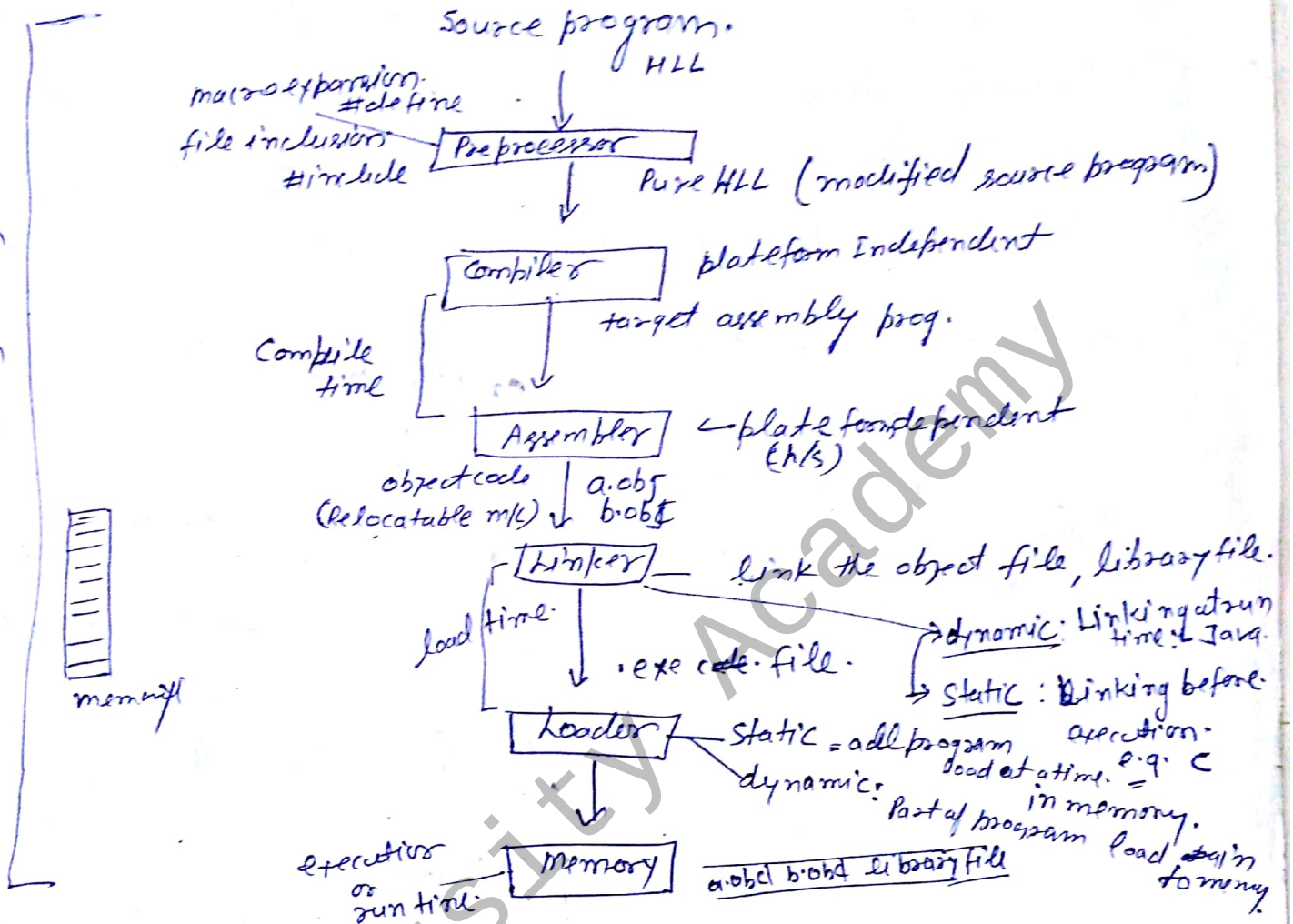
Compiler design.

Language processing System:



Source program

e.g

```
#include<stdio.h>
main()
{
int a,b;
int c;
c= add(a,b);
printf("%d",c);
}
```

a.c

```
int add(int x, int y)
{
 int z
 z = x+y;
 return z;
}
```

b.c

We know that computer understand only machine code i.e. 0 and 1.
it is difficult to write computer program directly in machine code.
Now days the programs are written in high-level language i.e. BASIC, PASCAL, C, C++ etc. A program written HLL is called source program.

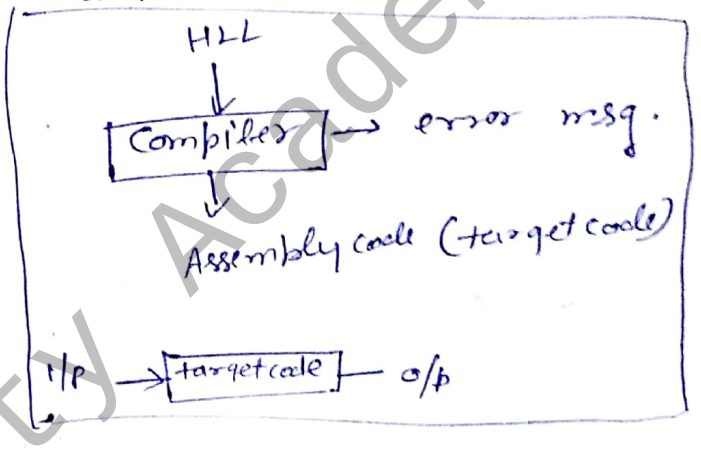
the source code cannot be executed directly by the computer. the source program must be converted into machine code to run it on computer. Every language has its own translator.

The high-level language is converted into binary language. in various phase. A compiler is a program that convert HLL to Assembly language, similarly Assembler is a program that convert the Assembly language to machine level language.

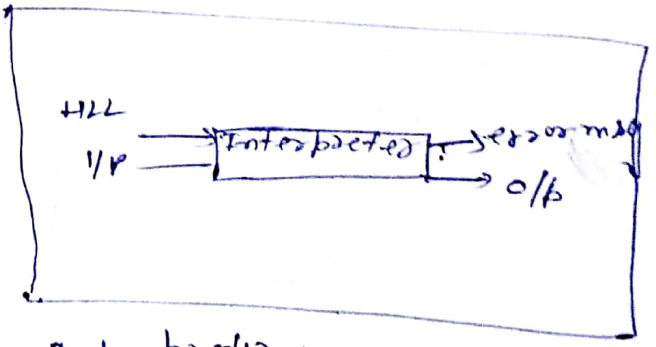
Compiler Vs Interpreter

speed: Compiler is fast.
error diagnosis → Interpreter is better.

compilation:



Interpreter



Compiler

1. It translate source code into object code as a whole
2. It create object file
3. execution is fast
4. translator program is not required to translate each time to run the program
5. it does not make easier to correct the mistakes in the source code.
6. most HLL uses compiler

Interpreter

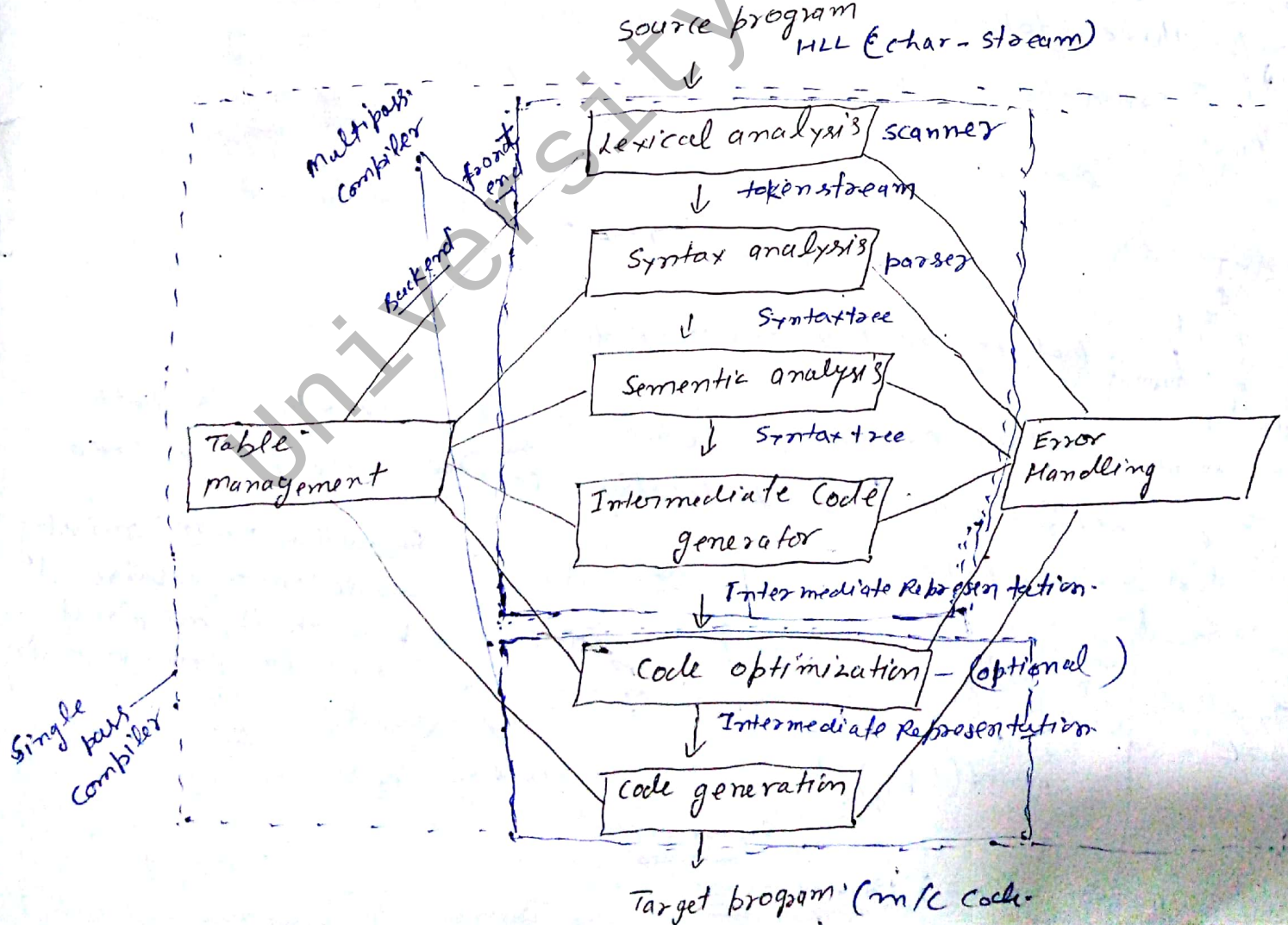
1. It translate the statement of the source code one by one and execute Immediately.
2. It doesn't.
3. execution is slow
4. translator program is required to translate the program each time to run.
5. it makes easier to correct the mistakes in the source code
6. few HLL uses Interpreter.

1- Translators and Compiler :-

A translator is a program that takes as input a program written in one programming language and produces as output a program in another language.

if the source language is a high level language such as FORTRAN, C, C++ and the object language is low level language such as assembly or machine language then the such translator is called compiler.

the compilation process is so complex that it is not resonable either logical or implementation point of view to understant this process we partition it into series of subprocess called Phases.



in Java a:=b
lexical error

or

|int| sum|;|
(smu)=15
lexical error.

float x, a, b;
x = a+b*60

|float| |x|,|a|,|b|;|
|x|=|a|+|b|*|60|
<id,1> <=> <id,2>
<+> <id,3>
<*> <60>
or
$id_1 = id_2 + id_3 * 60$

|a|+|b|
|i|+|+|i|
(longest lexeme matching.)
<token name, attribute value>

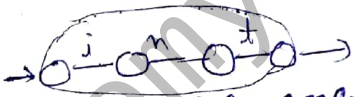
(1) Lexical Analysis:

It is also called Scanner, it separates character of source language into groups that logically belong together, these groups are called lexemes and generate tokens. Lexical analyzer reads the source program one character at a time. that each tokens represents a sequence of character that can be treated as single logical entity.

Identifier, Keyword, constant, operator, symbol etc. are typical token

e.g |int| |a|;|

int is pattern



example:

total = count + rate * 10

there are 7 tokens in above statement

Lexical Analysis dose not generate token for white space, and remove comment. it is used for compaction of white space into one.

Symbol table

S.N	V. Name	V. Type
1	x	float
2	a	float
3	b	float

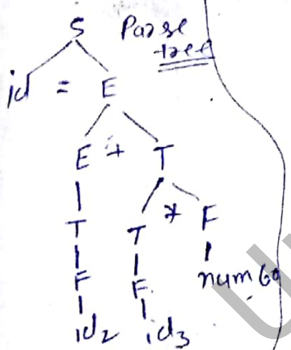
2. Syntax Analysis:

it is also called a parsing. The syntax analyzer groups tokens together into syntactic structures for example the three tokens represent A+B might be grouped into a syntactic structure called an expression.

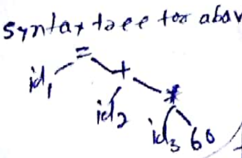
$id_1 = id_2 + id_3 * 60$

S → id = E
E → E+T / T
T → T*F / F
F → id / num

Parse tree



syntax tree for above



we can't generate e.g parse tree for a+b=c

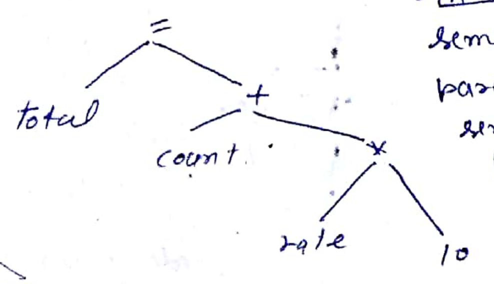
example:

A + / B
id op op id

Lexical analyzer

but syntax analyzer detect an error

another example: total = count + rate * 10 is both Lexically and syntactically correct. the parse tree for that is



Syntactical error: missing semicolon or unbalanced parenthesis, misplaced semicolon, extra or missing braces. etc.

e.g

① while (1
{
printf("hi);
}

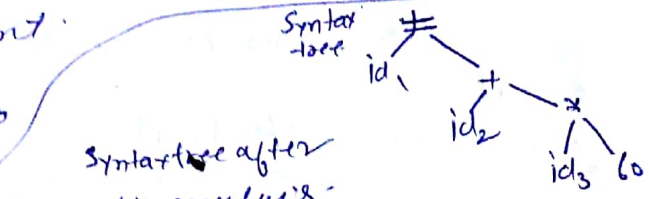
Syntax error

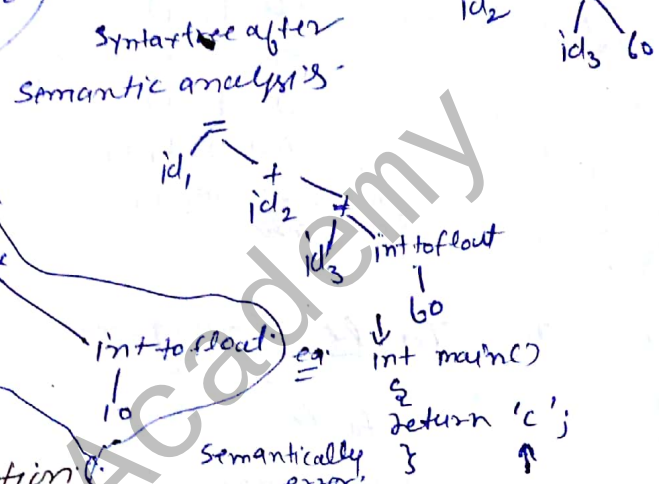
② int x = ; Syntax error

3. Semantic Analysis

Once the syntax is checked in the syntax analyzer phase the next phase semantic analysis determines the meaning of the source string. it also detect the types of data or variable used in statement.

eg. total = count + rate * 10





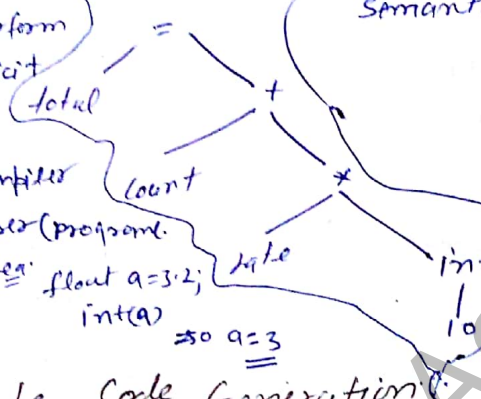
Note: Semantic analyser perform the implicit or explicit type casting.

Implicit → perform by compiler

Explicit → perform by user (programmer)

explicit - int + int
int + float

Implicit - float + float



eg. float a=3.2;
int(a)
⇒ a=3

eg.
int main()
{
return 'c';
}
Semantically error

4. Intermediate Code Generation

The Intermediate code is kind of code which is easy to generate and this code can be easily converted to target code. This code is in form of three address code, quadruple, triple, etc.

eg. three address code of the statement

A / B * C is

$T_1 = A / B$

$T_2 = T_1 * C$

T_1 & T_2 are temporary variable.

3 address code.

t_1 = int to float(60)
t_2 = $id_3 * t_1$
t_3 = $id_2 + t_2$
id_3 = t_3

Note - the work till ICG will be platform independent.

5. Code optimization

It is an optional phase designed to improve the Intermediate code so that the ultimate object program runs faster and take less space. Its output is another intermediate code program that does the same job as the original

t_1 = id_3 * 60.0
id_1 = id_2 + t_1

6. Code generation. produce the object code by ~~directly~~ deciding on memory location for data. Designing a code generator that produce truely efficient object program is one of the most difficult part of compiler design.

e.g. machine code for $A := B + C$ is

$R_2 \leftarrow id_3$
$R_2 \leftarrow R_2 * 60.0$
$R_1 \leftarrow id_2$
$R \leftarrow R_1 + R_2$
$id_1 \leftarrow R_1$

```
LD  R2, id3          LOAD B
MUL R2, R2, #60.0
LD  R1, id2          ADD C
ADD R1, R1, R2       STORE A
ST  id1 R1
```

7. Table Management or Bookkeeping

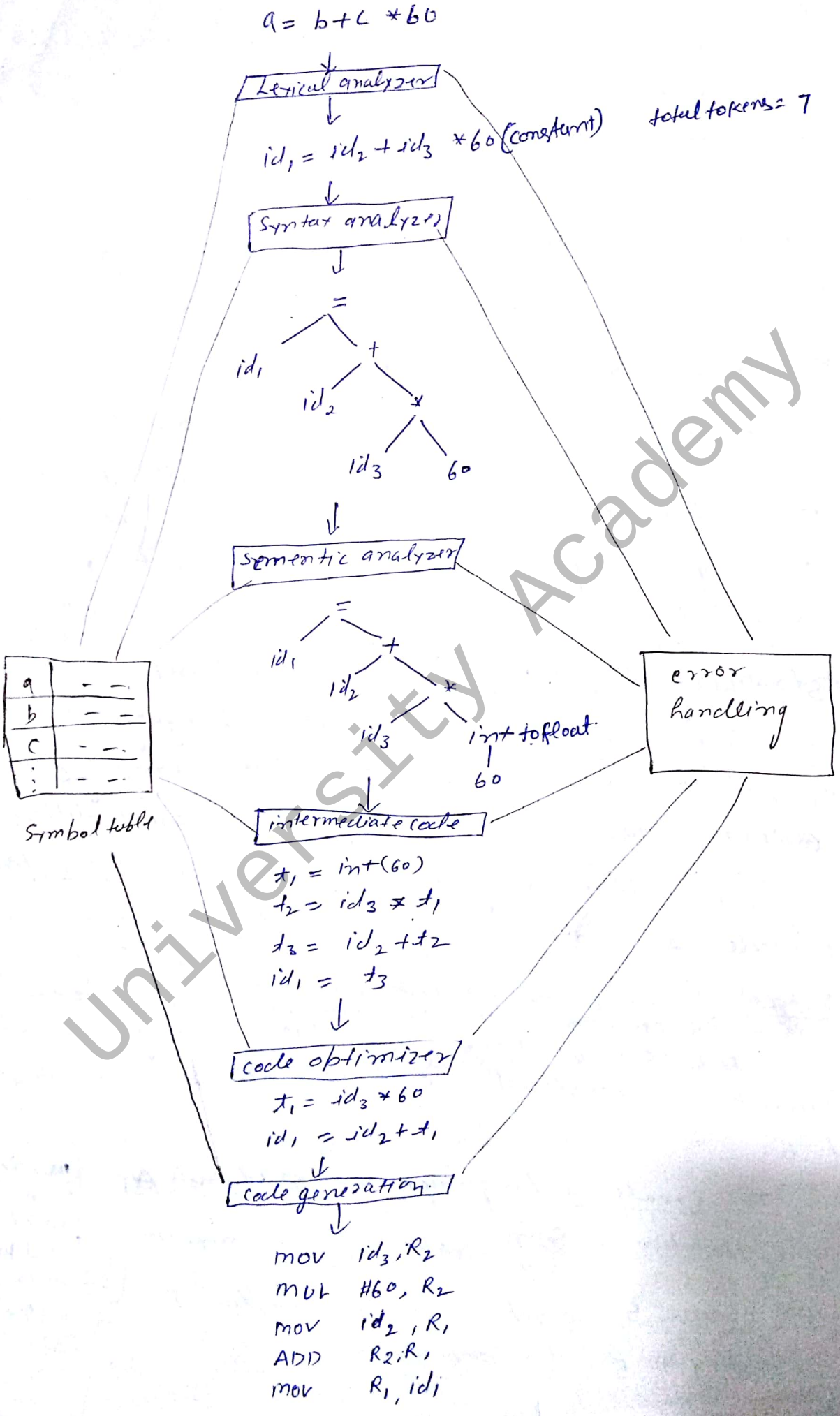
table management portion of the compiler keeps record of the name used by the program and essential information about each data type used in program. (integer, real etc.) the Data structure used to record this information is called Symbol table.

8. Error handling

error handler is invoked when a error in the source program is detected. It must warn the programmer by issuing a message.

both the table Management and error handling intract with all phase of the compiler.

