RAJASTHAN TECHNICAL UNIVERSITY, KOTA

B.Tech Computer Science and Engineering (Artificial Intelligence)

RAJASTHAN TECHNICAL UNIVERSITY, KOTA

Syllabus

III Year-V Semester: B.Tech. Computer Science and Engineering

5CAI4-02: Compiler Design

Credit: 3 **Max. Marks: 100(IA:30, ETE:70)**

3L+0T+0P **End Term Exam: 3 Hours**

SN	Contents	Hours
1	**Introduction:** Objective, scope and outcome of the course.	01
2	**Introduction:** Objective, scope and outcome of the course. Compiler, Translator, Interpreter definition, Phase of compiler, Bootstrapping, Review of Finite automata lexical analyzer, Input, Recognition of tokens, Idea about LEX: A lexical analyzer generator, Error handling.	06
3	**Review of CFG Ambiguity of grammars:** Introduction to parsing. Top down parsing, LL grammars & passers error handling of LL parser, Recursive descent parsing predictive parsers, Bottom up parsing, Shift reduce parsing, LR parsers, Construction of SLR, Conical LR & LALR parsing tables, parsing with ambiguous grammar. Operator precedence parsing, Introduction of automatic parser generator: YACC error handling in LR parsers.	10
4	**Syntax directed definitions;** Construction of syntax trees, S-Attributed Definition, L-attributed definitions, Top down translation. Intermediate code forms using postfix notation, DAG, Three address code, TAC for various control structures, Representing TAC using triples and quadruples, Boolean expression and control structures.	10
5	**Storage organization;** Storage allocation, Strategies, Activation records, Accessing local and non-local names in a block structured language, Parameters passing, Symbol table organization, Data structures used in symbol tables.	08
6	**Definition of basic block control flow graphs;** DAG representation of basic block, Advantages of DAG, Sources of optimization, Loop optimization, Idea about global data flow analysis, Loop invariant computation, Peephole optimization, Issues in design of code generator, A simple code generator, Code generation from DAG.	07
	Total	42

Office of Dean Academic Affairs
Rajasthan Technical University, Kota

RAJASTHAN TECHNICAL UNIVERSITY, KOTA

B.Tech Computer Science and Engineering (Artificial Intelligence)

Swami Keshvanand Institute of Technology,Management&Gramothan,Jaipur

Programme: B.Tech. CS(AI)
Semester: V
CourseName(CourseCode):COMPILER DESIGN (5CAI4-02))

CourseOutcomes

Aftercompletionofthiscourse,students willbeable to–

5CAI4-02.1	**Discuss** the major phases of compilers and to **use** the knowledge of Lex tool
5CAI4-02.2	**Develop** the parsers and **experiment** the knowledge of different parsers design without automated tools.
5CAI4-02.3	**Describe** intermediate code representations using syntax trees and DAG's as well as **use** this knowledge to generate intermediate code in the form of three address code representation.
5CAI4-02.4	**Classify** various storage allocation strategies and **explain** various data structures used in symbol tables
5CAI4-02.5	**Summarize** various optimization techniques used for data flow analysis and **generate** machine code from the source code of an ovel language.

Name of Faculty:
(Signature)

Verified by Course Coordinator

Signature
(Name:.....)

Verified by Verification and Validation Committee, DPAQIC

Signature
(Name:.....)

RAJASTHAN TECHNICAL UNIVERSITY, KOTA

B.Tech Computer Science and Engineering (Artificial Intelligence)

Swami Keshvanand Institute of Technology,Management&Gramothan,Jaipur

CO	Outcomes	Bloom's Level	PO Indicators	PSO Indicators
Upon successful completion of this course, students should be able to:				
5CAI04-02.1	**Discuss** the major phases of compilers and to **use** the knowledge of Lex tool.	2,3	PO(1.3.1, 1.4.5,2.1.1, 2.1.2, 2.1.3, 2.2.1, 2.2.2, 2.2.3, 2.2.4, 5.1.1, 5.1.2, 5.2.1, 12.1.1, 12.1.2, 12.2.1, 12.2.2)	PSO 1.1.1 , 1. 1.3
5CAI04-02.2	**Develop** the parsers and **experiment** the knowledge of different parsers design without automated tools.	3,6	PO(1.3.1,1.4.1, 2.1.1, 2.1.2,2.1.3, 2.2.1, 2.2.2, 2.2.3, 2.2.4, 2.3.2, 2.4.3, 2.4.4, 3.1.1, 3.1.6, 3.2.3,3.4.2, 4.1.1, 4.1.2, 4.2.1, 4.3.1, 4.3.2, 4.3.3, 4.3.4, 12.1.1, 12.1.2, 12.2., 12.2.2)	
5CAI04-02.3	**Describe** intermediate code representations using syntax trees and DAG's as well as **use** this knowledge to generate intermediate code in the form of three address code representation.	1,3	PO(1.3.1,1.4.1,2.1.1, 2.1.2,2.1.3, 2.2.1, 2.2.2, 2.2.3, 2.2.4, 2.3.2, 2.4.3, 2.4.4, 3.1.1, 3.1.6, 3.2.1, 3.2.3,3.4.2, 4.1.1, 4.1.2, 4.2.1, 4.3.1, 4.3.2, 4.3.3, 4.3.4, 12.1.1, 12.1.2, 12.2.1, 12.2.2)	PSO1.1.1

UNIT-I

COURSE DESCRIPTION

The goal of the course is to provide an introduction to the system software like assemblers, compilers, and macros. It provides the complete description about inner working of a compiler. This course focuses mainly on the design of compilers and optimization techniques. It also includes the design of Compiler writing tools. This course also aims to convey the language specifications, use of regular expressions and context free grammars behind the design of compiler.

COURSE OBJECTIVES

The objective of this course is to:

1. Provide an understanding of the fundamental principles in compiler design
2. Provide the skills needed for building compilers for various situations that one may encounter in a career in Computer Science.
3. Learn the process of translating a modern high-level language to executable code required for compiler construction.

COURSE OUTCOMES

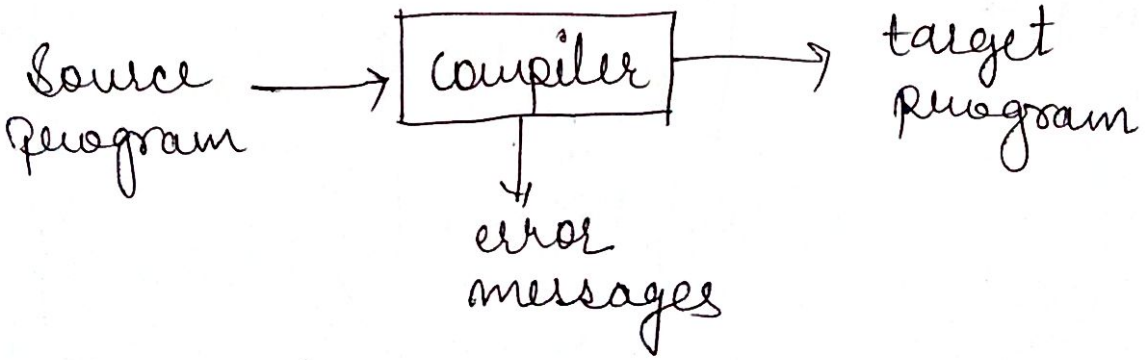
At the end of course students will be able to:

1. Discuss the major phases of compilers and to use the knowledge of Lex tool
2. Develop the parsers and experiment the knowledge of different parsers design without automated tools.
3. Describe intermediate code representations using syntax trees and DAG's as well as use this knowledge to generate intermediate code in the form of three address code representation.
4. Classify various storage allocation strategies and explain various data structures used in symbol tables
5. Summarize various optimization techniques used for dataflow analysis and generate machine code from the source code of a novel language.

Introduction to compilers:-

Translators:- a translator is a generic term that could refer to a compiler, assembler or interpreter; anything that converts higher level code into another high-level code or lower-level language.

Compilers:- is a program that reads a program written in one language - the source language - and translates it into an equivalent program in another language - the target language. In this process, compiler reports to its user the presence of errors in source code.

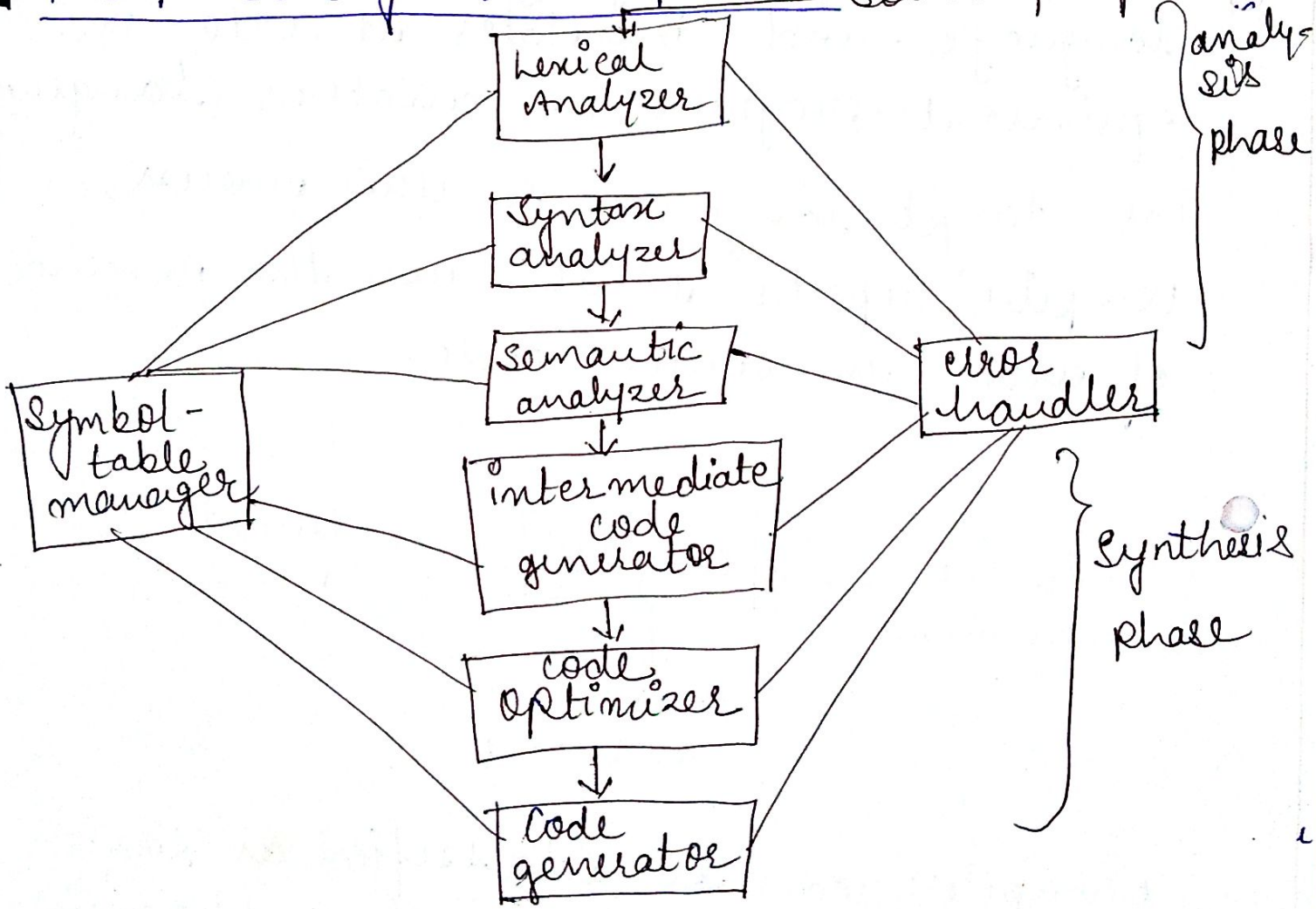


* compilers can be classified as single-pass, multi-pass, load-and-go, debugging or optimizing etc.

Interpreter:-

An interpreter translates code like a compiler but reads the code and immediately executes on that code and therefore is initially faster than a compiler. Interpreters execute a single line of code at a time.

- **The phases of a compiler:-**



The Analysis Phase:-

1) Lexical Analysis:-

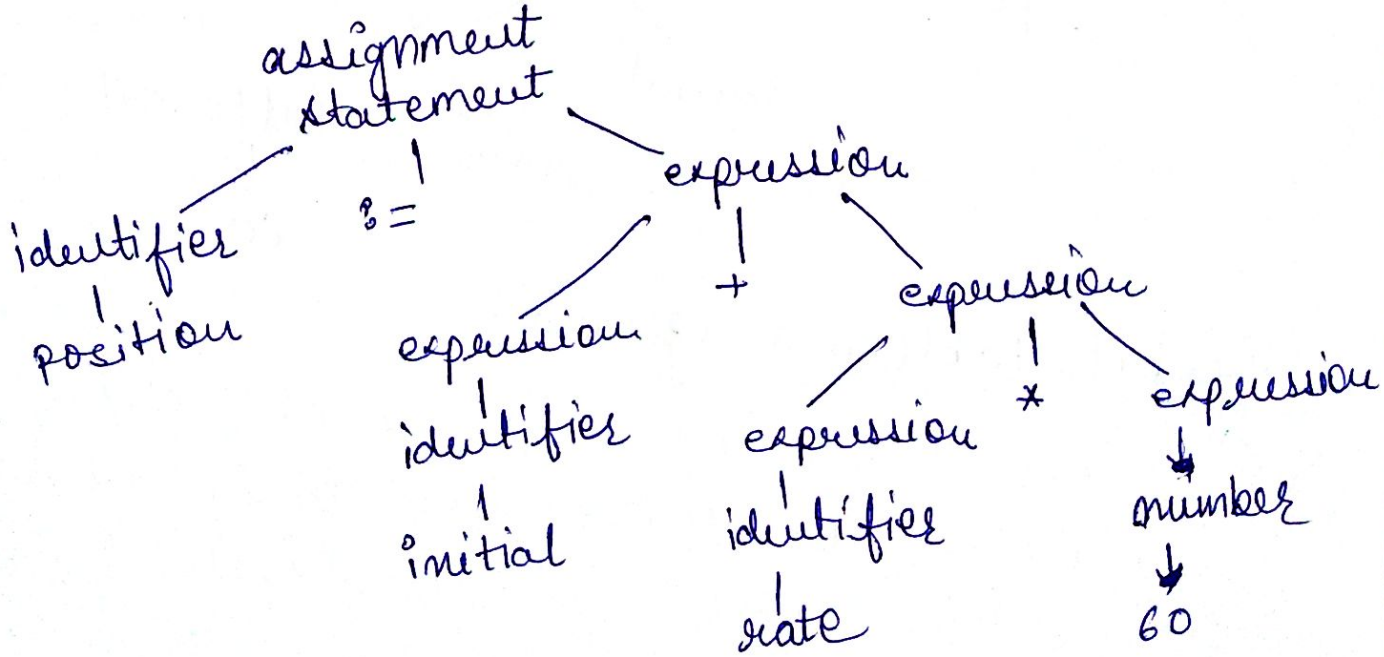
- Also called linear analysis.

