduce by F → id.
$0 E 1 + 6 F 3	* id $	reduce by T → F.
$0 E 1 + 6 T 9	* id $	Shift.
$0 E 1 + 6 T 9 * 7	id $	Shift.
$0 E 1 + 6 T 9 * 7 id 5	$	~~shift~~ reduce by F → id.
$0 E 1 + 6 T 9 * 7 F 10	$	reduce by T → T * F
$0 E 1 + 6 T 9	$	reduce by E → E + T
$0 E 1	$	~~Acc~~ reduce by E' → E
$0 E'	$	Accept.

Canonical LR (CLR)

- The general form of the canonical form item is represented as $A \rightarrow .\alpha\beta.a$ where $A \rightarrow \alpha\beta$ is a terminal present as lookahead of the corresponding production rule.
- such an item is called as LR(1) item where (1) refers to the length of second component called as look-ahead of the item.

Q consider the following grammar and construct the cannonical LR parsing table.

$S \rightarrow CC$

$C \rightarrow .cC \mid d$

Step 1 : Number the production rule.

1. $S \rightarrow CC$
2. $C \rightarrow cC$
3. $C \rightarrow d$

Step 2: Find the argumented grammar

$S' \rightarrow S$

$S \rightarrow CC$

$C \rightarrow cC$

$C \rightarrow d$

Step 3 : Find out the LR(1) item set

$I_0 : S' \rightarrow .S , \$$

$S \rightarrow .Cc , \$$

$C \rightarrow .cC , c/d$

$C \rightarrow .d , c/d$

$I_1 : S' \longrightarrow S. \ , \$$

$I_2 : S \rightarrow C . C , \$$

$C \rightarrow . CC , \$$

$C \rightarrow . d , \$$

$I_3 : C \rightarrow C . C , c/d$

$C \rightarrow . CC , c/d$

$C \rightarrow . d , c/d$

$I_4 : C \rightarrow d. , c/d$

$I_5 : S \rightarrow CC . , \$$

$I_6 : C \rightarrow C . C , \$$

$C \rightarrow . CC , \$$

$C \rightarrow . d , \$$

$I_7 : C \rightarrow d. , \$$

$I_8 : C \rightarrow cC , c/d$

$I_9 : C \rightarrow cC. , \$$

Now, the cannonical LR(1) DFA graph.

Now the parsing table of the given grammar is generated as

state	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	π_3	π_3			
5			π_1		
6	S_6	S_7			9
7			π_3		
8	π_2	π_2			
9			π_2		

Look Ahead LR (LALR)

- It states that in CLR if the production rule are same with different look ahead in any item set generated we can take the union of those item set to generate the equivalent item set.

$S \rightarrow cC$

$C \rightarrow .cc/d$

Step 1 : Production rule

$S \rightarrow CC$

$C \rightarrow cC$

$C \rightarrow d$

for the above example the item set generated may have the same production with different look ahead.

Ex-1 :- I_3 and I_6

I_4 and I_7

I_8 and I_9

having same production but diff. look ahead

I_{36} : $I_3 \cup I_6$

$C \rightarrow c.C$, $c/d/\$$

$C \rightarrow .cC$, $c/d/\$$

$C \rightarrow .d$, $c/d/\$$

I_{47} : $I_4 \cup I_7$

$C \rightarrow d.$, $c/d/\$$

I_{89} : $I_8 \cup I_9$

$C \rightarrow cC.$, $c/d/\$$

DFA

I_0

State
0
1
2
36
47
5
89

I_3

I_{89}

State	ACTION			GOTO	
	c	d	$	S	C
0	S_{36}	S_{47}		1	2
1			Accept		
2	S_{36}	S_{47}			5
36	S_{36}	S_{47}			89
47	π_3	π_3	π_3		
5			π_1		
89	π_2	π_2	π_2		

Operator precedence parsing

- This technique states that the CFG must consist of operators and based on the precedence of operators the parsing is to be done.
- It is suitable for the grammar that has no production with ε and there is no two adjacent non-terminals.

These kind of grammars are called operator grammar.

For example - $E \rightarrow EAE \mid (E) \mid -E \mid id$

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

The above grammar is not operator grammar because the right side EAE is two consequative non-terminal.

We can convert it into an operator grammar by substitution of 'A'.

$E \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid (E) \mid -E \mid id$

Advantage

- Simple and easy to implement

Disadvantage

- sometimes it is hard to handle the tokens like the (−) sign which has 2 different precedences depending on whether it is binary or unary.

Impler

- In
- pre
- <
- of
- If
- pre
- If
- If

Example

Consid
opera
E → E
ω →

Implementation

- In this parsing mechanism 3 disjoint precedence relations such as $<\cdot$, $\doteq$, $\cdot>$ are used between pair of terminals.
- If $a <\cdot b$, we say that a yields precedence to b.
- If $a \doteq b$, a has same precedence as b.
- If $a \cdot> b$, a takes precedence over b.

Example - Consider the grammar : $E \to E+E \mid E*E \mid id$
Then for the string id + id * id the precedence relation is as follows.

T \ T	id	+	*	$
id	$\doteq$/-	$\cdot>$	$\cdot>$	$\cdot>$
+	$<\cdot$	$\cdot>$	$<\cdot$	$\cdot>$
*	$<\cdot$	$\cdot>$	$\cdot>$	$\cdot>$
$	$<\cdot$	$<\cdot$	$<\cdot$	$\doteq$/-

Consider the following grammar & generate the operator precedence relation for the given string
$E \to E+E \mid E-E \mid E*E \mid E/E \mid E \uparrow E \mid (E) \mid id$
$w \to id * (id \uparrow id) - id/id$

	id	*	-	/	↑	(	)
id							
*							
-							
/							
↑							
(							
)							

Semantic Analysis

Syntax directed Translation

Generally the compilation process is driven by the syntax and semantic routines and performs interpretations based on the syntax structure.

The syntax directed translation refers to a method where a source language is translated to generate a meaningful parse tree from the abstract parse tree.

Syntax directed Definition

This refers to the definition of the attributes of each variable.

In SDD each grammar symbol is associated with a set of attributes that falls in two categories - 1) synthesized attribute
2) Inherited attribute

Synthesized

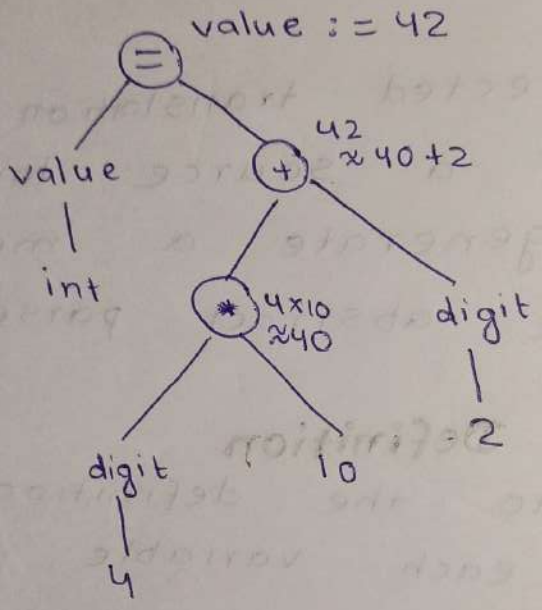
- These attributes are parsed up a parse tree that means the left side attribute of a production is computed to form right side attribute.
- In general the values are computed from the children node for that node.
- The lexical analyzer usually supplies the attributes of the terminals and the synthesized one is built up from the corresponding non-terminal.

Ex - $X \longrightarrow Y_1 Y_2 \ldots Y_n$

$X.a \longleftarrow Y_1.a \; Y_2.a \ldots Y_n.a$

value = digit * 10 + digit

4 2



Module-2

Inherited attribute

- These are the attributes used to passed down a parse tree that is the right hand side attribute are derived from left hand side attribute.
- In other words the child node values are obtained from its parent node.
- These attributes are used for passing the information about the context of a node further down the parse tree.

For example -

$$X \longrightarrow Y_1\ Y_2 \ldots Y_n$$

then $x.a = f(Y_1.a\ Y_2.a \ldots Y_n.a)$

Q Consider the following CFG

$P \longrightarrow DS$

$D \longrightarrow \text{var}\ V ; D \mid \epsilon$

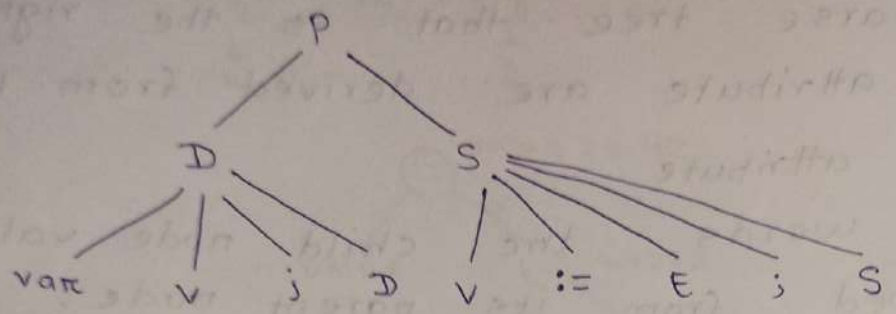
$S \longrightarrow V := E ; S \mid \epsilon$

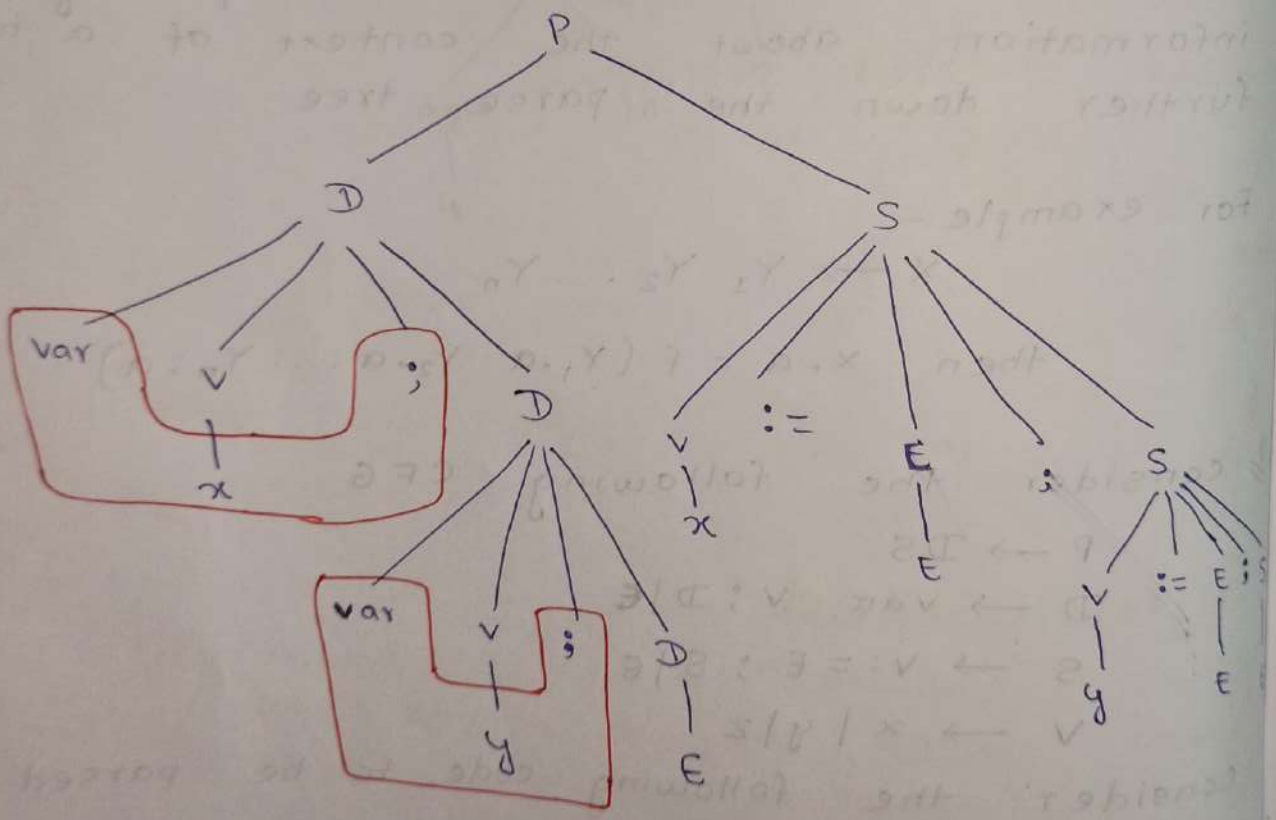
$V \longrightarrow x \mid y \mid z$

Consider the following code to be parsed.

```
var x ;
var y ;
x :=  ;
y :=  ;
```

- We can generate the inherited attributes for the given grammar for the given code.