Question Show how an input $a = b + c * 60$ get processed in Compiler

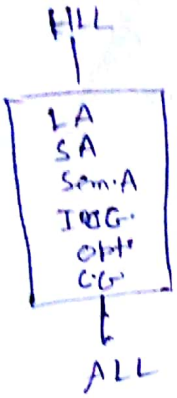


Passes:

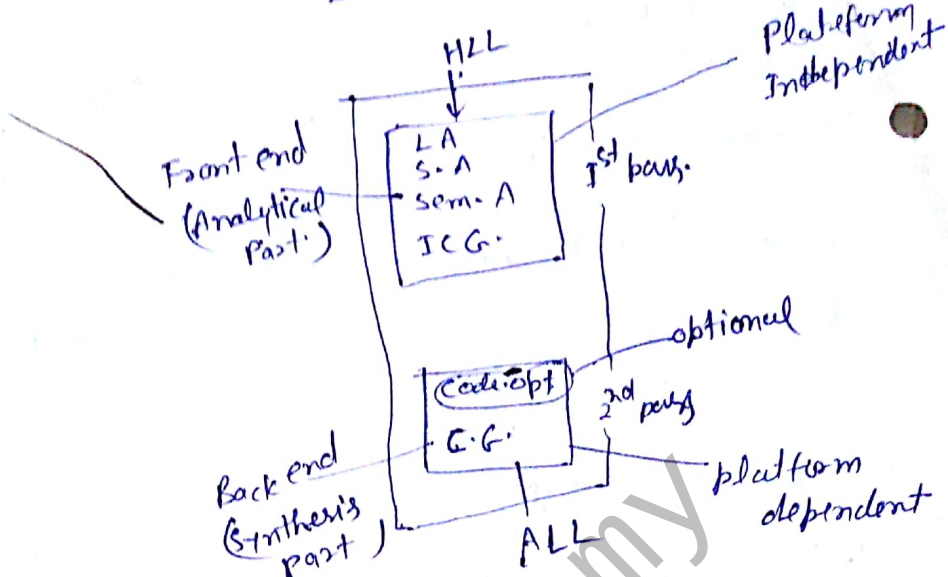
In a compiler portions of one or more phases are combined into a module called pass. A pass read the source program or output of previous pass, makes the transformation specified by its phase and write output into an intermediate file which may be read by subsequent pass.

generally there are two passes in compiler. in the ~~first~~ ~~Front-end phases~~ Front-end-pass of lexical analysis, syntax analysis, semantic analysis and Intermediate code generation might be grouped into one pass. code ~~gene~~ optimization is an optional phase. then Back-end-pass consisting of code generation for a particular target ~~m~~ code

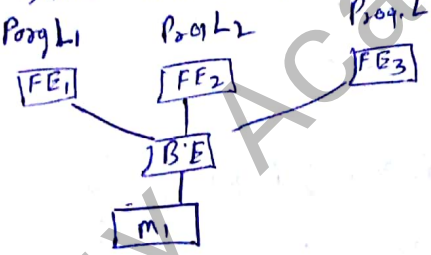
Single pass compiler



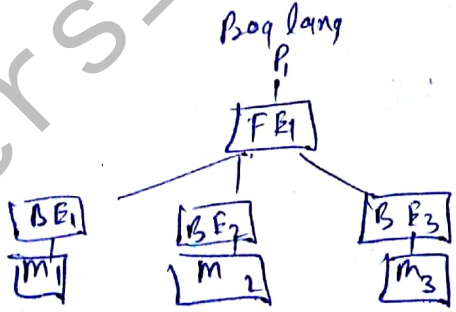
multipass compiler



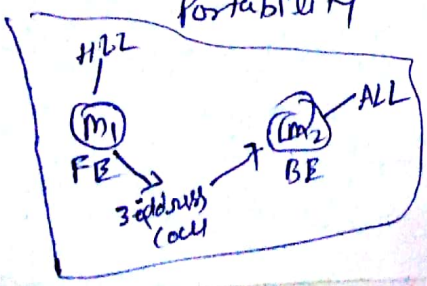
if we want to design compiler for three different programming lang for same machine:



if want to same prog. Language to different machine.



	Single pass	multipass
Communication gap.	No	Yes
speed.	fast	slow
memory	more	less
time.	less	more
Portability	No	Yes



Bootstrapping

Bootstrapping is the process of writting a compiler in the source programming language that it intends to compile. this technique leads to a self-hosting compiler. many prog. lang. are bootstrapped including compiler for BASIC, C, JAVA, LISP, PYTHON etc.

A compiler is characterized by three Languay, its source Language, its object (target) language and and the Language in which it is written. these language may all be qtuit different.

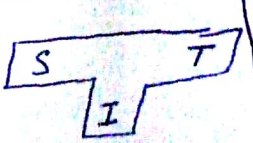
A combiler may run on one machine produce object code for another machine such a compiler called cross-combiler. A cross compiler is a compiler capable of creating executable code for a platform other than the one on which the compiler is running. eg. A compiler that run on a window 7 but generate code that runs on Android.

suppose we have a new language L which we want to construct combiler on a machine A and B.

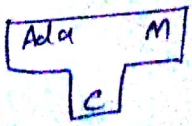
To create a Language for MachineA

Notation $_{I}^{S}T$ represent a compiler for Source S, Target T, Implementation I

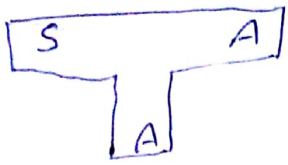
T-diagram for this combiler is



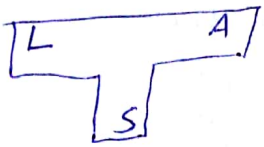
eg. writting compiler for Ada language in clanguge and produced m Languge as output.



1- Create ${}^{S}C^{A}_{A}$ a compiler for subset S of the desired Language L, using Language A which run on machine A (Language A may be Assembly language)

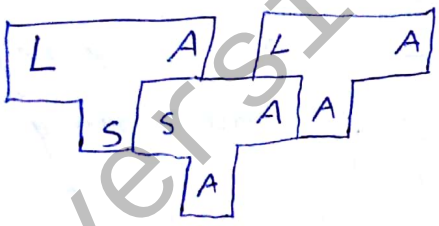


2. create ${}^{L}C^{A}_{S}$ a compiler for Language L written in a subset of L



3. Compile ${}^{L}C^{A}_{S}$ using ${}^{S}C^{A}_{A}$ to obtain ${}^{L}C^{A}_{A}$:- a compiler for Language L, which runs on machine A and produce code for machine A

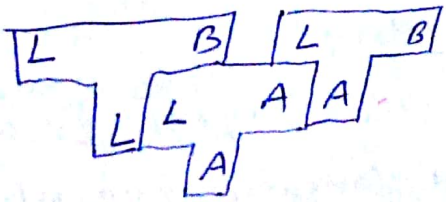
$${}^{L}C^{A}_{S} \rightarrow {}^{S}C^{A}_{A} \rightarrow {}^{L}C^{A}_{A}$$



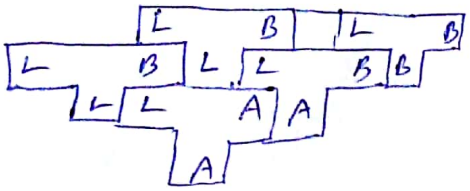
To produce a compiler for different machine:

1- Convert ${}^{L}C^{A}_{S}$ into ${}^{L}C^{B}_{L}$, it is easy if machine A and B both are not different. the language S is a subset of Language L.

2- Compile ${}^{L}C^{B}_{L}$ using ${}^{L}C^{A}_{A}$ to produce ${}^{L}C^{B}_{A}$



3. Now we cross compiler ${}^{L}C^{B}_{A}$ using ${}^{L}C^{B}_{L}$ to produce ${}^{L}C^{B}_{B}$.



Regular Expressions:

The regular expression are usefull for representing certain sets of string in an algebric fashion. Actually these describe the language accepted by Finite automata.

we call regular expression over alphabet 'Σ' are exactly those ~~int~~ expression that can be constructed from following rule.

1- ϵ is a regular exp. denoting $\{\epsilon\}$, that is language containing only the emb~~t~~y string.

2. for each 'a' in Σ, 'a' is a regular expr. denoting $\{a\}$ for language only one string that consisting of single symbol.

3. if R and S are Regular expression denoting language L_R and L_S respectively then.

(i) $(R)|(S)$ is regular expression denoting $L_R \cup L_S$.

(ii) $(R).(S)$ is Regular expression denoting $L_R \cdot L_S$.

(iii) $(R)^*$ is regular expression denoting L_R^*

Example: let $\Sigma = \{a, b\}$

1- RE a/b denotes the language $\{a, b\}$

2. $(a/b).(a/b)$ denote the language $\{aa, ab, ba, bb\}$

3. a^* denote the language $\{\epsilon, a, aa, aaa \cdots\}$

4. $(a/b)^*$ denotes the set of all string ~~consists~~ consisting of zero or more instances of a or b that is

$\{\epsilon, a, b, aa, ab, bb, aaa \cdots\}$

5. a/a^*b denotes the language $\{a, b, ab, aab, aaab \cdots\}$

Finite Automata

Finite Automata are recognizers they simply say yes or no about each possible input string.

F.A. comes in two type:

(i) Non-Deterministic Finite Automata:

Have no restriction on the level on their edge. A symbol can level several edge out of the same state, and ϵ, the empty string is a possible string.

A NFA consist of Five tubles $(Q, \Sigma, \delta, q_0, F)$ where

Q is finite non-empty set of states.

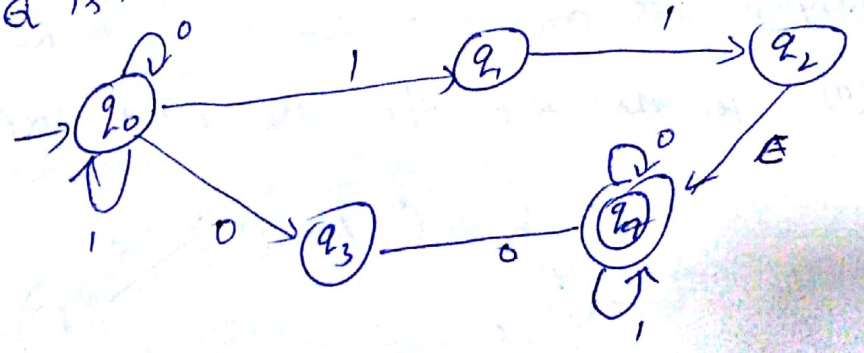
Σ is a Finite Non empty set of inputs

δ is transition function mapping from $Q \times \Sigma$ into 2^Q which is power set of Q is the ~~initial~~ ~~start~~ set of all subsets of Q

$q_0 \in Q$ is the initial state.

$F \subseteq Q$ is the set of Final state.

eg.



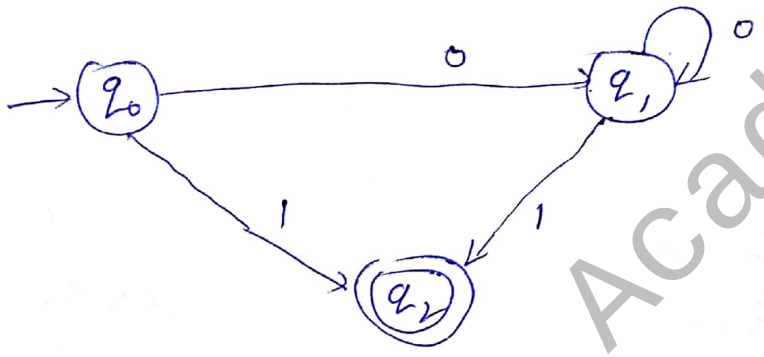
Deterministic Finite Automata (DFA) :

DFA have each state · and for each symbol of its input alphabet exactly one edge with that symbol leaving that state.

A DFA is a special case of an NFA where.

1- there are No moves input ∈.

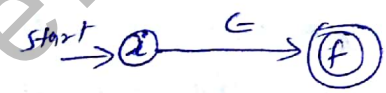
2- for each state 's' and input 'a' there is exactly one edge out of 's' leveled 'a'.



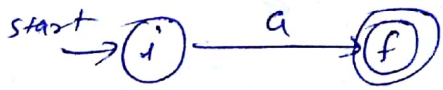
Regular Expression to NFA

Mcnaughton - Yamada - Thompson algorithm;

1- For ∈, construct the NFA

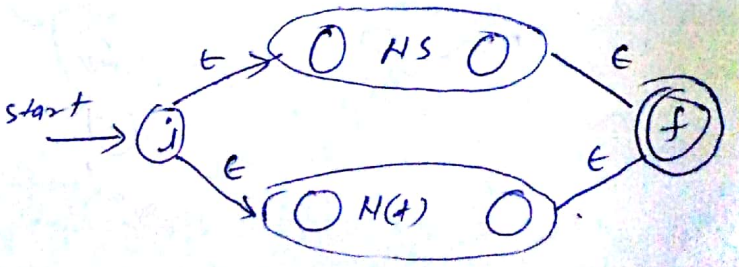


2. For 'a' in Σ construct NFA



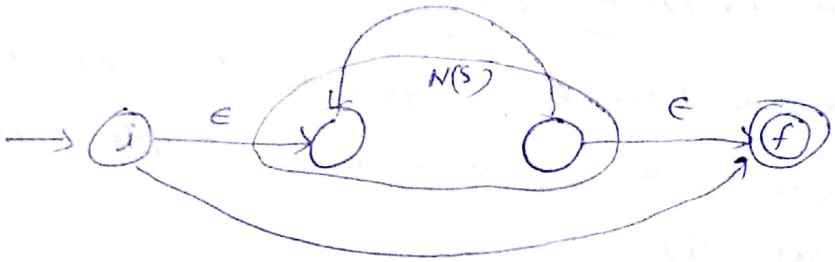
3. Suppose N(s) and N(t) are NFA for RE s and t.

(a) for the R.E s/t the NFA N(s/t)



(b) For the R.E 'st' the NFA N(st).

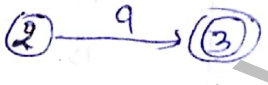
(c) For RE s^* the NFA $N(s^*)$



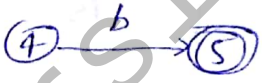
Example: Construct the NFA for RE = $(a/b)^* abb$.

Note
$a/b = a+b$

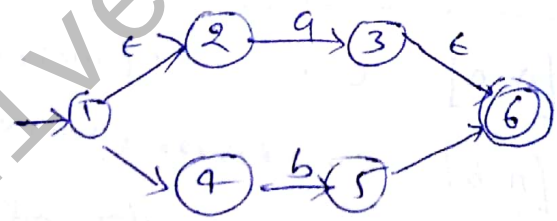
step-1



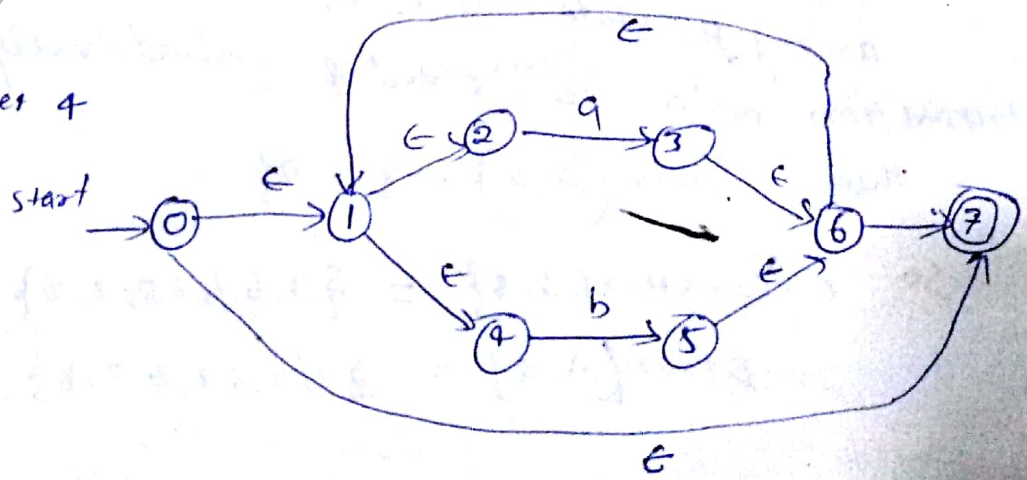
step 2.



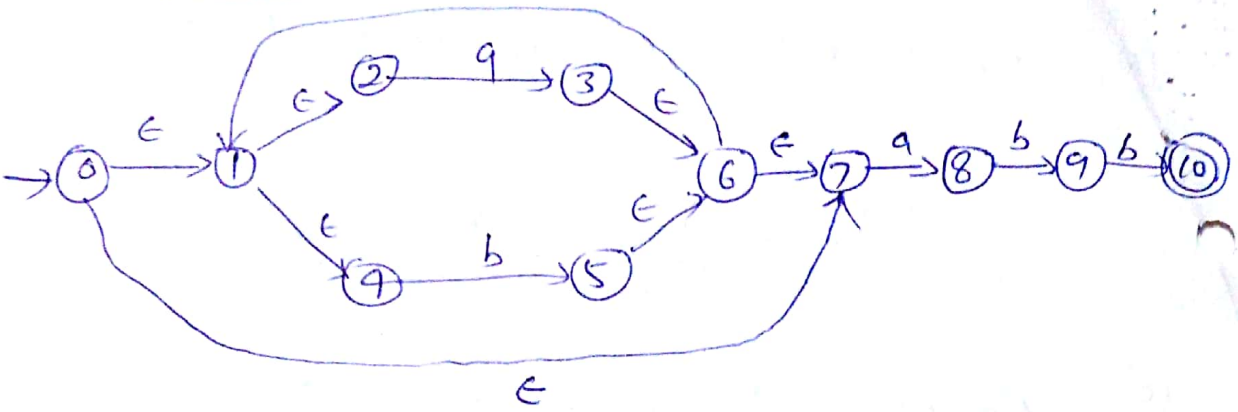
step3



step 4



NFA for RE $(a/b)^*abb$



Conversion NFA to DFA using Subset construction

Example: ~~Set~~ Construct the DFA equivalent to above NFA

We start state A of the equivalent DFA is

ϵ-closure(0)

$A = \{0, 1, 2, 4, 7\}$

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

Compute Dtran [A, a] = ϵ-closure (move(A, a))

Dtran [A, b] = ϵ-closure (move(A, b))

among the state 0, 1, 2, 4 and 7 only 2, 7 have transition on 'a' to 3, and 8 respectivelly.

thus move (A, a) = {3, 8}

so ϵ-closure{3, 8} = {3, 6, 1, 2 4, 7, 8}

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

How we compute

$Dtran[A, b] = \epsilon\text{-closure}(\{5\})$

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

$= C$

$Dtran[B, a] = \epsilon\text{-closure}(move(B, a))$

$= \epsilon\text{-closure}(\{3, 8\})$

$= B$

$Dtran[B, b] = \epsilon\text{-closure}(move(B, b))$

$= \epsilon\text{-closure}(\{5, 9\})$

$= \{5, 6, 1, 2, 4, 7, 9\}$

$= D$

$Dtran[C, a] = \epsilon\text{-closure}(move(C, a))$

$= \epsilon\text{-closure}(\{3, 8\})$

$= B$

$Dtran[C, b] = \epsilon\text{-closure}(move(C, a))$

$= \epsilon\text{-closure}(\{5\})$

$= C$

$Dtran[D, a] = \epsilon\text{-closure}(move(D, a))$

$= \epsilon\text{-closure}(3, 8\})$

$= B$

$Dtran[D, b] = \epsilon\text{-closure}(move(D, b))$

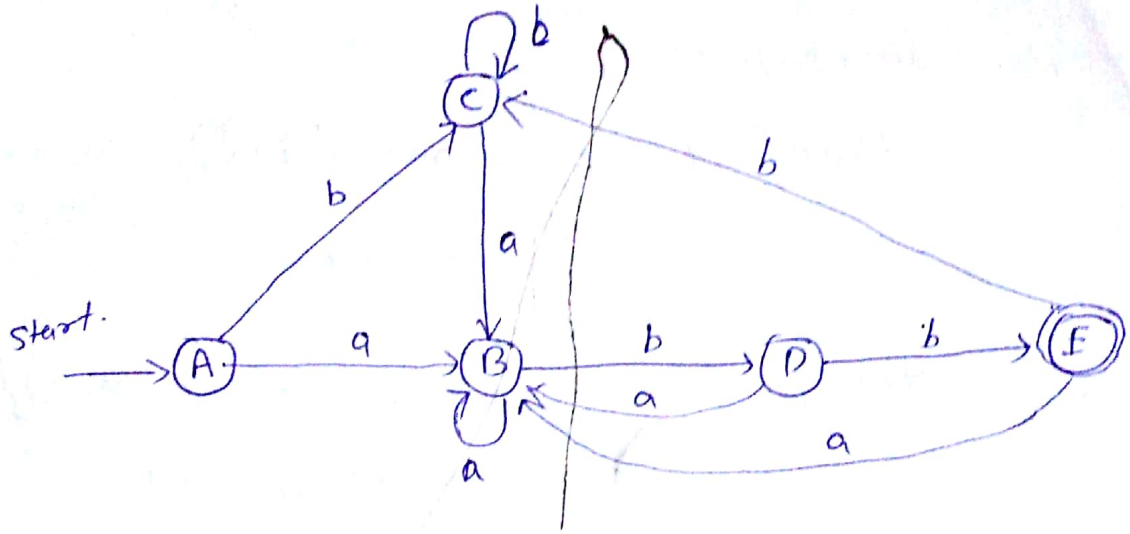
$= \epsilon\text{closure}(\{5, 10\})$

$= \{5, 6, 7, 1, 2, 4, 10\}$

$= E$

$Dtran[E, a] = B$

$Dtran[E, b] = C$



DFA

Example: design DFA for $10 + (0+11)0^*1$

Optimization of DFA - Based pattern matchers:

In this section we present three algorithm that have been used to implement and optimize pattern matchers constructed from regular expression.

1- Algorithm for construct a DFA Directly from a RE.
2. The algorithm to minimize the number of state of DFA.
3. The algorithm to produce representation of transition table.