Ex position := initial + rate * 60

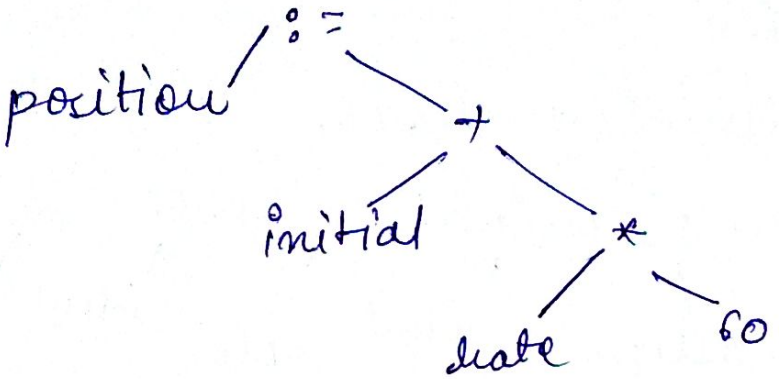
identifier — position
assignment symbol — :=
ident — initial
plus sign — +
ident — rate
multiply sign — *
number — 60

2) Syntax Analysis:-

- Also called hierarchical analysis / parsing.
- It involves grouping the tokens of the source program into grammatical phrases that are used by the compiler to synthesize output.
- Represented by parse tree.



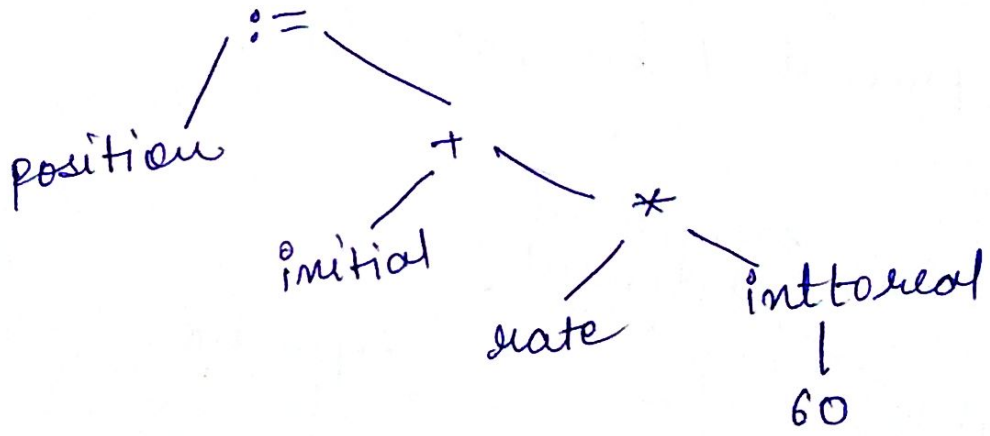
- A more common internal representation of above tree,



3) Semantic Analysis:-

- Checks the source program for semantic errors and type checking.

Ex: Assume in above example all operands/identifiers are in real and 60 is an integer.



Symbol-table management:-

- A symbol table is a data structure containing a record for each identifier, with fields for the attributes of the identifier.

- fields can store attributes like storage allocated for identifier, its type, its scope, and in case of procedures it can store number/type of arguments etc.

- Symbol table allows us to find the record for each identifier quickly and to store/retrieve data from record quickly.

- when an identifier is detected in source program by lexical analyzer, the identifier is entered into the Symbol table. The attributes of identifier is generally not known at this time.

Ex:- (Pascal) var position, initial,
rate : real;

when lexical analyzer identifies the variable 'position', its data type will not known and entered into the symbol table by remaining phases.

Error Detection and Reporting:-

- Each phase can encounter errors.
- Syntax & semantic analysis phases usually handle a large fraction of errors.
- lexical phase can detect errors where the characters remaining in the I/P do not form any token.
- Errors where token stream violates the structure rules of the language are determined by the syntax analysis phase.
- Semantic phase detect constructs that have right syntactic structure but no meaning to the operation involved.
  Ex:- Add two identifiers. One is array and one is procedure.

Intermediate Code Generation:-

- The intermediate representation should have two important properties:-
  1) it should be easy to produce
  2) it should be easy to translate into the target program.
- It can has multiple forms/methods.
  Figure on page 7 has intermediate form called "three-address code".

Position := initial + rate * 60

↓

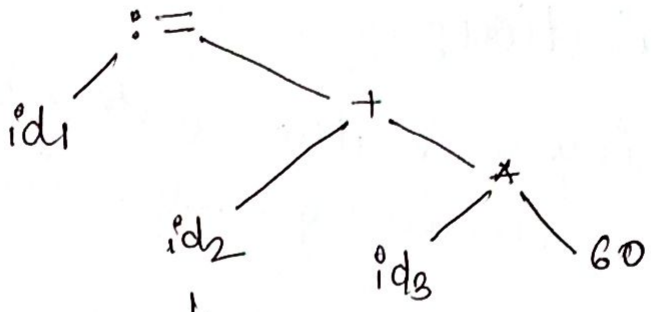
lexical analyzer

↓

$id_1 := id_2 + id_3 * 60$

↓

Syntax Analyzer

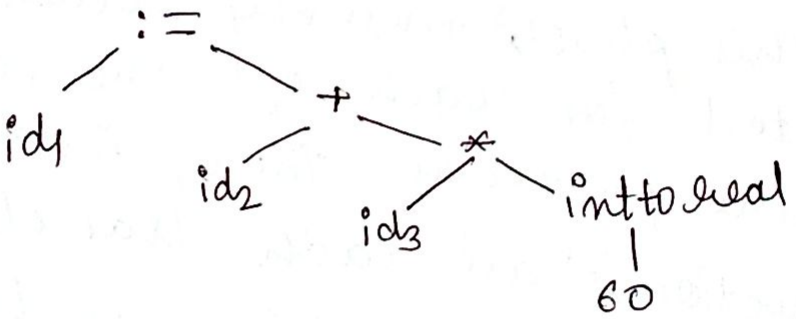


Symbol Table

1	Position	....
2	initial	.....
3	rate	...
4		

↓

Semantic Analyzer



↓

intermediate code generator

temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 = temp3

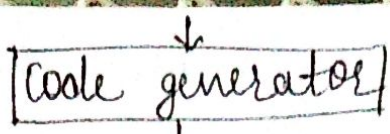
↓

code optimizer

↓

temp1 := id3 * 60.0
id3 := id2 + temp1

↓




```
MOVF  id3, R2
MULF  #60.0, R2
MOVF  id2, R1
ADDF  R2, R1
MOVF  R1, id1
```

Code Optimization:-

- Attempts to improve the intermediate code, so that faster-running machine code will result.

Code generation:-

- In this phase, memory locations are selected for each of the variables used by the program. Then, intermediate instructions are each translated into a sequence of machine instructions that perform the same task.

■ Bootstrapping:-

- Bootstrapping is the technique for producing a self-compiling compiler, that is compiler written in the source programming language that it intends to compile.

- Using the facilities offered by a language to compile itself is the essence of bootstrapping.
- Use of bootstrapping to create compilers and to move them from one machine to another by modifying the back end.
- For bootstrapping purposes,
  - S → source language
  - T → Target language
  - I → in which language compiler is written

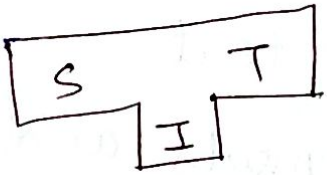
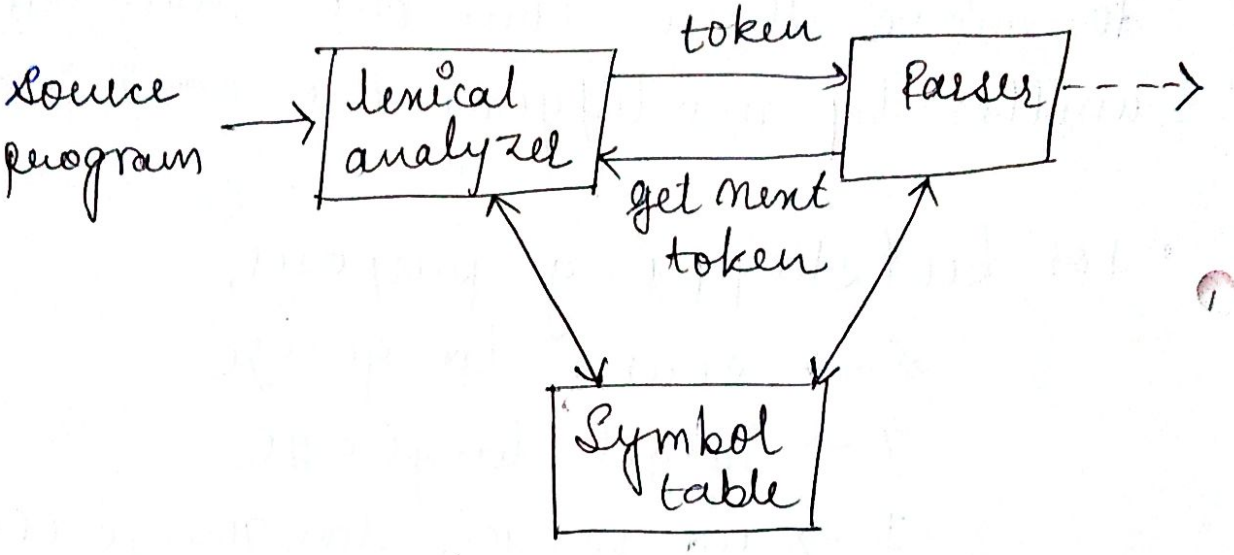


Fig: T-Diagram

- S, T & I can be different.
- A compiler may run on one machine and produce target code for another machine. Such a compiler is often called a cross-compiler.

Lexical Analysis: – (Role of lexical analyzer)

- Its main task is to read the i/p characters and produce as o/p a sequence of tokens that the parser uses for syntax analysis.



Lexical analyzer's tasks:-

1) Reads source text
2) Strip out comments and white spaces from source program
3) correlating error messages from the compiler with the source program. (Keep track of newline character for giving line number in error messages)
4) Makes a copy of source program with the error messages marked in it.
5) Preprocessor functions may be implemented

as lexical analysis takes place.

Issues in lexical analysis:-

- Reasons for separating analysis phase in lexical analysis and parsing,
  - → It simplifies the design (lexical analyzer already eliminate the comments / white spaces before passing)
  - → Compiler efficiency is improved (Specialized buffering techniques for reading i/p characters and processing tokens can significantly speed up the performance of a compiler)
  - → compiler portability is enhanced (I/P alphabet peculiarities and other device-specific anomalies can be restricted to the lexical analyzer.)

Tokens, Patterns, Lexemes:-

- Following constructs are treated as tokens: keywords, operators, identifiers, constants, literal strings and punctuation symbols.

- Set of strings described by a rule called a pattern associated with the token.
- A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.

Ex:- const pi = 3.1416;

- pi is a lexeme for the token "identifier".

Token	Sample lexemes	Informal description of pattern.
1) const	const	const
2) if	if	if
3) relation	<, <=, =, <>, >, >=	any kind of relational operators
4) id	pi, count, D2	letter followed by letters & digits.
5) num	3.14, 0, 6.02	any numeric constant

Attributes for Tokens:-

- When more than one pattern matches a lexeme, the lexical analyzer must provide additional information about it, so that additional attributes are provided to token.
  → so that code generator can know what string was actually matched.

- Ex:- the pattern num matches both the string 0 and 1.

int a = 2
float a = 2
these additional attribute (int, float) tells which kind of no a is.

Ex:- Tokens and associated attributes - values,

$$E = M * C ** 2$$

<id, pointer to symbol-table entry for E>
<assign-op,>
<id, pointer ——— M>
<mult-op, >
<id, pointer ——— C>
<exp-op, >
<num, integer value 2>

Lexical errors:-

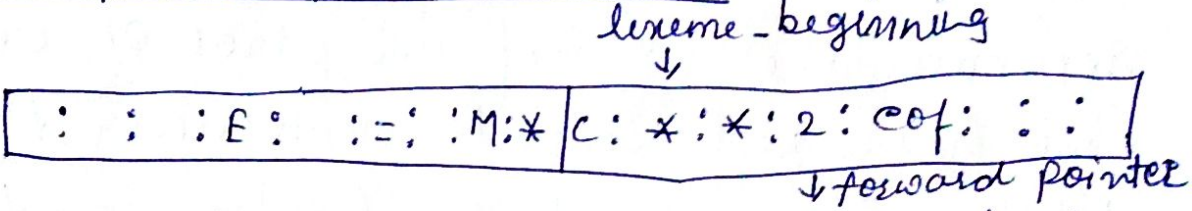
- Few errors are discern at the lexical level alone.

Ex `fi(a==f(x))...`

- Lexical analyzer will return 'fi' as identifier.
- Suppose a situation in which lexical analyzer is unable to proceed bcoz none of the patterns for tokens matches a prefix of the remaining input.
- possible error-recovery actions,
  - → ~~deleting~~
  - → "panic mode" recovery method, we delete successive characters from the remaining i/p until the lexical analyzer can find a well-formed token.
  - → delete an extra char
  - → insert a missing char
  - → replace an incorrect char by correct char
  - → transposing two adjacent char.

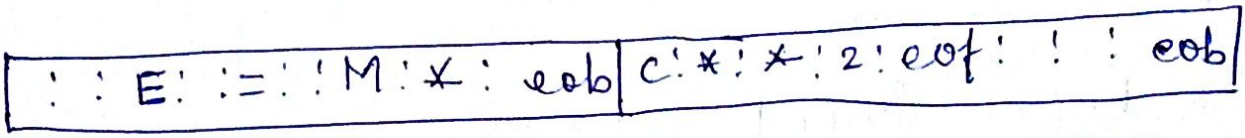
Input Buffering:-

Two buffer input scheme:-



- We divide the buffer in two halves, each having capacity of N. N is the no of chars on one disk block, eg, 1024 or 4096.
- Read N chars ~~or~~ into each half.
- Two pointers lexeme_beginning and forward. The string of chars b/w two pointers is the current lexeme.
- If the forward pointer is at the end of first half then right buffer is filled with N chars and if forward pointer is at the end of second half then left buffer is filled with N chars.
- Problem is that each time, (while moving forward) we have to check that we have not moved off one half of the buffer, so that we must reload other half. This will increase the overload. So we use a special sentinel char ~~to check whether~~

at end of ~~one~~ each half ~~arrived~~ ~~or~~ ~~not~~.



- we can take as eof in place of eob, but we have put special mechanism to check whether it is end of file or end of buffer.

specification of tokens:-

- Regular expressions are used to specifing patterns.

String and Languages:-

- Alphabet or char class can denotes any finite set of symbols.
  ex:- {0,1} binary class
- A string over some alphabet is a finite sequence of symbols drawn from that alphabet.

  ex:- Symbol → s
  Empty string → ∈
  length of string → |s|

- The term language denotes any set of strings over some fined alphabet.

= Abstract languages → ϕ

empty set → $\{\epsilon\}$

$S\epsilon = \epsilon S = S$

= Ex:- x = dog , y = house

xy = doghouse

Operation on languages:-

1) union of L and M (L ∪ M)

$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$

2) concatenation of L and M (LM)

$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$

3) Kleene closure of L (L^*)

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

4) Positive closure of L (L^+)

$$L^+ = \bigcup_{i=1}^{\infty} L^i$$

Regular expressions:-

Ex:- identifier:- a letter followed by zero or more letters or digits.

letter (letter | digit)*

Regular Definitions:-

- We may wish to give names to regular expressions and to define regular expressions using these names as if they were symbols.