Synthesized and Inherited Translation

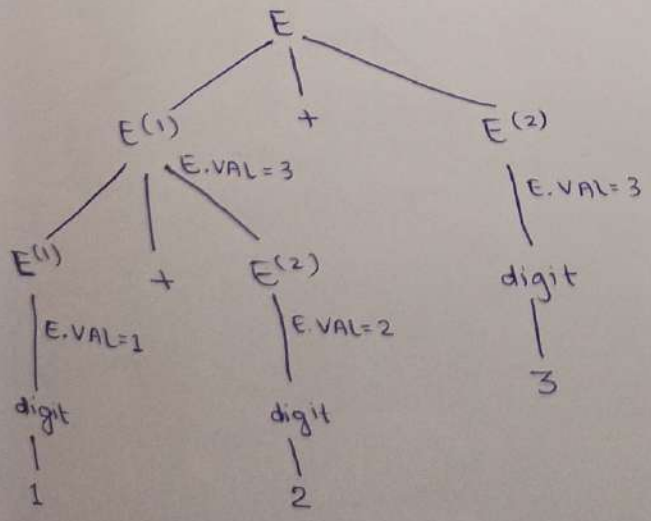
- This process define the value of translation of the non-terminals on the left side of the production as a function of the translation of the non-terminals on the right side.
- This approaches are called as syntax directed translation scheme.

Ex- consider the syntax directed translation scheme & construct the parse tree for the # 1 + 2 + 3.

CFG	Syntax-directed Translation scheme
$E \rightarrow E^{(1)} + E^{(2)}$	$\{E.VAL := E^{(1)}.VAL + E^{(2)}.VAL\}$
$E \rightarrow digit$	$\{E.VAL := digit.VAL\}$

Synthesized Attribute

Inherited attribute



Q Consider the following CFG and construct the corresponding semantic action and equivalent parse tree.

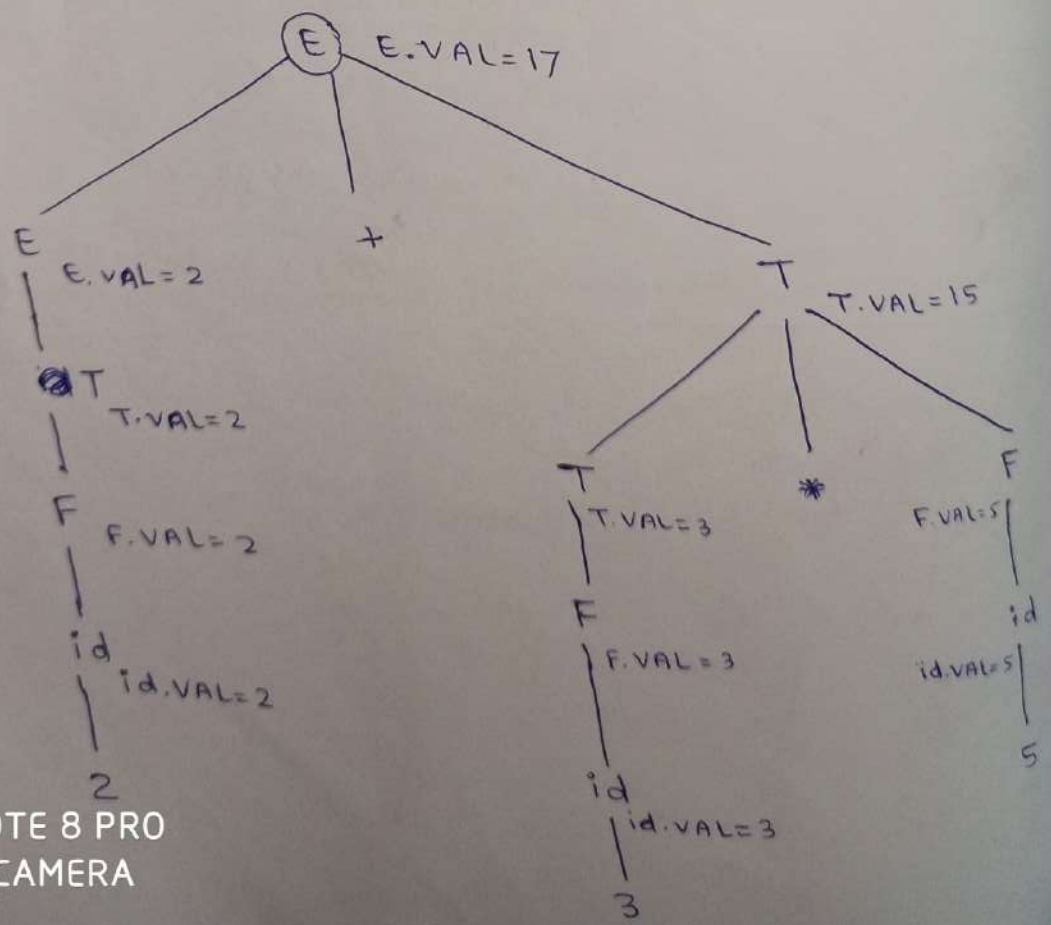
$w = 2 + 3 * 5$

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow id$

CFG	Syntax Directed Translation Scheme
$E \rightarrow E + T$	$E.VAL := E.VAL + T.VAL$
$E \rightarrow T$	$E.VAL := T.VAL$
$T \rightarrow T * F$	$T.VAL := T.VAL * F.VAL$
$T \rightarrow F$	$T.VAL := F.VAL$
$F \rightarrow id$	$F.VAL := id.VAL$



Q Design a syntax directed translation scheme to specify a desk calculator program for the input 23 * 5 + 4 and show the sequence of moves of the input using bottom-up parsing technique.

$S \longrightarrow E\ \$$

$E \longrightarrow E^{(1)} + E^{(2)} \mid E^{(1)} * E^{(2)} \mid (E^{(1)}) \mid I$

$I \longrightarrow I^{(1)}$ digit

$I \longrightarrow$ digit

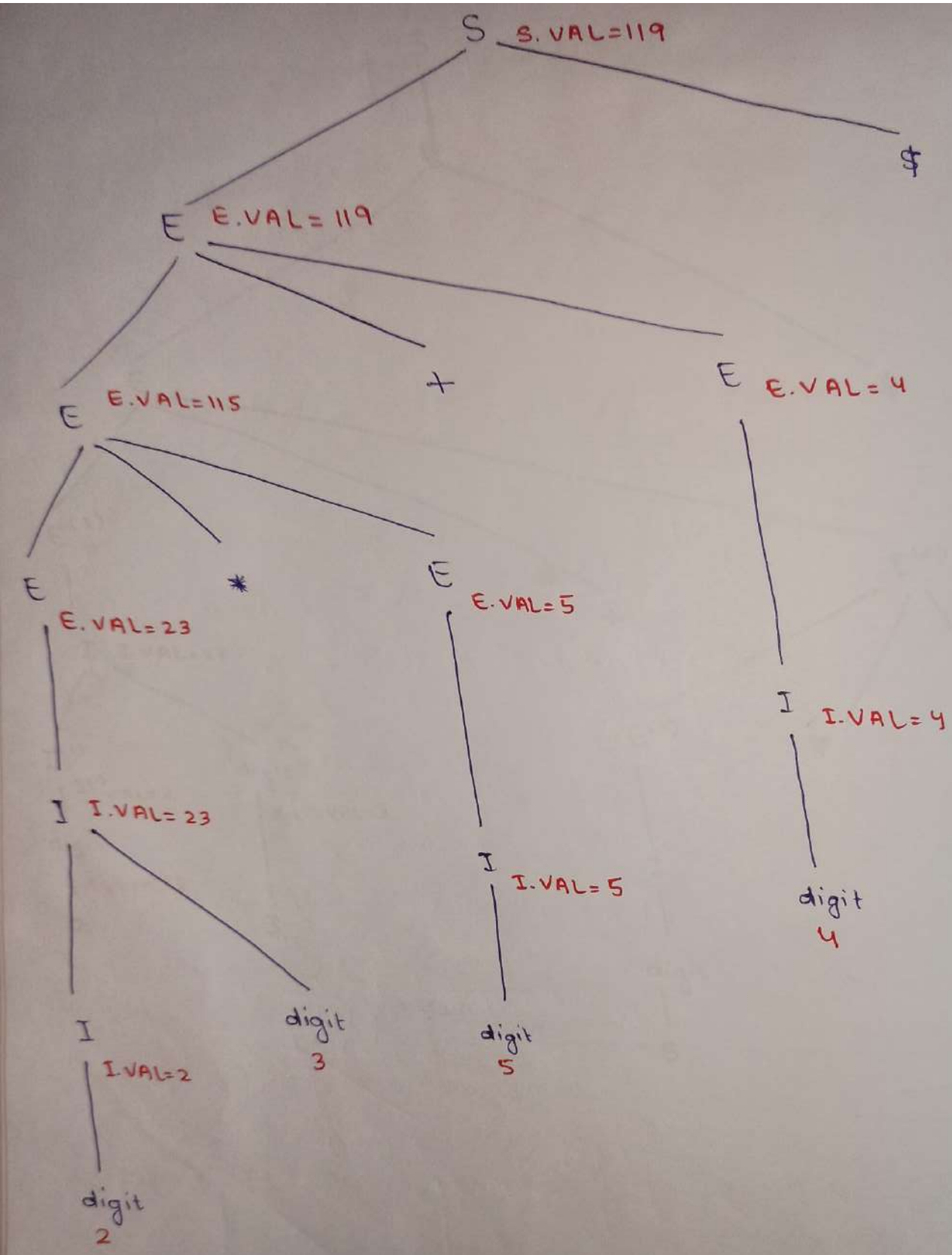
SDTS

trns -

$S \longrightarrow E\ \$$	$\{S.VAL := E.VAL\ \$\}$
$E \longrightarrow E^{(1)} + E^{(2)}$	$\{E.VAL := E^{(1)}.VAL + E^{(2)}.VAL\}$
$E \longrightarrow E^{(1)} * E^{(2)}$	$\{E.VAL := E^{(1)}.VAL * E^{(1)}.VAL\}$
$E \longrightarrow (E^{(1)})$	$\{E.VAL := (E^{(1)}.VAL)\}$
$E \longrightarrow I$	$\{E.VAL := I.VAL\}$
$I \longrightarrow I^{(1)}$ digit	$\{I.VAL := 10 * I^{(1)}VAL + digit\}$
$I \longrightarrow$ digit	$\{I.VAL := digit\}$

Implementation of desk calculator

CFG	Desk calculator
$S \longrightarrow E\ \$$	{print VAL[TOP]}
$E \longrightarrow E^{(1)} + E^{(2)}$	{VAL[TOP] := VAL[TOP] + VAL[TOP-2]}
$E \longrightarrow E^{(1)} * E^{(2)}$	{VAL[TOP] := VAL[TOP] * VAL[TOP-2]}
$E \longrightarrow (E^{(1)})$	{VAL[TOP] := (VAL[TOP-1])}
$E \longrightarrow I$	{VAL[TOP] := VAL[TOP]
$I \longrightarrow I^{(1)}$ digit	{VAL[TOP] := 10 * VAL[TOP] + VAL[TOP-1]
$I \longrightarrow$ digit	{VAL[TOP] := VAL[TOP]}


S
S.VAL=119
$
E
E.VAL=119
E
E.VAL=115
+
E
E.VAL=4
E
*
E
E.VAL=23
E.VAL=5
I
I.VAL=4
I
I.VAL=23
I
I.VAL=5
digit
4
I
digit
3
digit
5
I.VAL=2
digit
2

sequence of moves for the given input w = 23 * 5 + 4 $ is as follows

Input	State	VAL	Production used
23 * 5 + 4 $	- $	-	-
3 * 5 + 4 $	2 $	-	-
* 5 + 4 $	I $	23	I → I digit
* 5 + 4 $	E $	23	E → I
5 + 4 $	E *	23	-
+ 4 $	E * 5	23 5	
+ 4 $	E * I	23 5	I → digit
+ 4 $	E * E	23 5	E → I
+ 4 $	E	115	E → E * E
4 $	E +	115	-
$	E + 4	115	-
$	E + I	115 4	I → digit
$	E + E	115 4	E → I
$	E	119	E → E + E
$	S	119	S → E