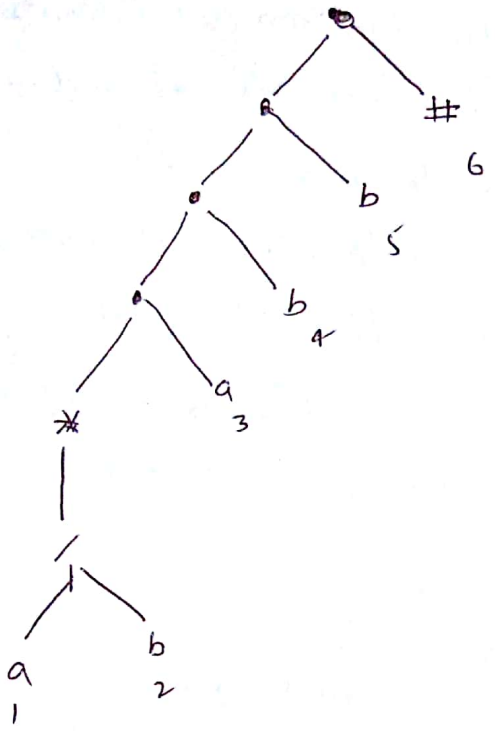
1- Converting RE Directly to DFA:

(i) Construct a Syntax tree T from an Augmented RE $(r)\#$.
(ii) Comput nullable, firstpos, lastpos, and followpos for T.
(iii) Construct Dstates, the set of states of ~~DFA~~ D and Dtran, the transition table for D.

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

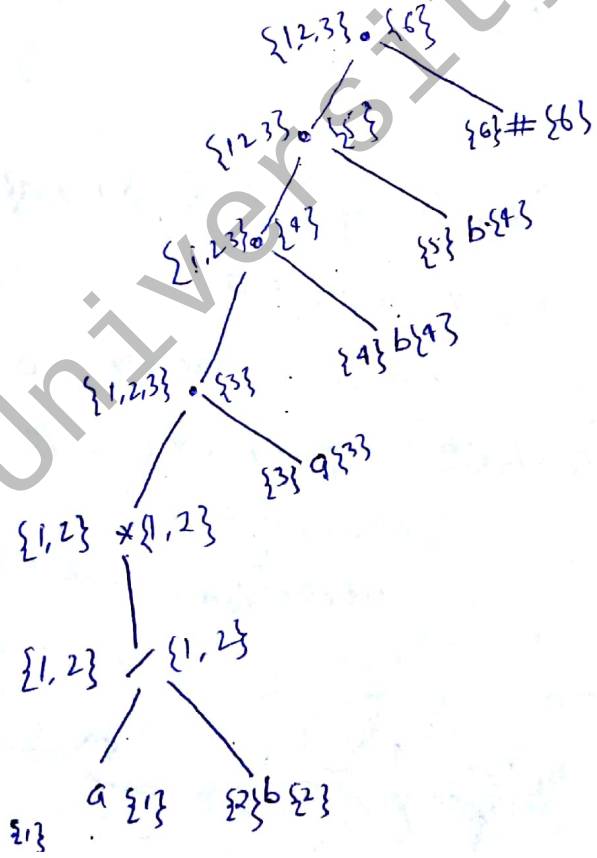
$(r)\# = (a|b)^* abb\#$

(i)



(ii) Nullable (n) true for only star node.

firstpos and lastpos as.



followpos(i) tell us what position can follow position 'i' in the syntax tree. two rule define all all followpos.

1- if n is a cat-node with child c_1 and c_2 and 'i' is a position in lastpos(c_1), then all position in firstpos(c_2) are in followpos(i).

2. if n is a star node ~~then all~~ ~~possible~~ i.e. 'i' is a position in lastpos(n), then all position in firstpos(n) are in followpos(i).

node	followpos
1	{1, 2, 3}
2	{1 2 3}
3	{4}
4	{5}
5	{6}
6	—

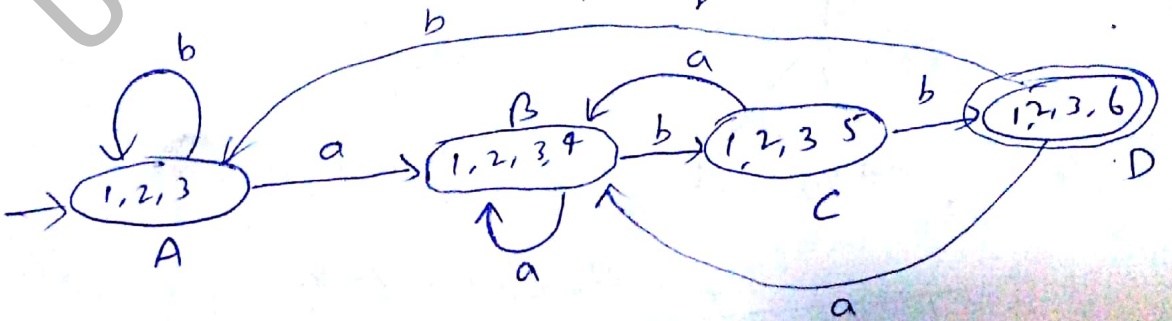
in the syntax tree firstpos of root is {1, 2, 3}. Let set be A and consider input symbol a

position 1 and 3 are for a so let B =

followpos(1) ∪ followpos(3) = {1, 2, 3, 4} = B

Dtran[A, a] = B

Dtran[A, b] = followpos(2) = {1, 2, 3} = A

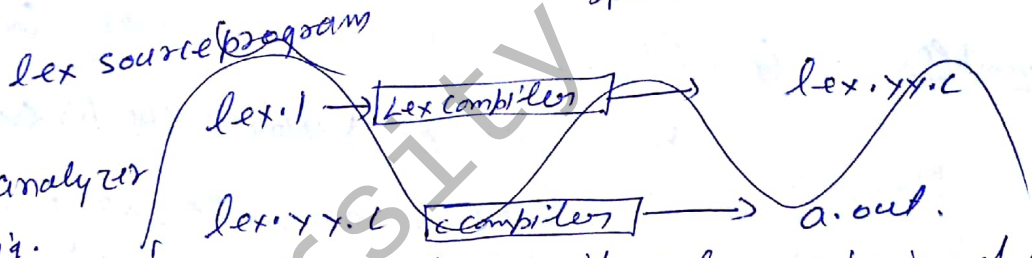


Lexical Analyzer Generator : Lex:

LEX is tool that allows one to specify a lexical analyzer by specifying regular expressions to describe pattern for tokens:

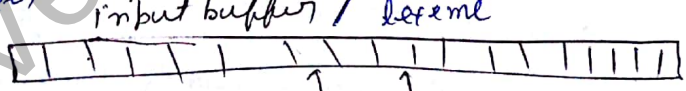
The input notation for the Lex tool is referred as a Lex language and the tool itself is the lex compiler. the lex compiler transforms the input patterns into a transition diagram and generate code.

e.g. An input file which we call lex.l is written in the lex language and describe the lexical analyzer to be generated. the lex compiler transform lex.l to c program and is compiled by c compiler

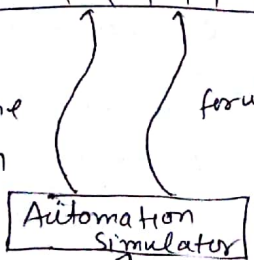


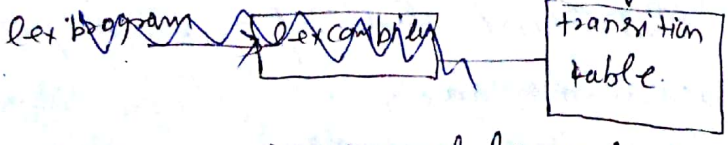
A lexical analyzer shown in fig:

There is an input buffer with two pointers to it A lexeme beginand a forward pointer. the lex compiler construct a transition table for finite automata from R.E. patterns in the Lex specification.



The lexical analyzer itself consists of a finite automata that uses transition table to look for R.E. patterns in the input buffer.

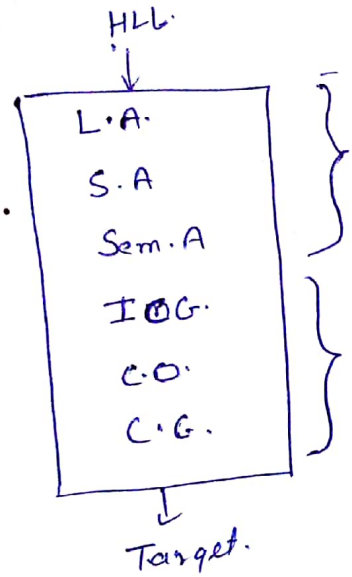




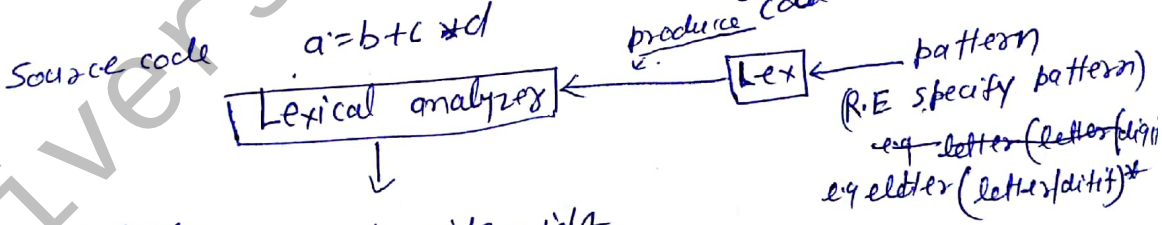
Design of lexical analyzer generator.

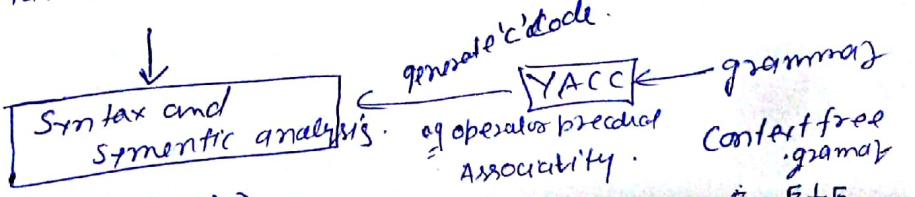
Lex and YACC

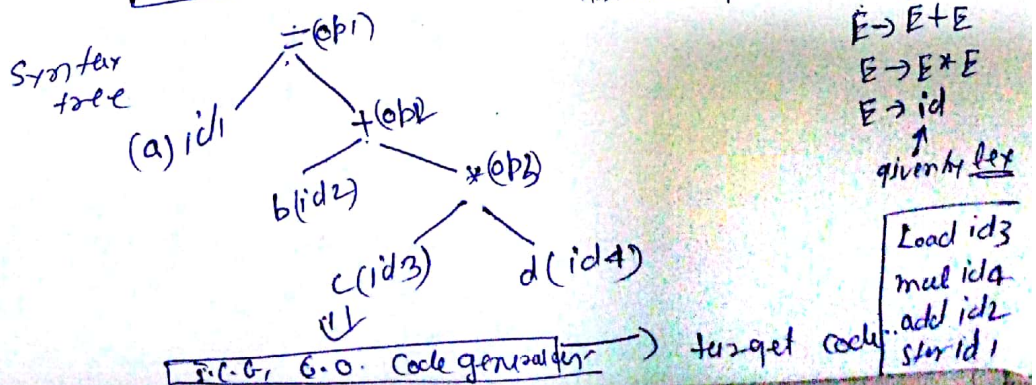
Before 1975 writting a compiler was very time consuming process. In 1975 Lesk and Johnson published papers on Lex and YACC, these Utilities greatly simplify compiler writting.



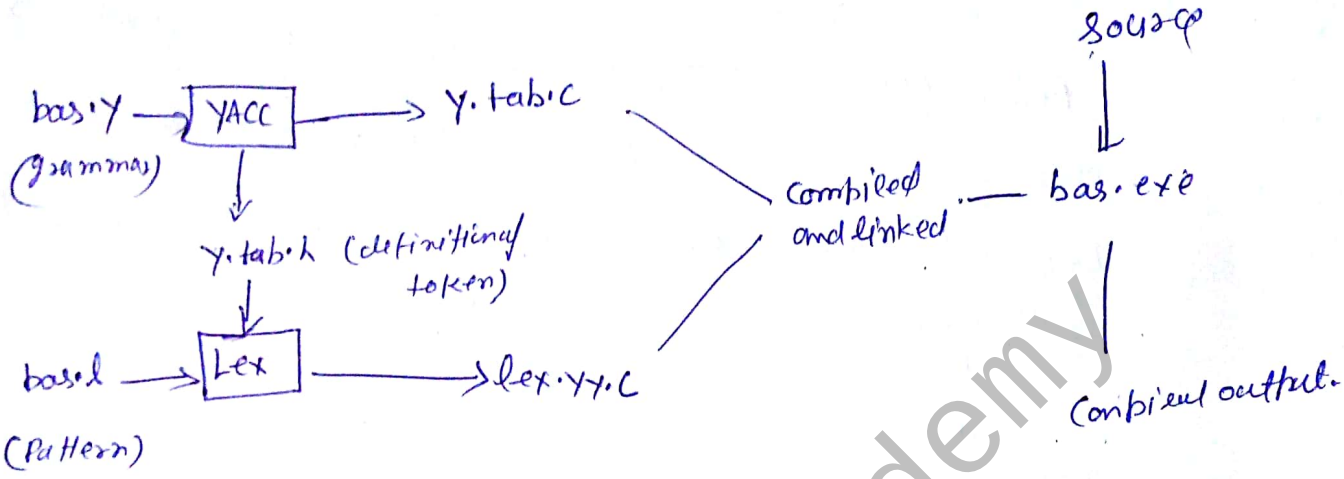
the role of Lex and YACC (yet another compiler compilers) is on first subtask (Read the source file and discover structure)







Writing a BASIC compiler.



lex is program

Syntax:

```
... definition ...
%%
... rules ...
%%
... subroutine ...
```

Yacc is program

```
... definition ...
%%
... rules ...
%%
... subroutine ...
```

Formal Grammars.

A Formal grammar is a set of rules for rewriting strings, along with a "start symbol" from which rewriting start. therefore a grammar is usually act as language generator.

A grammars have four tuble (V_N, Σ, P, S)

V_N is a set of Non-terminal sometime called Syntactic variable.

Σ is a set of terminal symbol sometime called tokens.

P is a set of production rule

S is a start symbol.

Example: $G = (V_N, \Sigma, P, S)$ is a grammar consisting.

$V_N = \{$ <Sentence> <Noun> <verb> <adverb> $\}$

$\Sigma = \{$ Ram, sam, ate, sang, well $\}$

$S = \{$ sentence $\}$

P consist of following production.

Sentence ⟶ <Noun> <Verb>

Sentence → <Noun> <Verb> <adverb>

Noun → Rame / Sam

Verb → age / sang

well → well.

there are two important type of Formal grammar.

1. Context-Free-Grammars
2. Context-Sensitive-grammar.

BNF Notation:

BNF (Backus-Naur Form) is main Notation techniques for Grammar. It is developed by John Backus and peter Naur to describe the syntax of a given language. It is Formally define the grammar of a language.

A BNF Notation is written as:

<symbol> ::= - expression -

where <symbol> is a non-terminals and - expression - consist one or more sequence of symbol seperated by vertical bar "|" indicating the choice symbol that never appear on a left side are terminals and symbol that appearing left side are non-terminals the Non-terminal always enclosed between the pair "<>"

→ the ::= means the symbol on left side must be replace with the expression on the right.

example: <number> ::= <digit> | <number> <digit>

<digits> ::= 0/1/2/3/4/5/6/7/8/9

Context-Free Grammars:

A CFG consist of terminals, non-terminals, a start Symbol and production rule, terminals are basic symbols from which strings are formed. Non-terminal are syntactic variable that denote set of string.

the production of a grammar, specify the manner in which the terminals and non-terminals can be combined to form string.

Terminals: the following symbols are terminals.

(i) lower case letter a, b, ---- z.

(ii) operator symbol +, -, /, * ----

(iii) punctuation symbol (,), ;, etc.

(iv) the digit 0. --- 9

(v) Bold ~~face~~ string such as id.

Non-terminals:

(i) uppercase letters A, B, C ... Z

(ii) S - start symbol.

(iii) lower case italic name such as expr or stmt.

~~(iv)~~

example:

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

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

E and A are non-terminal and E is a start symbol. the remaining symbols are terminals.

Derivations:

The central idea here is that a production is treated as a rewriting rule in which the non-terminals on the left is replaced by the string on the right hand side of the production.

For example: the derivation for string -(id+id) is a sentence of grammar $E \rightarrow E+E \mid E*E \mid (E) \mid -E \mid id$

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

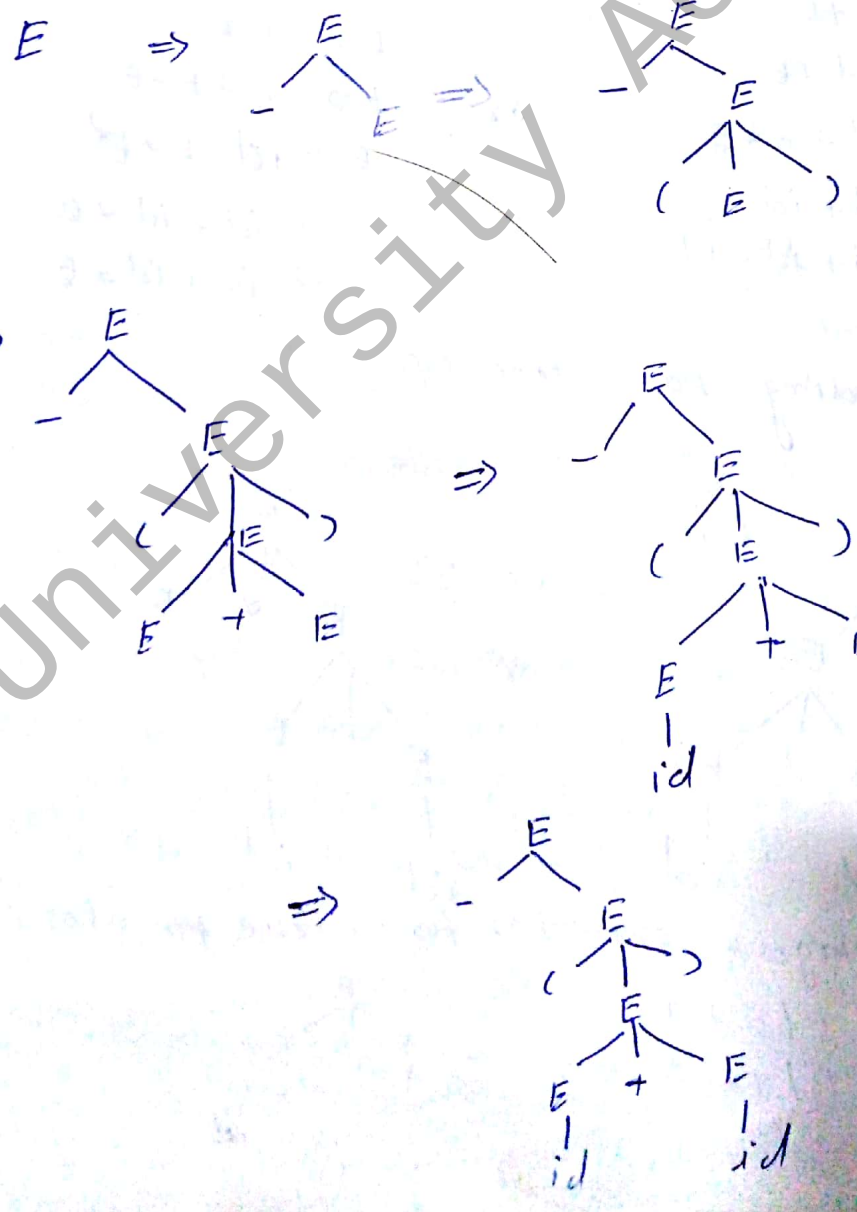
Parse trees:

A parse tree is a graphical representation of a derivation that filters out the choice regarding replacement order. Each interior node of a parse tree is labeled by some nonterminal and the leaves of the parse tree are labeled by nonterminal or terminals and read from left to Right.

forexample: the parse tree for $-(id+id)$.

the derivation for the string is

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(id+E) \Rightarrow -(id+id)$$



Ambiguity:

A Grammar that produce more than one parse tree for some language is said to be ambiguous. An ambiguous grammar is one that produce more than one leftmost or more than one right most derivation for the same sentences.