If Σ is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form,

$$d_1 \rightarrow r_1$$
$$d_2 \rightarrow r_2$$
$$\vdots$$
$$d_n \rightarrow r_n$$

Eg:- letter $\rightarrow$ A | B | ... | Z | a | b | ... | z

digit $\rightarrow$ 0 | 1 | ... | 9

id $\rightarrow$ letter (letter | digit) *

Notational shorthands:-

1) One or more instances:- + is used for this.

2) Zero or one instance:- ? is used for this.

3) Character classes:-

Eg:- [A-Za-z] [A-Za-z0-9]*

Non regular Sets:-

- Some languages cannot be described by any regular expression.

Ex:- Repeating strings cannot be described by regular expressions.

Pushdown Automata → $\{ww \mid w \text{ is a string of 0's \& 1's}\}$

- Regular expressions can be used to denote only a fixed no of repetitions

Recognition of Tokens:-

Ex:- Language →

stmt → if expr then stmt
| if expr then stmt else stmt
| ϵ

expr → term relop term
| term

term → id | num

Regular definitions

keyword

if → if
then → then
else → else

lexeme

relop → < | <= | = | <> | > | >=
id → letter (letter | digit)*
num → digit+ (. digit+)? (E(+|-)? digit+)?

- In addition, we assume lexemes are separated by white space,

  delim → blank | tab | newline
  ws → delim +

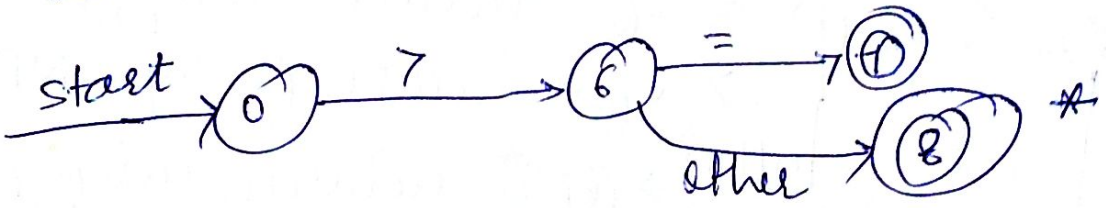
- Our goal is to construct a lexical analyzer that will isolate the lexeme for the next token in the i/p buffer and produce as o/p a pair consisting of the appropriate token and attribute value, using the translation table,

Regular Expression	Token	Attribute Value
ws	—	—
if	if	—
then	then	—
else	else	—
id	id	pointer to table entry
num	num	"
<	relop	LT
<=	"	LE
=	"	EQ
<>	"	NE
>	"	GT
>=	"	GE

Transition Diagram:-

- Transition diagrams are intermediate step in the construction of a lexical analyzer.
- These diagrams depict the actions that take place when a lexical analyzer is called by the parser to get the next token.
  - → positions in diagram – states (circles)
  - → connected by – edges (arrows)
- we assume these diagrams are deterministi

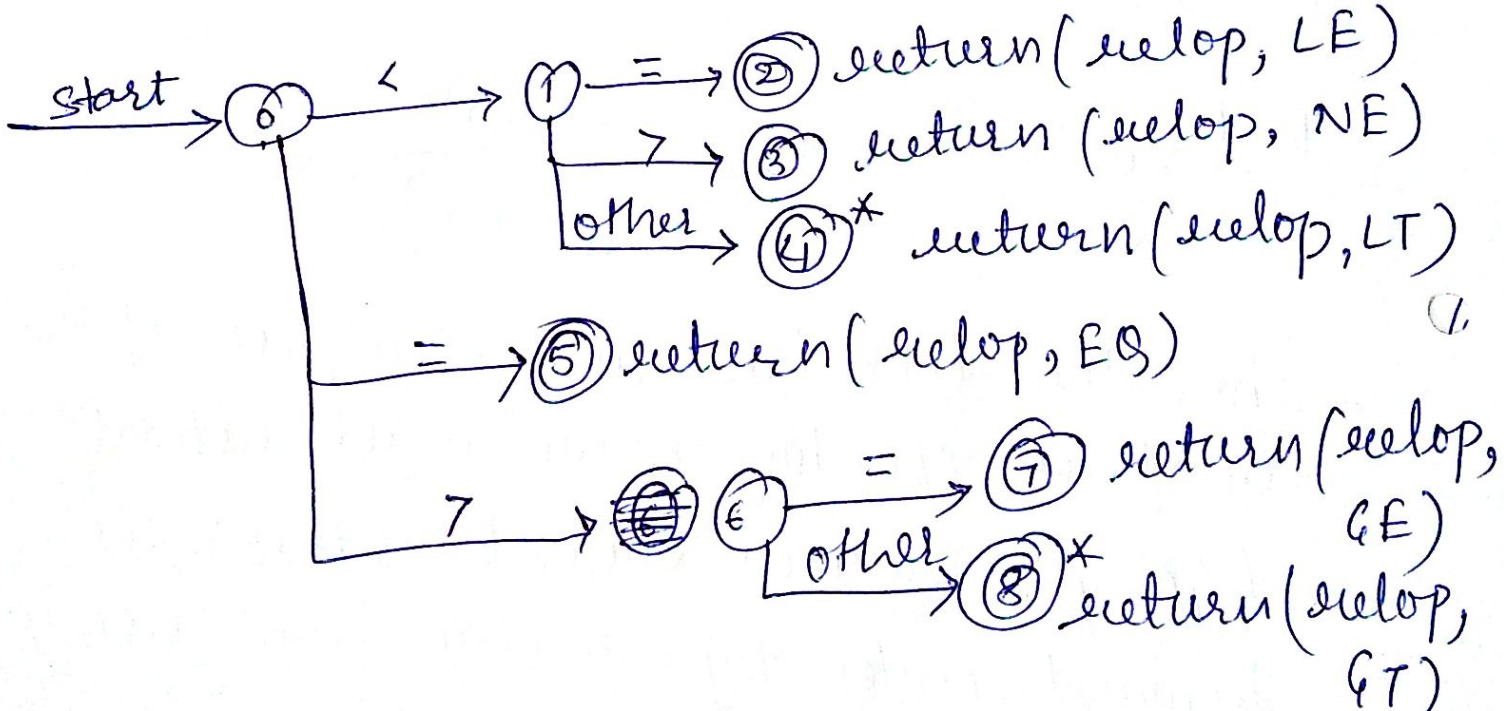
e.g. >=



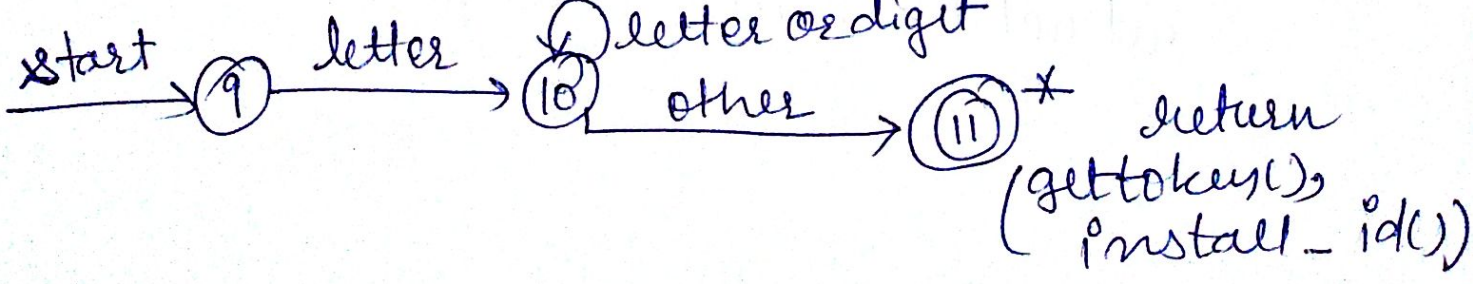
- Since the extra char is not a part of the relational operator >, we must retract (going to previous state, backtract) the forward pointer by one char. we use a * to indicate states on which this i/p retraction must take place.

• In general, there may be several transition diagrams, each specifing a group of token. If failure occurs while we are following one transition diagram, then we retract the forward pointer to where it was in the start state of this diagram, and activate the next transition diagram. (Forward pointer is retracted to the position marked by the lexeme - beginning pointer).

Ex :- Transition diagram for relop,



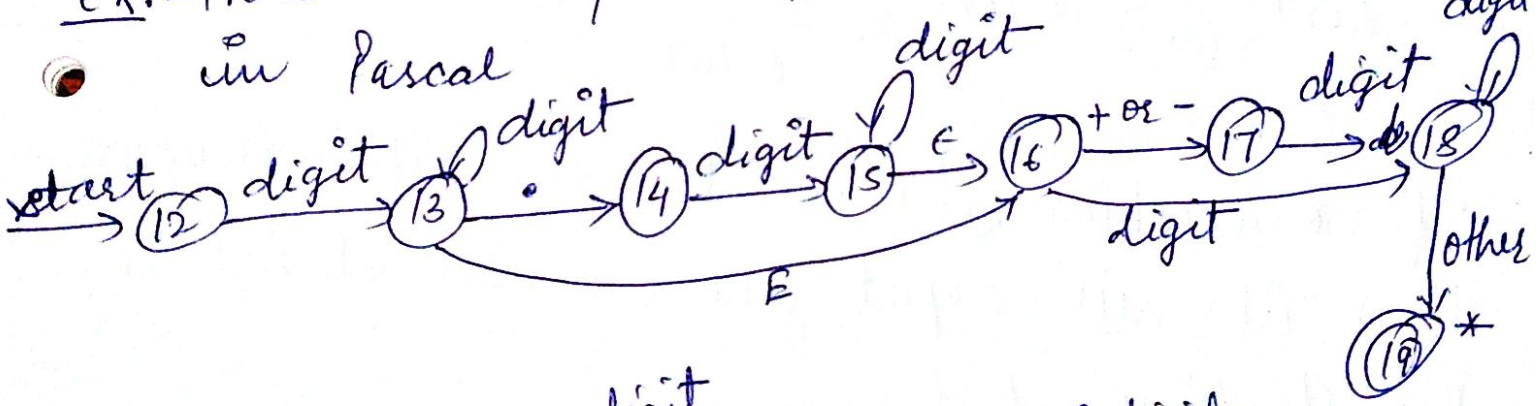
Ex : Transition diagram for keywords & identifiers

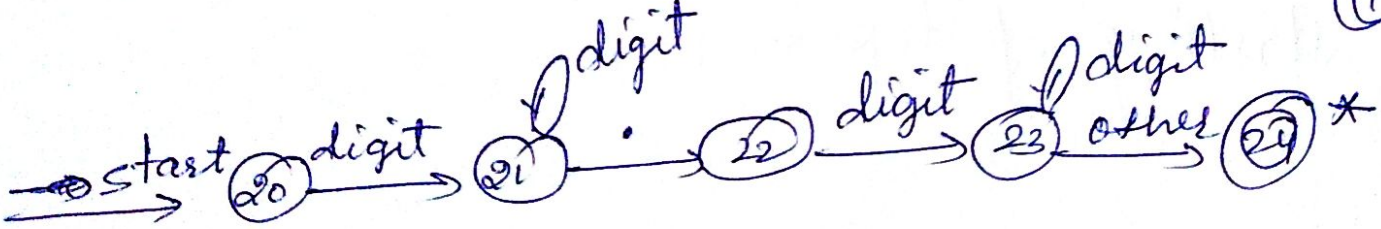


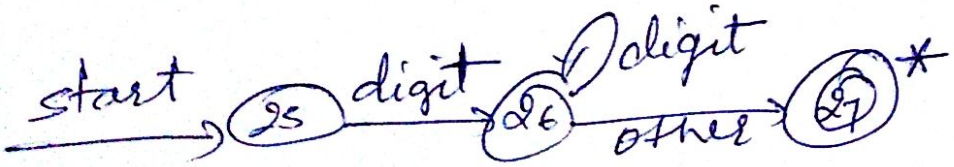
- In starting all keywords / identifiers are detected by above transition diagram. Then matched with symbol table, if exist then it will be a keyword otherwise identifier. All keywords are already placed in symbol table before i/p any char.

- install-id() returns 0 if i/p is keyword according to symbol table. Otherwise it will returns a pointer to the symbol table entry.
- gettoken() returns the token if i/p is token, otherwise token id is returned.

Ex:- Transition diagrams for unsigned numbers in Pascal







- **Error-handling:-** Info about the language that is not in the regular definition of the tokens can be used to pinpoint errors in i/p. Ex:- i/p → 1.0<x,
  we fail in states 14 and 22
  ~~with next i/p char <~~
- Rather than returning the no 1, we may wish to report an error and continue as if the i/p were 1.0<x. Such knowledge can be used to simplify the transition diagrams, bcoz error-handling may be used to recover from the situations that would otherwise lead to failure.

Ex:- white space

If we combine above all transition diagrams, then it will depict the example stated in the starting of topic.

Finite Automata:-

Finite automaton → Generalized transition diagram

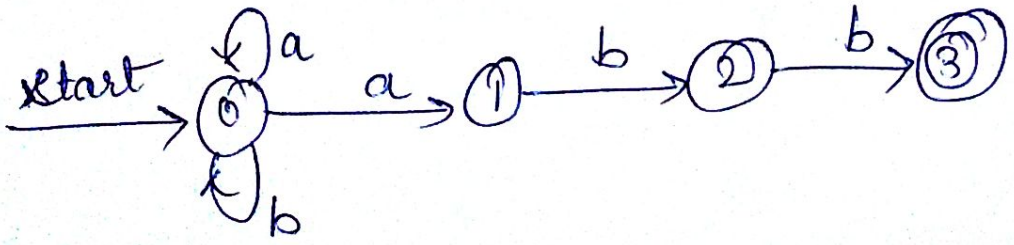
- Deterministic (faster recognizers)
- nondeterministic (less in size than deterministic)

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

Nondeterministic finite Automata:-

- NFA is a mathematical model that consists of,
  1) S → states
  2) P Σ → set of i/p symbols
  3) Transition function
  4) s_0 → start state
  5) F → set of accepting states
- Represented by transition graph, in this same char can label two or more transitions out of one state and edges can be labeled by ϵ as well.