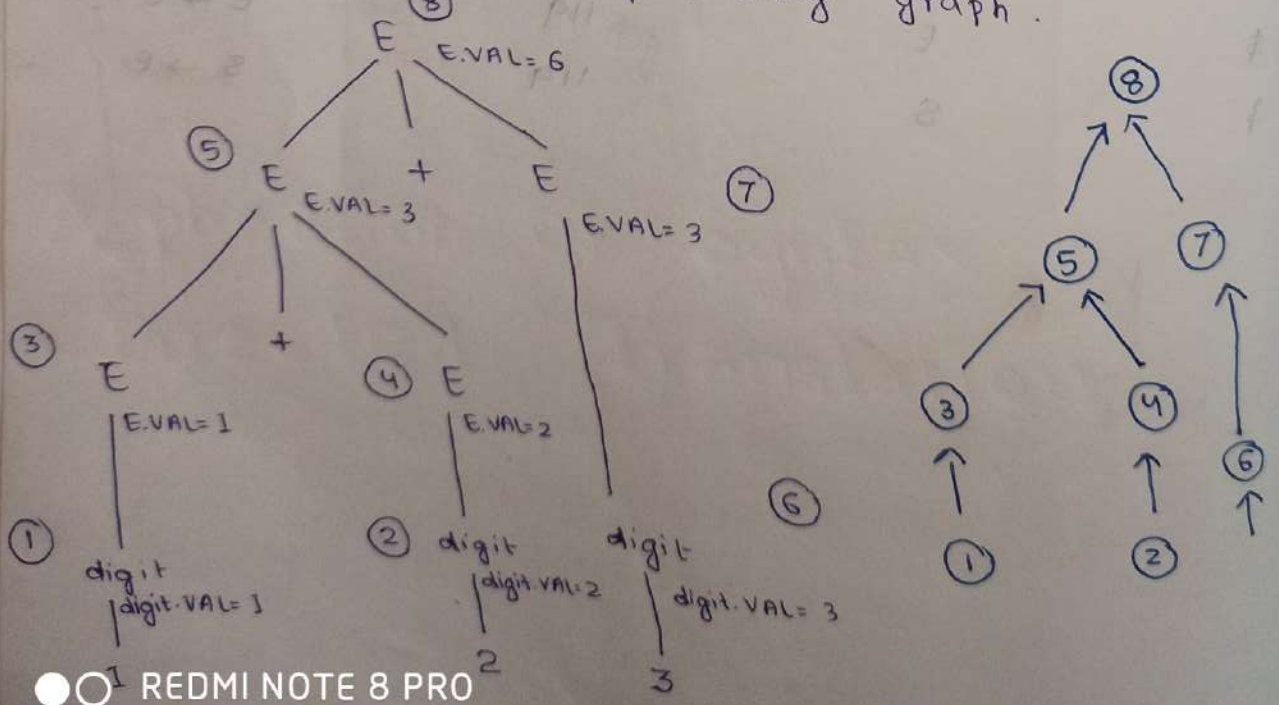
Dependency graph

- The interdependencies between the inherited and synthesized attributes at the nodes in a parse tree can be depicted by a directed graph called a dependency graph.
- That means if an attribute 'b' at a node in a parse tree depends on attribute 'c' then the semantic rule for b at that node can be evaluated after the semantic rule for node c is defined.

For example -

Suppose $A.a := f(x.a, y.a)$ is a semantic rule for production $A \rightarrow x, y$ then it defines that the synthesized attribute A.a depends on its inherited attribute x.a & x.y.

Q Consider the CFG, $E \rightarrow E + E \mid digit$ what's the tree for $w = 1+2+3$ and draw its equivalent dependency graph.

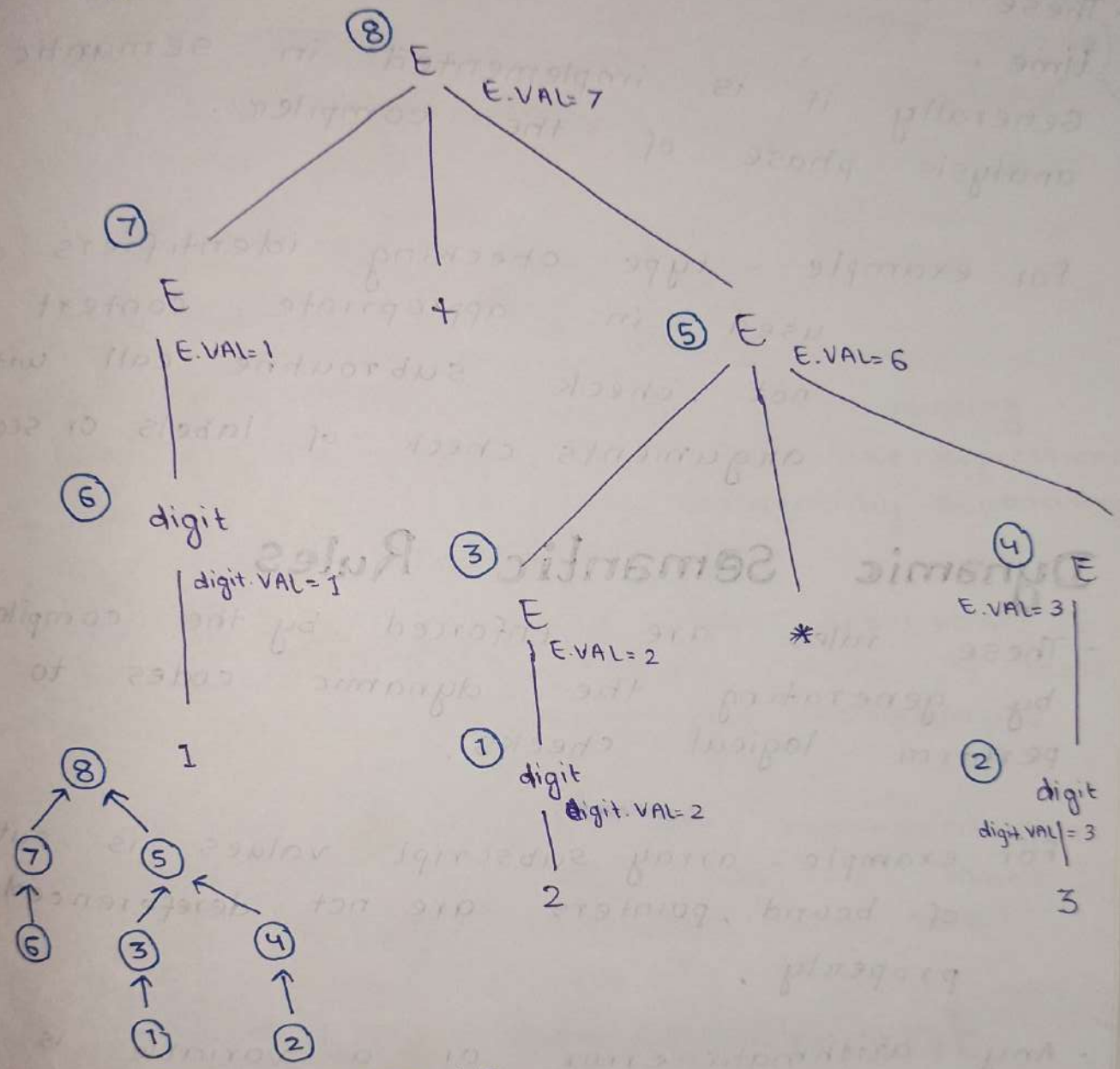


Q $E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow digit$

$\omega = 1 + 2 * 3$



Semantic Rules

- Semantic rules are used to describe the meaning of a program.

It can be categorized into 2 types

1) Static

2) Dynamic

Static Semantic Rules

- These are enforced by a compiler at compile time.
  Generally it is implemented in semantic analysis phase of the compiler.

  For example - type checking identifiers are used in appropriate context or not, check subroutine call with arguments, check of labels or scope etc.

Dynamic Semantic Rules

- These rules are enforced by the compiler by generating the dynamic codes to perform logical checks.

  For example - array subscript values is out of bound, pointers are not dereferenced properly.

- Any arithmatic error or a variable is used but not initialized

Application of Syntax Directed Translation

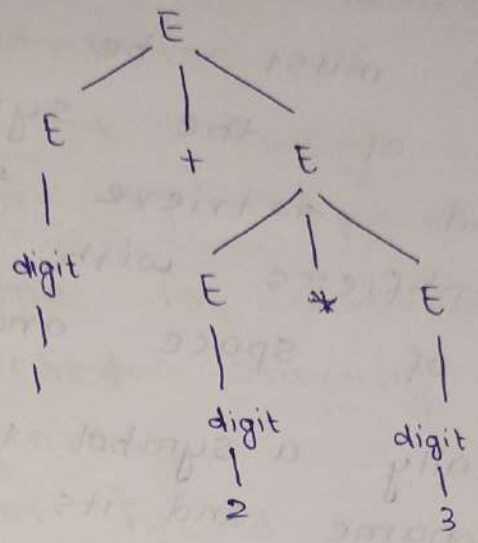
- Generally SDT is used for construction of syntax tree which is a condensed form of parse tree.

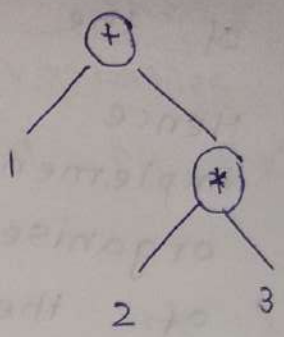
For example - $w = 1 + 2 * 3$

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow \text{digit}$





- Syntax trees are useful for representing programming language constructs like expressions
- They help the compiler design by separating parsing from translation
- Each node of a syntax tree represents a construct & the children node of these construct are its meaningful components.

For ex - a syntax tree node representing an expression $id_1 + id_2$ has a label + at the construct and two children representing id_1, id_2.

Symbol Table Management

- Symbol table is a data structure used by a compiler to keep track of scope and binding information about variables constants, functions as well as labels of the statements.
- Symbol table is searched everytime the variables or methods encounter in the source code. When a new name or information about the existing identifiers is discovered then the content

of the symbol table is updated.

. Hence there must be an efficient implementation of the symol table to organise and retrieve the information of the identifiers with minimal cost in terms of space and time.

- More specifically a symbol table stores

1) Each type name and its type definition

2) Each variable name and its datatype

3) Each array variable name, its data type as well as dimension that is size.

4) If it is a constant then it's name, type and value.

5) For each procedure or function it stores its return type, name, list of formal parameters and their type as well as type of passing the arguments.

Data Structure for Symbol Table

- we can use list, self-organising list, hash table or binary search tree for the symbol table representation

List

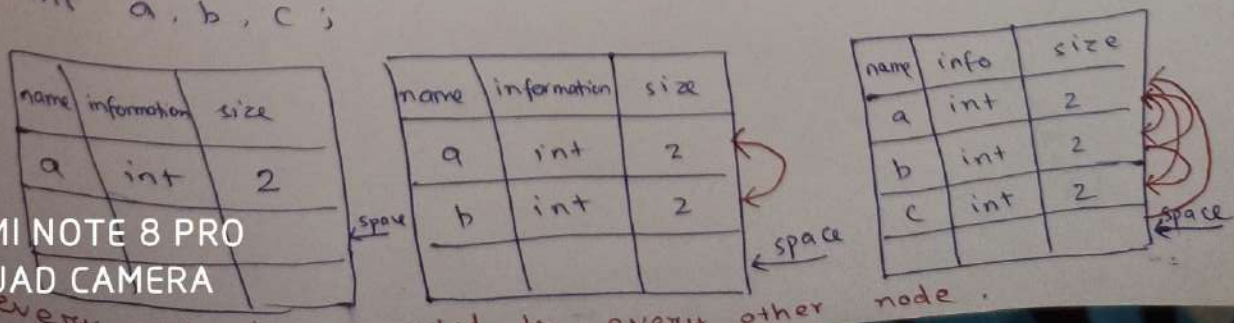
- The simplest and easiest way to implement data str or symbol table is a linear list of records
- A single array or collection of several used for this purpose

- It can store the names and the associated attributes of the variables.
- End of the list always marked by the pointer known as "space".
- when a name is inserted then the list is searched starting from "space" to the beginning
- If it is not found in the list then an entry in the space for the corresponding name is created and the space is incremented by 1 or with the value of data type (with the corresponding bytes)
- But if the name is already present in the list then it returns the index of the name.