Example: the grammar is

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

and sentance $id+id*id$

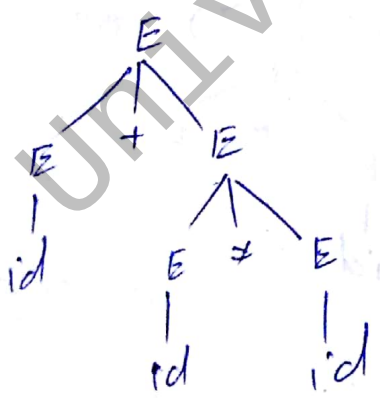
solution it permit two distinct Left most derivation for sentance

$E \Rightarrow E+E$
$E \Rightarrow id+E$
$\Rightarrow id+E*E$
$\Rightarrow id+id*E$
$\Rightarrow id+id*id$

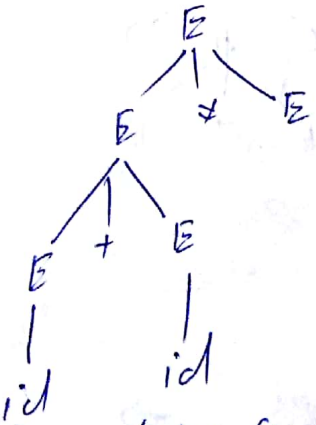
or

$E \Rightarrow E*E$
$\Rightarrow E+E*E$
$E \Rightarrow id+E*E$
$\Rightarrow id+id*E$
$\Rightarrow id+id*E$

corresponding parse tree are

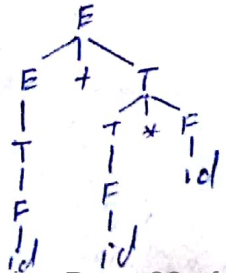


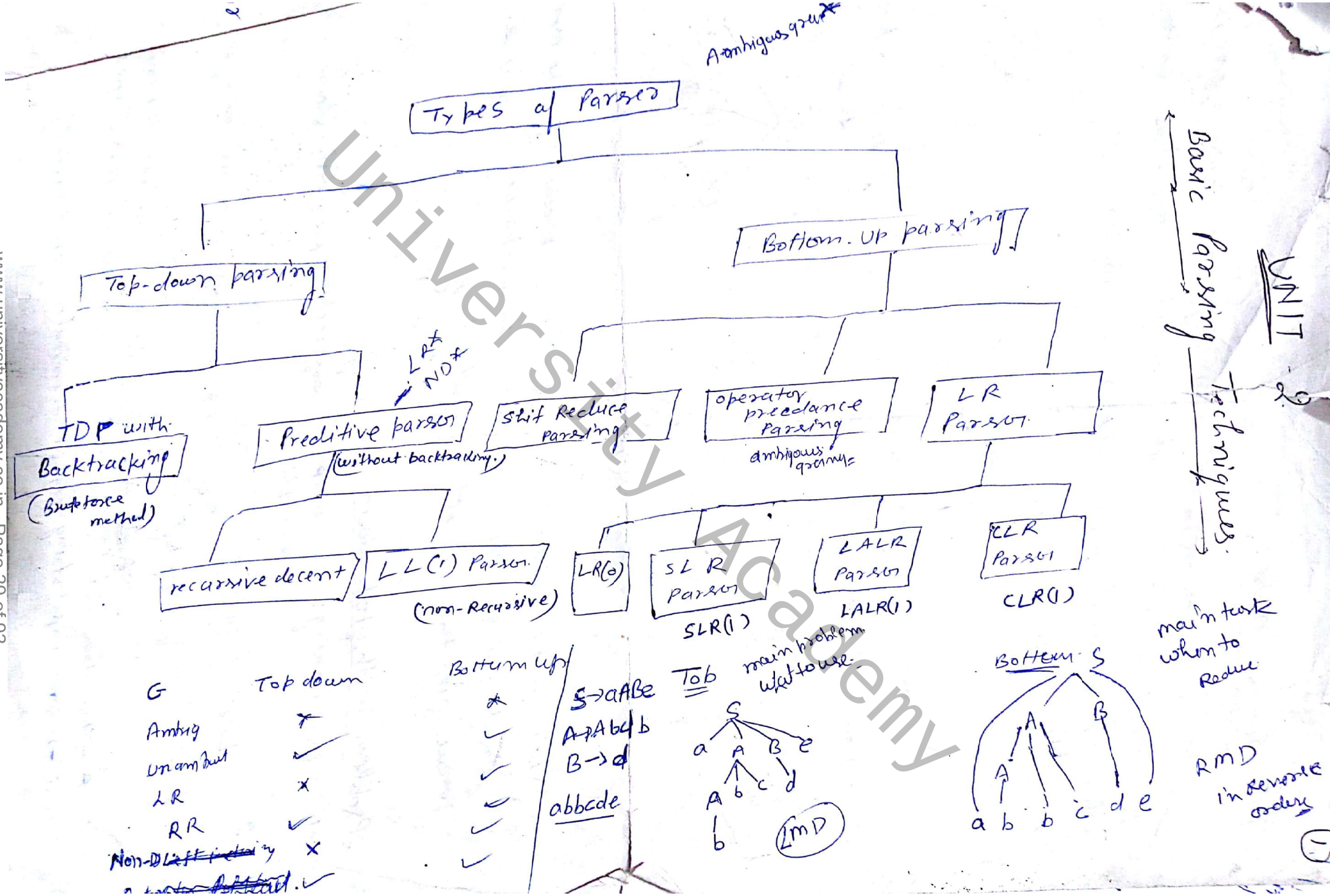
or



the unambigous grammar for above for above is

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





Top-down parser

In the top-down parsing the parse tree is gene from top to bottom (from root to leaves). The derivation term when the required input string terminates

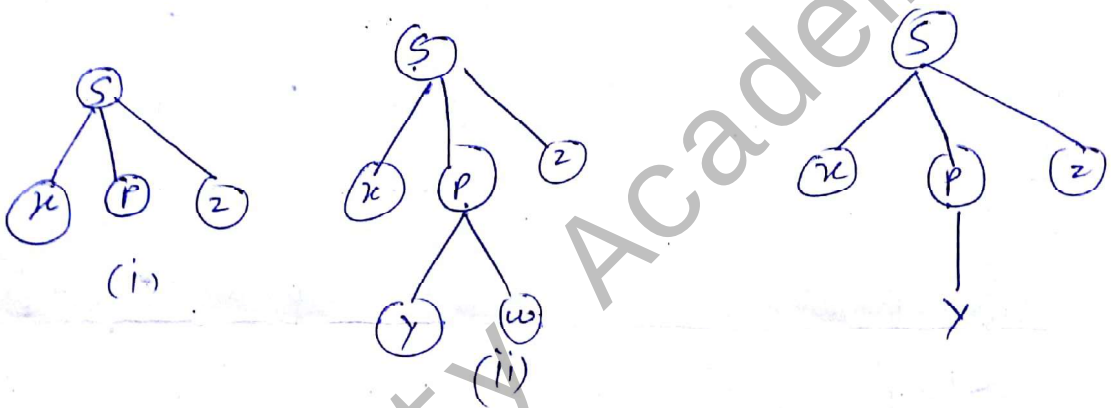
Consider the grammar

$$S \rightarrow xPz$$

$$P \rightarrow yw / y$$

and input string xyz

Now we construct the parse tree in top down approach



Note: if the production rule is not producing the correct input string then we need to Backtrack and then we get the correct input string.

Problem with top down parsing:

1. Backtraking: the concept of Backtracking problem shown in abo example:

2. Left Recursion:

The left recursive grammar is a gramm which is as given below

$$A \overset{+}{\Rightarrow} A\alpha$$

here $\overset{+}{\Rightarrow}$ means the input in one or more steps.

A – Non-terminal

α → some input string (either terminal or non-terminal)

A left recursion production can be eliminated by rewriting the production.

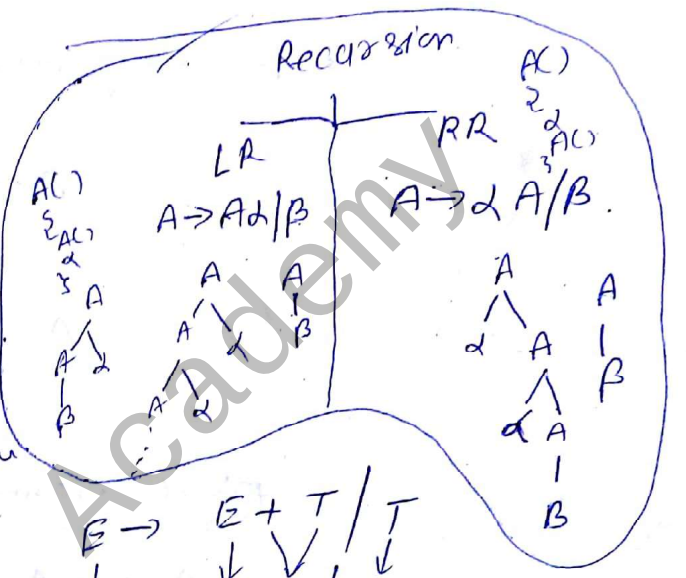
consider the example $A \to A\alpha / \beta$

then we eliminate left recursion by following rule;

$A \to \beta A'$

$A' \to \alpha A'$

$A' \to \epsilon$



consider

$E \to E+T / T$

we have $A \to A\alpha / \beta$

then

$E \to E+T / T$

$A \to A\alpha / \beta$

then

$E \to TE'$

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

similarly

$T \to T*F / F$

$A \to A\alpha / \beta$

then

$T \to FT'$

$T' \to *FT / \epsilon$

Example: Consider the following grammar.

$A \rightarrow ABd / Aa / a$

$B \rightarrow Be / b$

Consider the rule

$A \rightarrow ABd / Aa / a$

we map this gram with rule $A \rightarrow A\alpha / \beta$

$A \rightarrow ABd / a \Rightarrow A \rightarrow aA'$

$A' \rightarrow BdA' / \epsilon$

and

$A \rightarrow Aa / a$

$\Rightarrow A \rightarrow aA'$

$A' \rightarrow aA' / \epsilon$

Final grammar without Left recursive

$A \rightarrow aA'$

$A' \rightarrow BdA' / aA' / \epsilon$

the rule $B \rightarrow Be / b$ mab with $A \rightarrow A\alpha / \beta$ grammar becomes

$B \rightarrow bB'$

$B' \rightarrow eB' / \epsilon$

3- Left Factoring

if the grammar is left factored then it becomes suitable for the use. Basically left factoring is used when it is not clear that which of two alternative is used to expand the non-terminal.

in general form $A \rightarrow \alpha\beta_1 / \alpha\beta_2$

then left factored are

$A \rightarrow \alpha A'$

$A' \rightarrow \beta_1 / \beta_2$

for example: ①

$S \to iEtS \mid iEtSeS \mid a$

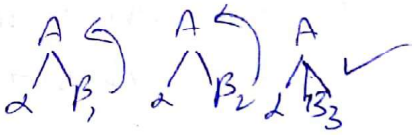
$E \to b$

the left factored grammar becomes.

$S \to iEtS \mid iEtSeS \mid a$

$A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3$

$\alpha\beta_3$



~~$A \to aA'$~~

$A' \to \beta_1 \mid \beta_2 \mid \beta_3$

$\alpha \to iEtS$, $\beta_1 \to \epsilon$

$\beta_2 \to eS$

$S \to iEtSS' \mid a$

$S' \to eS \mid \epsilon$

Example 2 do left factoring in the grammar

$A \to aAB \mid aA \mid a$

$A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3$

$\alpha \to a$

$\beta_1 \to AB$

$\beta_2 \to A$

$\beta_3 \to \epsilon$

$A \to aA'$

$A' \to AB \mid A \mid \epsilon$

④ Ambiguity: Already discussed in previous lecture

Predictive parsing

there are two type of predictive parsers

1- Recursive descent
2- LL(1) Parser

1- Recursive descent parser:

A parser that uses collection of recursive procedure for parsing the given input string is called Recursive descent parser. In this type of parser the CFG is used to build recursive routines.

~~steps for construction of RD parser:~~

the R.H.S of the production rule is directly converted to a program. for each Non-terminal a seperate procedure is written and body of the procedure is R.H.S. of the corresponding non-terminal.

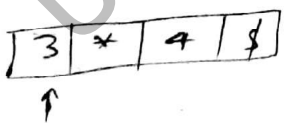
example: consider the grammar

$E \rightarrow num\ T$

$T \rightarrow *num\ T \mid \epsilon$

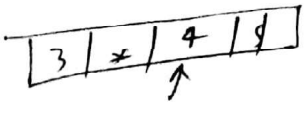
$num \rightarrow 0/1/2 \cdots /9$

and consider the input string 3*4

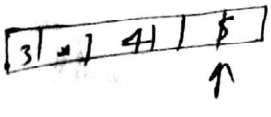


$E \rightarrow num\ T$

$T \rightarrow *num\ T$



$T \rightarrow *num\ T$



declare success

Recursive descent parser

$E \rightarrow E+E \mid i$

$E \rightarrow iE'$

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

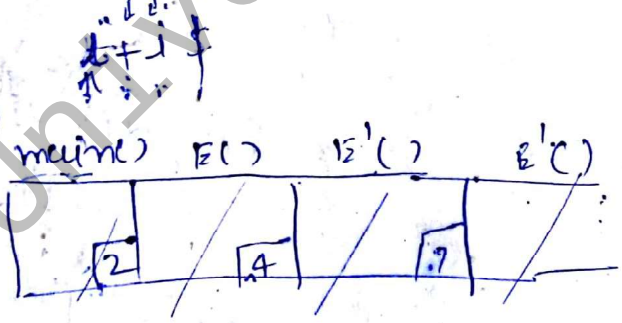
```
E()
{
1. if (l == 'i')
2.   { match('i');
3.     E'();
     }
4. }
```

```
E'()
{
1. if (l == '+')
   {
2.   match('+');
3.   match('i');
4.   E'();
   }
5. else
6.   return;
7. }
```

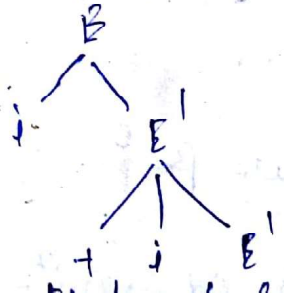
```
match(char t)
{
  if (l == t)
    l = getchar();
  else
    printf("Error");
}
```

```
main()
{
1. E();
2. if (l == '$')
3.   printf("Parsing successful");
}
```

Eg.



Stack Record.



Predictive LL(1) Parser:

This top-down parsing algorithm is non-recursive type parsing. in this type of parsing a table is built.

for LL(1) - the first L means the input is scanned from left to right. the second L means it uses leftmost derivation for input string and (1) means the input symbol uses only one symbol.

Construction of predictive LL(1) Parser:

1. Computation of FIRST and FOLLOW function.
2. Construct table using FIRST and FOLLOW function.
3. Parse the input string with help of parsing table.

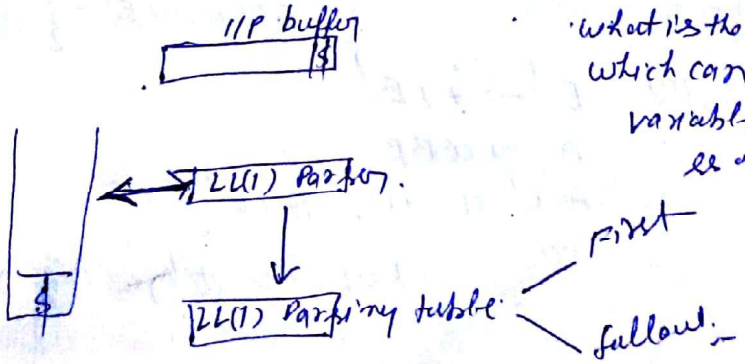
FIRST function:

1. if the terminal symbol 'a' then FIRST(a) = {a}
2. if there is a rule $X \to \epsilon$ then FIRST(X) = {ϵ}
3. for the rule $A \to X_1 X_2 X_3$ then
   FIRST(A) = FIRST(X_1)

FOLLOW Function:

1. For the start symbol S place $ in follow (S)
2. if there is a production $A \to \alpha B \beta$, $\beta \neq \epsilon$ then everything in FIRST(β) without ϵ is placed in FOLLOW(B).
3. if there is a production $A \to \alpha B$ or $A \to \alpha B \beta$ where FIRST(β) contain ϵ ($\beta \Rightarrow \epsilon$) then everything in FOLLOW(A) is in FOLLOW(B).

e.g. S → aABCD First(S) = a
A → b first(A) = b
B → c first(B) = c
C → d first(C) = d
D → e first(d) = e



Architecture of LL(1)

· what is the terminal which can follow a variable in the process of derivation

Example:

Consider the grammar

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

$E \to E+T / T$
$T \to T*F / F$
$F \to (E) / id$

First function:

$FIRST(E) = FIRST(T) = FIRST(F) = \{(, id\}$

$FIRST(E') = \{+, \epsilon\}$

$FIRST(T') = \{*, \epsilon\}$

FOLLOW function.

FOLLOW(E) →

the production $F \to (E)$ shows that E followed ')'. Hence ')' is placed in follow(E).

since E is a start symbol add $ to FOLLOW(E).

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

FOLLOW(E') :-

(i) $E \to TE'$ by rule 3

$A \to \alpha B \beta$

$A \to E$, $\alpha = T$, $B = E'$ $\beta = \epsilon$

then everything in follow A is in follow B since FOLLOW(E) is in FOLLOW(E').

$FOLLOW E' = \{), \$\}$

(ii) $E' \to +TE'$

$A \to \alpha B \beta$

$A = E'$ $\alpha = +T$, $B = E'$ $\beta = \epsilon$

$FOLLOW(E') = \{), \$\}$

FOLLOW(T)

(i) $E \to TE'$

$A \to \alpha B \beta$

$A = E$, $\alpha = \epsilon$, $B = T$, $\beta = E'$ by rule 2

everything FIRST(β) $- \epsilon$ in FOLLOW(B)

since FIRST(E') $- \epsilon = \{+\}$

FOLLOW(T) $= \{+\}$

(ii) $E' \to +TE'$

$A \to \alpha B \beta$ by rule 2.

FIRST(E') $- \epsilon = \{+\}$

since in both production T following E'

since all follow E' is also in follow T.

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

FOLLOW (T')

$T \to FT'$ by rule 3

$A \to \alpha B$

Follow (A) is in FOLLOW(B) since

FOLLOW(T) is in FOLLOW(T')

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

FOLLOW(F) -

$T \to FT'$ or $T' \to *FT'$

by rule 2 by rule 3

$A \to \alpha B \beta$ $A \to \alpha B \beta$

$A = T$, $\alpha = \epsilon$, $B = F$ $\beta = T'$ $A = T'$, $\alpha = *$, $B = F$ $\beta = T'$

FOLLOW(F) $= \{$FIRST(T') $- \epsilon\}$ FOLLOW $T' =$ FOLLOW(F)

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

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

Summarize of FIRST and FOLLOW

Symbol	FIRST	FOLLOW
E	{(, id}	{), $}
E'	{+, ε}	{), $}
T	{(, id}	{+,), $}
T'	{*, ε}	{+,), $}
F	{(, id}	{+, *,), $}

Parsing table for ~~for~~ grammar:

	id	+	*	(	)	$
E	E → TE'			E → TE'		
E'		E' → +TE'			E' → ε	E' → ε
T	T → FT'			T → FT'		
T'		T' → ε	T' → *FT'		T' → ε	T' → ε
F	F → id			F → (E)		

Now the input string id + id * id $ can be parsed.

stack	input	output
$E	id + id * id $	
$E'T	id + id * id $	E → TE'
$E'T'F	id + id * id $	T → FT'
$E'T'id	id + id * id $	F → id
$E'T'	+ id * id $	
$E'	+ id * id $	T' → ε
$E'T+	+ id * id $	E' → +TE'
$E'T	id * id $	
$E'T'F	id * id $	T → FT'
$E'T'id	id * id $	F → id
$E'T'	* id $	

Bottom Up Parsing:

The parse tree is constructed from bottom to up that is from leaves to root is called Bottom up parsing. In this process, the input symbols are placed at the leaf nodes after successful parsing. The parse tree is created starting from leaves, the leaf nodes reduced further to internal node and these internal node further reduced ~~root~~ and root node obtained.

Example Consider the grammar

$S \rightarrow aABe$

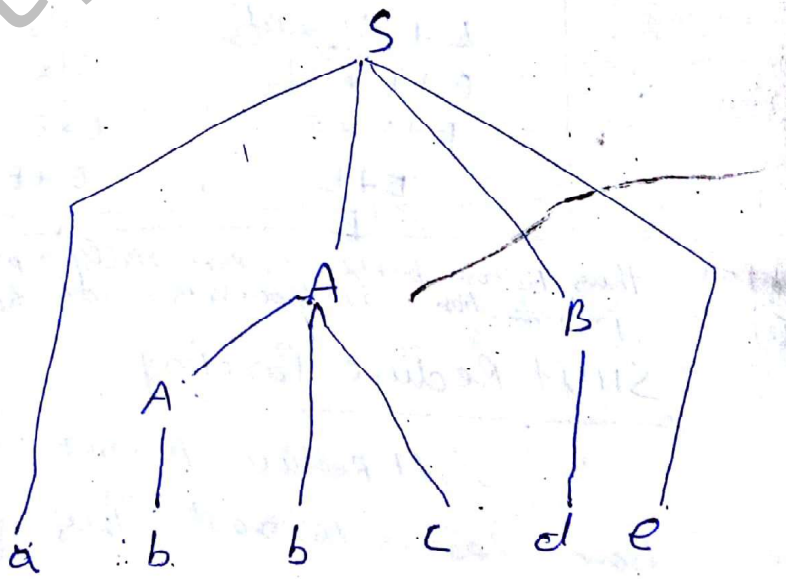
$A \rightarrow Abc \mid b$

$B \rightarrow d$

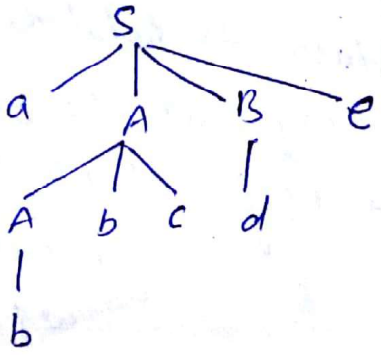
the input string abbcde can be reduced to S by following steps.

abbcde
aAbcde
aAde
aABe
S

Bottom up procedure-



or



Handles and handle pruning:

In bottom-up parsing is to find substring that that could be reduced by appropriate non-terminal. such a substring is called handle.

Handle is a string of substring that matches the right side of production and we can reduce such string by a non-terminal on left hand side production

Example: Consider the grammar

$E \rightarrow E+E$

$E \rightarrow E*E$

$E \rightarrow id$ and input string $id_1 + id_2 * id_3$

Right sentential form	Handle	Reducing production.
$id_1 + id_2 * id_3$	id_1	$E \rightarrow id$
$E + id_2 * id_3$	id_2	$E \rightarrow id$
$E + E * id_3$	id_3	$E \rightarrow id$
$E + E * E$	$E * E$	$E \rightarrow E * E$
$E + E$	$E + E$	$E \rightarrow E + E$
E		

thus Bottom parser is essentially a process of detecting handles and using them in reduction. This process is called hand pruning.

SHift Reduce Parsing:

Shift Reduce parser attempts to construct parse tree from leaves to root. thus it works on Bottom up parser principle. A convenient way to implement a shift-Reduce parser is to use a stack to hold the grammar symbol and an buffer to hold the string w to be parsed. initially the stack is empty and the string w is on input as follows:

stack	input
$	w $

There are Four possible actions a Shift-Reduce parser can make:

1- **Shift**: In a shift action, the next input symbol is shifted onto the top of the stack.

2- **Reduce**: In Reduce Action, the parser knows the right end of the handle is at the top of the stack. then reduction of it by appropriate rule.

3- **Accept**: in accept action the parser announces successfull completion of parsing.

4- **Error**: A situation in which parser can not either shift or reduce the symbols, it cannot even perform accept action it's called error.

Example: Consider the following grammar

$E \rightarrow E+E$
$E \rightarrow E*E$
$E - id$ and input string $id_1+id_2*id_3$.

Stack	Input	action
$	$id_1+id_2*id_3$ $	shift
$id_1	$+id_2*id_3$ $	reduce by $E \rightarrow id$
$E	$+id_2*id_3$ $	shift
$E+	id_2*id_3 $	shift
$E+id_2	$*id_3$ $	reduce by $E-id$.
$E+E	$*id_3$ $	shift
$E+E*	id_3 $	shift
$E+E*id	$	Reduce by $E \rightarrow id_3$
$E+E*E	$	Reduce by $E \rightarrow E*E$
$E+E	$	Reduce by $E \rightarrow E+E$
$E	$	accept.