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



- Transition graph can be implemented by transition table,

State	Input Symbol	
	a	b
0	{0,1}	{0}
1	–	{2}
2	–	{3}

Advantage:- fast access to transitions
Disadvantage:- take lot of space

- Adjacency list representation can also be used,
  Advantage:- compact
  Disadvantage:- access is slower

- In a NFA, same i/p string can lead to accepting state by different sequences,

  Ex:- aabb

$$0 \xrightarrow{a} 0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$$

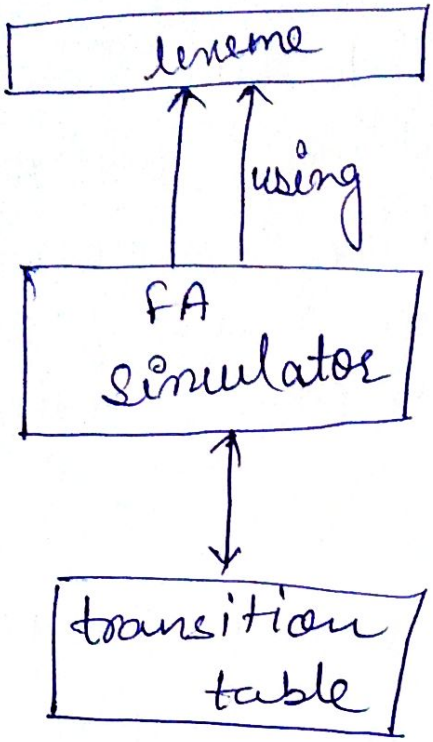
$$0 \xrightarrow{a} 0 \xrightarrow{a} 0 \xrightarrow{b} 0 \xrightarrow{b} 0$$

Deterministic Finite Automata:-

- DFA is a special case of NFA,
  - → no state has ϵ-transition
  - → for each state s and i/p symbol a, there is at most one edge labeled a leaving s

Design of a lexical analyzer generator:-

- Aim is to design a s/w tool that automatically constructs a lexical analyzer from a program in the lex language.
- Specifications
  P_1 {action$_1$}
  P_2 {action$_2$}
  ⋮
  P_n {action$_n$}
- P_i = pattern i is a regular expression & each action$_i$ is a program fragment that is to be executed whenever a lexeme matched by P_i.

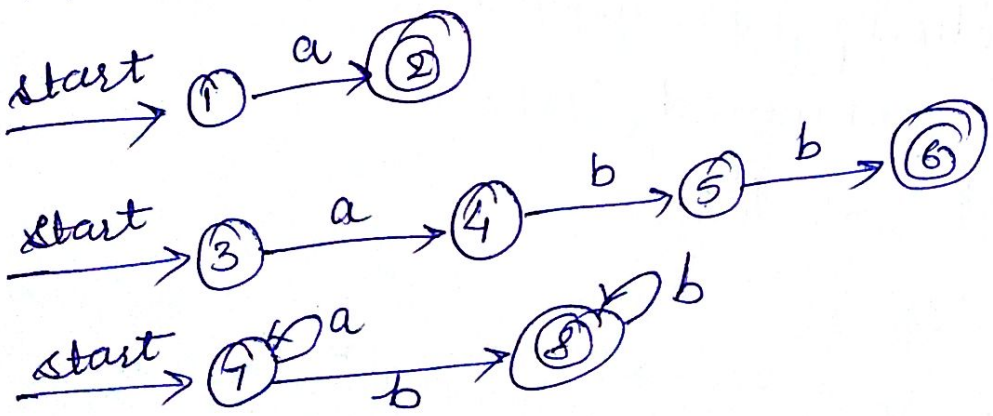


- Finite automata simulator uses transition table to look for the regular expression patterns in the i/p buffer.

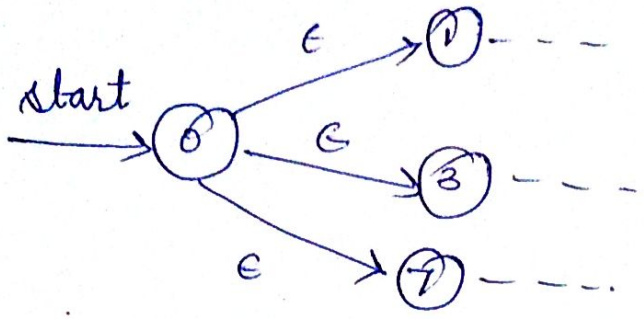
Pattern matching based on NFA's :-

Eg
```
a       { }   /* actions are omitted here */
abb     { }
a* b+   { }
```

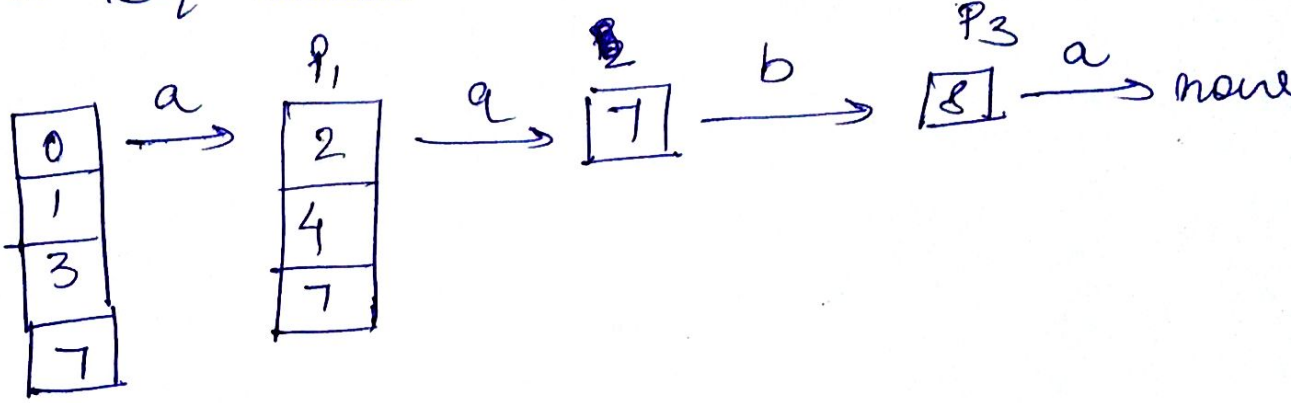
a) NFA for each regular exp



b) Combined NFA



c) Check seq aaba



DFA for lexical analyzers:-

- Convert NFA into DFA (transition table).
- Use transition table to check the seq.

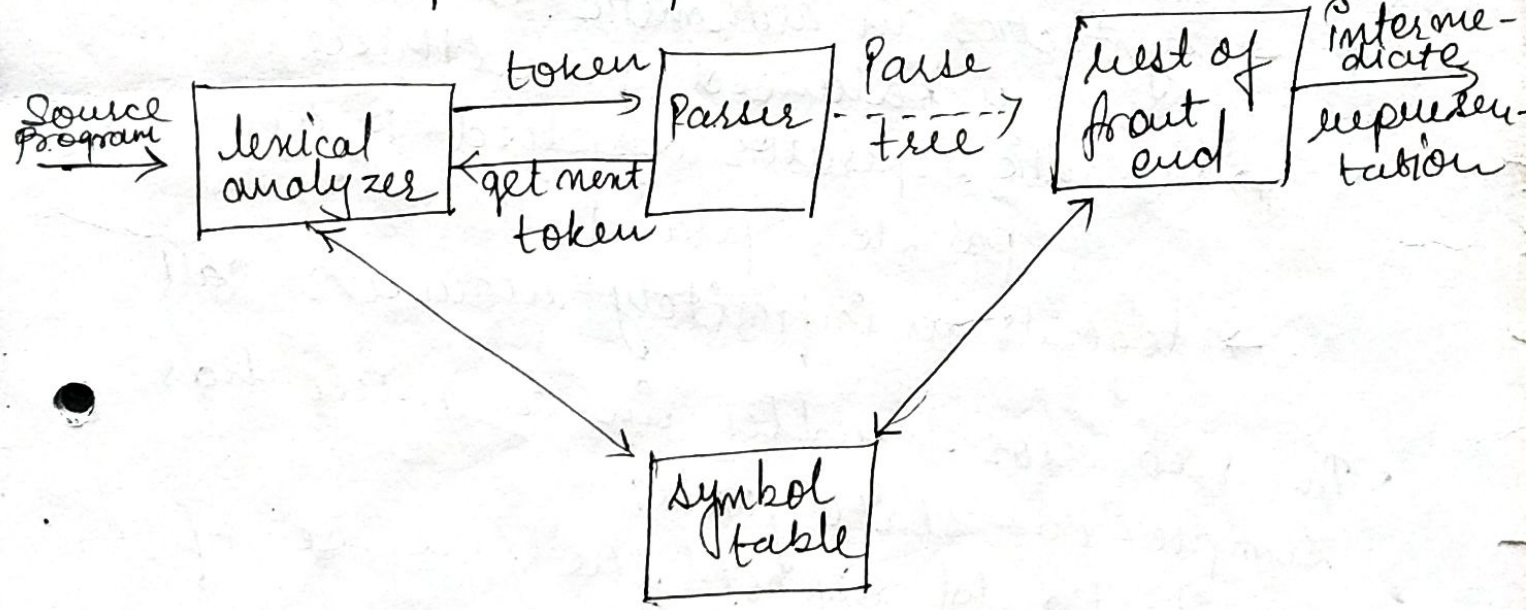
State	I/P symbol a	I/P symbol b	Pattern
		8	none
0137	247	8	a
247	7	58	a^*b^+
8	–	8	none
7	7	8	a^*b^+
58	–	68	abb
68	–	8	

Syntax Analysis

Introduction (Role of Parser):-

- Parser obtains a string of tokens from the lexical analyzer and verifies that the string can be generated by the grammer for the source language.

- It should also recover from commonly occurring errors so that it can continue processing the remainder of its i/p.



- Grammers
  - top-down (build parse tree from top to bottom)
  - bottom-up

Syntax error Handling :-

- A good compiler should assist the programmer in identifying and locating errors.
- Few languages have been designed with error handling.
- Programming language do not describe how a compiler should respond to errors; the response is left to the compiler design.

- Errors can be,
  - → lexical: misspelling an identifier, keyword
  - → syntactic: an arithmetic expression with unbalanced parentheses
  - → semantic: operator applied to an incompatible operand
  - → logical: an infinitely recursive call
- The ~~error~~ error handler in a parser has simple-to-state goals:
  - → it should report the presence of errors clearly and accurately.
  - → should recover from each error quickly
  - → should not significantly slow down the processing of correct programs.

- Common errors are simple one.
- In LL and LR methods, they have the viable-prefix property, meaning they detect that an error has occurred as soon as they see a prefix of i/p that is not a prefix of any string in the language.

Ex: Common errors can be,
- use a comma at place of semicolon.
- Forgot semicolon

- Rules can vary from language to language. A compiler should report the place in source program where an error is detected.
- In some parsers, when want to recover from error, some data can be curread. So that some variables can be unknown.

Error-Recovery strategies:-

1) Panic Mode:-
- Simplest method & can be used by most parsing methods.
- On discovering an error, the parser discards i/p symbols one at a time until one of a designated set of synchronizing tokens (semicolon, end) is found.
- This method often skips a considerable amount of i/p without checking it for additional errors.

2) Phrase-level Recovery:-

- On discovering an error, this method performs local corrections on the remaining i/p; that is, it may replace a prefix of the remaining i/p by some string that allows the parser to continue.

Ex:- replace a comma by semicolon, delete an extra semicolon, insert a missing semicolon.

- We must be careful to choose replacements that do not lead to infinite loops.

3) Error Productions:-

- By using common errors we augment grammer and make parser for erroneous constructs. So that we can recognized the erroneous constructs in the i/p.

4) Global correction:-

- Given an incorrect i/p string x and grammer G, these algorithms will find a parse tree for a related string y, such that no of insertions, deletions and changes of tokens required to transform x into y is as small as possible.

- They try to obtain a globally least-cost correction.
- These methods are in general too costly in terms of time and space, so these techniques are currently only of theoretical interest.

Context-Free Grammers:-

- Some type of statements cannot be specified using regular exp, so that CFG is used.

Ex:- stmt → if expr then stmt else stmt

→ In Regular exp → right side of production will be start from terminal only.

→ In CFG → right side of production can be terminal / non terminal

→ CSG → It will have same content in left and right side

→ unrestricted - no rules

- Terminals, non terminals, a start symbol & productions are diff parts of CFG.

→ Terminals:- symbols from which strings are formed. Also called token.

- lower-case letters early in alphabet a, b, c
- operators +, -, *
- Punctuation like parentheses, comma
- Digits 0, 1, ... 9
- Boldface string id or if.

→ Non terminals:- Syntactic variables that denote set of strings.

- Upper-case letters early in alphabet

  A, B, C

- letter S (start state)
- Lower-case italic names like expr/stmt.

→ Start state:- one nonterminal, and the set of strings it denotes is the language defined by grammar.

→ Productions specify the manner in which the terminals & nonterminals can be combined to form strings.

- Upper-case letters late in alphabet X, Y, Z represent grammar symbols, that is either nonterminals or terminals.
- lower-case letters late in alphabet u, v, z represent strings of terminals.
- Lower-case Greek letters, α, β, γ represent strings of grammer symbols.
- if $A \to \alpha_1$, $A \to \alpha_2$ --- $A \to \alpha_k$
  then $A \to \alpha_1 | \alpha_2 | \cdots | \alpha_k$ called alternatives of A.
- The ~~start~~ left side of first production is start symbol.

Ex:- expr → expr op expr
expr → (expr)
expr → - expr
expr → id

op → +
op → -
op → *
op → /
op → ↑

According to above rules it can be stated as,

E → EAE | (E) | -E | id ——— (1)
A → + | - | * | / | ↑

Derivations :-

- A production is treated as a rewriting rule in which the nonterminal on the left is replaced by the string on the right side of the production. ——— (2)

Ex:- E → E+E | E*E | (E) | -E | id

- E ⇒ -E (means E derives -E)
- E*E ⇒ (E)*E or E*(E) (Both are correct)
- E ⇒ -E ⇒ -(E) ⇒ -(id) (derivation of -(id) from E)

- $\Rightarrow$ derives
  $\overset{*}{\Rightarrow}$ derives in zero or more steps
  $\overset{+}{\Rightarrow}$ derives in one or more steps
- $L(G)$ → language generated by G. strings in $L(G)$ may contain only terminal symbols of G.
- if $S \overset{\alpha}{\Rightarrow} \alpha$, where α may contain nonterminals, then we say that α is a sentential form of G.
- if $S \overset{+}{\Rightarrow} w$, here w is a string of terminals, then w is called a sentence of G.

Ex:- string -(id+id) is a sentence of grammer 2.