Self-organising list

- To reduce the time of searching we can add an additional field called linker to each record or array index.
- When a name is inserted then the space is incremented as well as all the linkers are managed to other existing names.

int a, b, c;



Hash table

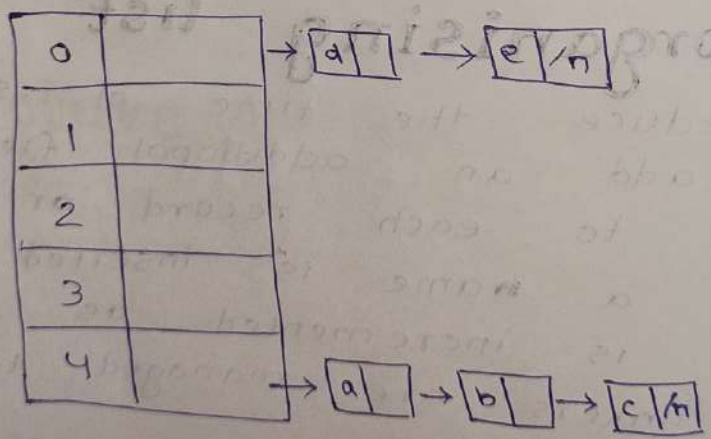
- A hash table or hash map is a data structure that associates key values with each name using open hashing.
- The organisation of hash table reserves the time and space complexity required for list & self organising list respectively.

```
#include <stdio.h>
void main()
{
int a, b, c;
float d, e;

}
}
```

4 % 5 = 4

5 % 5 = 0



Sear

- Anoth
table
that
searc
tree
- All
the
proper
any
it
befor

```
int a
int su
{
a = x
return
}
main()
{
int a;
u = sum
}
```

Consider
search

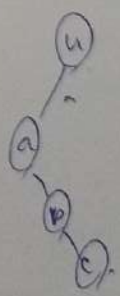
Search Tree

- Another approach to organise the symbol table is that we can add 2 link fields that is left and right child hence the search tree is called as binary search tree.
- All names are created as child of the root node that always follow the property of binary tree that is if any variable created after name, it will be it's right child and before name it will be its left child.

```
int a, b, c;
int sum (int x, int y)
{
a = x + y;
return (a);
}
main()
{
int u;
u = sum (5, 6);
}
```

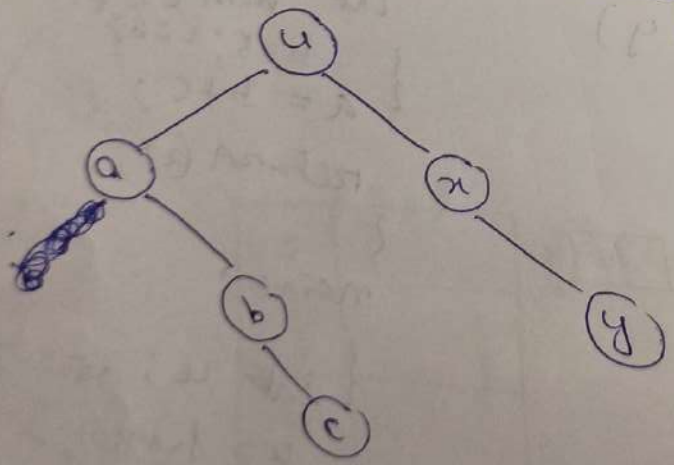
```
int a, b, c;
int fun()
{ b=5; c=6;
a = b + c;
return (a);
}
main()
{ int u;
u = fun();
}
```

Consider the above code and create list, search tree, hash table for it.



Name	Information	size
u	int	2
a	int	2
b	int	2
c	int	2
x	int	2
y	int	2

← space



Intermediate code Generation

While translating a source program into a functionally equivalent object code representation the compiler must first generate an intermediate representation

- Intermediate code is easy to generate and can be easily converted to the target code or object code.
- Following are commonly used intermediate representations.
  - Postfix notation
  - syntax tree
  - Three-address code

1. Post-fix notation

a+b : infix
ab+ : postfix
+ab : prefix

- In this notation the operator follows the operands.

(a+b) * c/d - e

ab+ * c/d - e

ab+c * /d - e

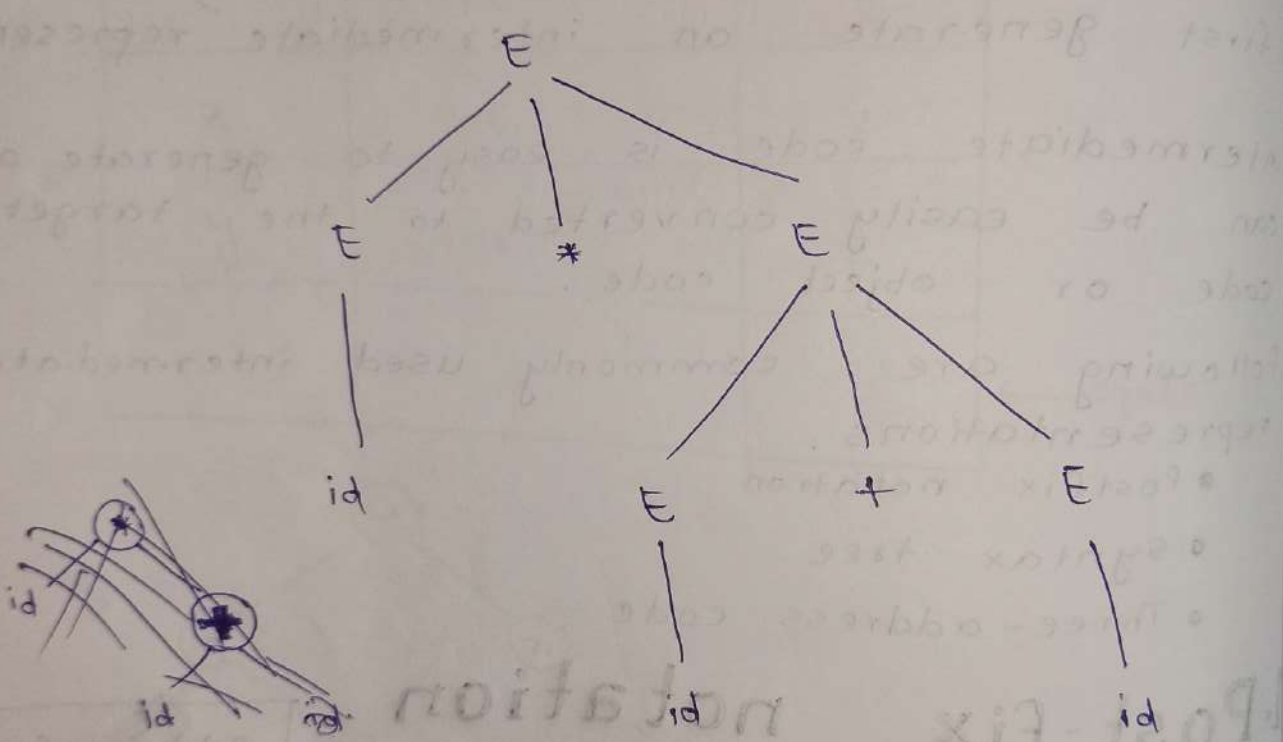
ab+c*d/ - e

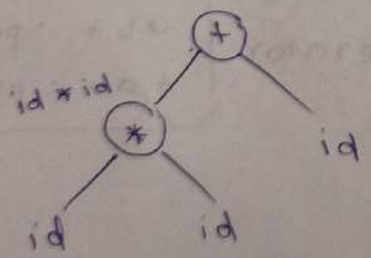
ab+c*d/e - postfix

2. Syntax tree

- The syntax tree is generally known as concrete parse tree or derivation tree.
- Consider the CFG; $E \rightarrow E+E \mid E*E \mid id$

$\omega = id * id + id$





id * id + id

3. Three-address code

- This code is the sequence of statements of the form

$$X := Y \text{ op } Z$$

since a statement in three address code involves not more than three address references hence called a three address statement and the sequence of such statements is called three address code.

- For example -
three address code for the given expression $a + b * c - d$

$$T_1 := b * c$$
$$T_2 := a + T_1$$
$$T_3 := T_2 - d$$

- Sometimes a statement might contain less than 3 references but also called as 3 address statement.

$$c = -a * b$$
$$T_1 := -a$$
$$T_2 := T_1 * b$$
$$c := T_2$$

} optimize

$$T_1 := -a$$
$$c := T_1 * b$$

- The three address statements used to represent various programming language constructs as follows -

The three address statements used to represent various programming language constructs as follows -

- used for representing arithmatic expressions in the form of
  - $x := Y$ op Z (binary operator)
  - $x :=$ op Y (unary)
  - $x := Y$ (assignment)
- used for representing boolean expression
  - if (E_1 relational / logical operator E_2) goto L
- used for representing array references and de-referencing operators
  - $x := Y[i]$
  - $x[i] := Y$

  ] array
  - $x := *Y$
  - $*x := Y$

  ] pointer
- used for representing a procedure call
  - Param T
  - call P, n

Three adress code Representation

- The three address code is represented in any one of the following ways
  1) Quadruple
  2) Triple
  3) Indirect triple

1. Quadruple

- It is the most simplest method to represent the three address code with four fields where the first field holds the operator, second and third field hold the operand 1 and operand 2 respectively and the last field holds the results.

$a + b * c - d$

$T_1 := b * c$

$T_2 := a + T_1$

$T_3 := T_2 - d$

Statement \ Fields	operator	op1 / Arg1	op2 / Arg2	Result
(1)	*	b	c	T_1
(2)	+	a	T_1	T_2
(3)	-	T_2	d	T_3

$X = (a + b) * (-c) / d$

$T_1 := a + b$

$T_2 := -c$

$T_3 := T_1 * T_2$

$T_4 := T_3 / d$

$X = T_4$

Statement \ Fields	operator	op1 / Arg1	op2 / Arg2	Result
(1)	+	a	b	T_1
(2)	-	c		T_2
(3)	*	T_1	T_2	T_3
(4)	/	T_3	d	T_4
(5)	:=	T_4		X

2. Triple

- To avoid the entering temporary names into the symbol table we can allow the statements computing the temporary values to represent that value.
- Hence three address statements are represented by a structure with only 3 fields where the first field contains the operator and the next two fields contain operand 1 and 2 resp

$x = (a+b) * (-c)/d$

① := a + b

② := - c

③ := ① * ②

④ := ③ / d

field / statement	operator	op1 / Arg 1	op2 / Arg 2	
①	+	a	b	
②	-	c		
③	*	①	②	
④	/	③	d	

3. In

- This
- For

(1)
(2)
(3)
(4)
(5) x

3. Indirect triple

- This is another representation of 3 adress code which use an additional array to list the pointers to ~~list~~ the corresponding triples in desired order

- For example - consider the statement

$X = (a+b) * -c/d$

Item no.	Pointer
(1)	(10)
(2)	(20)
(3)	(30)
(4)	(40)
(5)	(50)

fields / Pointer	operator	OP 1	OP 2
(10)	+	a	b
(20)	−	c	
(30)	*	(10)	(20)
(40)	/	(30)	d
(50)	=	X	(40)

(1) := a + b

(2) := − c

(3) := (1) * (2)

(4) := (3) / d

(5) X := (4)

Q. Write the quadruple, triple, indirect triple.

$(x + y) * (y + z) + (x + y + z)$

(1) $T_1 = x + y$

(2) $T_2 = y + z$

(3) $T_3 = T_1 + z$

(4) $T_4 = T_1 * T_2$

(5) $T_5 = T_4 + T_3$

① := x + y

② := y + z

③ := ① + z

④ := ① * ②

⑤ := ④ + ③

field / operator	op 1	op2	Result
①	+	x	y	T_1
②	+	y	z	T_2
③	+	T_1	z	T_3
④	*	T_1	T_2	T_4
⑤	+	T_4	T_3	T_5

field / operator	op 1	op 2
①	+	x	y
②	+	y	z
③	+	①	z
④	*	①	②
⑤	+	④	③

item no	Pointer
(1)	(10)
(2)	(20)
(3)	(30)
(4)	(40)
(5)	(50)

field / Pointer	operator	op 1	op 2
(10)	+	x	y
(20)	+	y	z
(30)	+	(10)	z
(40)	*	(10)	(20)
(50)	+	(40)	(30)

Translation of Boolean operation

- (Translation of Boolean expression) Three adress code be used in generating truth value 0 or 1.
- It has two primary functions
  1) Used as conditional expressions in statements that alter the flow of control.

     Ex - Loops or conditional statements
  2) Used to compute logical values for example using the logical operators and relational operators

For example - Boolean expression in flow of control statement can be represented ~~as~~ in three address code as.

```
goto L
if A goto L
if A relop B goto L
if (A relop B) logop (C relop D) goto L
```

Where A, B, C or D are simple variables or constants, L is the statement no./label to jump. relop is any of

< , > , <= , >= , == , !=

and logop . && , || , ! , ~

AND OR NOT

Q Using the translation scheme translate the expression. A < B to corresponding three address code.

- if (A < B) then 1 else 0

A < B is equivalent to the above conditional statement.

- Three Address code

(1) if (A < B) goto 1+3
4

(2) $T_1 := 0$

(3) goto 5

(4) $T_1 := 1$

(5)

The translation scheme for above 3 address code is as follows -

Production	Semantic Action
$E := id^{(1)}$ relop $id^{(2)}$	{ $T_1 :=$ NEW TEMP ( ) E.PLACE := T GEN (if $id^{(1)}$.PLACE relop $id^{(2)}$.PLACE goto NEXT QUAD +3) GEN ($T_1 := 0$) GEN (goto NEXTQUAD +2) GEN ($T_1 := 1$) }

* To create new temp names we use a function NEW TEMP ( ) which returns the location of new temporary resister created.

GEN( ) is used to generate a value for the corresponding quadruple & for temp. resistors.

* E.PLACE defines the place value that holds the value of the corresponding expression or variable.

Using translation scheme translate the following expression into 3 address code.