Question: Consider the grammar

$S \rightarrow TL;$

$T \rightarrow int \mid float$

$L \rightarrow L, id \mid id$

Parse the input string int id, id; using shift-Reduce parser.

Operator Precedence Parser

A grammar is said to be operator precendence if there are no production with ϵ in right side. and no two adjacent non-terminal in any production.

Consider the grammar for arithmetic expression

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

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

This grammar is not an operator precedence grammar. if we substitute for A each of its alternatives, we obtain the following operator grammar

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

In operator precedence parsing, we define three precedence relations, $\lessdot, \doteq, \gtrdot$ between certain pairs of terminals.

these precedence relation have following meaning

$a \lessdot b$	a "yields precedence to" b
$a \doteq b$	a "has same precedence as" b
$a \gtrdot b$	a "takes precedence over" b

the operator precedence relation for id, +, *, $ shown in table

	id	+	*	$
id		⋗	⋗	⋗
+	⋖	⋗	⋖	⋗
*	⋖	⋗	⋗	⋗
$	⋖	⋖	⋖	

Left associativity

Now consider the string id + id * id

we will insert $ symbol at the start and end of the input string. we will also insert operator precedence by referring table.

$⋖ id ⋗ + ⋖ id ⋗ * ⋖ id ⋗ $

the handle can be found by the following process.

1- scan the string from the left to right until first ⋗ is encountered.

2- then scan bakwards over any ≐ until a ⋖ is encountered

3- the handle is a string between ⋖ and ⋗

the parsing can be done as follows'

input	obtain handle and Reduction.
$⋖ id ⋗ + ⋖ id ⋗ * ⋖ id ⋗ $	handle id is obtain E→id
E + ⋖ id ⋗ * ⋖ id ⋗ $	handle id " " "
E + E * ⋖ id ⋗ $	" " "
E + E * E	Remove all non-terminals.
+ *	insert $ at begin and end, insert precedence operators
$⋖ + ⋖ * ⋗ $	* becomes handle. E→E*E
$⋖ + ⋗ $	+ becomes handle E→E+E
$ $	Parsing done.

Example 4.28 try self'

LR Parsers:

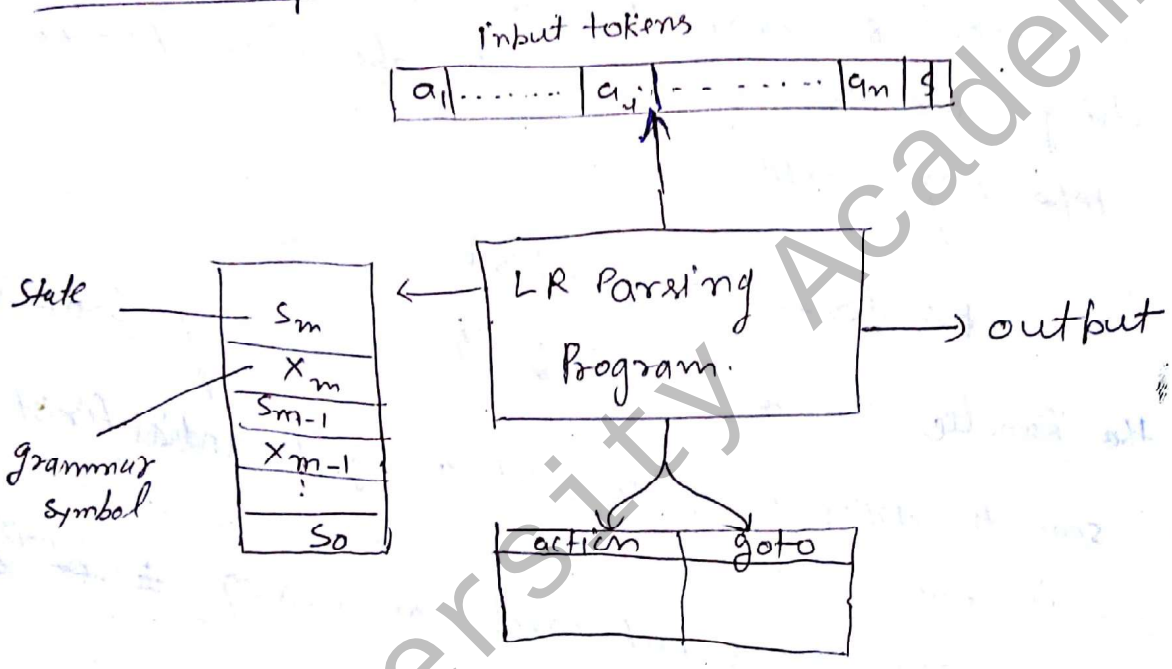
This is most efficient method of bottom-up parsing which can be used to parse the large class of CFG. this method is also called LR(k) Parsing. here the LR parser detect syntactical error very efficiently. here

L stands for left to right scanning

R stands for right most derivation in reverse

K is no. of input symbol. when k is omitted k is assumed to be 1.

Structure of LR Parsers



LR Parser Consist of an input, an output, a stack, a driver program and a parsing table that has two part (action and goto).

→ A driver program is the same for all LR Parsers any the parsing table changes from one parser to another.

→ The parsing table consist two part action and goto. it determines S_m the state currently on top of the stack and a_i the current input symbol. it then consult action $[S_m, a_i]$, the parsing action table entry for state S_m and input a_i, which can have one of the four value.

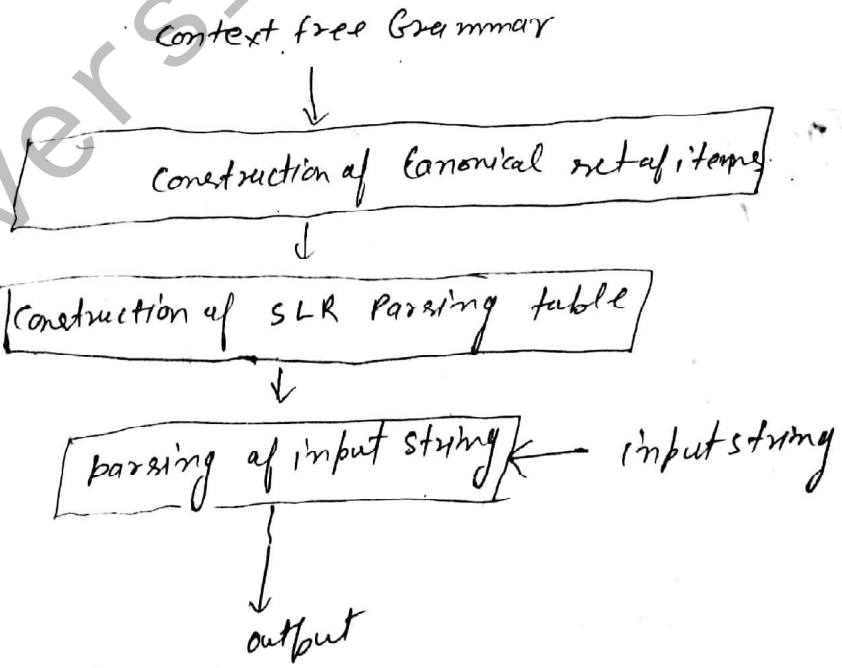
1- Shift s, where s is state
2- Reduce by a grammar
3- Accept
4- error

→ the goto takes a state and grmmar symbol as arguments and produce a state

Types of LR Parser:
1. LR(0).
2. SLR (Simple LR parser)
3. LALR (Lookahead LR parser)
4. CLR (canonical LR parser

Simple LR parser. (LR(0))

It is the weakest of the three method but it is easiest to implement. the parsing can be done as fallows.



Construction of Canonical set of items:
1- for grammar G initially add S' → ·S in set of grammar production and construct the augmented grammar for G.

2. for each set of item I_i in G and for each grammar symbol α (may be terminal or non-terminal), add closure (I_i, α). this process should be repeated by applying goto (I_i, α) for each α in I_i such that goto (I_i, α) is not empty and not in G.

Example:

consider the grammar

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

in this grammar we will add $E' \to .E$ and make augmented grammar such as

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

goto (I_0, E)
I_1: $E' \to E.$
$E \to E.+T$

goto (I_0, T)
I_2: $E \to T.$
$T \to T.*F$

goto (I_0, F)
I_3: $T \to F.$

goto $(I_0, ()$
I_4: $F \to (.E)$
$E \to .E+T$
$E \to .T$
$T \to .T*F$
$T \to .F$
$F \to .(E)$
$F \to .id$

goto (I_0, id)
I_5: $F \to id.$

goto $(I_1, +)$
I_6
$E \to E+.T$
$T \to .T*F$
$T \to .F$
$F \to .(E)$
$F \to .id$

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

goto(I_2, *)

I_7 : $T \to T*\cdot F$
$F \to \cdot(E)$
$F \to \cdot id$

goto(I_4, E)

I_8 $F \to (E\cdot)$
$E \to E\cdot + T$

goto(I_6 T)

I_9 : $E \to E + T\cdot$
$T \to T\cdot * E$

goto(I_7 F)

I_{10} $T \to T*E\cdot$

goto(I_8,))

I_{11} $F \to (E)\cdot$

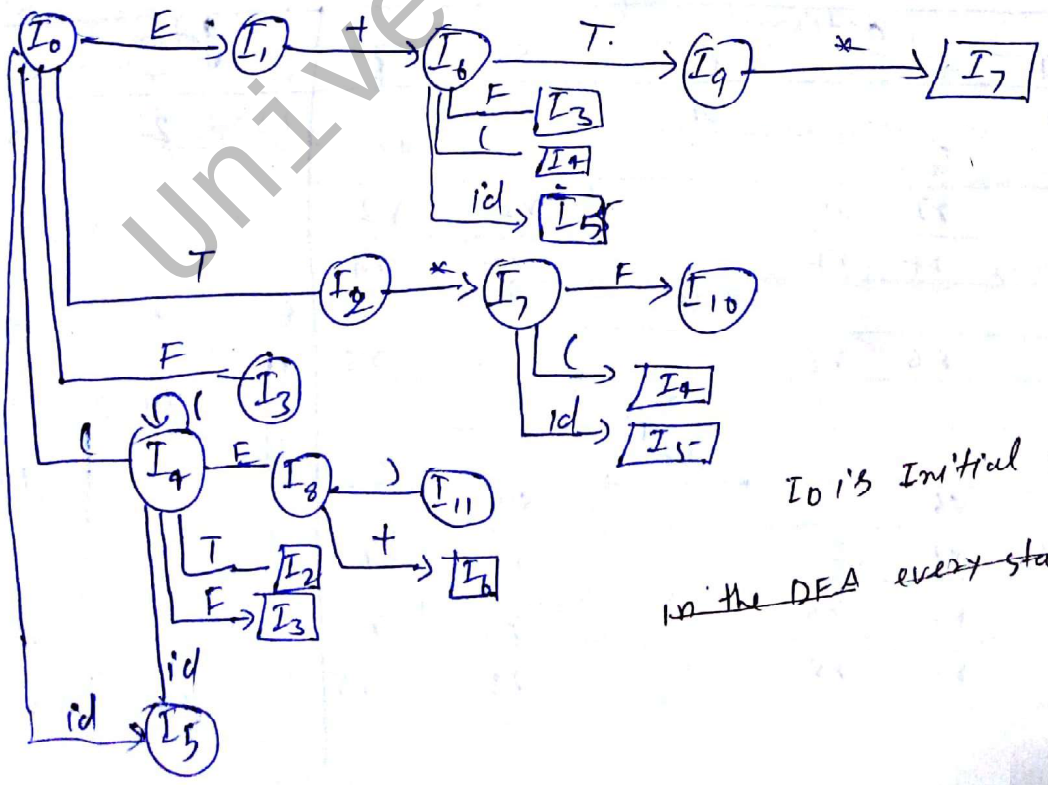
goto(I_4, T) = I_2
goto(I_4, F) = I_3
goto(I_4, () = I_4
goto(I_4, id) = I_5

goto(I_6, F) = I_3
goto(I_6, () = I_4
goto(I_6, id) = I_5

goto(I_7, () = I_4
goto(I_7, id) = I_5

goto(I_8, +) = I_6
goto(I_9, *) = I_7

Now we can desine the DFA for above set of Item:



I_0 is Initial state and other state is Final state.

In the DFA every state i.

Construction of SLR Parsing table:

(1) Initially construct set of item $C = \{I_0, I_1, \ldots I_n\}$ for Input grammar G.

(2) The parsing action are based on each item I_i as given below

(a) if $A \to \alpha . a\beta$ is in I_i and $goto(I_i, a) = I_j$ then set action $[i, a]$ as "shift j" or "sj"

(b) if there is a rule $A \to \alpha .$ is in I_i then set action $[i, a]$ to reduce $A \to \alpha$ for all symbols a, where $a \in FOLLOW(A)$. A must not be an augmented grammar S'.

(c) if $S' \to S .$ is in I_i then the entry in action table action $[i, \$]$ = accept.

(3) the goto part of SLR table can be filled as the goto transition for state i is considered for non terminals only if $goto(I_i, A) = I_j$ then $goto[I_i, A] = j$

4. all the entries not defined considered as error.

State	action						goto		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Parsing of Input string:

Let us take the example input: id * id + id

Stack	Input Buffer	action table	goto table	Parsing action.
$0	id * id + id $	[0, id] = S5		shift
$0 id 5	* id + id $	[5, *] = r6	[0, F] = 3	reduce F → id
$0 F 3	* id + id $	[3, *] = r4	[0, T] = 2	reduce T → F
$0 T 2	* id + id $	[2, *] = S7		shift
$0 T 2 * 7	id + id $	[7, id] = S5		shift
$0 T 2 * 7 id 5	+ id $	[5, +] = r6	[7, F] = 10	Reduce F → id
$0 T 2 * 7 F 10	+ id $	[10, +] = r3	(0, T] = 2	Reduce by T → T * F
$0 T 2	+ id $	[2, +] = r2	(0, E) = 1	Reduce by E → T
$0 E 1	+ id $	[1, +] = S6		shift
$0 E 1 + 6	id $	[6, id] = S5		shift
$0 E 1 + 6 id 5	$	[5, $] = r6	[6, F] = 3	F → id
$0 E 1 + 6 F 3	$	[3, $] = r4	[6, T] = 9	Reduce T → F
$0 E 1 + 6 T 9	$	[9, $] = r1	[0, E] = 1	E → E + T
0 1 E	$	accept		accept.

Example 2. Construct the SLR parsing table for following grammar.

S → SS
S → a
S → ε

Ans. It not SLR(1) parsing.

Example 2

E → E + T / T
T → TF / F
F → F* / a / b

Construct SLR parsing parsing table

First Follow (E) = {+, $}
(T) = {+, $, a, b}
F = {+, *, $, a, b}

CLR (Canonical LR parser)

The canonical set of item is the parsing technique in which a lookahead symbol is generated while constructing set of item. hence collection of set of item is reffered as LR(1).

we follow the following steps in CLR:

1. Construction of canonical set of item along with Lookahead.
2. Building canonical LR Parsing table.
3. Parsing the put string using canonical LR table.

Construction of canonical set of item along with lookahead

1- for the grammar G initially add $S' \to \cdot S$ in set of item.

2. the closure function can be computed as follows.

for each item ~~$A \to \alpha \cdot B \beta$~~ $[A \to \alpha \cdot B\beta, a]$ and rule ~~B~~ $\to \gamma$ and $b \in FIRST(\beta a)$ such that $[B \to \cdot\gamma, b]$ is not in I then add $\boxed{B \to \cdot\gamma, b}$ to I.

3. similarly the goto function can be computed as follows.

for each item $[A \to \alpha \cdot B\beta, a]$ is in I and rule $[A \to \alpha B \cdot \beta, a]$ is not in goto item then add $[A \to \alpha B \cdot \beta, a]$ to goto items.

Consider the grammar

$S \to CC$
$C \to cC \mid d$

$\Rightarrow$

$S' \to .S$
$S \to .CC$
$C \to .cC \mid .d$

Augmented grammar.

initially we add $S' \to S, \$$ as the first rule in I_0

now match

$S' \to .S, \$$ with
$A \to \alpha . B\beta, a$

$A = S', \alpha = \epsilon, B = S, \beta = \epsilon, a = \$$

if there is a production $B \to \gamma$

$\therefore S \to .CC$

and

$b \in FIRST(\beta a)$
$\in FIRST(a)$
$\in FIRST(\$)$
$\in (\$)$

so $S \to .CC, \$$ will be added to I_0

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

Now $S \to .CC, \$$ match with

$A \to \alpha . B\beta, a$ $\quad A = S, \alpha = \epsilon, B = C, \beta = C, a = \$$

if there is a production $B \to \gamma$

$\Rightarrow$ $C \to .cC$
$C \to .d$

and $b \in First(\beta a)$
$\in First(Ca)$
$\in First(C)$
$\in \{c, d\}$

then

$C \to .cC, c/d$
$C \to .d, c/d$ will added to I_0

Now we compute goto function.

goto (I_0, S)

I_1 $S' \to S.$, \$

goto (I_0, C)

I_2

$S \to C.C$, \$

$C \to .cC$, \$

$C \to .d$, \$

$S \to C.C$, \$

$A \to \alpha.B\beta$, a

$A \to S$, $\alpha = C$, $B = C$, $\beta = \epsilon$ $a = \$$

The production $B \to \gamma$

$\Rightarrow C \to .cC$ and $b \in FIRST(\beta a)$

$C \to .d$ $\in FIRST(a)$

$\in First(\$)$

$\in \{\$\}$

since $C \to .cC$, \$, $C \to .d$, \$ } added to I_2

goto (I_0, c)

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

$C \to .cC$, c/d

$C \to .d$, c/d

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

$A \to \alpha.B\beta$, a

$A = C$, $\alpha = c$, $B = C$, $\beta = \epsilon$ $a = c/d$

The production

$B \to \gamma \Rightarrow C \to .cC$

$C \to .d$

and $b \in first(\beta a)$

$\in First(a)$

$\in First(c, d)$

$\in \{c, d\}$

So

$C \to .cC$, c/d added to I_3

$C \to .d$, c/d

goto (I_0, d)

I_4: $C \to d.$ a/d

goto (I_2, C)

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

goto(I_2, C)

I_6: C → c.C, $
C → .cC, $
C → .d, $

Now: C → c.C, $
A → αC.Bβ, $

B → γ ⇒ C → .cC
C → .d

and d ∈ First(Bα)
∈ {$}

then C → .cC, $
C → .d, $ added to I_6

goto(I_2, d)

I_7: C → d., $

goto(I_3, C)

I_8 C → aC., a/d

goto(I_6, C)

I_9 C → aC., $

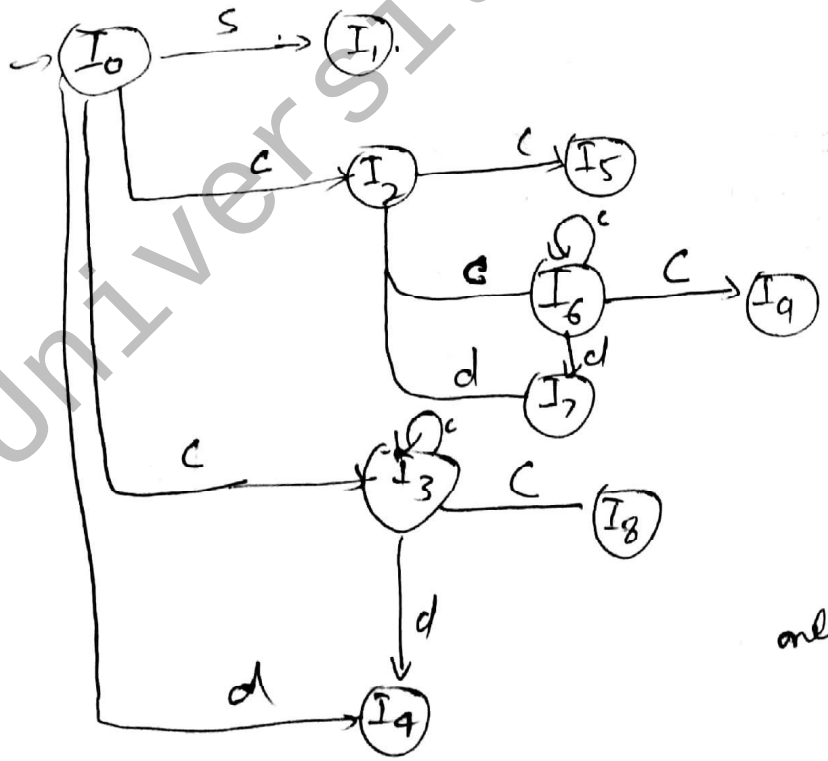
goto(I_3, c) = I_3

goto(I_3, d) = I_4

goto(I_6, c) = I_6

goto(I_6, d) = I_7

DFA for above canonical set.



only I_0 is initial all other state is final state.

Construction of CLR Parsing table:

the algorithm for CLR parsing table is similar as SLR parsing table:

state	action			goto	
	c	d	$	S	C
0	s3	s4		1	2
1	~~s3~~	~~s4~~	accept		
2	s6	s7	~~accept~~		5
3	s3	s4			8
4	r3	r3			
5	r1	r1	r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

parsing the input using CLR or LR(1) table:

input string is "ccdd"

stack	input Buffer	action	goto	Parsing action.
$0	ccdd$	[0, c] = s3		shift.
$0c3	cdd$	(3, c) = s3		shift
$0c3c3	dd$	(3, d) = s4		shift
$0c3c3d4	d$	(4, d) = r3	(3, C) = 8	C→d
$0c3c3C8	d$	(8, d) = r2	(3, C) = 8	C→cC
$0c3C8	d$	(8, d) = r2	(0, C) = 2	C→cC
$0C2	d$	[2, d] = s7		shift
$0C2d7	$	[7, $] = r3	(2, C) = 5	C→d
$0C2C5	$	[5, $] = r1		S→CC
$0S1	$	[1, $] = accept		

LALR Parser (Lookahead LR Parser)

This method is often used in practice because the table obtained by it are considerably smaller than CLR table. in fact the states of SLR and LALR parsing are always same. most of the programming Languages use LALR parser.

we follow the same steps as discussed in SLR and CLR parsing techniques and those are

1- Construction of canonical set of items along with Lookahead
2. Building LALR Parsing table.
3. Parsing the input string using LALR Parsing table.