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

Means $E \overset{*}{\Rightarrow} -(id+id)$

One more choice
$-(E+E) \Rightarrow -(E+id) \Rightarrow -(id+id)$

- In general leftmost nonterminal in any sentential form is replaced at each step.

such derivations are termed leftmost.

$$E \underset{lm}{\Rightarrow} -E \underset{lm}{\Rightarrow} -(E) \underset{lm}{\Rightarrow} -(E+E) \underset{lm}{\Rightarrow} -(id+E)$$

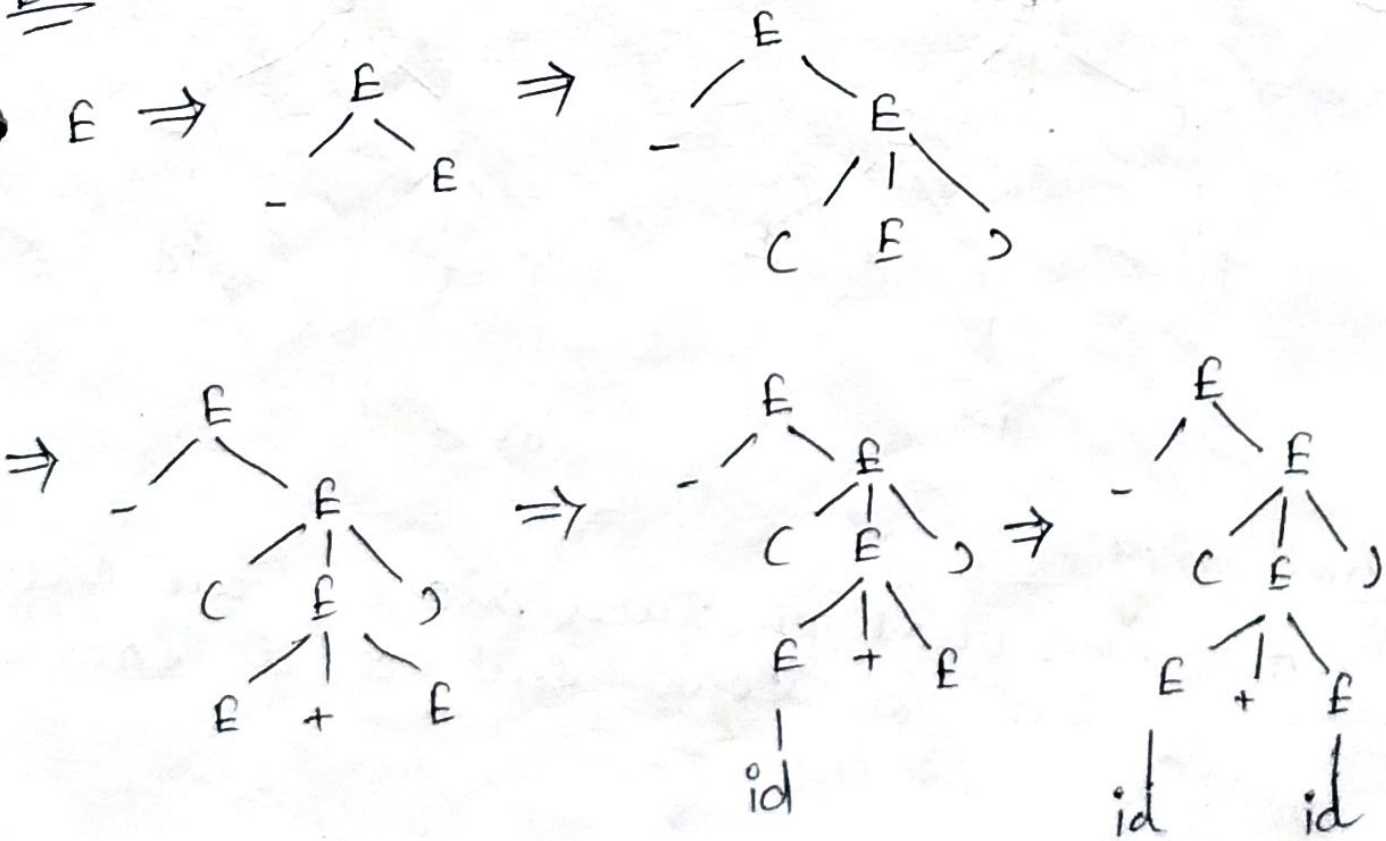
$$\underset{lm}{\Rightarrow} -(id+id)$$

- In which rightmost nonterminal is replaced is called canonical derivations.

Parse Trees and Derivations:-

- A parse tree may be viewed as a graphical representation for a derivation that filters out the choice regarding replacement order. Each interior node has nonterminals while leaves has terminals • called yield of tree.

Ex:- Derive $-(id+id)$

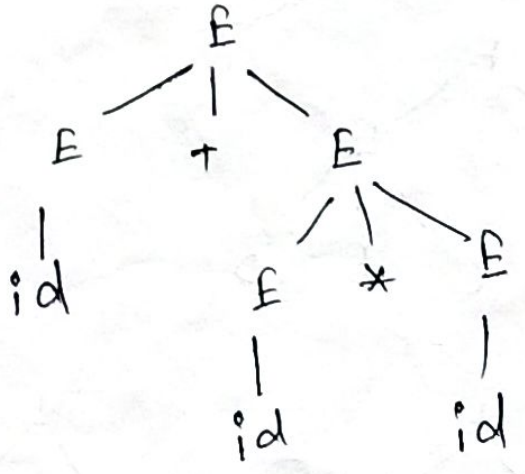


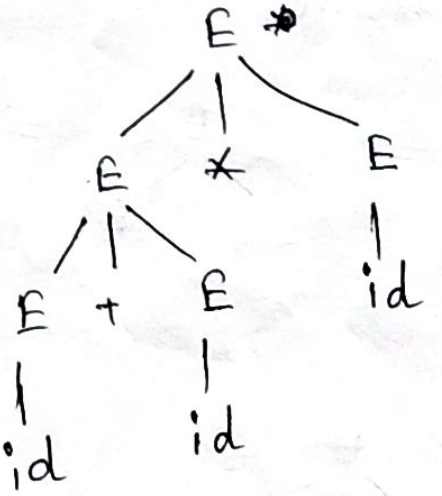
- The variations in the order in which productions are applied can also be eliminated by considering only leftmost derivations.
- Generally every parse tree has associated with a unique leftmost and unique rightmost derivation. sometimes it may be wrong.

Ex id + id * id

① $E \Rightarrow E+E$
$\Rightarrow id+E$
$\Rightarrow id+E*E$
$\Rightarrow id+id*E$
$\Rightarrow id+id*id$

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





Ambiguity:-

- A grammer that produces more than one parse tree for some sentence is said to be ambiguous.

Writing a grammar:-

- **Regular expressions vs. context free grammar:-**

→ constructs that can be described by a regular exp can also be described by a grammar.

Ex Regular exp $(a|b)^*abb$

grammar
$$A_0 \to aA_0 \mid bA_0 \mid aA_1$$
$$A_1 \to bA_2$$
$$A_2 \to bA_3$$
$$A_3 \to \epsilon$$

→ Why use regular exp:-

* lexical rules are simple and do not require a notation as powerful as grammars.
* Regular exp provide a more concise & easier to understand notation for tokens.
* More efficient lexical analyzers can be constructed automatically from regular exp

- **Eliminating Ambiguity:-**

Ex
stmt → if expr then stmt
| if expr then stmt else stmt
| other

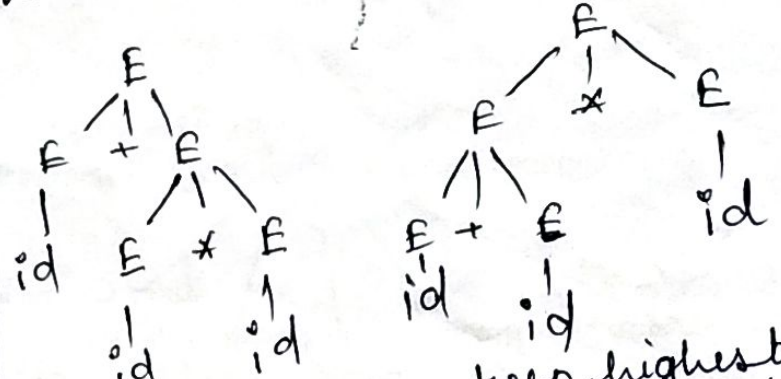
Eliminating Ambiguity:-

Ex:- $E \to E+E$
$E \to E*E$
$E \to (E) \mid id$
$E \to \varepsilon$

Sol:- $E \to E+T \mid T$
$T \to T*F \mid F$
$F \to (E) \mid id$

precedence ambiguity, solve this using level

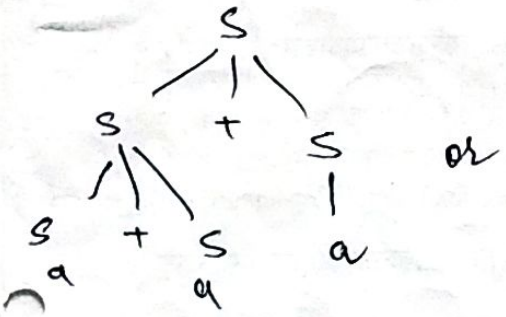
$w = id+id*id$



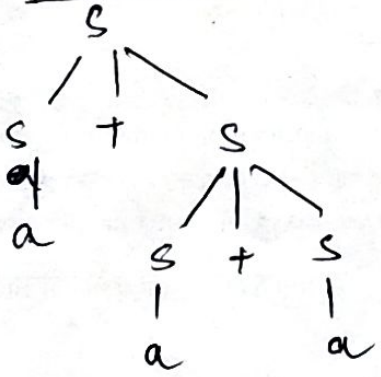
In this case we keep highest precedence operator far from start symbol.

Ex:- $S \to S+S \mid a$

$w = a+a+a$



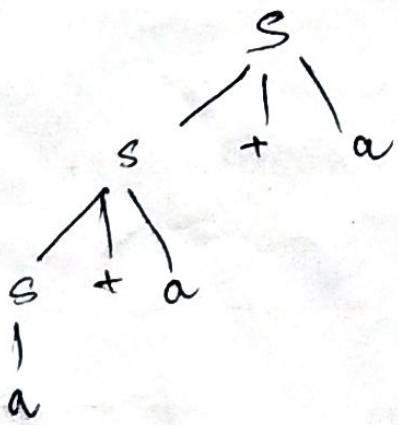
or



Associativity ambiguity, solve by using left/right recursion

Sol:- $S \to S+a \mid a$

$w = a+a+a$



(left recursion)

• Elimination of left Recursion:-

→ A grammar is left recursive if it has a nonterminal A such that there is a derivation

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

→ Rule to eliminate left recursion

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

then $A \to \beta A'$

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

consider

$E \to E+T \mid T$

here $A = E$

$\alpha = +T$

$\beta = T$

Ex $E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid id$

↓

$E \to TE'$

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

$T \to FT'$

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

$F \to (E) \mid id$

- **Left factoring :-**

→ When it is not clear which of two alternative productions to use to expand a nonterminal A, we may be able to rewrite the A-productions.

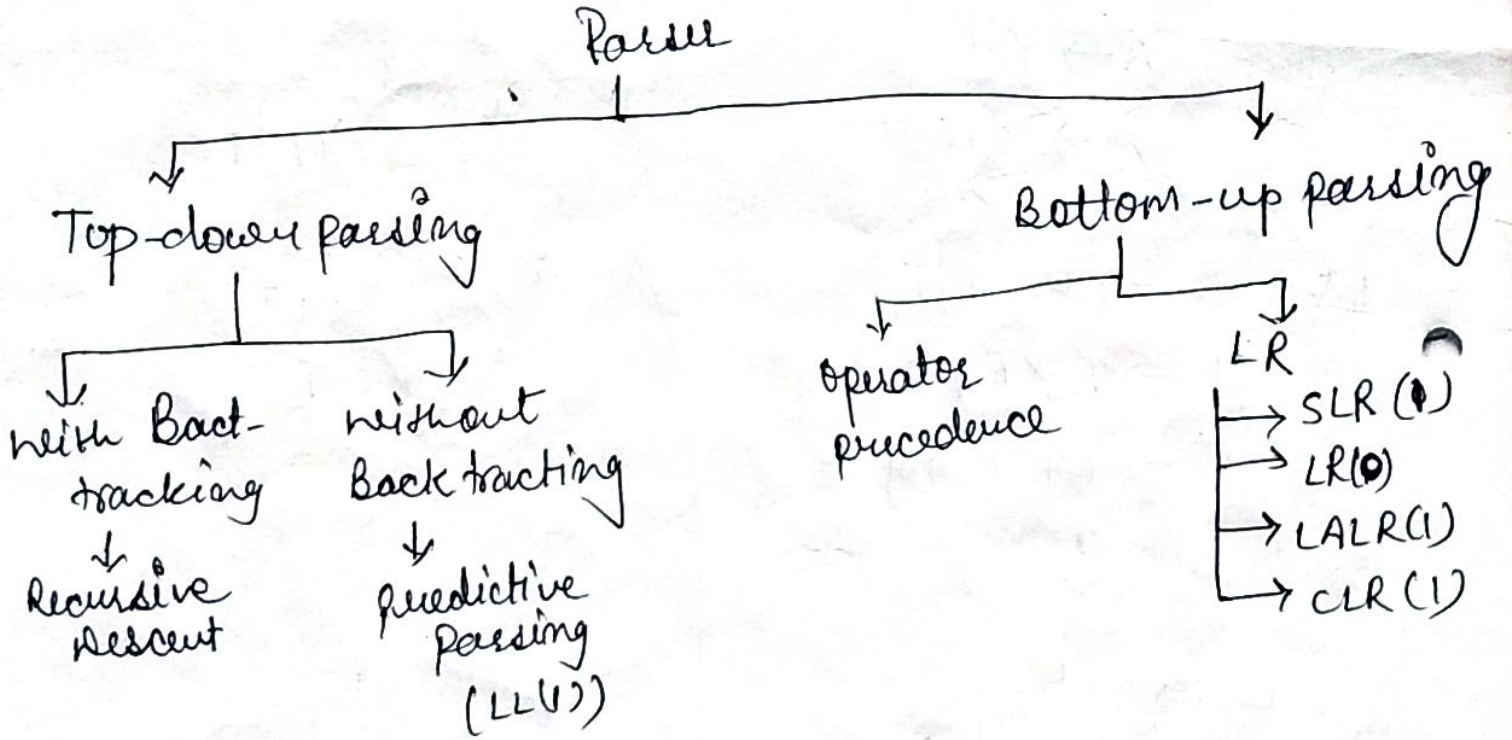
Ex :-

if $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$

then $A \rightarrow \alpha A'$

$A' \rightarrow \beta_1 \mid \beta_2$

- ~~Top - Down Parsing~~ :-



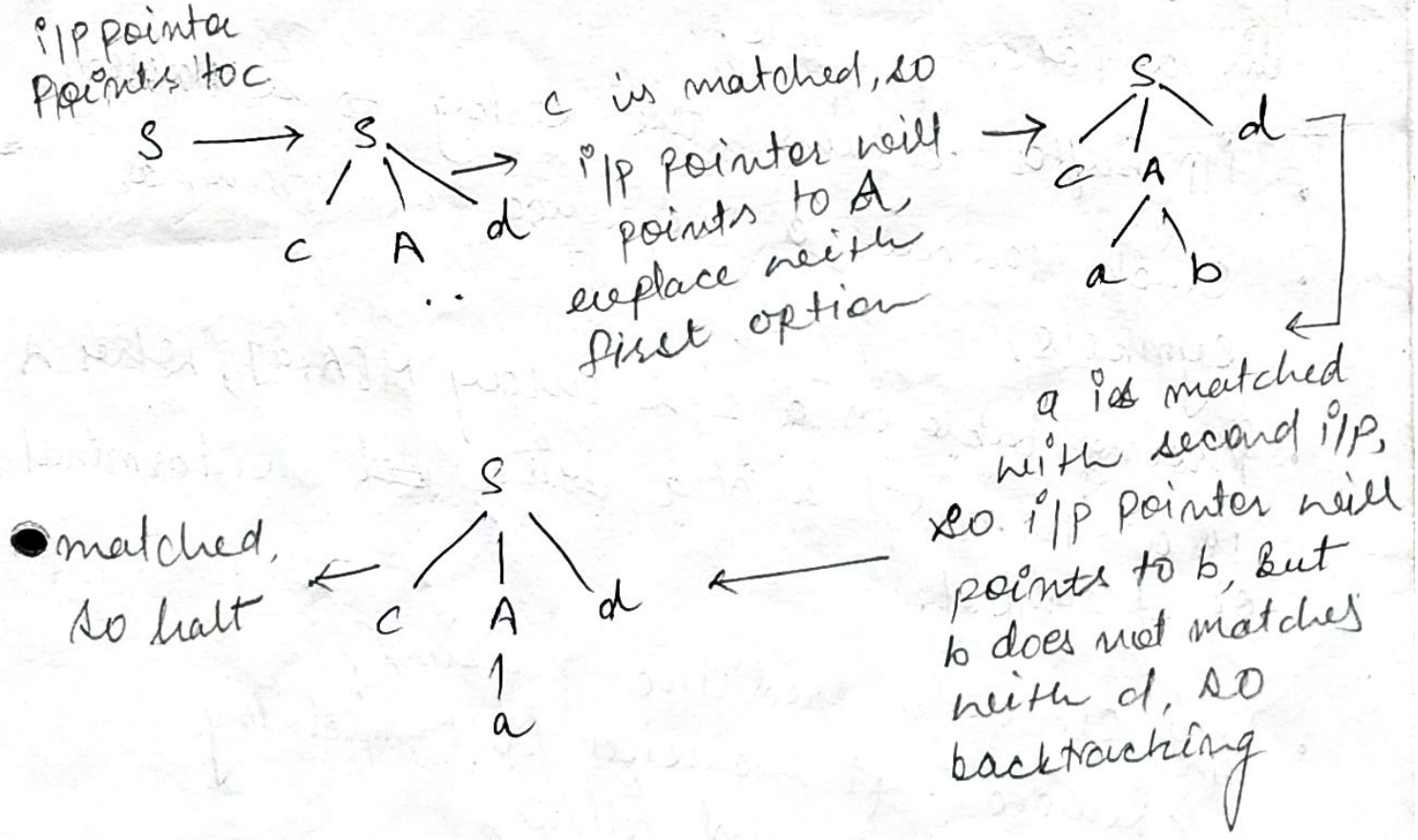
Recursive-Descent Parsing:-

- Top-down parsing can be viewed as an attempt to find a leftmost derivation for an i/p string.
- This may involve backtracking.