(a < b) && (c > d)

- if (a < b) then jump to (c > d)
  if (c > d) then 1 else 0

(1) if (a < b) goto (i+3)
(2) T_1 := 0
(3) goto
(4) if (c > d) goto
(4) T_2 := 0
(5) T_2 := 1

- (i) if (a < b) goto i+4
(i+1) $T_1 = 0$
(i+2) goto i+9
(i+3) $T_1 = 1$
(i+4) if c > d goto i+8
(i+5) $T_2 = 0$
(i+6) goto i+9
(i+7) $T_2 = 1$
(i+8) $T = T_1$ && T_2
(i+9)

Production	Semantic Action
$E := E_1$ logop E_2	{ $T_1 := NEW\ TEMP()$
$E_1 := id^{(1)}$ relop $id^{(2)}$	$T_2 := NEW\ TEMP()$
$E_2 := id^{(1)}$ relop $id^{(2)}$	$E.PLACE := T_1$ && T_2 $E_1.PLACE = T_1$ $E_2.PLACE = T_2$ GEN(if $id^{(1)}$.PLACE Relop $id^{(2)}$.PLACE goto NEXTQUAD +4) GEN($T_1 := 0$) GEN(goto NEXTQUAD +3) GEN($T_1 := 1$) GEN(if $id^{(1)}$.PLACE Relop id.PLACE goto NEXTQUAD +2) GEN($T_2 := 0$) GEN(goto NEXTQUAD +3) GEN($T_2 := 1$) GEN(T_1 && T_2)

Q) if (a < b or c < D) x = Y + Z

- if (a < b) then 1 else jump to (c < d)
  if (c < d) then 1 else 0.
- i) if (a < b) goto i+3
  i+1) if (c < d) goto i+3
  i+2) goto i+5
  i+3) $T_1 := Y + Z$
  i+4) $x := T_1$
  i+5)

TAC
$x = Y + Z$
$T_1 := Y + Z$
$x := T_1$

Q) For while

if (

if (

i) wh

i+1)

i+2)

i) whi

i+1) go

i+2) if

i+3) g

i+4) T

i+5) x

i+6)

Q For the following statement generate TAC

while (A < B) do

if (C < D)

then X = Y + Z

- if (A < B) then jump to (C < D)
  if (C < D) then 1 else 0.
- i) while (A < B) goto ii
  i+1) if (C < D) goto iii
  i+2) T_1 := Y + Z
  X := T_1

```
while (A < B)
{ if (C < D)
  { X = Y + Z;
  }
}
```

i) while (A < B) goto i+2

i+1) goto i+6

i+2) if (C < D) goto i+4

i+3) goto i

i+4) T_1 := Y + Z

i+5) X := T_1

i+6)

Production	Semantic Action
[illegible]	$T_1 := NEWTEMP()$ GEN (while ($id^{(1)}$ relop $id^{(2)}$) goto NEXTQUAD+2 GEN (goto NEXTQUAD+5) GEN (if ($id^{(3)}$ relop $id^{(4)}$) goto NEXTQUAD+2 GEN (goto NEXTQUAD) GEN ($T_1 := Y + Z$) GEN ($X.PLACE := T_1$)

Backpatching

- One problem that arise when we are generating 3-adress code for boolean expression is that we may not have generated the actual quadraples to which jumps are to be made.
- Once we generate the code we fill the corresponding label numbers to which the statement jumps.
- That concept is backpatching.

TAC for control structures and looping

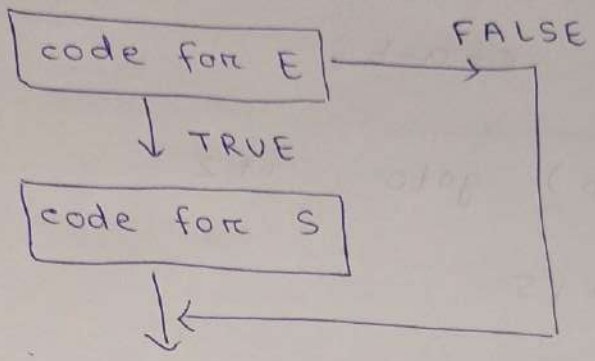
- if
  - Since a simple if or an if-then statement is comprised of two components that is a boolean expression E and a statement S that will be executed when E is true.

Hence
repre

Ex -
if
the
i) if
i+1)
i+2
i+3
i+4

- if - the
- Since
three
E an
if E
execut
- Contro

Hence it's control flow can be can be represented as



Ex -

```
if (a < b)
then  c = a + b ;
```

i) if (a < b) goto i+2

i+1) goto i+4

i+2) $T_1 := a + b$

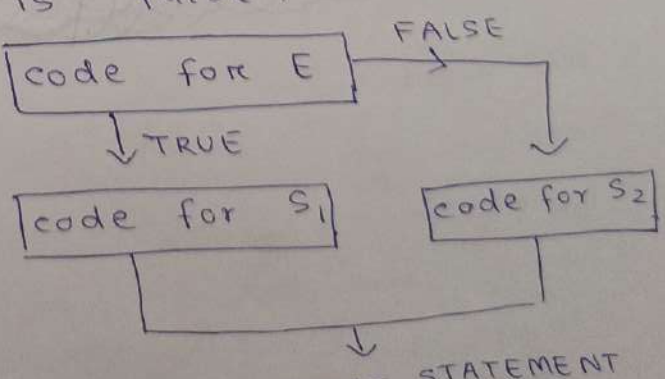
i+3) $c := T_1$

i+4)

- if - then - else

- Since if-then-else statement is composed of three components ie boolean expression E and statement S_1 to be executed if E is true and statement S_2 to be executed if E is false.

- Control flow



Ex - if (a < b)
then c = a+b
else c = a-b ;

i) if (a < b) goto i+2

i+1) goto i+5

i+2) $T_1 := a+b$

i+3) $C := T_1$

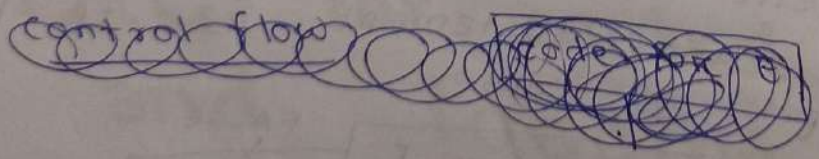
i+4) goto i+7

i+5) $T_2 := a-b$

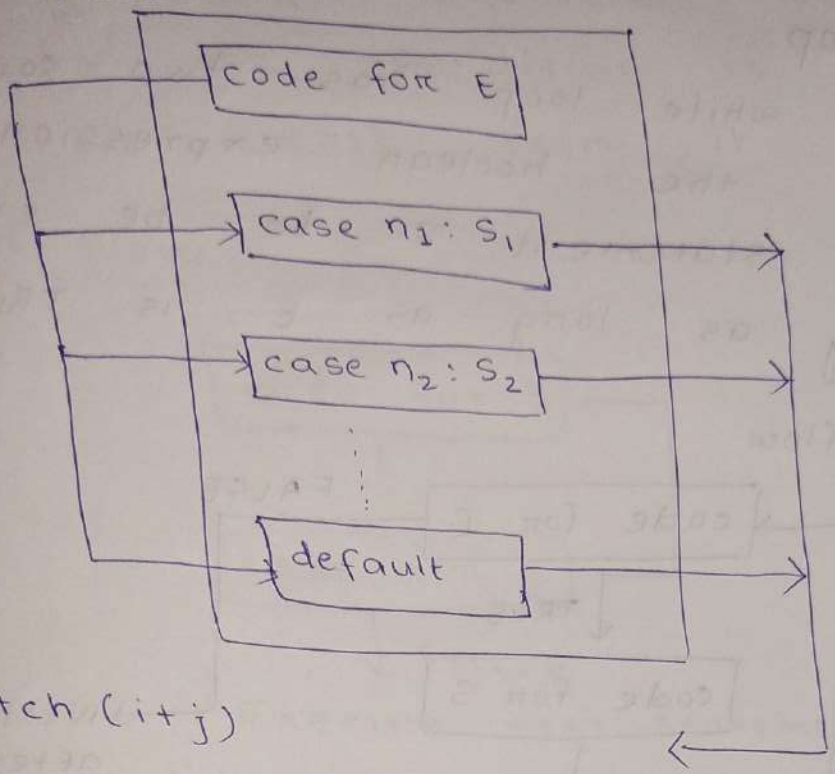
i+6) $C := T_2$

i+7)

- switch
  - A switch statement comprised of two components and an expression E which is used to select a particular 'n' no. of cases including the default case.



control flow




```
ex - switch (i+j)
     {
       case 1 : x = y+z;
                break;
       case 2 : u = v+w;
                break;
       default : p = q+r;
                 break;
     }
```

~~i) switch (i+j) goto~~

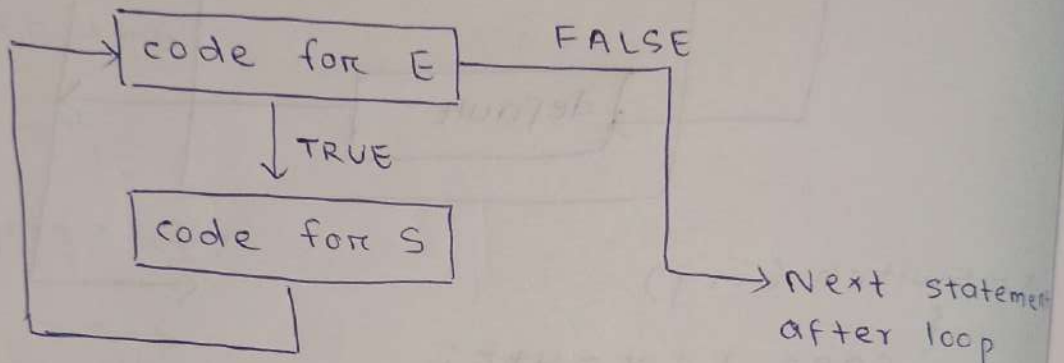
i) T := i + j
i+1) switch (T)
i+2) T_1 := y + z
i+3) x := T_1
i+4) goto i+10
i+5) T_2 := v + w
i+6) u := T_2
i+7) goto i+10
i+8) T_3 := q + r
i+9) P := T_3
i+10)

i) t = i + j
i+1) switch (T)
i+2) if (T == 1) goto i+4
i+3) goto i+7
i+4) T_1 := Y + z
i+5) X := T_1
i+6) goto i+14
i+7) if (T == 2) goto i+9
i+8) goto i+12
i+9) T_2 := V + w
i+10) U := T_2
i+11) goto i+14
i+12) T_3 := q + r
i+13) P := T_3
i+14)

- while loop

- Since a while loop has two components that is the boolean expression E and a statement S to be executed repeatedly as long as E is TRUE

control flow



Ex - Generate the TAC, while (a < b) do c = a + b ;

i) while(a < b) goto i+2

i+1) goto i+5

i+2) $T_1 := a + b$

i+3) $C := T_1$

(+4) goto i

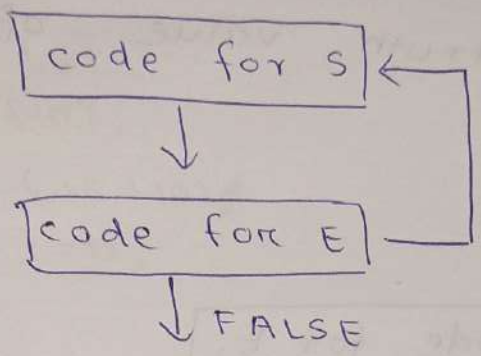
i+5)

- do-while loop

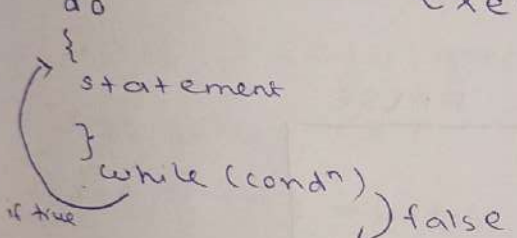
- Since a do-while loop consists of two components, a boolean expression E and statement S which is to be executed repeatedly as long as E is true.