1- Construction of canonical set of items along with Lookahead.

same procedure as CLR used in LALR canonical set of item along with Lookahead symbol.

For Example consider the Previous example of CLR.

$S \to CC$
$C \to cC$
$C \to d$

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

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

I_2 goto(I_0 C)
$S \to C.C, \$$
$C \to .cC, \$$
$C \to .d, \$$

I_3 goto(I_0, c)
$C \to c.C, c/d$
$C \to .cC, c/d$
$C \to .d, c/d$

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

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

I_6: goto(I_2, c)
$C' \to c.C, \$$
$C \to .cC, \$$
$C \to .d, \$$

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

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

I_9: goto(I_6, C)
$C \to cC. \$$

Construction of LALR Parsing table:

Step:1 ~~Merg~~ Construct $C = \{I_0, I_1, \ldots I_n\}$ the collection of set of LR(1) items.

2. Merge the two states I_i and I_j if the ~~first~~ component (the production rule with dots) are matching and create a new state replacing one of the older state such as

$$I_{ij} = I_i \cup I_j$$

3. the parsing action are given below.

(a) if $[A \to \alpha \cdot a\beta, b]$ is in I_i and goto $(I_i, a) = I_j$ then create an entry in the action table action$[I, a]$ = shift I

(b) if there is a production $[A \to \alpha \cdot, a]$ in I_i then in the action table action$[I_i, a]$ = reduce by $A \to \alpha$. A should not be S'.

(c) if there is a production $S' \to S \cdot, \$$ in I_i then ~~action~~ ~~(I_i, A)~~ action$[i, \$]$ = accept.

4. the goto part of LALR table can be filled as:

if goto$(I_i, A) = I_j$ then goto$[I_i, A] = j$

5. if the parsing action conflict then the algorithm fails to produce LALR parser. and grammar is not LALR. all the entries not defined by rule 3 and 4 are considered to be error.

In the previous example after the step 2 of LALR parsing table algorithm. the canonical Item set will be.

I_0
$S \to \cdot S, \$$
$S \to \cdot C\ C\ \$$
$C \to \cdot cC, a/d$
$C \to \cdot d, a/d$

I_1 goto(I_0, S)
$S' \to S \cdot, \$$

I_2 goto(I_0, C)
$S \to C \cdot C, \$$
$C \to \cdot cC, \$$
$C \to \cdot d\ \$$

I_{36}: goto(I_0, c)
$C \to c \cdot C, c/d/\$$
$C \to \cdot cC, c/d/\$$
$C \to \cdot d, c/d/\$$

goto$(I_{36}, C) = I_{89}$

I_{47} goto(I_0, d)
$C \to d \cdot, a/d/\$$

I_5: goto(I_2, C)
$S \to CC \cdot \$$

I_{89}: goto(I_3, C)
$C \to cC \cdot, a/d/\$$

goto$(I_2, c) = I_{36}$
goto$(I_2, d) = I_{47}$

goto$(I_{36}, c) = I_{36}$
goto$(I_{36}, d) = I_{47}$

Construction of LALR Parsing table:

State	action			goto	
	c	d	$	S	C
0	S36	S47		1	2
1					
2	S36	S47			5
36	S36	S47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Parsing input string "aadd".

Stack	input Buffer	action table	goto table	Parsing action

Ex: Show the grammar $S \to Aa / bAc / Bc / bBa$

$A \to d$

$B \to d$

is LR(1) but not LALR(1) grammar.

UNIT-III

Syntax Directed Translation.

SDT = Grammar + Semantics rule.

Syntax Directed translation are augmented rule to the grammar that facilitate semantic analysis.

There are two notation for associating semantic rule.

(1) Syntax Directed Definition

(2) Syntax Directed translation scheme.

1- Syntax Directed Definition: In Syntax Directed ~~translation~~ definition, Each grammar production $A \to \alpha$, $\alpha \in (V \cup T)^*$ has associated with it a set of semantic rules ~~for~~ of form $b := f(c_1, c_2, \ldots c_k)$ where f is function and either.

i. b is synthesized attribute of A and $c_1, c_2 \ldots c_k$ are attribute belonging to the grammar symbol of production

(ii) b is Inherited Attribute of one of the grammar Symbol on R.H.S of production and $c_1, c_2 \ldots c_k$ are attributes belonging to the grammar symbol of the production.

2. Syntax Directed translation scheme: . It indicate the order in which ~~translation take~~ semantic rule are to be evaluated.

input string → parse tree → do binary graph → evaluation order.

Arithmetic calculation

$E \to E + T$ { E.value = E.value + T.value }

$E \to T$ { E.value = T.value }

$T \to T * F$ { T.value = T.value * F.value }

$T \to F$ { T.value = F.value }

$F \to num$ { F.value = num.lexval }

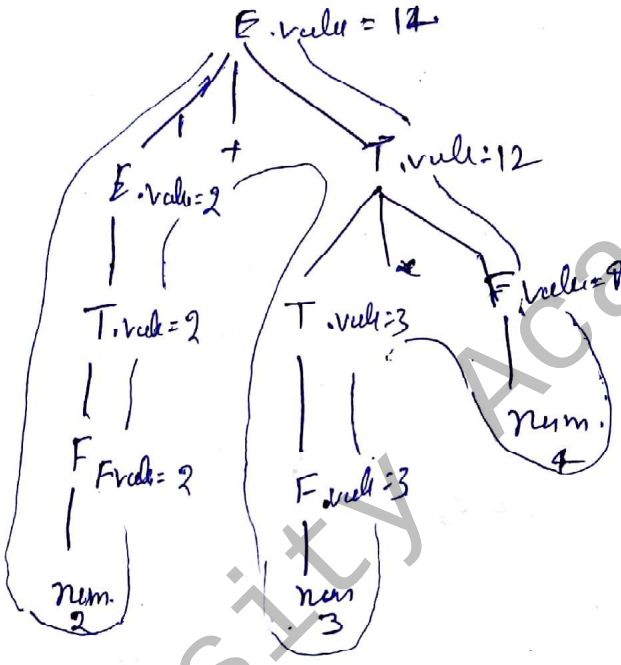
Synthesized Attribute

Attribute

2+3*4 = 14

num num num

left recursion



top down left to right

reduction action

Convert Infix to Postfix

2+3*4.

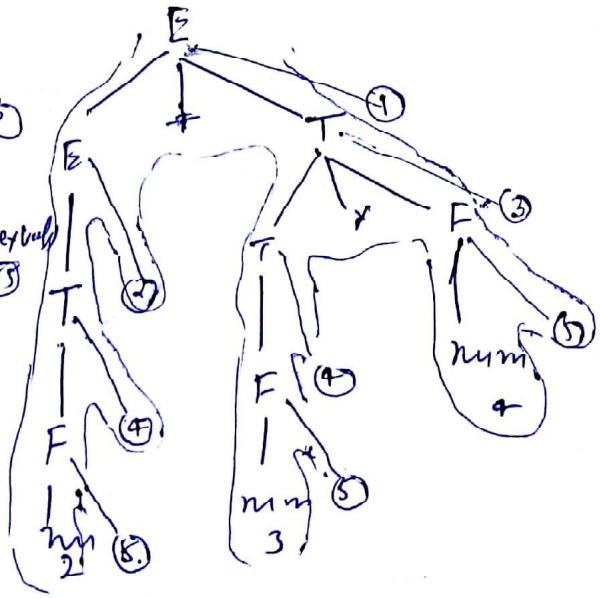
$E \to E + T$ { Print("+") } (1)

$E \to T$ { } (2)

$T \to T * F$ { Print("*") } (3)

$T \to F$ { } (4)

$F \to num$ Print(num.lexval) (5)



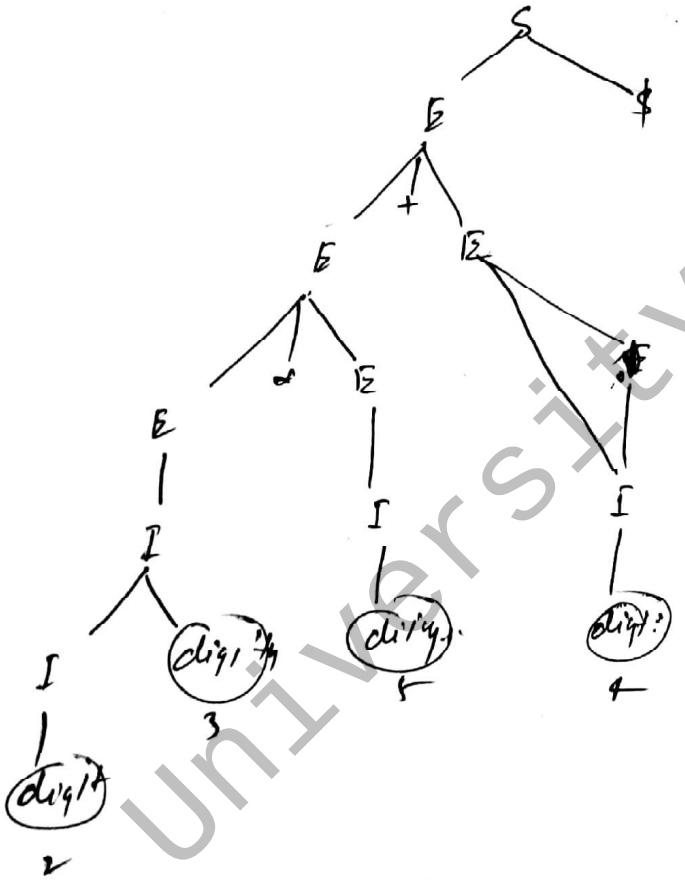
Top down Parse tree

2 3 5 * +

Q. Give the SDT for Desk calculator.

$S \to E\$$	{Print E.val}
$E \to E^1 + E^2$	{E.val = E^1.val + E^2.val}
$E \to E^1 * E^2$	{E.val = E^1.val + E^2.val}
$E \to (E)$	{E.val = E.val}
$E \to I$	{E.val = I.val}
$I \to I^1$ digit	{I.val = 10 * I^1.val + digit.lexval}
$I \to$ digit	{I.val = digit.lexval}

93*5+4



SDT are two types

S-attributed SDT

1- use only synthesized attributes.

2. Semantic action are placed at right end of production

A → B C D {- - - -}

Postfix SDT.

3. Attributed are evaluated during BUP.

L-Attributed SDT

1. uses both inherited and synthesized. Each inherited is restricted to inherit either from parent or left sibling.

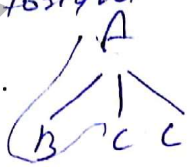
2. Semantic action are placed any where in R.H.S

A → { } B C

/ D { } E

/ F { }

3. Attributes are evaluated by traversing parse tree depth first left to right



Depth first left two right

Eg: A → L M { L.i = f(A.i); M.i = f(L.s); A.s = f(M.s); }

A → Q R { R.i = f(A.i); Q.i = f(R.i); A.i = f(Q.s.); }

a) S-attributed b) L-attributed c) both d) none.

A → B C D

A.s = f(B.s, C.s, D.s) Synthesized attributed. → Node take a value from child node.

C.i = A.i child taked from Parent

C.i = B.i or Inherited: left ~~value~~ from ~~left sib~~ value from sibling

C.i = D.i

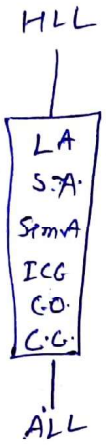
A → B C { B.s → A.s }

a) S-attribute b) L attribute c) both d) none.

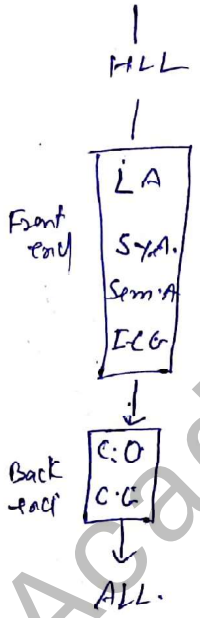
Intermediate code.

"Three Address Code!"

Single pass Compiler.



multi pass / two pass compiler.



Benifit of multi pass compiler.

1. Beneficial for design compiler to different ~~machine~~ programming langue for same machine.
2. Beneficial for design compiler for same programming langue for different machine. [Lecture-6]

There are three kind of Intermediate Code.

1. Postfix Notation.
2. Syntax tree or Parse tree
3. Three-Address code. → Quadruples representation
   → Triples Representation

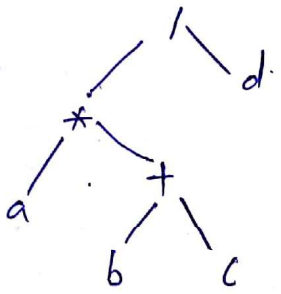
1- Postfix Notation.

infix Notation	Prefix Notation	Postfix Notation
a+b	+ab	ab+
(a+b)*c		ab+c*
a*(b+c)		abc+*

1. No Parentheses are needed
2. No operator precedence needed
3. No Associativity needed.
4. Easy Implementation and scanning left to right.
5. Memory efficient (stack)

2. Syntax tree or Parse tree: The parse tree is usefull Intermediate language representation for a source code. Syntax tree is a tree in which each leaf represent operand and interior node an operator.

$a * (b+c) / d$



infix = Left, root, right

Prefix = root, left, right

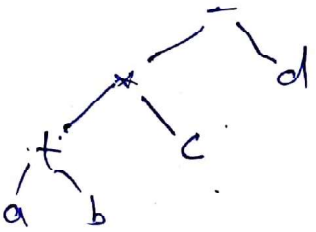
Postfix = left, right, root

infix: $a * (b+c) / d$

Prefix: $/ * a + b c$

Postfix: $a b c + * d /$

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



Prefix: $- * + abc d$

Postfix: $ab + c * d -$

3. Three Address code:

Three Address code is a sequence of statement of the general form $x := y \; op \; z$

x, y, z are variable, constant or compiler generated temporary variable

op is operator

e.g. $x + Y * Z$ is translated into:

$t_1 := Y * Z$

$t_2 := x + t_1$

t_1, t_2 are compiler generated temporary variable

Typical Three address statement:

1. Assignment Statement of form $x := y \text{ op } z$ 'op' is binary or logical operation.
2. unary operation $x := op\ y$
3. Copy statement $x := y$
4. conditional jump if x relop y goto lineL.

   relop: <, =, >, etc.
5. Procedure call:

   Param x_1

   param x_2
6. Index Assignment: $x := y[i]$

   $x[i] = y$
7. Address and pointer $x := \&y$

   $x = *y$

Implementation of three address code:

Quadruples: A quadruple is a record structure with four field called op, arg1, arg2, result.

$x := y \text{ op } z$

result ← arg1, arg2, op.

$a = b * -c + b * -c$

Three address code

$t_1 = -c$
$t_2 = b * t_1$
$t_3 = -c$
$t_4 = b * t_3$
$t_5 = t_2 + t_4$
$a = t_5$

	op	arg1	arg2	result
(0)	− unary	c		t_1
(1)	*	b	t_1	t_2
(2)	− unary	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	Assign.	t_5		a.

Triples.

A triple is a record with three field. op, arg1, arg2 arg1 and arg2 contain variable name or pointer to symbol table.

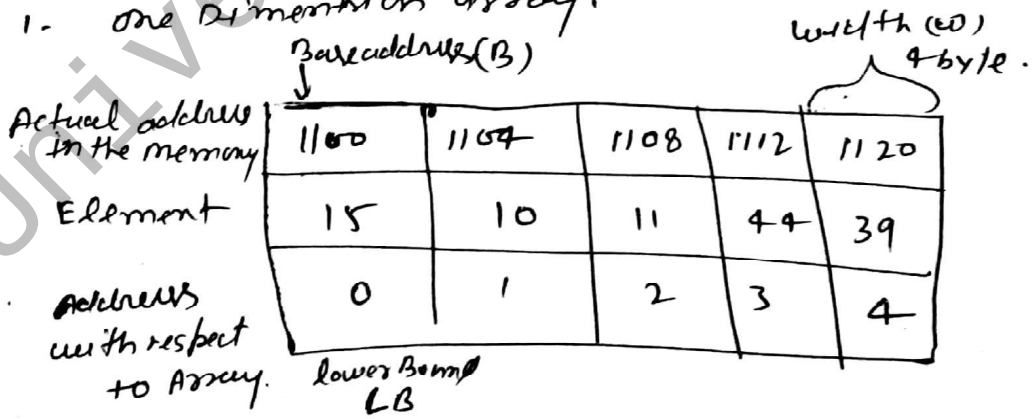
	op	arg1	arg2
(0)	-(unary)	c	
(1)	*	b	(0)
(2)	-(unary)	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

More about translation.

1. Array Reference in arithmatic
2. Procedure Call; ~~declaration~~
3. Declaration and case statement.

1- Array Reference in Arithmatic:

1- One Dimension array:



Address of $A[i] = B + w * (i - LB)$

e.g. given value $B = 1020$, $LB = 1300$, $w = 2$ ...

$A[1700] = ?$ (I)

$1020 + 2 * (1700 - 1300)$

$= 1020 + 2 * 400$

$= 1820$ Ans

Translate the following code in Three Address code.

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

```
1.  i = 1
2   if i <= 10 goto (4)
3   goto (8)
4   t1 = 4*i
5   a[t1] = 0
6   i = i + 1
7   goto (2)
8   stop
```

$a[i] = B + w*(i - LB) = B + 4*(i - 0) = B + 4i$

Address calculation in two Dimensional Array:

① row major

② column major

Row Index \ Column Index	0	1	2	3
0	8	6	5	4
1	3	2	9	6
2	7	3	2	1

A[3][4]
M N

① row major :

8	6	5	4	3	2	9	6	7	3	2	1

row-0 row-1 row-2

Add A[i][j] = B + w*[N*(i - Lr) + (j - Lc)]

② Column major :

8	3	7	6	2	3	5	9	2	4	6	1

Col 0 Col 2 Col 3 Col 3

Add A[i][j] = B + w*[(i - Lr) + (j - Lc)*m]

① row wise: B = 1500, w = 1 byte Lr = 0 Lc = 0 N = 4, M = 3

$A[2][3] = 1500 + 1*[4*2 + 3]$
(i j)
$= 1500 + 1*11]$
$= 1511$ ✓

② Column wise: $A[2][3] = 1500 + 1*[2 + 3*3]$

$= 1500 + 11$
$= 1511$ ✓

Translate following fragment into three address code.

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

```
1   i = 0
2   if i<10 goto(4)
3   goto(8)
4   t1 = 44 * i
5   a[t1] = 1
6   i = i+1
7   goto (2)
8   Stop.
```

We assume row major form and 4 byte allocation per word.

$$a[i][i] = B + 4 \times [10 \times (i) + i]$$

$$= B + 4 \times [11 \times i]$$

$$= B + 44 \times i$$

Procedure call

A procedre call like $P(A_1 A_2 A_3 \cdots A_n)$ may have many Addresses for one statement in three Address code. So it is shown as sequence of $n+1$ statements.

```
Param A1
Param A2
  :
Param An
call P, n
```

P is name of procedre.

n is integer indicating number of actual parameters

Example swapping two numbers using function.

```
Int main()
 {
  int x, y
  swap (&x, &y)
 }
void swap (int *a, *b)
 {
  int t;
  t = *b
  *b = *a
  *a = t;
 }
```

three Address code

```
1.  Param &x
2   Param &y
3   call swap, 2
4   t = *b
5   *b = *a
6   *a = t
7
8
```

Case Statement:

```
switch expression
{
  case value : statement
  case value : Statement
       :
  case value : student
}
```

Generate the Three Address code for

Ex.

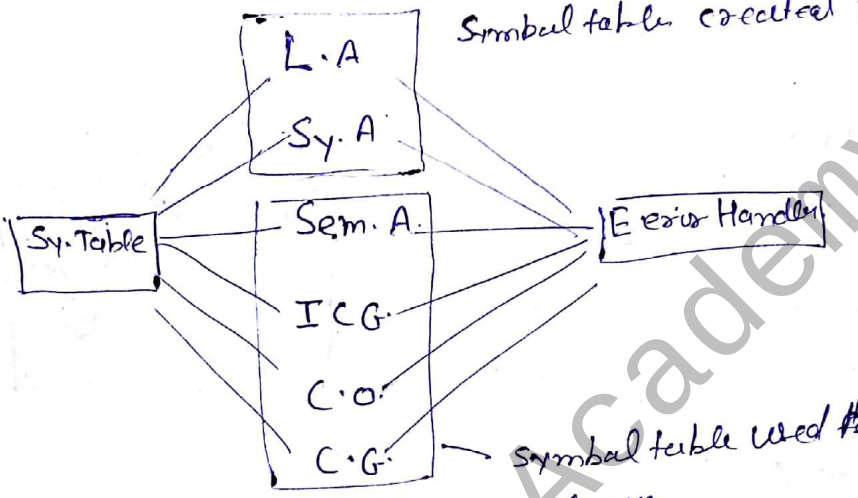
```
switch (ch)
{
case 1; c = a+b;
        break
case 2 : c=a-b;
        break;
}
```

```
1  if ch = 1  goto (3)
2  if ch = 2  goto (6)
3  t1 = a+b
4  c = t1
5  goto 9
6  t1 = a-b
7  c = t1
8  goto_9
9  stop.
```

UNIT-IV

Symbol table

Symbol table is an data structure created and maintained by compiler in order to store information about. variable Name, function name. object, class, compiler generated temp. var. etc.



Format of symbol table

Compiler uses following type of information in symbol table,

1) Data type, 2) Name/Id 3) scope. 4) Address
5- other Attributes

e.g. static int a;

S.No.	Name.	Type	Attribute
1	a	int	static

Symbol table Representation : example. Int calculate; sum, a, b;

1. Fixed length.

Name.	Type.
c a l c u l a t e	int
s u m	int
a	
b	

2. variable length.