Ex:- $S \rightarrow cAd$

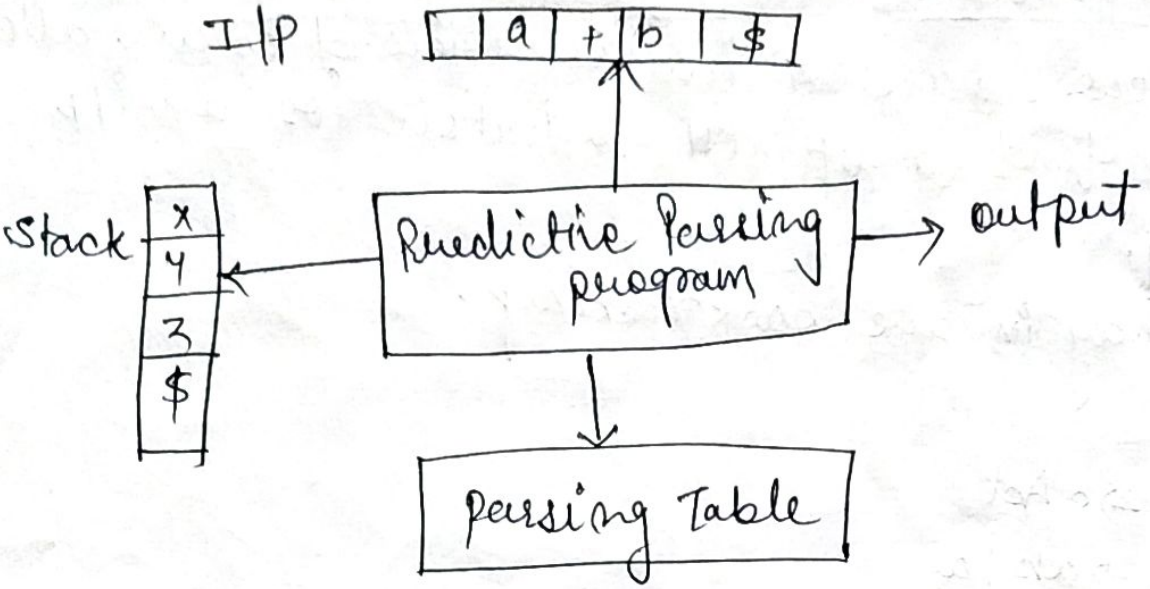
$A \rightarrow ab \mid a$

Sol $w = cad$



Predictive Parsers:-

- For this type of parsing we have to remove the left recursion and left factoring from the grammar.



- parser looks up the production to be applied in a parsing table.
- I/P buffer contains the string to be parsed.
- Stack contains a sequence of grammar symbols.
- Parsing table is a 2-D array M[A, a], where A is a terminal and a is a nonterminal or $ symbol.
- Four steps for predictive parsing.
  1) Remove left recursion / left factoring
  2) Find first and follow
  3) Parsing tabl
  4) Stack implementation
  5) Parse tree

$S \rightarrow ACB \mid Cbb \mid Ba$

$A \rightarrow da \mid BC$

$B \rightarrow g \mid ? \mid \varepsilon$

$C \rightarrow h \mid ? \mid \varepsilon$

First(S) = d, g, ?, h, ε, b, a

(A) = d, g, ?, ε, h

(B) = g, ?, ε

(C) = h, ?, ε

Follow(S) = {$}

Follow(A) = first(C) = h, ?, ε = first(B)

h, ?, g, ε = follow(S)

= h, ?, g, $

Follow(B) = first(C) = h, ?, ε = follow(A)

= h, ?, g, $

Follow(C) = b, follow(A)

= b, h, ?, g, $

Ex :- $S' \to S$
$S \to CC$
$C \to cC \mid d$

first $(S') = c, d$
$(S) = c, d$
$(C) = c, d$

follow $(S') = \{\$\}$
$(S) = \{\$\}$
$(C) = \text{first}(C) = c, d$

Ex
$S \to Aa$
$A \to BD$
$B \to b \mid \varepsilon$
$D \to d \mid \varepsilon$

first $(S) = b, \varepsilon, d, a$
$(A) = b, \varepsilon, d$
$(B) = b, \varepsilon$
$(D) = d, \varepsilon$

follow $(S) = \{\$\}$
$(A) = \{a\}$
$(B) = \text{first}(D) = d, \varepsilon = \text{follow}(A)$
$= d, a$
$(D) = \text{follow}(A) = \{a\}$

Ex Consider Grammer:-

$E \rightarrow E+E$

$E \rightarrow E*E$

$E \rightarrow (E) \mid id$

After removing ~~left recursion~~ ambiguity,

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

- After removing left recursion,

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

- Calculating first and follow,
  - → first and follow is calculated only for non terminals.
  - → for calculating first we see the left Non-terminals and follow the path till it generate a terminal production. Now first char of terminal production will be first.

First(E) = First(T) = First(f) = {(, id}

First(E') = {+, ε}

First(T') = {*, ε}

→ For calculating follow we see nonterminals in right side and see which nonterminal follows it. Now ~~follow~~ first of following nonterminal will also be follow of followed nonterminal

follow(E) = {), $} [By the production f → (E) and $ bcoz its start state]

follow(E') = follow(E) [By E → T E']

= {), $}

follow(T) = ~~follow~~ first(E') = {+, ε}

But ε can't come in follow so follow of ε!

follow(T) = {+,), $}

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

follow(f) = First(T') = {*, ε}

= {*, +,), $}

- **Construction of Parsing table:-** Doesn't contains ϵ.

Non-terminals	Input Symbols					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to id$			$F \to (E)$		

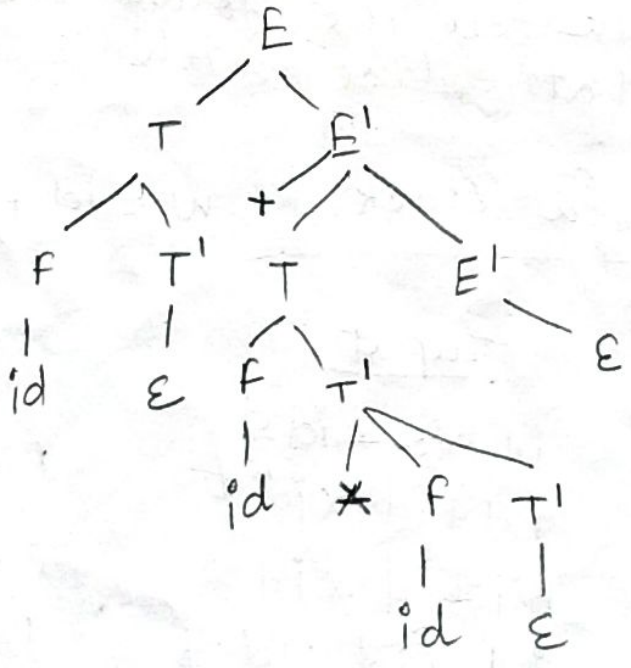
- See first of E, that are {(, id}, entries will be here. But entries that are producing these first symbols
- If a ϵ occurs in first of some non-terminal, then see the follow and put entry in that place with ϵ production.

- **Doing entries in stack:-** $w = id + id * id$

Stack	Input	Output
$ E $	id + id * id $	$E \to TE'$
$ E' T	id + id * id $	$T \to FT'$
$ E' T' F	id + id * id $	$F \to id$
$ E' T' id	id + id * id $	
$ E' T'	+ id * id $	
~~$ E' T' E' T +$~~	~~+ id * id $~~	~~$T' \to +FT'$~~
$ E'	+ id * id $	$T' \to \epsilon$
$ E' T +	+ id * id $	$E' \to +TE'$
$ E' T	id * id $	
$ E' T' F	id * id $	
$ E' T' id	id * id $	
$ E' T'	* id $	
$ E' T' F *	* id $	
$ E' T' F	id $	
$ E' T' id	id $	
$ E' T'	$	
$ E'	$	
$	$	

Stack	Input	Output
$E'T	TE'$	id * id $	
$E'T'F	FT'E'$	id * id $	T → FT'
$E'T'id	~~id~~T'E'$	~~id~~ * id $	F → id
$E'T'	T'E'$	* id $	
$E'T'F*	~~*~~FTE'$	~~*~~id $	T' → *FT'
$E'T'F	FTE'$	id $	
$E'T'id	~~id~~TE'$	~~id~~ $	F → id
$E'T'	T'E'$	$	
$E'	E'$	$	T' → ε
$	$	$	E' → ε

Making Parse Tree



Error Recovery in Predictive Parsing:- (LL Parser)

* Panic-mode error recovery:-
  - We insert synchronizing tokens in "first" and "follow" of non-terminals.
  - if parser looks up entry M[A, a] and finds that it is blank, then i/p symbol 'a' is skipped.
  - if the entry is synch, then the non-terminal on top of the stack is popped in an attempt to resume parsing.

NonTerminal	i/p Symbol					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$	Synch	Synch
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$	Synch		$T \to FT'$	Synch	Synch
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to id$	Synch	Synch	$F \to (E)$	Synch	Synch

- Input →) id * + id $

Stack	Input	Remark
$ E	) id * + id $	error, skip) (As E is first symbol and we can't remove this)
$ E	id * + id $	
$ E' T	id * + id $	
$ E' T' F	id * + id $	
$ E' T' id	id * + id $	
$ E' T'	* + id $	
$ E' T' F *	* + id $	
$ E' T' F	+ id $	error, M[F, +] = synch F has been popped
$ E' T'	+ id $	
$ E'	+ id $	
$ E' T +	+ id $	
$ E' T	id $	
$ E' T' F	id $	
$ E' T' id	id $	
$ E' T'	$	
$ E'	$	
$	$	

* Phrase-level recovery:- it fills blank entries of table with pointers to error routines. These routines may change, insert or delete symbols on i/p and issue appropriate error msg.

LL(1) Grammars:-

- A grammar whose parsing table has no multiple-defined entries is said to be LL(1).
- First 'L' stands for scanning the input from left to right, second 'L' for producing a leftmost derivation and the '1' for using one i/p symbol of lookahead at each step to make parsing action decisions.

Bottom-up Parsing:-

- Also known as shift-reduce parsing.
- Attempts to construct a parse tree for an i/p string beginning at the leaves and working up towards the root. We can think of this process as one of "reducing" a string w to the start symbol of a grammar. At each reduction step a particular substring matching the right side of a production is replaced by the symbol on the left of that production.

Ex:- $S \to aABe$
$A \to Abc \mid b$
$B \to d$

$w = abbcde$
$= aAbcde$
$= aAde$
$= S$

$S \underset{rm}{\Rightarrow} aABe \underset{rm}{\Rightarrow} aAbcde \underset{rm}{\Rightarrow} aAbcde \underset{rm}{\Rightarrow} abbcde$

Handles:-

- A handle of a string is a substring that matches the right side of a production, and whose reduction to the nonterminal on the left side of the production represents one step along the reverse of a rightmost derivation.

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

Stack implementation of shift-reduce parsing:-

- To start, the stack is empty and the string w is on the i/p, as follows,

Stack	Input
$	w $

- The parser operates by shifting zero or more i/p symbols onto the stack until a handle β is on top of the stack. The parser then reduces β

to the left side of the appropriate production. The parser repeats this cycle until it has detected an error or until the stack contains the start symbol and i/p is empty.

Ex:- $w = id_1 + id_2 * id_3$

	Stack	Input	Action
1)	$	$id_1 + id_2 * id_3 \$$	shift
2)	$ id_1	$+ id_2 * id_3 \$$	reduce $E \to id$
3)	$ E	$+ id_2 * id_3 \$$	shift
4)	$ $E+$	$id_2 * id_3 \$$	shift
5)	$ $E+id_2$	$* id_3 \$$	reduce $E \to id$
6)	$ $E+E$	$* id_3 \$$	shift
7)	$ $E+E*$	$id_3 \$$	shift
8)	$ $E+E*id_3$	$\$$	reduce $E \to id$
9)	$ $E+E*E$	$\$$	reduce $E \to E*E$
10)	$ $E+E$	$\$$	reduce $E \to E+E$
11)	$ E	$\$$	accept

- There are four possible actions,
  1) shift :- next i/p symbol is shifted onto the top of stack
  2) reduce :- parser replace handle with nonterminal
  3) accept :- successful
  4) error :- syntax error

Operator - Precedence Parsing :-

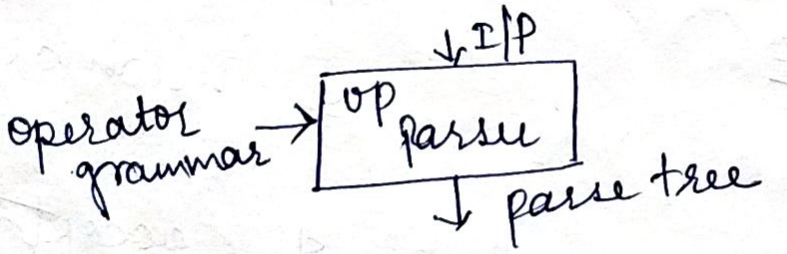
- operator-precedence parsers constructs parse tree for operator grammer only,
  - → no production in right side is ϵ
  - → no production in right side has two adjacent nonterminals.