But S will execute atleast once as the condition is checked while we exit from it.

control flow



Execute next statement after loop



Ex - do c = a + b ;
while (a < b)

i) $t_1 = a + b$

(i+1) $c := t_1$

(i+2) while (a < b) goto i

(i+3)

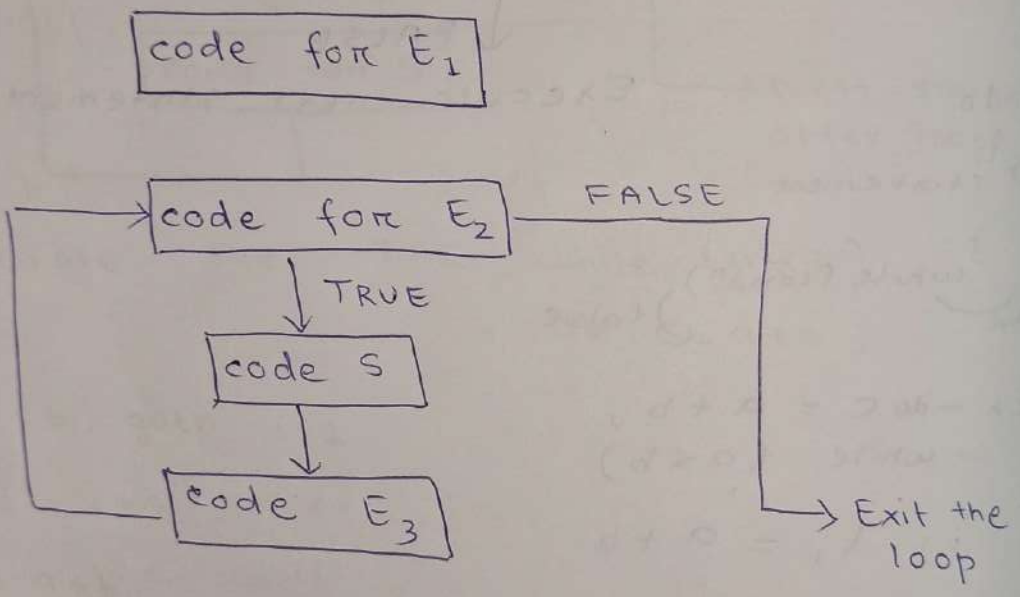
- for loop

for (initialization ; condⁿ ; increment/decrement)
{
}

- A for statement is composed of four components - an expression E_1 used to initialize the iteration variable, an expression E_2 that is an boolean expression that

checks the condition, an expression E_3 that specify the state by which the iteration variable incremented or decremented and the statement S which is to be executed based on the truth value of expression E_2.

control flow



Ex - for (i = 1 ; i <= 10 ; i++)

```
{
    a = b + c ;
}
```

i) i = 1

i+1) if (i <= 10) goto i+3

i+2) goto i+8

i+3) t_1 := b+c

i+4) a := t_1

i+5) t_2 := i+1

i+6) i := t_2

i+7) goto i+1

i+8)

TAC for Array elements

- consider the following code and generate the three address code for it.

```
main ( )
{
int i = 1 ;
int a[10];
while (i<=10)
  a[i] = 0;
}
```

i) i = 1

i+1) if i<=10 goto i+3

i+2) goto i+7

i+3) T_1 := i * width

i+4) T_2 := addr(a)

i+5) $T_2[T_1]$:= 0

i+6) goto i+1

i+7)

TAC for procedure/function

- Generate the three array code for

```
int dot_Prod (int x[] , int Y[ ])
{
int d,i ;
d=0;
for (i=0; i<10; i++)
d= d + x[i] * Y[i];
return d ;
}
main( )
{
int a[10] , b[10] ;
Prod (a,b);
```

TAC for function dot_Prod

```
funcn begin dot_prod
d = 0;
i = 0;
L1: if (i >= 10) goto L2
T1 = addr [x]
T2 = i * width
T3 = T1 [T2]
T4 = addr [Y]
T5 = T4 [T2]
T6 = T3 * T5
T7 = d + T6
d  = T7
T8 = i + 1
i = T8
goto L1
L2: return d
funcn end
```

TAC for main( )

```
funcn begin main
ref param a
ref param b
ref param result
call dot_prod, 3
p = result
funcn end
```

Module-3

Run time Environment

- During execution of the input source program some data objects also called as variables are used in code generation phase.
- These data objects are referred by their addresses.
- The addresses of these data objects are dependent upon the organisation of memory.
- The allocation and deallocation of these data objects are managed by run-time support packages consisting of routines loaded with the generated target code.
- Each execution of a procedure is referred to as an activation of the procedure if the procedure is recursive several of its activation records may be alive at the same time.
- Each call of the procedure leads to an activation that may manipulate the data objects allocated for its use.

Source language issues during execution

- There are various language features that affect the organisation of memory.
- These issues are - 1) recursive procedure
  2) handling the recursive calls

- there may be several instances of recursive procedures that are active simultaneously.
- Memory allocation must be needed to store each instance with its copy of local variables and parameters passed to that recursive procedure but the no of active instances is determined at runtime.

2) method of passing the parameter to function / procedure

- There are two methods for parameter passing ie the call by value and callby reference or callby address.
- The allocation principle for these methods are different based on what are the arguments we are passing.

3) Procedures referring a non-local name

- Each procedure has a access to its local names but a language must supp the method of accessing the non-local that is global variables by the procedur
- The portion of the program to which a declaration applies is called the scope of the declaration.

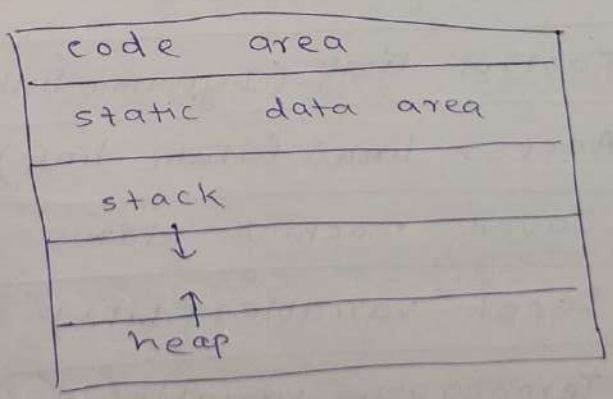
- An occurance of a variable within that scope is called local, otherwise the occurance is said to be non-local.

4) language that support memory allocation and de-allocation dynamically

- Dynamic allocation and de-allocation of memory brings the effective utilization of the memory at run-time.

Storage Organisation

- The compiler for a programming language demands for a block of memory to OS.
- The compiler use this block for execution of the compiled program.
  This block of memory is called runtime storage
- This block is subdivided into following parts.



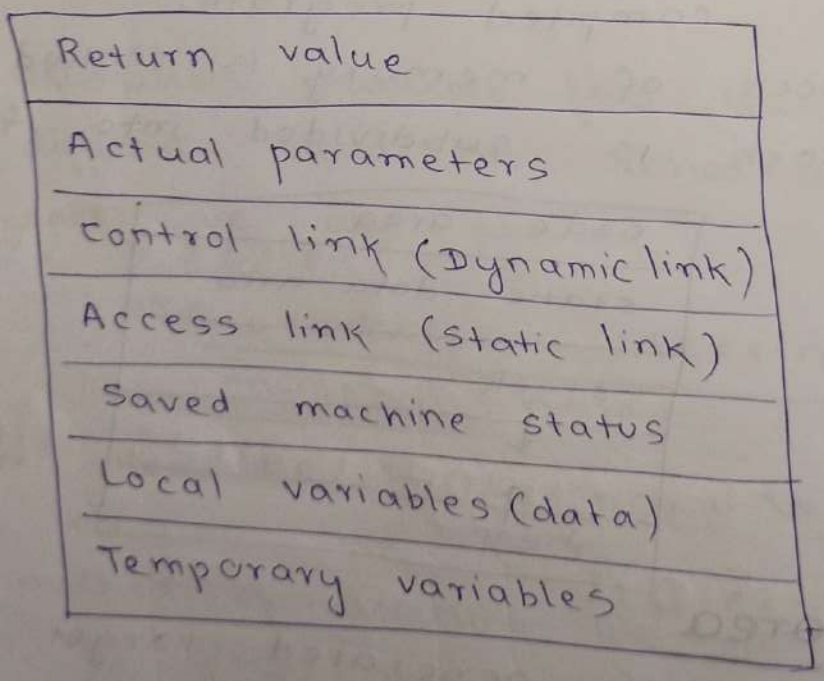
Code area

- The size of generated target code is stored in code area.
- The size of the data object is determined during compile time hence are stored in static data area

but the size of the stack and heap grows and shrinks according to the number of procedures activated during execution.

Activation Record

- Each execution of a procedure is referred to as an activation of the procedure information needed by each execution is managed using a contiguous block of storage called as an activation record or frame.
- The activation record consists of the collection of fields such as



- The purpose of the fields of an activation record is
  i> Temporary variables
  These are used for evaluation of the expressions within the procedure

ii > Local variables

The field for local data holds the data that is declared within the procedure.

iii> Saved machine status

It holds the information about the state of the machine during the procedure is called.

This information includes the values of the program counter and machine registers that have to be restored when the control returns from the procedure.

iv) Access link

The optional field Access link also called static link is used to refer to the non-local data held in other activation records.

v) Control link

The optional field control link points to the activation record of the caller function.

Hence it is a dynamic link.

vi> Actual parameters

The field of actual parameters is used by the calling procedure to supply the parameters to the called procedure.

vii> Return value

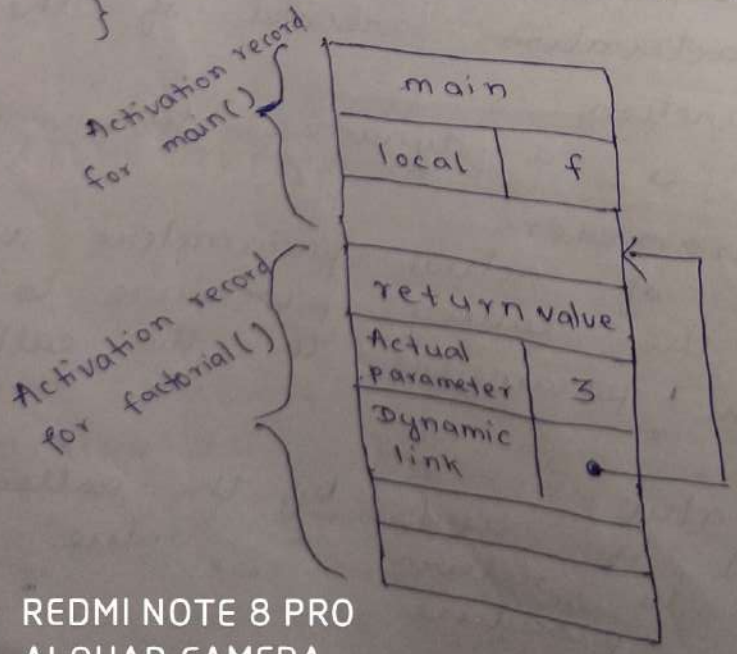
This field is used by the called procedure to return a value to the calling procedure.

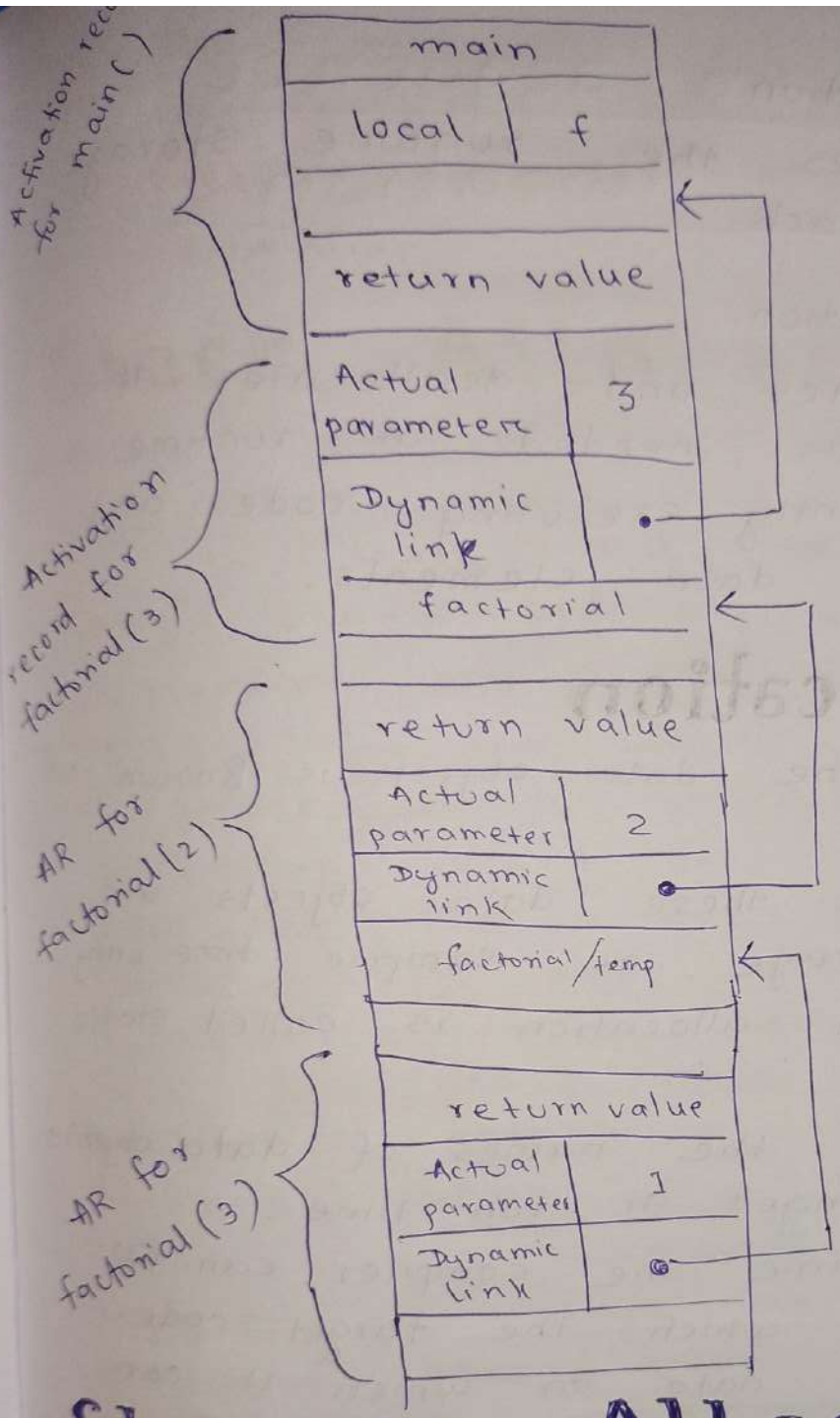
Example

Activation record for each procedure call for the following program segment is represented as follows.