Name. Starting index	length	Type.
0	10	
10	4	int.
14	2	
16	2	

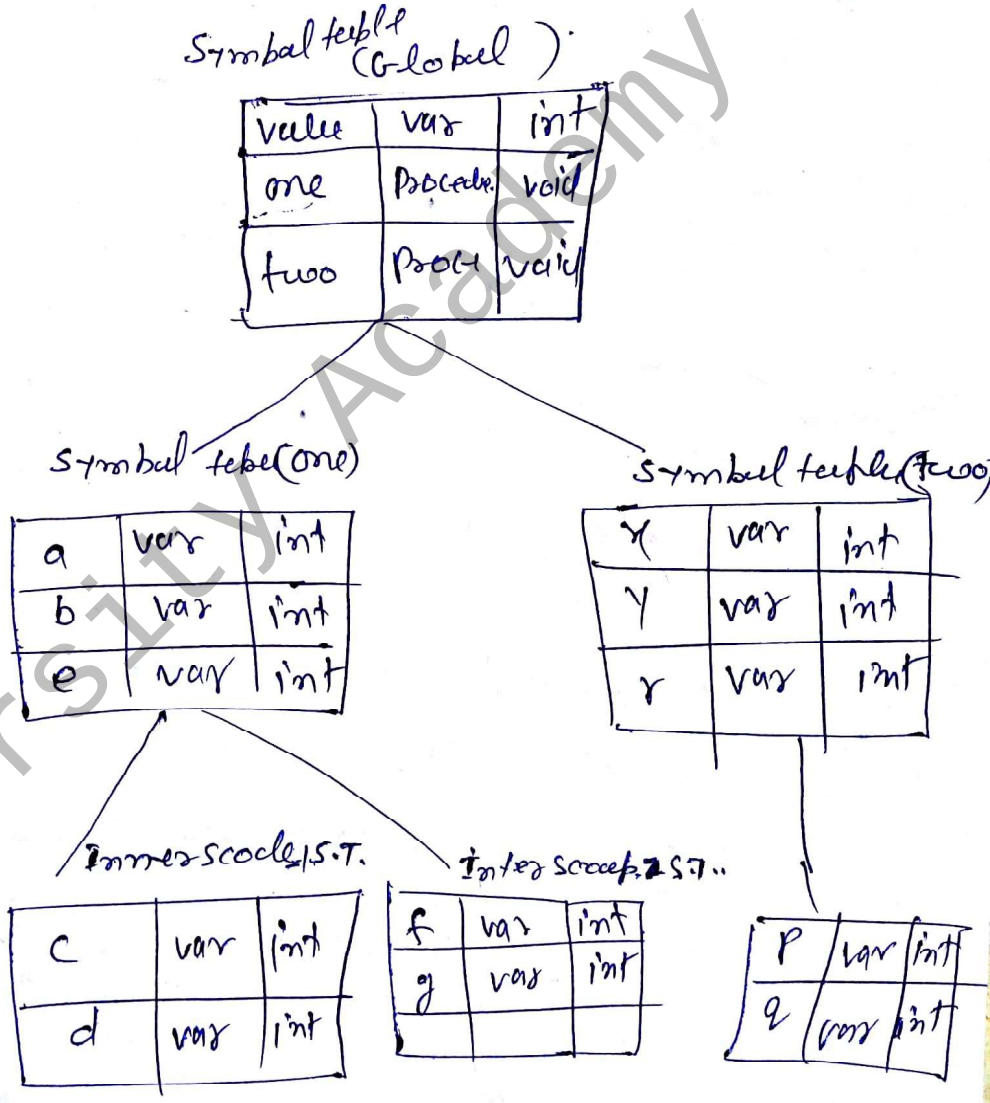
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
c	a	l	c	u	l	a	t	e	$	b	u	m	$	a	$	b	$

operation on Symbal teble.

1. Insert(): eg int a ⇒ Insert(a, int
2. Lookup(): Lookup(Symbal)
3. Scope mgmt:- global, and local: scope:
4. Delete(): delect(Symbal)

examples

```
int value=10;
void one()
  {
    int a
    int b;
      {
        int c
        int d;
      }
   int e;
      {
        int f;
        int g;
      }
  }
void two;
    {
     int x;
     int y;
       { int k;
         int q;
       }
     int r;
    }
```

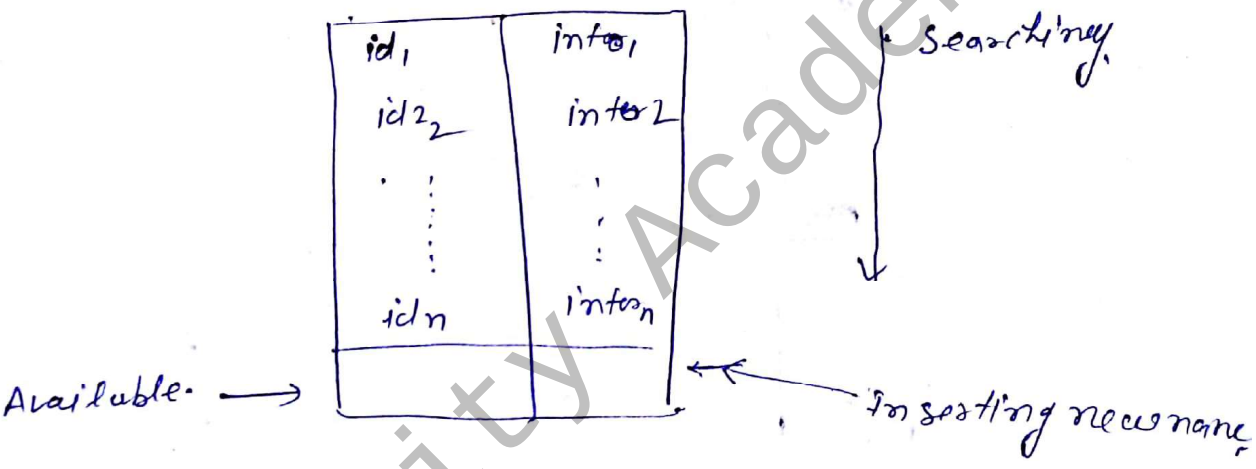


Implimentation of Symbol table / Data Structure for Symbol Table

1. Linear list
2. Self organized list
3. Binary search Tree.
4. Hash tables.

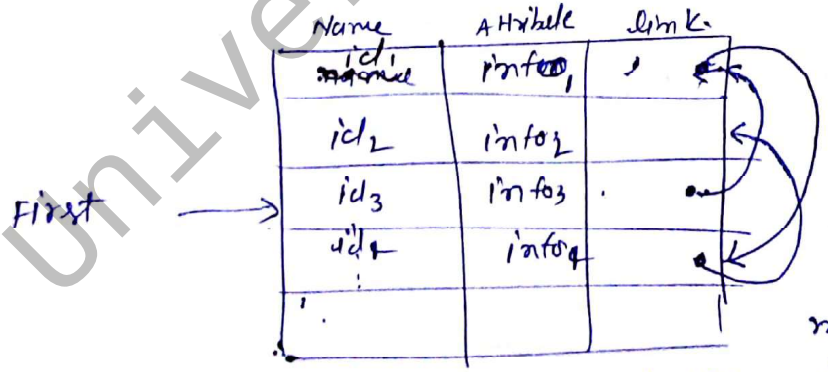
1. Linear list:

Linear list is simplest format of mechanism to implement S.T. An array is used to store name and associated information.



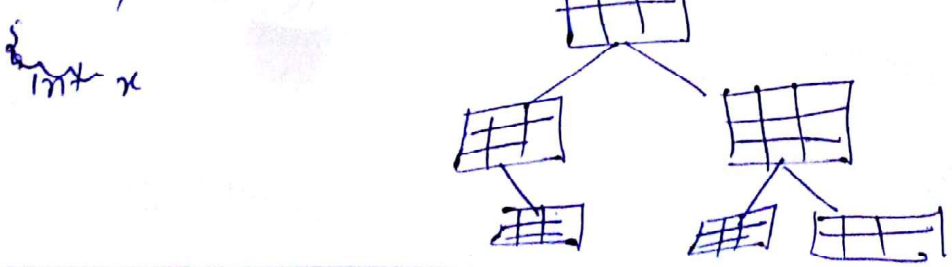
2. Self organizing list:

This Symbol table implementation is using linked list. A link field added to each record.



Fast searching.
most frequently referred name will tend to be found of list.

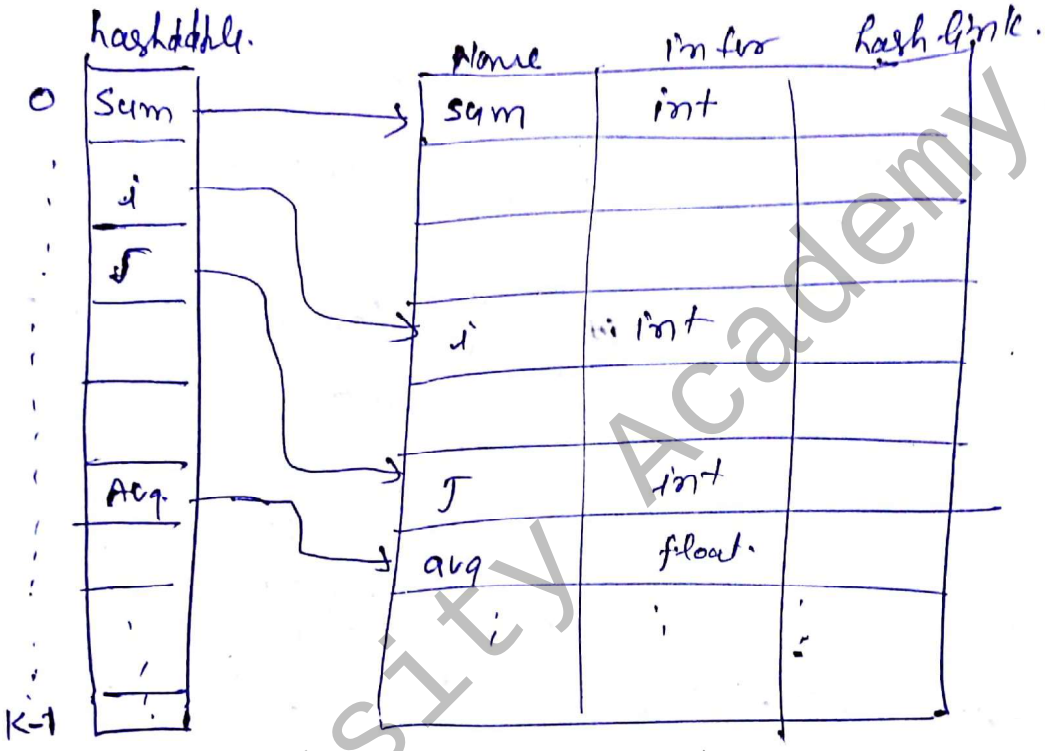
(3.) Binary search Tree:



Hash table-

Hashing is an important technique used to search the record of symbol table.. In hashing scheme two table are maintained.

1. Hash table
2. Symbol table



search by has function 'h' as h(name) that will return integer bet 0 - k-n.

searching of name is so fast in hashing.

Run Time Storage Administration

Compiler Demands for a block of memory to operating system. the Compiler Utilizes this block of memory for running the compiled program. this Block of memory called run time storage.

operating system

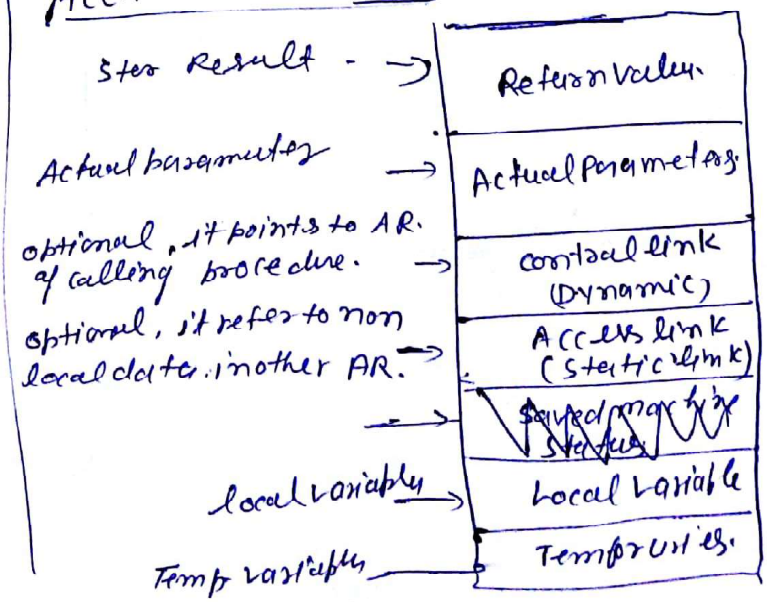
1- Implementation of A Simple Stack Allocation Scheme.
2- Block - Implementation of Block Structured Language.

Code Area has fixed size and hold target code.
Constant, Global variable etc fixed memory.

eg:
```
main()
{
  one();
}
one()
{
  two();
}
two()
{
  ---
}
```

two() A.R
↓
one() AR.
|
main() AR.

Activation Record.



Eg:

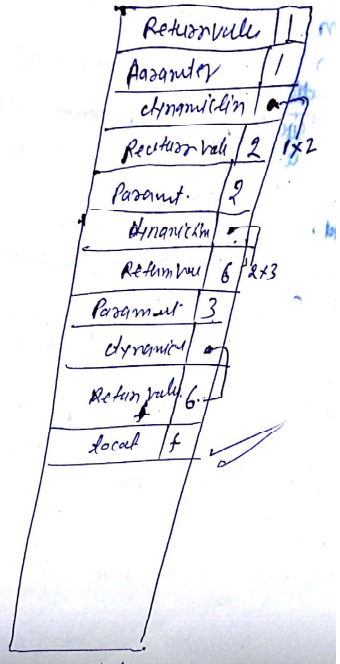
```
main()
{
 int f;
 f=fact(3);
}
int fact(int n)
{
 if(n==1)
  return 1;
 else
  return(n*fact(n-1));
}
```

AR - main

AR - fact(3)

AR - fact(2)

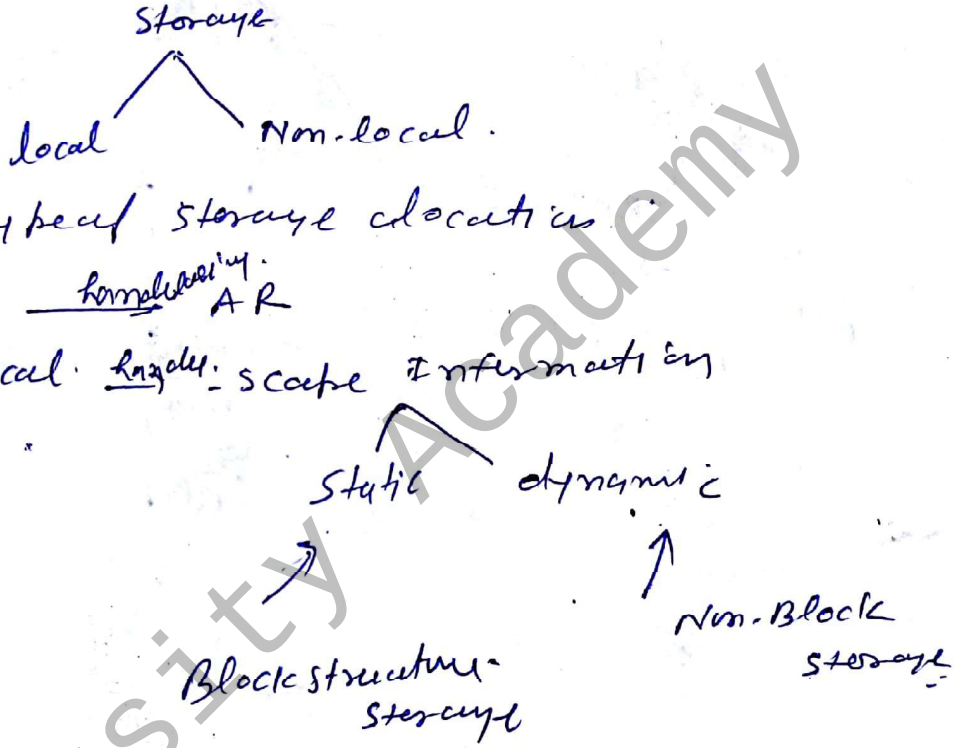
AR - fact(1)



Implementation of Block-structured Language: and Non Block structure

Languages with block structure such as ALGOL or PL/I present certain complexities not found in C.

In Block structured language not only procedure but Block as well may define their own data.



There are two types of storage allocation.

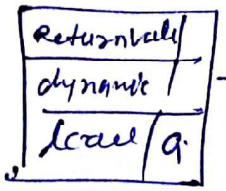
1- local — compile time. AR

2- Non-local. Lazy: scope Information

(1) Local data:

local data can be accessed with help of AR eg:

```
A()
{
    int a;
}
```



(2) Non local data:

A procedure may sometime refer to variables with are not local to it; such variable is called Non-local variable.

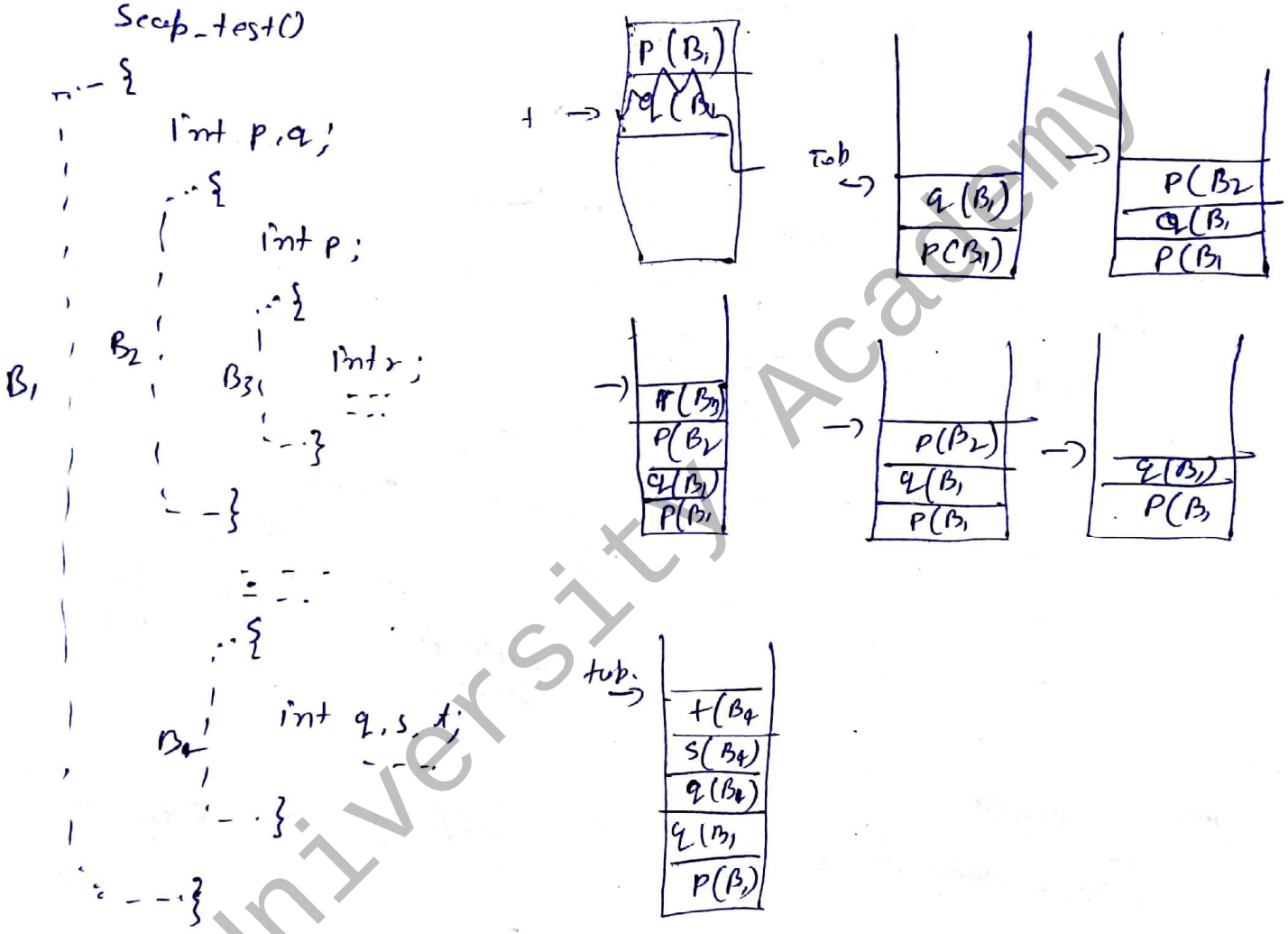
there are two types of scope static and dynamic.

Static scope rule

The static scope rule is also called lexical scope. the languages PASCAL, C, ADA are used static scope. the language are also called Blocks structured Langu:

e.g

```
Scop_test()
{
    int p,q;
    {
        int p;
        {
            int r;
        }
    }
    {
        int q,s,t;
    }
}
```



Dynamic Scope Rule

for non block structured language this dynamic scope allocation rule are used. the dynamic scope rule determines the scope of declaration of the names at such time by considering the current activation. LISP and SNOBAL are the language with the dynamic scope rule.

Error detection and Recovery.

Symbol table.

Error handler

- L.A. → unable to identify tokens.
- S.A → unable to passe the tokens.
- Sem. A. → unable to extract many of Parse tree.
- I.I.O.C. → incompatible type operator used
- C.O. → unable to identify unreachable statement.
- Cons. → if compiler created constant that is too large to fit in the target machine.

Error handler

- detection
- Reporting
- Recovery

error

- Compiler time
  - Lexical Phase error
  - Syntax phase error ①②③④
  - Semantic phase error
- run time.

① Panic mode.
② Phrase level Recovery
③ Error production
④ Global production.

① Lexical phase Error

1. Exceeding length of identifier or numerical constant (C-31, FORTRAN-132)
2. Appearance of illegal character → $, @.
3. unmatched string. or comments. i.e "..." or /*...*/

example

```
void main()
{ int a, @ ; variable declaration */
  a = 10;
  printf("%d", a); }
```

```
void main ()
{ int x=10, y=20;
  char *a;
  a = &x;
  x = 1*ab;
}
```

with variable or number.

Dis: skipped input without checking any additional error.

Panic Recovery: in this method successive characters from the input are removed one at a time until designated set of synchronizing token is found.

Advantage: never go in infinite loop. Dis: skipped input without checking

Syntactic Phase error:

(1) missing paranthesis. eg: printf("hello" ;

(2) Missing operator eg. a*bc

(3) misspelled keyword. eg: switch(ch)
{

}.

4. Colon in place of semicolon. a=1 :-
x=2;

5. Extra blank. /* comment */

Recovery:

① Panic mode :- same as lexical.