Ex :- E → EAE | (E) | -E | id

A → + | - | * | / | ↑

change in operator grammar,

E → E+E | E-E | E*E | E/E | E↑E | (E) | -E | id



Ex:- $T \rightarrow T+T / T*T / id$

$w = id + id * id$

steps - 1) Check given grammar is operator precedence grammar or not
2) operator precedence relation table.
3) Parse the given string
4) Generate parse tree

operator precedence relation table:-

	+	*	id	$
+	>	<	<	>
*	>	>	<	>
id	>	>	-	>
$	<	<	<	A

Basics

id, a, b, c → High

$ → low

+ > +

* > *

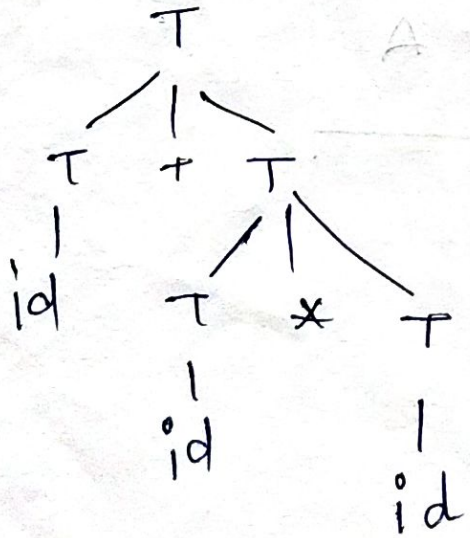
id ≠ id

$ A $

Parsing string

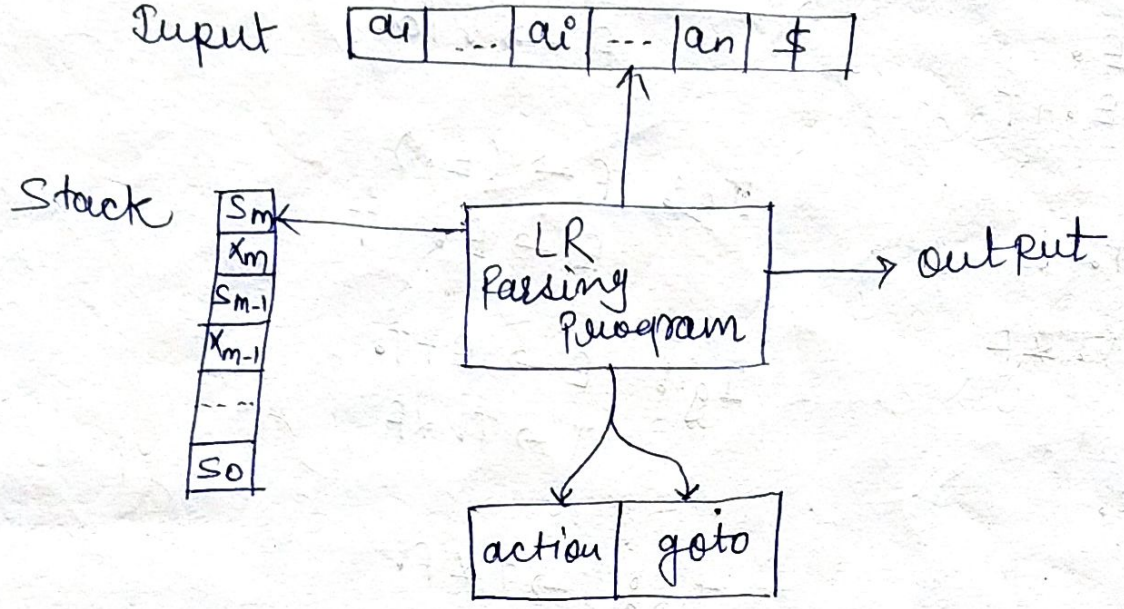
Stack	Relation	Input	Comment
$	<	id + id * id $	shift id
$ id	>	+ id * id $	Reduce id (T→id)
$ T	<	+ id * id $	shift +
$ T +	<	id * id $	shift id
$ T + id	>	* id $	Reduce T→id
$ T + T	<	* id $	shift *
$ T + T *	<	id $	shift id
$ T + T * id	>	$	Reduce T→id
$ T + T * T	>	$	Reduce T→T*T
$ T + T	>	$	Reduce T→T+T
$ T	A		

Parse Tree



LR Parsers:- LR(k) LR(0)

- 'L' stands for left to right scanning of input.
  'R' stands for constructing a rightmost derivation ~~of~~ ~~too~~ in reverse
  'k' stands for ~~constructing~~ no of i/p symbols of lookahead.



- Xi = grammar ~~of~~ symbol
- Si = symbol called a state
  - Each state symbol summarizes the information contained in the stack below it.

SLR Parsers :- SLR(1)

Ex:- E → E+T | T
T → T*f | f
f → (E) | id

Step1 : Augmented expression grammar

E' → E E → E+T | T
T → T*f | f f → (E) | id

I_0: $E' \to .E$
$E \to .E+T$ — (1)
$E \to .T$ — (2)
$T \to .T*f$ — (3)
$T \to .f$ — (4)
$f \to .(E)$ — (5)
$f \to .id$ — (6)

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

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

I_3: $T \to f.$

I_4: $f \to (.E)$
$E \to .E+T$
$E \to .T$
$T \to .T*f$
$T \to .f$
$f \to .(E)$
$f \to .id$

I_5: $f \to id.$

I_6: $E \to E+.T$
$T \to .T*f$
$T \to .f$
$f \to .(E)$
$f \to .id$

I_7: $T \to T*.f$
$f \to .(E)$
$f \to .id$

I_8: $f \to (E.)$
$E \to E.+T$

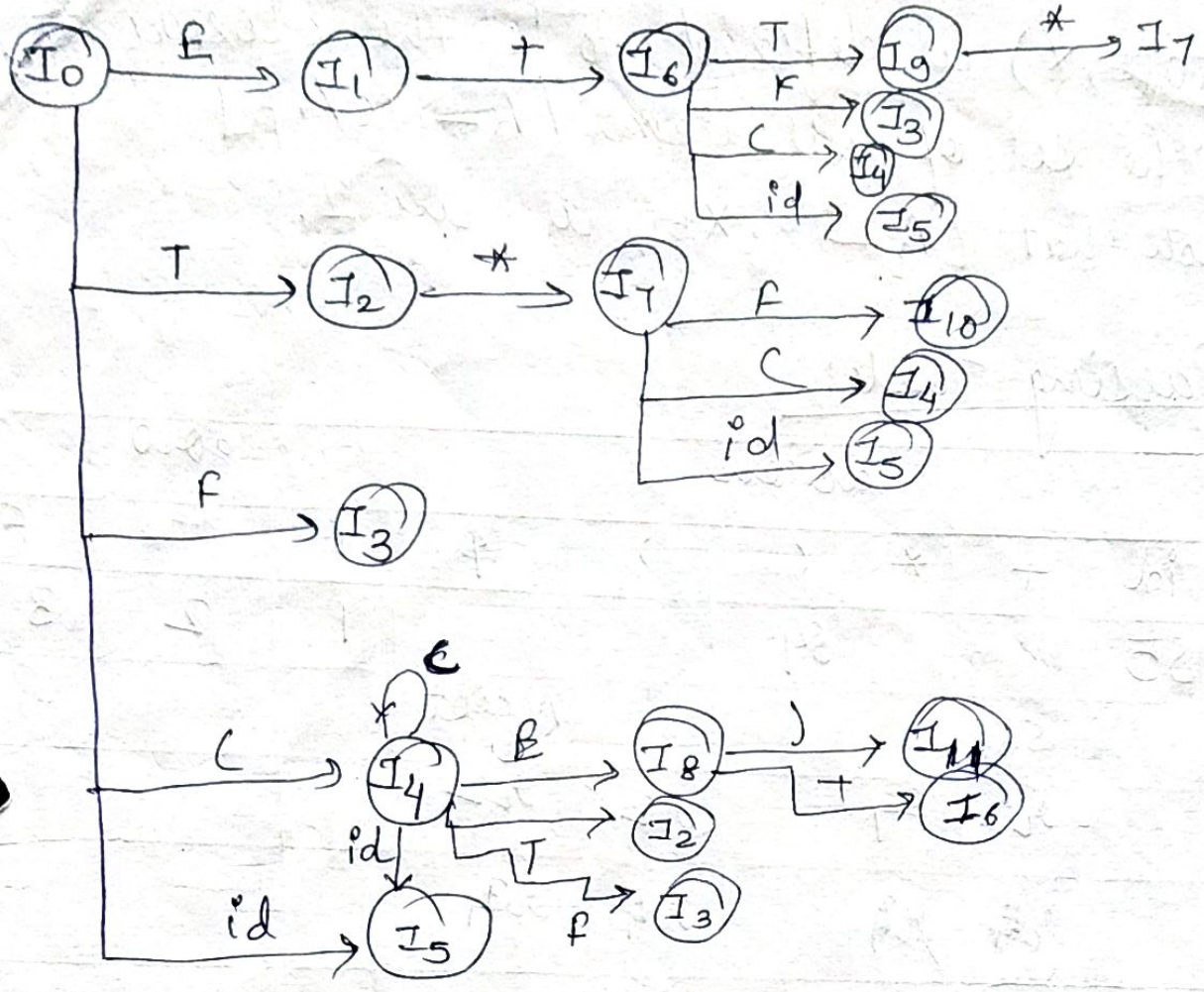
I_9: $E \to E+T.$
$T \to .T.*f$

I_{10}: $T \to T*f.$

I_{11}: $f \to (E).$

First(E) = { (, id }
first(T) = { (, id }
first(F) = { (, id }
follow(E) = { +,), $ }
follow(T) = { *, +,), $ }
follow(f) = { *, +,), $ }

After removing left recursion,
$E \to TE'$ $E' \to +TE' / \varepsilon$
$T \to fT'$ $T' \to *fT' / \varepsilon$
$F \to (E) / id$



- The closure Operation
  → if I is a set of items for a grammar G, then closure(I) is the set of items constructed from I by two rules:
  a) Initially, every item in I is added to closure(I).
  b) if $A \to \alpha \cdot B\beta$ is in closure(I) and $B \to \gamma$ is a production, then add the item $B \to .\gamma$ to I, if it is not already there.

- The Goto Operation:- goto(I, X) where I is a set of items and X is a grammar Symbol.

goto(I, X) is defined to be the closure of the set of all items $[A \to \alpha X . \beta]$ such that $[A \to \alpha . X \beta]$ is in I.

Make Passing Table:-

State	action						goto		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				Accept			
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			

- Check which symbol is starting from a state and going to which state.
- For finding reduce find such state which are final (having a . in the end) states. Then

check the follow of symbol just before '.'. Now place reduce action in such places in the parsing table with production numbers from I_0.

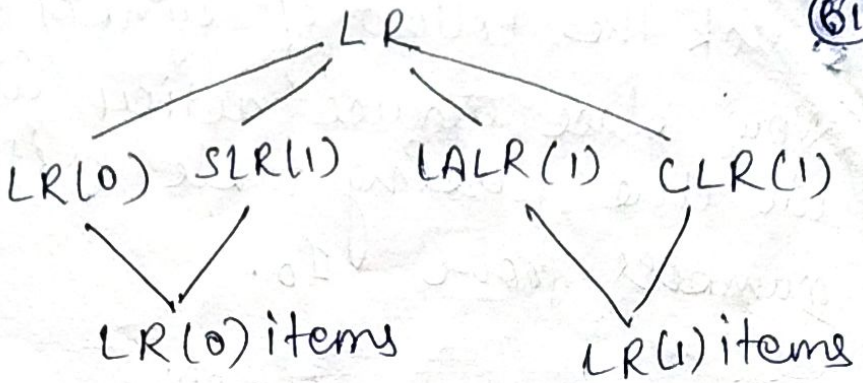
Stack Implementation:-

w = id * id + id.

Stack	Input	Action
1) 0	id * id + id $	shift
2) 0 id 5	* id + id $	reduce F → id
3) 0 F 3	* id + id $	reduce T → F
4) 0 T 2	* id + id $	shift
5) 0 T 2 * 7	id + id $	shift
6) 0 T 2 * 7 id 5	+ id $	reduce F → id
7) 0 T 2 * 7 F 10	+ id $	reduce T → T * F
8) 0 T 2	+ id $	reduce E → T
9) 0 E 1	+ id $	shift
10) 0 E 1 + 6	id $	shift
11) 0 E 1 + 6 id 5	$	reduce F → id
12) 0 E 1 + 6 F 3	$	reduce T → F
13) 0 E 1 + 6 T 9	$	reduce E → E + T
14) 0 E 1	$	Accept

- In line 1, 0 and id will be processed and we look at the parsing table, it is shift to state 5. So shift 'id' to stack with state no 5 at top.
- In line 2, 5 to * is reduce to production which is F → id, Now put F into state and then see 0 to F, that is 3. So stake is 0F3.

LR(0):-



Example:-

Steps:-

1) Augment the given grammar
2) Draw canical collection of LR(0) items/DFD
3) Number the production
4) Create the parsing table
5) stack implementation
6) Draw parse tree

Example:-

$E \rightarrow BB$
$B \rightarrow cB/d$

Step1:- augment grammar

$E' \rightarrow E$
$E \rightarrow BB$
$B \rightarrow cB/d$

Step2: Draw conical collection

I_0 : $E' \to .E$
$E \to .BB$
$B \to .cB$
$B \to .d$

I_1 : $E' \to E.$

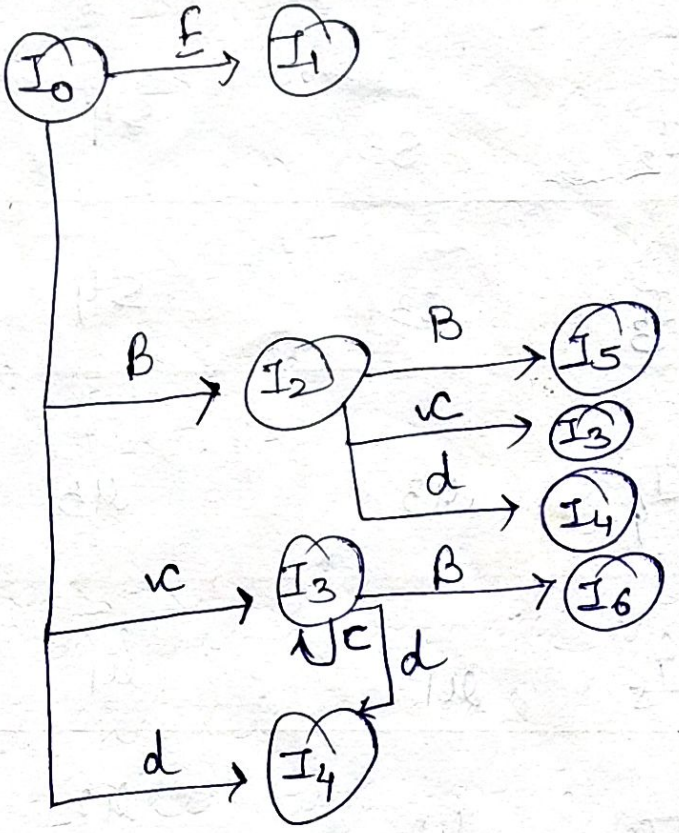
I_2 : $E \to B.B$
$B \to .cB$
$B \to .d$

I_3 : $B \to c.B$
$B \to .cB$
$B \to .d$

I_4 : $B \to d.$

I_5 : $E \to BB.$

I_6 : $B \to cB.$



Step3:- No the productions

$E' \to E$
$E \to BB$ — (1)
$B \to cB$ — (2)
$B \to d$ — (3)

State	Action			Goto	
	c	d	$	£	B
I_0	S3	S4		1	2
I_1			Accept		
I_2	S3	S4			5
I_3	S3	S4			6
I_4	R3	R3	R3		
I_5	R1	R1	R1		
I_6	R2	R2	R2		

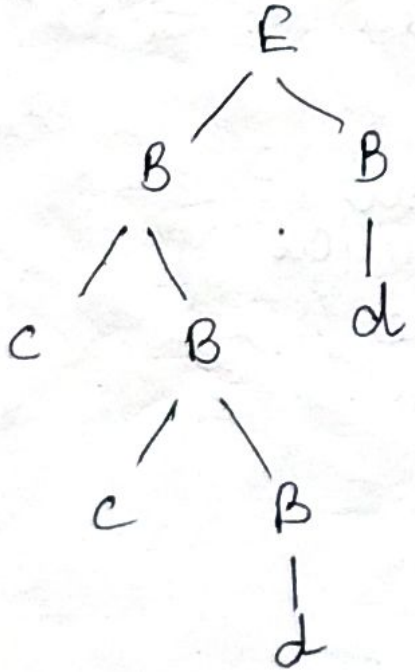
As we have (0) lookahead symbols, so we can't find follow, as it requires next symbol. So conflict can occur. To remove conflicts we use SLR(1) parse with one lookahead symbol.