Program segment

```
main ( )
{
int f ;
f = factorial(3);
}
int tutorial (int n)
{
 if (n==I)
  return I;
 else
 return (n * factorial (n-I));
}
```





Storage Allocation Strategy

Different storage allocation strategy is used in each of the three data areas.

① Static Allocation (layout the storage for all instructions and data objects)

② Stack allocation
It manages the runtime storage as a stack.

③ Heap allocation
It allocates and deallocates the storage as needed at runtime for currently executing code and respective data elements.

Static Allocation

- The size of the data object is known at compiler time.
- The names of these data objects are bound to storage at compile time only. Hence, such an allocation is called static allocation.
- The binding of the names of data objects can't be changed at run time.
- At compile time the compiler can fill the addressess which the target code can find the data on which it can operate.

Limitation

- The size of the data object and its address location is fixed at compiler time.
- Recursive procedures are not supported by this type of allocation because all activation of procedure has to use same bindings for the local names.

- The data structures can't be created dynamically since there is no mechanism for storage allocation at run time.

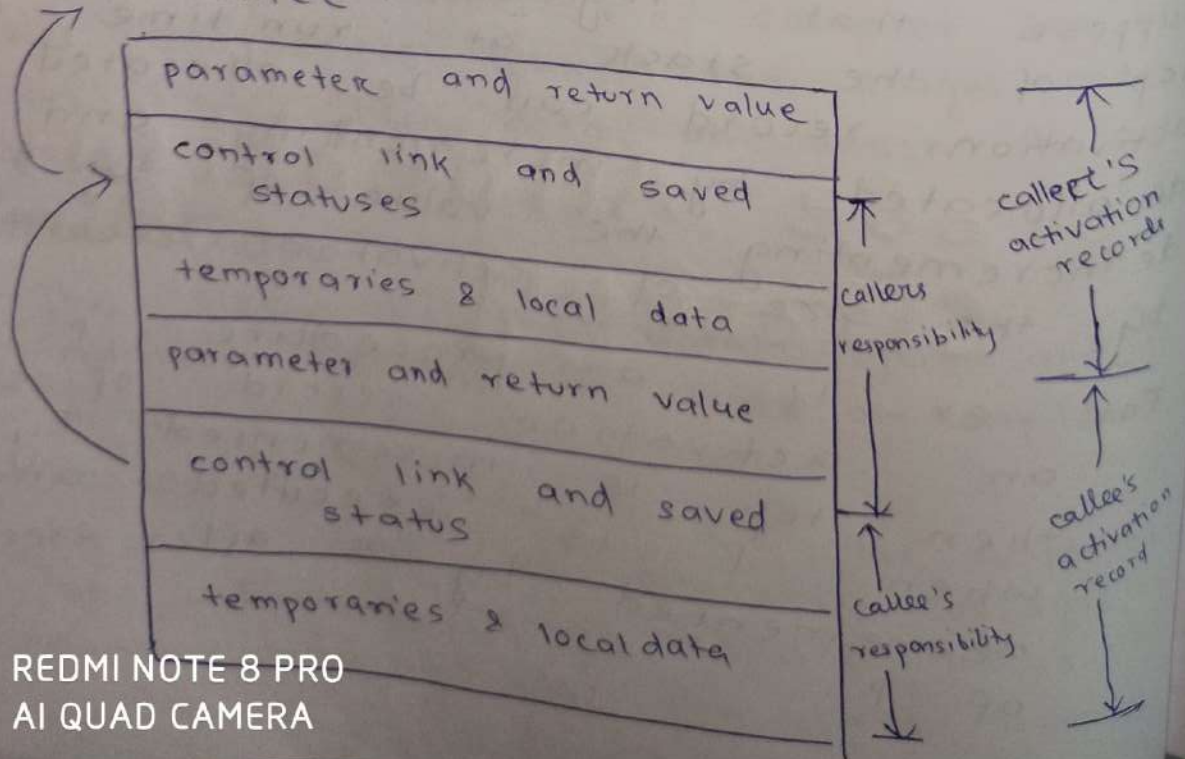
Stack Allocation

- Stack allocation based on implementation of control stack and storage is organised as a stack and the activation record are pushed and popped when the activation begins and ends respectively.
- Storage for the locals in each call of a procedure is content in the activation record for that call.
- Thus, the locals are bound to the fresh storage in each activation because a new activation record is pushed onto the stack when the call is made.
- Also the values of locals are deleted when the activation ends.
- Suppose that register "top" marks the top of the stack at run time an activation record can be allocated and de-allocated by incrementing and de-incrementing the value of top resp. by the size of activation record.

For ex- If a procedure 'p' has an activation record of size 's' then top is incremented by s and when p is executed and after execution decremented by s.

Call and return sequence for stack Allocation

- Procedure calls are implemented by generating calling sequences in the target code
- All call sequence allocates an activation record and enters the information related to the currently executing procedure into its resp. fields.
- A return sequence returns to the calling procedure based on saved machine status so that the calling procedure can continue its execution.
- The code in a calling sequence is often divided between the calling procedure (caller) and the procedure that is called - callee.



Heap Allocation

- If the values of the non-local variables required to be ~~wr~~ retained even after the activation record is deallocated then such a retaining is not possible by stack allocation.
- This limitation of stack allocation is because of its last in first out nature.
- So, To retain such variables we require heap allocation
- The efficient heap management can be done by 1) creating a linked list for the free blocks when ~~creat~~ any memory is deallocated. Those freed blocks are appended in the linked list.

2- Allocate the most suitable block of memory from the linked list using best fit technique for allocation of a block.

Access to non-local names

- A procedure may sometimes refer to the variables which are not local to it.
- Such variables are called non-local variables having two types of access scope
  1. Static scope rule
  2. Dynamic scope rule

① Static Scope rule (also called lexical scope)

- In this type the scope is determined by examining the source code.
- The languages like PASCAL, C uses the static scope rule hence called block structure language.

② Dynamic Scope rule

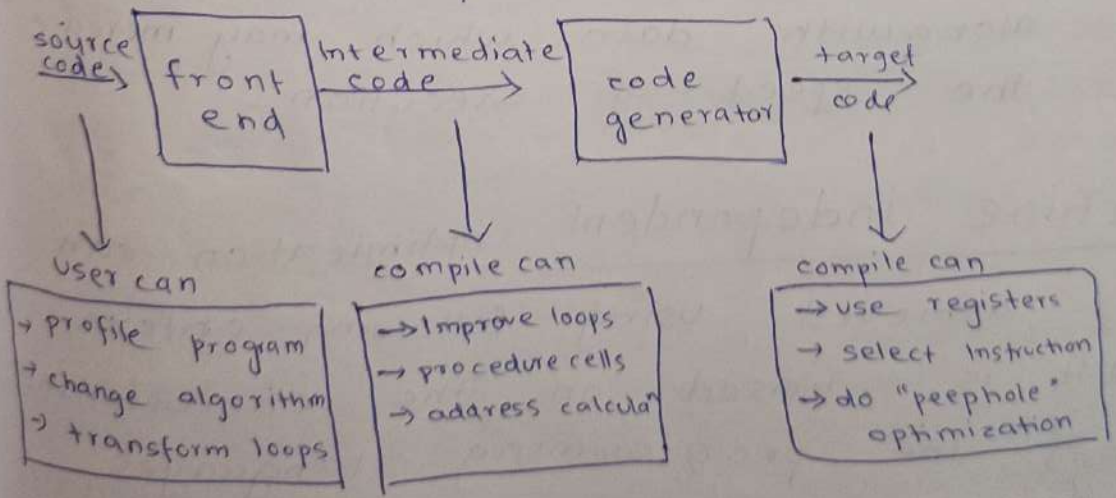
- This determines the scope of declaration of names at run time by considering the current activation.

Code Optimization

- Code optimization is required to produce an efficient target code.
- There are two factors to be considered while optimising the generated code.
  1) Semantic equivalence of the source program must not be changed.
  2) The improvement over the program efficiency must be achieved without changing the algorithm of the program.

* The compilers that apply code improvement transformation are called optimizing compilers.

- The potential improvement by the user and the compiler is done by the compilation process as follows



Classification of Optimization

- The classification can be done in two categories ① Machine dependent ② Machine independent

- Machine dependent optimization is based on the characteristics of the target machine for which the instruction set are used and the equivalent addressing modes used for those instructions to produce the efficient target code.
- This optimization can be acheived using following criteria.
  ① Allocation of sufficient numbers of resources to improve the execution efficiency of the program
  ② Using immediate instructions wherever necessary
  ③ The use of intermix instructions alongwith data which may increase the speed of execution.

- Machine independent optimization can be achieved using following criteria
  ① It is based on the characteristics of the programming languages for appropriate programming structure and uses of efficient arithmatic properties in order to

reduce the execution time

(2) The code should be analyzed completely and used alternative equivalent sequence of source code that can be useful to produce a minimum amount of target code.

(3) Move two or more identical computations at one place and make use of the resultant value instead of each time computing the expression.

(4) From the source code eliminate the unreachable code

Principal Sources of Optimization

- The optimization can be done locally or globally if the transformation is applied on the same basic block then that kind of transformation is done locally otherwise the transformation is done globally.
- Generally the local transformations are done first.
- Following are the techniques of optimization

(1) Compile time evaluation

- This means shifting of computations from runtime to compile time.

- There are two methods to obtain compile time evaluation

  → folding
  → constant propagation

Folding

In this technique the computation of the constants is done at compile time instead of at execution time.

for ex - If the instruction is

$$a = (22/7) * r * r$$

then by using the folding technique we can replace 22/7 with value 3.141

Constant Propagation

In this technique the values of the variable are replaced and computation of an expression is done at compile time.

Ex - { Pi = 3.141 ; r = 5 ; a = Pi * r * r } (1)

{ a = 3.141 * 5 * 5 } 2

Here at compile time the value of Pi is replaced by and r=5 so that the computation is done during

(ii) Common Sub-expression elimination

- This is the technique to remove an expression appearing repeatedly in a code which is computed previously.
- Then if the operants of this sub-expression don't get changed at all the result of such sub expression is used instead of recomputing it each time.

- For ex -

$t_1 = 4 * i$

$t_2 = a[t_1]$

$t_3 = 4 * i$

$t_4 = b[t_3]$

$t_5 = t_4 + n$

Here $t_3 = 4 * i$ is eliminated as it is already computed in t_1.

After eliminating common subexpression

$t_1 = 4 * i$

$t_2 = a[t_1]$

$t_3 = b[t_1]$

$t_4 = t_3 + n$

(iii) Variable propagation

- It means to use one variable instead of another.

for example - if $x = pi$

$a = x * r * r$.

we can directly write, $a = pi * r * r$

the optimization can be done by eliminating variable x by the variable pi.

(iv) Code movement

- There are two basic goals of code movement, (1) To reduce the size of the code that is to obtain space complexity (2) To reduce the frequency of execution of the code that is to obtain time complexity.

Ex -

```
for (i=0; i<=10; i++)
 {
    x = y+5;
    k = (y+5)+50;
 }
```

Optimizing :

```
x = y*5;
for (i=0; i<=10; i++)
 {
    k = x+50;
 }
```

The code for y*5 will be generated only once and can be used in the computation whenever necessary.

(v) Loop invariant computation or code motion

- This method can be obtained by moving some amount of code outside the loop and placing it just before entering in the loop

- This method is also called as code motion.

```
while (i<=max-1)
    sum = sum + a[i];
```

Optimized code:

```
a = max - 1
while (i<=n)
    sum = sum + a[i]
```

This above code is optimized by removing the repeated computation of max-1 by removing it outside the loop

(vi) Dead code elimination

- A code is said to be live if it is used subsequently during execution of the program.
- But if a code is unreachable ie never been used in execution is called dead code and can be eliminated to optimize the code.

```
i = 0;
if (i == 1)
    a = a+5;
```

Here if statement is a dead code as the condition never be satisfied. Hence if block can be removed from the program.