② Statement mode Recovery:

(i) when a parser encounter an error, it perform necessary correction on remaining input and parse rest of input.

(ii) take care after not going in infinite Loop

③ Error production: if user has knowledge of common error that can be encountered then, these errors can be incorporated by augmenting the grammar with error production.
difficult to maintain.

4. Global correction.- the parser examines the whole program and tries to find out closest match for it which is error free.

the closest match program has less number of insertion, deletion, and changes of tokens to recover from erroneous input.

due to high time and space complexity it is not implemented practically

Semantic errors.

1- Incompatible type of operands.
2- Undeclared variable.
3- not matching actual argument with formal one.

e.g.

```
int a[10], b;
- - -
- - -
a = b;
```

Recovery: if error "Undeclared Identifier" is encountered then to recover from this a symbol table entry for corresponding identifier is made.

if data type of two operand are incompatible then automatic type conversion done by compiler.

UNIT - V

Code Optimization & code generation.

Code optimization:

The code optimization is required to produce an efficient target code. the improvement over Intermediate code by program transformation called optimizations.

the code optimization are classified into two categories machine - Independent optimization and machine dependent optimization

machine - Independent optimization used to improve the target code without taking into consideration any properties of the target machine. And machine dependent optimization used for register allocation and utilization of machine instruction sequence.

this type of optimization used in code generation.

Three Criteria that we have applied in optimization

1. transformation must preserve the meaning of program.
2. transformation must, on the average, speed up programs by a measurable amount.
3. A transformation must be worth the effort.

Consider the code for quicksort:

```
i=m-1; J=n; v=a[n];
while(1) {
  do i=i+1; while(a[i]<v);
  do J=J-1; while(a[J]>v)
  if(i>=J) break;
  }x=a[i];a[i]=a[J]; a[J]=x;
x=a[i]; a[i]=a[n]; a[n]=x;
```

three address code for above code.

```
1  i = m-1
2  j = n
3  t1 = 4*n
4  v = a[t1]
5  i = i+1
6  t2 = 4*i
7  t3 = a[t2]
8  if t3 < v goto (5)
9  j = j-1
10 t4 = 4*j
11 t5 = a[t4]
12 if t5 > v goto (9)
13 if i >= j goto (23)
14 t6 = 4*i
15 x = a[t6]
16 t7 = 4*i
17 t8 = 4*j
18 t9 = a[t8]
19 a[t7] = t9
20 t10 = 4*j
21 a[t10] = x
22 goto (5)
23 t11 = 4*i
24 x = a[t11]
25 t12 = 4*i
26 t13 = 4*n
27 t14 = a[t13]
28 a[t12] = t14
29 t15 = 4*n
30 a[t15] = x
```

Basic Block:

our first step to break code into basic block that is sequence of consecutive statement which may be entered only at the beginning and executed the sequence of statement without halt.

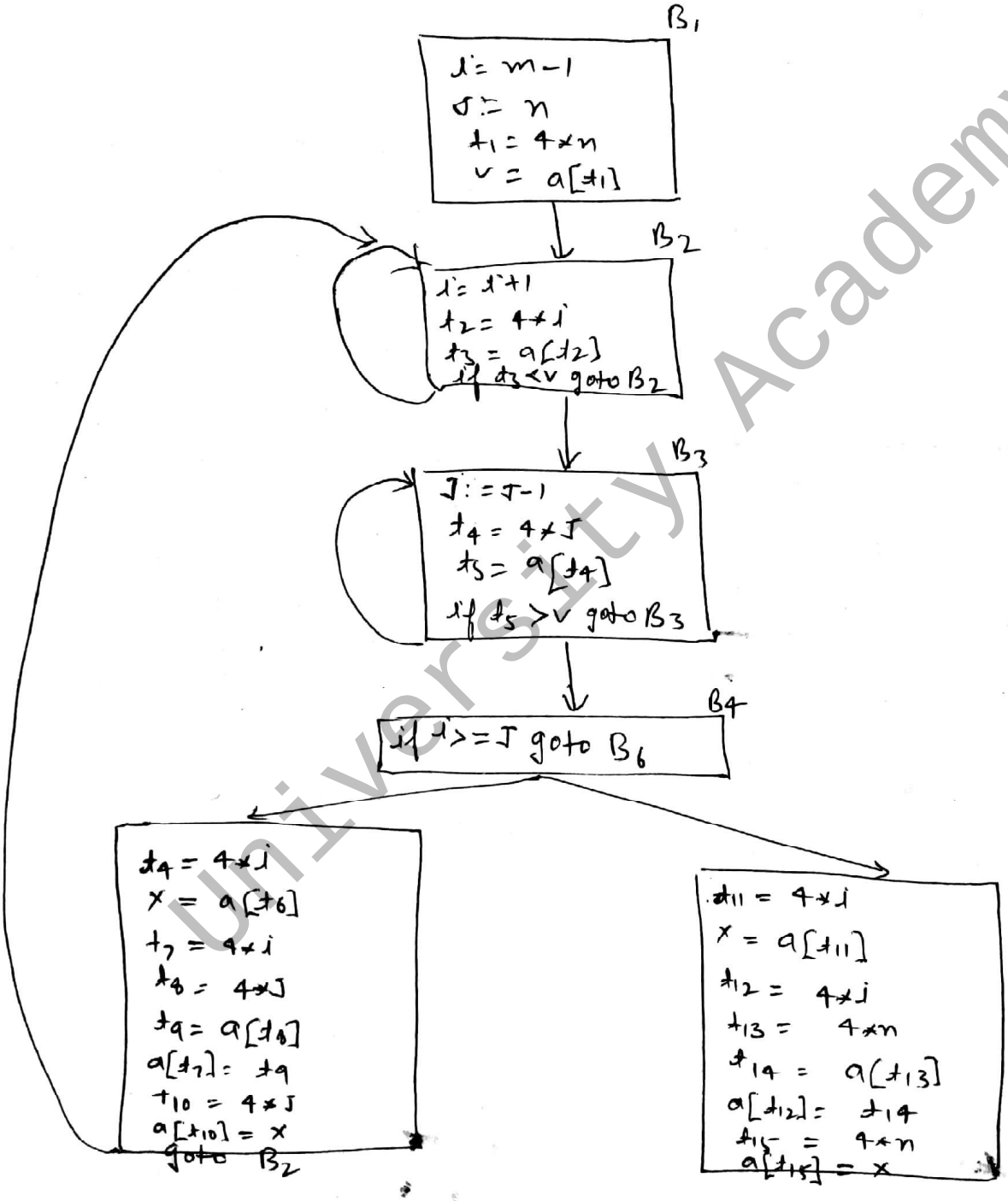
to ~~identify~~ determine the basic block we have to determine leaders. by following rule.

1. the first statement is leaders.
2. conditional or unconditional goto is a leaders
3. A statement immediately follows the conditional goto is a leaders.

thus we have obtain 6 Basic block for above code.

Flow graph. It is useful to show the Reationship between basic block by a directed graph called a flow graph. the nodes of the flow graph are the basic blocks.

Basic block and flow graph for above code:



flow graph

The Principle source of optimization:

The optimization of program is called local if it can be performed by looking only at the statement in a basic block; otherwise it is called global.

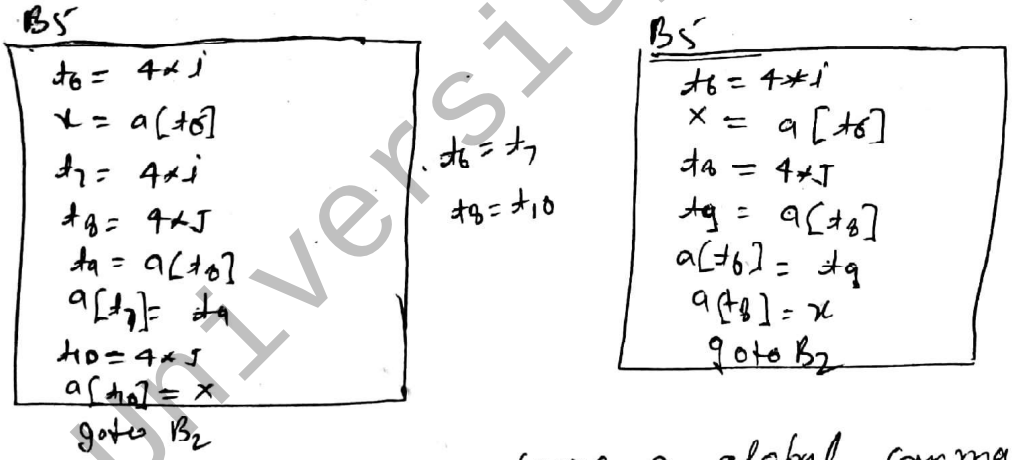
1- Function-Preserving transformation:

There are number of ways in which a compiler can improve a program without changing the function it compute. the function preserving transformation include following optimization:

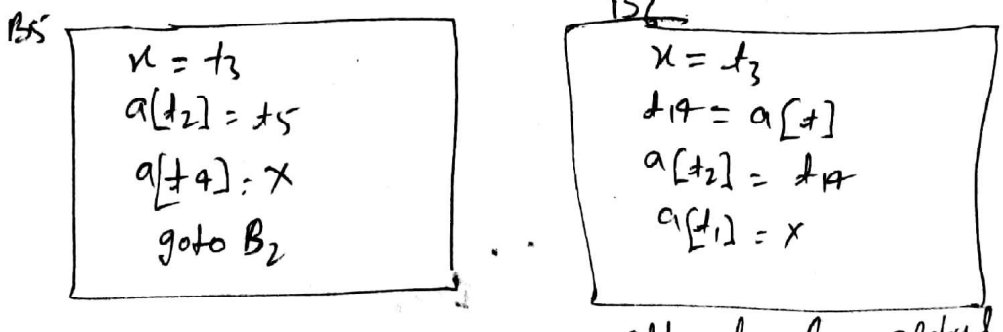
(a) common subexpression elimination
(b) copy propagation.
(c) dead code elimination
(d) constant folding.

(a) Common Subexpression elimination:

we observe local common subexpression in B_5



Now we observe a global common subexpression in B_5 and B_2, B_3 $t_6 = t_2$ and $t_8 = t_4$



after local and global common subexpression

(b) Copy propagation:

Copy propagation in the form of $f := g$:

we observe in B_5

B_5

$x = t_3$
$a[t_2] = t_5$
$a[t_4] = t_3$
goto B_2

after copy propagation

B_5

$x = t_3$
$a[t_2] = t_5$
$a[t_4] = t_3$
goto B_2

(c) Dead-Code Elimination:

A variable is live at a point in a program if its value can be used subsequently otherwise it is dead at that point.

Now again consider B_5

$x = t_3$
$a[t_2] = t_5$
$a[t_4] = t_3$
goto B_2

after dead code elimination

$a[t_2] = t_5$
$a[t_4] = t_3$
goto B_2

that value of x is removed.

(d) Constant folding

At compile time that the value of an expression is a constant and using the constant instead is known as constant folding.

Loop Optimization.

Loop optimization is very important optimization. there are three techniques are important for Loop optimizatn

a- code motion - which move code outside.

b. Induction variable Elimination

c- reduction in strength.

(a) Code motion!

code motion decreases the amount of code in a loop its code motion. this transformation takes an expression that yield the same result indpendent of the number of time a Loop is executed.

exam: while (i<= limit-2)

can be written as

t= limit-2

while (i<=t)

(b) Induction variable and Reduction in strength.

A variable x is called an induction variable of a Loop if the value of variable gets changed every time. It is either decrement or increment by some constant.

consider the Loop in B_3, Note that value of J and t_4 ~~xxx~~ every time the value of J decreases by 1 and t_4 decreases by 4 is assigned to t_4 such identifire are called induction variable.

Strength Reduction:

the strength of certain operators is higher than other such as strength of * is higher than +. in the strength Reduction technique the higher strength operator can be replaced by lower strength operator.

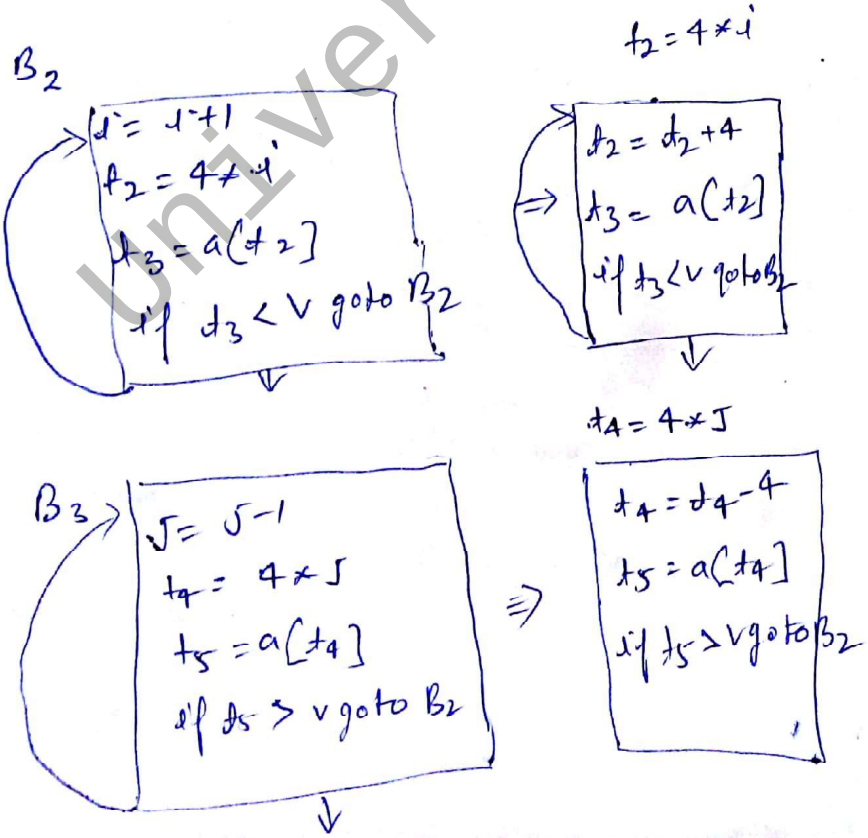
ex.

```
for (i=1; i<=50; i++)
{
    count = i*7;
}
```

it can be Replaced by following code.

```
temp=7
for (j=1; j<=50; j++)
{
    count=temp;
    temp=temp+7
}
```

Now we can apply the Reduction in strength in B_2 and B_3



DAG Representation of Basic Block.

Directed acyclic graph are useful data structure. for Implementing transformation of Basic Block.

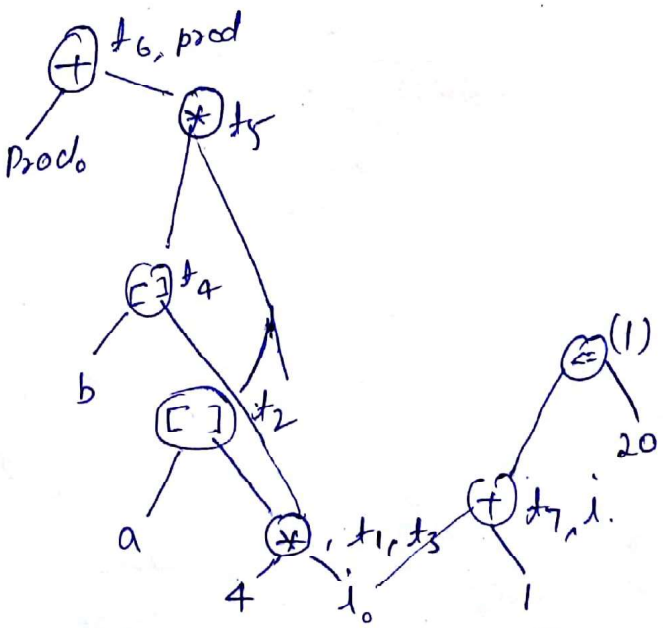
Constructing a DAG from three Address code is a good way to determining common subexpression within Block, determining which name are used inside the block and but evaluated outside the block, and determing which statement of the block could have there computed value used outside the block.

A DAG is Directed Acyclic graph with the following labels on nodes.

1. leaves are labeled by unique identifiers; either variable or constant.
2. Interior node are labeled by an operator.
3. nodes are also given a sequence of identifier for labels.

eg. e..

1 $t_1 = 4 * i$
2 $t_2 = a[t_1]$
3 $t_3 = 4 * i$
4 $t_4 = b[t_3]$
5 $t_5 = t_2 * t_4$
6 $t_6 = prod + t_5$
7 $prod = t_6$
8 $t_7 = i + 1$
9 $i = t_7$
10 if $i <= 20$ goto(1)



Application of DAG

1 - Determining common sub expression and apply DAG based local optimization.

2. Determining which statement of block could have their computed value outside the block.

3 determining which name are used inside the block and computed outside the block.

Code Generation.

Code generation is the final activity of compiler. Basically code generation is a process of creating Assembly / machine language. there are some properties of code generation phase.

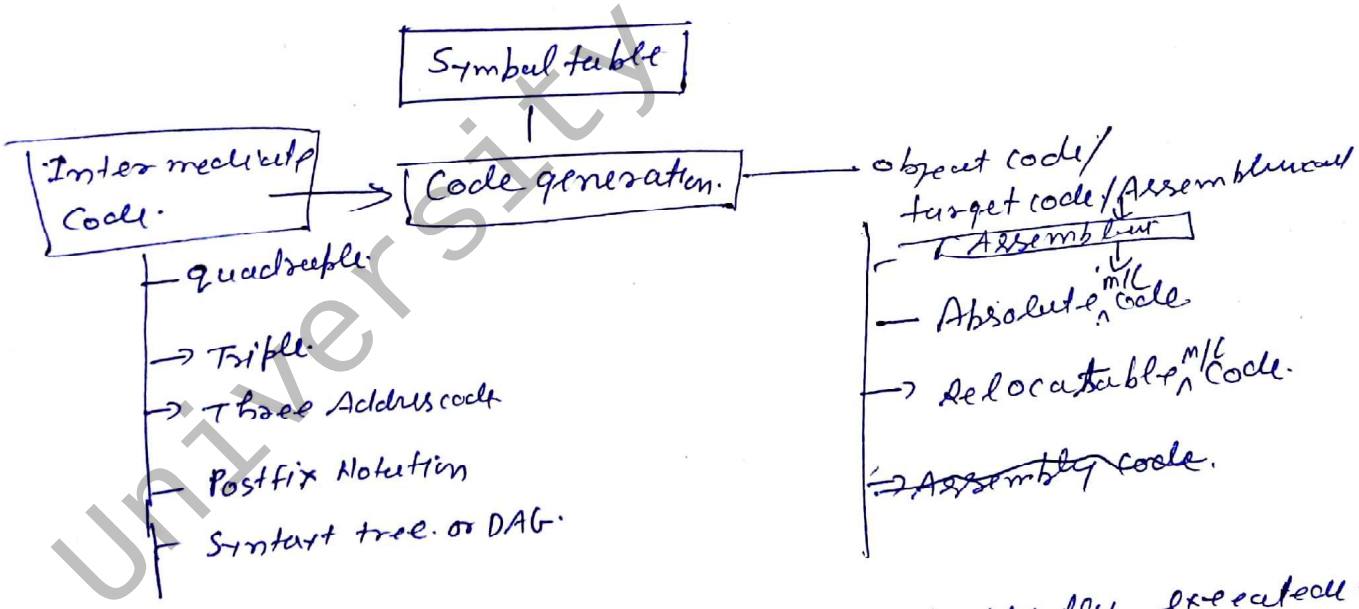
(i) Correctness.

(ii) High quality

(iii) Efficient use of resources of target machine.

(iv) quick code generation.

L.A
S.A.
Sem. A.
I.C.G
C.O. ← machine independent optimization
C.G. ← machine dependent optimization.



Absolute Code: fixed location in memory and immediately executed. "useful for small program". eg. .exe.

Relocatable code: Not a fixed location in memory, code can be placed wherever the linker finds a room. ~~wherever~~ RAM. Useful for commercial compiler. eg. .exe file format.

Assembly Code: easy to generate. uses symbolic code, especially for machine with small memory.

Design Issue

1. Input to the code generater:
2. Target program:
3. memory Management:
4. Instruction Selection.
5. Register Allocation.
6. ~~R~~

$x = y + z$

```
mov Y, R0
ADD Z, R0
mov R0, x.
```

$a = b + c$
$d = a + e$

```
mov b, R0
ADD c, R0
mov R0, a
mov a, R0
ADD e, R0
mov R0, d
```

redundant

```
mov b, R0
ADD c, R0
ADD e, R0
mov R0, d.
```

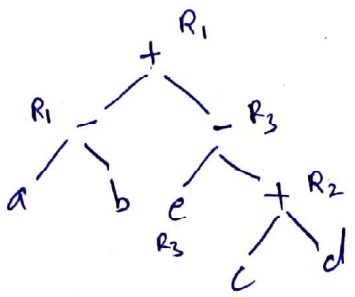
5. Consider Three-Address-Code.

$t = a + b$
$t = t * c$
$t = t / d$

```
mov a, R0
ADD b, R0
MUL c, R0
DIV d, R0
mov R0, t
```

Question: $(a-b) + (e-(c+d))$

Determine the minimum number of Registers needed to compute the expression



$R_1 \leftarrow a$
$R_2 \leftarrow b$
$R_1 \leftarrow R_1 - R_2$
$R_2 \leftarrow c$
$R_3 \leftarrow d$
$R_2 \leftarrow R_2 + R_3$
$R_3 \leftarrow e$
$R_3 \leftarrow R_3 - R_2$
$R_1 \leftarrow R_1 + R_3$

Peephole optimization.

A statement by statement code generation strategy generates target code that contain redundant instruction. the quality of such code is poor. To optimize such target code contain transformations need to be applied on the target code

The peephole optimization is simple and effective technique for locally improving target code.

the peephole optimization can be applied on the target code using following characteristic.

1- Redundant instruction:

(i) mov R0 a
(ii) mov a R0

then eliminate instruction (2).

2. Flow of control optimization.

```
goto test
  . . .
  . . .
test: goto done
  . . .
  . . .
done:
```

⇒

```
goto done
  . . .
test: goto done
  . . .
done:
```

3. Algebraic simplification.

$x = x + 0$
or
$x = x * 1$

} eliminated such type of instruction

4. ~~Algebraic simplification.~~ Reduction strength:

Addition is cheaper than multiplication.
Subtraction is cheaper than division.

5. Machine idioms: Auto increment and Auto decrement.