Loop Optimization

or Flow of Control optimization

- The code optimization can be done in loops of the program specially where the programs spend large amount of time.
- Hence, if number of instructions are less in a loop then the running time of the program decreases to a large extent.
- The loop optimization can be carried out using following methods
  1) Code motion
  2) Induction variable and strength reduction
  3) Loop invariant method
  4) Loop unrolling
  5) Loop fusion

1) Code motion

- Code motion is a technique which mo the code outside the loop.
- If there are some expressions in th loop whose result remains unchanged even after executing the loop severa times then such expression should be placed outside the loop.

```
while (x <= max - 1)
    sum = sum + a[x]
```

⇓

```
n = max - 1
while (x <= n)
sum = sum + a[x];
```

2) Induction variable and strength reduction

- A variable i is called induction variable of loop L if the value of the variable gets changed everytime.
- It is either decremented or incremented by some value.
- Before optimizing we have to identify those induction variables and keep them as they are in the loop.

- Reduction in strength states that we have to remove certain operators having higher strength with the operators having lower strength.

```
for ( i = 1 ; i <= 50 ; i++ )
 count = i * 5 ;
```

```
temp = 5 ;
for ( i = 1 ; i <= 50 ; i++ )
{
    count = temp ;
    temp = temp + 5 ;
}
```

- The replacement of multiplication by addition with speed up the object code execution without changing the meaning of the code.

3>

- In

4>

- In

and

writing

For

3) Loop invariant method

- In this optimization technique the computation inside the loop is avoided and thereby the computation overhead by the compiler is reduced.

```
for (i = 0; i < 10; i++)
sum = sum + a/b
```

⇓ after optimisation

```
temp = a/b
for (i = 0; i < 10; i++)
sum = sum + temp;
```

4) Loop unrolling

- In this method the number of jumps and tests can be reduced by writing the code two times.

For example -

```
int i = 1;
while (i <= 100)
{ a[i] = b[i];
  i++;
}
```

⇓ optimisation

```
int i = 1;
while (i <= 50)
{ a[i] = b[i];
  i++;
  a[i] = b[i];
} i++;
```

5> Loop fusion

- In loop fusion method several loops are merged to form one loop.

```
for (i=1; i<n ; i++)
for (i=1 ; i<m; i++)
  a[i] = 10;
```

⇓ optimisation

```
for (i=1; i<n*m ; i++)
  a[i] = 10;
```

BASIC BLOCKS AND FLOW GRAPHS

Basic blocks

- A basic block is a sequence of consequative statements in which flow of control enters at the beginning and leaves at the end without any halt or possibility of branching except at the end.

Terminologies of basic block

i) Define and use

ii) Live

i) Define and use

- A three address code is in form $X := Y + Z$ is said to be define X and use Y and Z.

ii) Live

- A name or a variable in a basic block is said to be live at a given point if its value is used after that point in the program or maybe in another basic block.
- Ex - Consider the following code segment

```
prod = 0;
i = 1;
do
{
prod = prod + a[i] * b[i]
} while (i<=20);
```

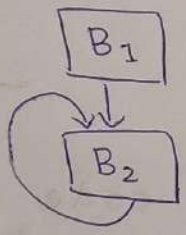
```
prod = 0;
i = 1;
do
{
  prod = prod + a[i] * b[i];
  i = i + 1;
}
while (i<=20);
```

```
1. prod := 0
2. i := 1
3. t1 := i * width
4. t2 := a[t1]
5. t3 := b[t1]
6. t4 := t2 * t3
7. t5 := prod + t4
8. prod := t5
9. t6 := i + 1
10. i := t6
11. if (i<=20) goto 3
```

B_1

B_2

Flowgraph

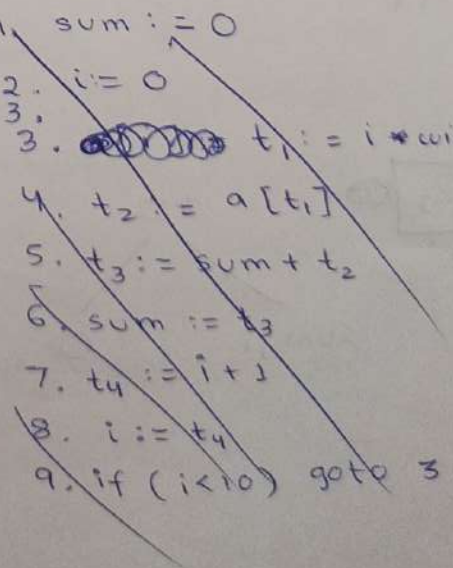


- The flowgraph is a directed graph that represents flow of control information in the set of basic blocks.
- The node of a flow graph are the different basic blocks from the three adress code.

DAG representation of a basic block

- Directed Acyclic graph are useful data structures for implementing transformation on basic blocks.
- It gives a picture of how the values are computed by each statement in a basic block and used in the subsequent statements of that block.
- DAG for a basic block is a graph with the following labels on its nodes
  1) The leaves are labelled by unique identifiers, either the variables or constants. generally the leaves represent r values (that is the right side value of assignment)
  2) The interior nodes are labelled by an operator symbol.

```
sum = 0;
for (i=0; i<10; i++)
sum = sum + a[i];
```

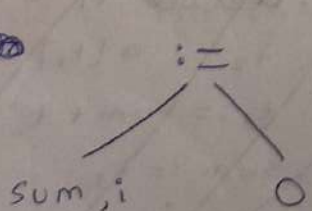


```
sum = 0;
for (i = 0; i < 10; i++)
    sum = sum + a[i]
```

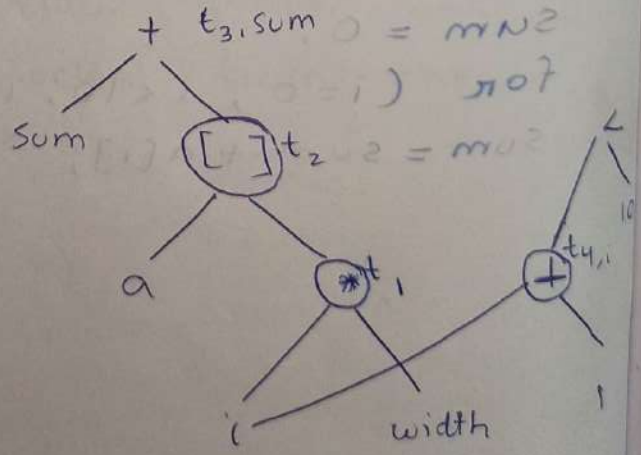
TAC

1. sum = 0;
2. i := 0;
3. if (i < 10) goto 5
4. goto 11
5. t_1 := i * width
6. t_2 := a[t_1]
7. t_3 := sum + t_2
8. sum := t_3
9. t_4 := i + 1
10. i := t_4
11.

B_1



B_2



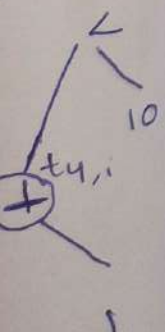
Application of DAG

- Determining the common sub-expressions
- Determining which variables are used inside a block and computed outside a block.
- Determining which statements of the block could have their computed values outside the block.

Peephole Optimization

- A peephole optimization is a simple and effective method for trying to improve the performance of the target program by examining a short sequence of target instructions also called as peepholes and replacing these instructions by a shorter and faster sequence whenever possible.
- The peephole optimization can be done using following characteristics

1) Redundant Instruction Elimination
2) Elimination of unreachable code
3) Algebric simplification
4) Use of machine idioms
5) Flow of control optimisation



1> Redundant Instruction Elimination

- Generally the redundant loads and stores can be eliminated in this type of transformation.

For ex – MOV R_0, X
STA X, R_0

The second instruction can be eliminated because value of R_0 is already in X.
Provided that R_0 is not modified in between the load & store instructions.

2> Unreachable code

- Another opportunity for peephole optimization is removable unreachable instructions.

```
# define [redacted] i 0

unreachable code
if ( i == 1 )
{
  printf ("%d", i);
}
```

3) Algebric Simplification

- Peephole optimization is an effective technique for algebric simplification.

For ex - $x = x + 0;$

$x = x * 1;$

- If the code contains additive and multiplicative identity then it can be eliminated.

- Also we can use reduction in strength for algebric simplification.

for ex - If we have the expression with multiplication

$x = x * 5$, then it can be removed by repeat additions.

- If we have expressions with power

$x^2 \approx x * 5$

then can be removed by repeat multiplication.

4) Machine idioms

- The target machine may have certain inbuilt instructions to implement some specific operations efficiently, we can allow these

instructions to reduce the execution time significantly by implementing them through peep hole.

For ex- some machines have auto-decrement & auto-increment adressing modes. These modes add or subtract 1 from an operand before or after using its value.

- The use of these modes can greatly improve the quality of the code by replacing the statements like i++, i=i+1, i-- etc.

5) Control flow optimization

- The intermediate code generation may produce jumps to conditional jumps or conditional jumps to jumps
- These unnecessary jumps can be eliminated in either the intermediate code or the generated target code by the following types of peephole optimization.

goto L_1

.....

L_1 : goto L_2

replaced by →

goto L_2

For example - The above code can be replaced.

goto L_1

.....

.....

L_1: if a < b goto L_2

L_3 : ....

replaced →

if a < b goto L_2

L_3 .....

CODE GENERATIO

- This is the final activity of compiler
- Basically it is a process of creating assembly language or machine language instruction which will perform the operations specified by the source program when they run.

Properties of a code generation phase are

- correctness
  - It should produce a correct code and do not alter the purpose of the source code.
- high quality
  - It should produce a high quality object code in terms of space and time complexity
- Efficient use of resources of the target machine
  - while generating the object code it's necessary to know the target machine on which the code is to be executed, that may efficiently use the available resources of the target machine.

For ex- efficient memory utilisation while allocating the registers or efficient utilisation of ALU while

performing certain operations.

- Quick code generation
  - This is desired feauture of code generation phase which should produce the code quickly after compiling the source program.

Issues In code gen

- common Issues during code generation are - 1) Input to the code generator
  - The input to the code generator consists of the intermediate code representation of the source program in a optimized form.
  - It works together with the information of the symbol table that is used to determine the runtime addresses of the data objects denoted by the names in the intermediate representation.

2) Target program
The output of the code generator is the target code which may be in the following three forms
- Absolute machine language
- relocatable machine code
- Assembly code

- The advantage of producing the target code in absolute machine language form is that it can be placed directly at the fixed memory location and can be executed immediately.

- The advantage of producing the relocatable machine code as the output is that the sub-routines can be compiled separately.

  Relocatable object modules can be linked together and loaded for execution by the linker and loader.

- The advantage of producing the assembly language code as the output makes the code generation easier because the symbolic instruction and macro facilities of assembler can be used to generate the code.

3) Memory management

- Both the front-end and code-generator performs the task of mapping the names in the source program to

the corresponding addresses of the data objects in runtime memory.

- The names used in TAC refers to the entries in the symbol table that contains the data type and the size of the variable required in memory.

4) Instruction selection

- The uniformity and completeness of instruction set is an important factor for code generator.
- The selection of instructions depends upon the instruction set of a target machine.
  The speed of instruction and machine idioms are two imp. factors in selection of instructions

5) Use of registers

- Instructions involving register operands are usually shorter and faster than those involving operands in memory.
- Therefore efficient utilisation of resistors is an important factor in generating good code
- The use of resistors is subdivided into two sub problems
  1) Resistor allocation
  2) Resistor assignment

1) Register Allocation

- During the register allocation select appropriate set of variables that will reside in the registers.

2)- During the register assignment pick up the specific register in which corresponding variable will reside.

Approaches to code Generation

- The most important criteria for code generator is that it can produce correct code.
- The code generation phase use the information about subsequent uses of an operant to generate the code for target machine.
- It considers each statement and keeps the operands in the registers as long as possible.

Gen

consid

d :

TAC

1. T
2. T
3. T
4. -
5. d

Now
abov

Statement
$t_1 := a - b$
$t_2 := a - c$
$t_3 := t_1 +$
$d := t_3 + t_2$

Generation of code

consider the following statement.

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

TAC

1. $T_1 := a-b$
2. $T_2 := a-c$
3. $T_3 := T_1 + T_2$
4. $T_4 := T_3 + T_2$
5. $d := T_4$

optimimised code

1. $T_1 := a-b$
2. $T_2 := a-c$
3. $T_3 := T_1 + T_2$
4. $d := T_3 + T_2$

Now the generated code of the above TAC can be represented as

Statements	code generator	Location	Register Assignment	Register Allocation
			All registers are empty	
$t_1 := a-b$	mov a, R_0 sub b, R_0	R_0	R_0 containing t_1	t_1 t_1 in R_0
$t_2 := a-c$	mov a, R_1 sub c, R_1	R_1	R_0 containing t_1 R_1 containing t_2	t_1 in R_0 t_2 in R_1
$t_3 := t_1 + t_2$	ADD R_0, R_1	R_0	R_0 containing t_3 R_1 contains t_2	t_3 in R_0 t_2 in R_1
$d := t_3 + t_2$	ADD R_0, R_1	R_0	R_0 contains d	d in R_0 and memory