Step 5: Stact implementation

String = ccdd

0	ccdd $	shift
0 c3	cdd $	shift
0 c3c3	dd $	shift
0 c3c3 d4	d $	reduce B → d
0 c3 c3 B6	d $	reduce B → cB
0 c3 B6	d $	reduce B → cB
0 B2	d $	shift
0 B2d4	$	reduce B → d
0 B2B5	$	reduce E → BB
0 E1	$	Accept

Step 6: Draw parse tree



CLR(1):- Canonical LR(1)

- Based on LR(1) items

Find lookahead:-

Ex:- E → BB
B → cB/d

E' → .E, $ (as start symbol do not has any lookahead)

E → .BB, $

B → .cB/.d, c/d

Ex:- A → α.Bβ, a

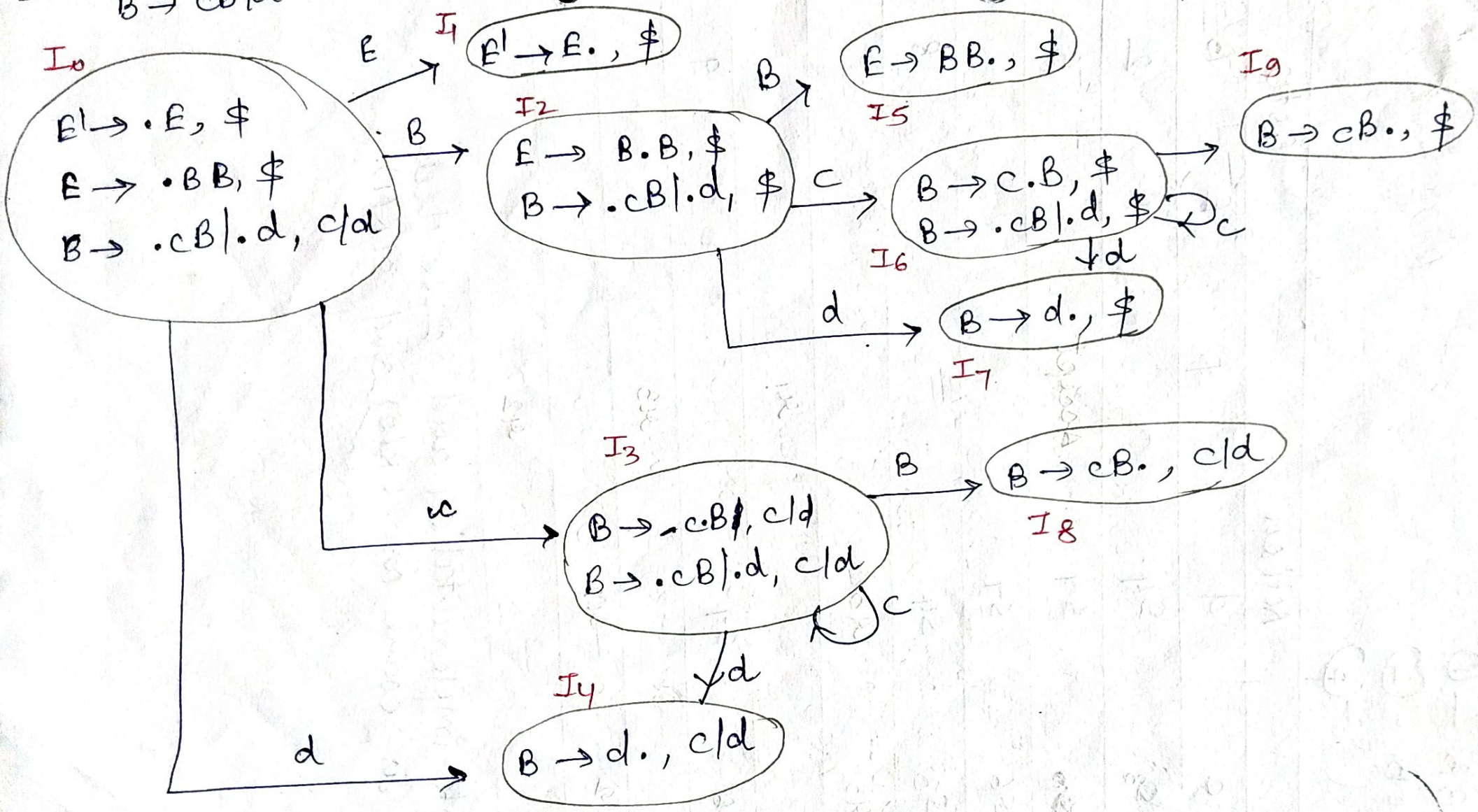
Suppose we are now opening B, so (B → γ) we take look-ahead as first(βa)

- suppose first of β is ε
  then B → .γ, a/c
- suppose A → α.B, a
  then B → .γ, a

Steps: Augment grammar
Draw conical collection
Number the production
create passing table
stack implementation
Draw parse table

Ex:- $E \to BB$
$B \to cB \mid d$

when we do goto, then lookahead doesn't change.
Lookahead only changes when we open some symbol.



$E' \rightarrow E$

$E \rightarrow BB$ — (1)

$B \rightarrow cB$ — (2)

$B \rightarrow d$ — (3)

State	Action			Goto	
	c	d	$	E	B
I_0	S_3	S_4		1	2
I_1			Accept		5
I_2	S_6	S_7			
I_3	S_3	S_4			8
I_4	r_3	r_3			
I_5			r_1		
I_6	S_6	S_7			9
I_7			r_3		
I_8	r_2	r_2			
I_9			r_2		

Now stack implementation and parse tree creation is same as LR(0) and SLR(1).

LALR(1):- Lookahead LR(1)

- using LR(1) items
- Combine those states which are having same production but different lookahead.

$I_3 + I_6 = I_{36}$

$I_4 + I_7 = I_{47}$

$I_8 + I_9 = I_{89}$

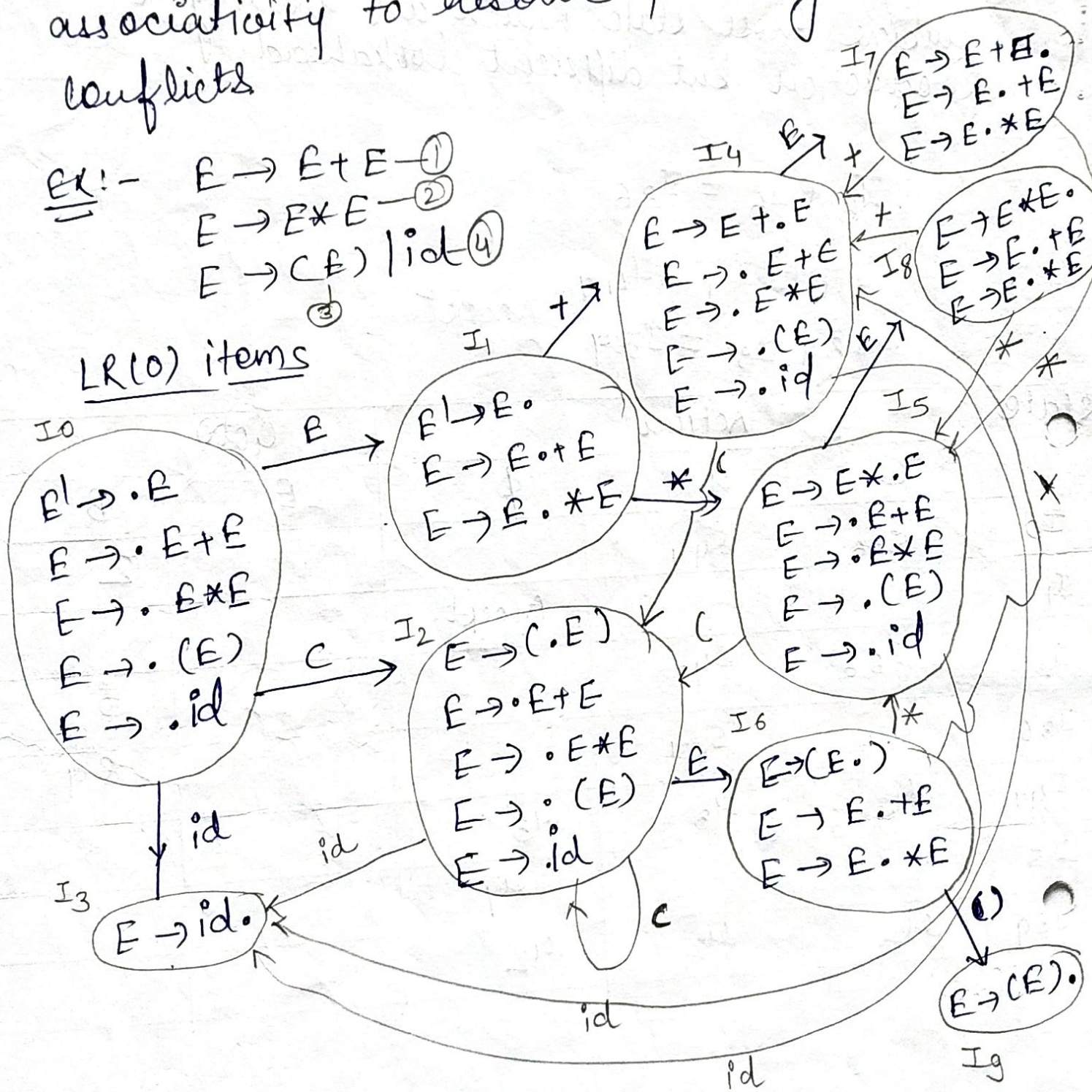
state	Action			Goto	
	C	d	$	E	B
I_0	S_{36}	S_{47}		1	2
I_1			Accept		
I_2	S_{36}	S_{47}			5
I_{36}	S_{36}	S_{47}			89
I_{47}	r_3	r_3	r_3		
I_5			r_1		
I_{89}	r_2	r_2	r_2		

Using Ambiguous Grammars :-

- This type of grammar uses precedence and associativity to resolve parsing action conflicts

Ex :- $E \to E+E$ — (1)
$E \to E*E$ — (2)
$E \to (E) \mid id$ — (4)
(3)

LR(0) items



State	Action						Goto
	id	+	*	(	)	$	E
I_0	S3			S2			1
I_1		S4	S5			Accept	
I_2	S3			S2			6
I_3		r_4	r_4		r_4	r_4	
I_4	S3			S2			7
I_5	S3			S2			8
I_6		S4	S5		S9		
I_7		r_4/S_4	S_5/r_4		r_4	r_4	
I_8		S_4/r_2	S_5/r_2		r_2	r_2	
I_9		r_3	r_3		r_3	r_3	

First (E) = { (, id }

Follow (E) = { +, *,), $ }

State	Input	Action
0	id + id * id $	shift
0 id 3	+ id * id $	reduce E → id
0 E 1	+ id * id $	shift
0 E 1 + 4	id * id $	shift
0 E 1 + 4 id 3	* id $	reduce E → id
0 E 1 + 4 E 7	* id $	

Now as precedence of * is higher on +, so shift * onto stack, otherwise reduce the stack.

Syntax Directed Definitions:-

- In this chapter, we are going to attach attributes to the grammar symbols. Values for attributes are computed by "semantic rules" associated with the grammar productions.
- Notations for semantic rules

Syntax-directed definitions → translation scheme.

(High level specifications for translation schemes and hide many implementation details)

(indicate the order in which semantic rules are to be evaluated)

- conceptual view of syntax-directed translation,

input string → Parse tree → dependency graph → evaluation order for semantic rules

Syntax-directed definitions:-

- A syntax-directed definition is a generalization of a content-free grammar in which each grammar symbol has an associated set of attributes, called synthesized and inherited attributes.

- An attribute can represent anything a string, number, type or memory location.
- Synthesized attribute :- computed from the values of attributes at children
- Inherited attribute :- computed from the values of attributes at the siblings and parent of that node.
- A dependency graph is computed from attributes, from this graph an evaluation order for semantic rules is derived. Evaluation of these rules defines the values of attributes at the node in parse tree for I/P string.
- A parse tree showing the values of attributes at each node is called an annotated parse tree.

form of a Syntax-Directed Definition:-

- In a syntax-directed definition, each grammar production $A \to \alpha$ has associated with it a set of semantic rules of the form $b := f(c_1, c_2 \ldots c_k)$ where f is a function, and either,

→ b is a synthesized attribute of A and $c_1, c_2 \ldots c_k$ are attributes belonging to the grammar symbols of the production, or

→ b is an inherited attribute of one of the grammar symbols on the right side of the production and $c_1, c_2 \ldots c_k$ are attributes belonging to the grammar symbols of the production.

Ex:-

Production	Semantic Rules
dummy attri ← $L \to E\,n$ ⇒ newline char for printing	print(E.val)
$E \to E_1 + T$	$E.val := E_1.val + T.val$
$E \to T$	$E.val := T.val$
$T \to T_1 * F$	$T.val := T_1.val \times F.val$
$T \to F$	$T.val := F.val$
$F \to (E)$	$f.val := E.val$
$F \to digit$	$f.val := digit.lexval$

↳ Value is assumed to be supplied by lexical analyzer

- Terminals are assumed to have synthesized attributes only.

Synthesized attributes:-

- Called S-attributed definition. A parse tree for an S-attributed definition can always

Ex:- 3*5+4n

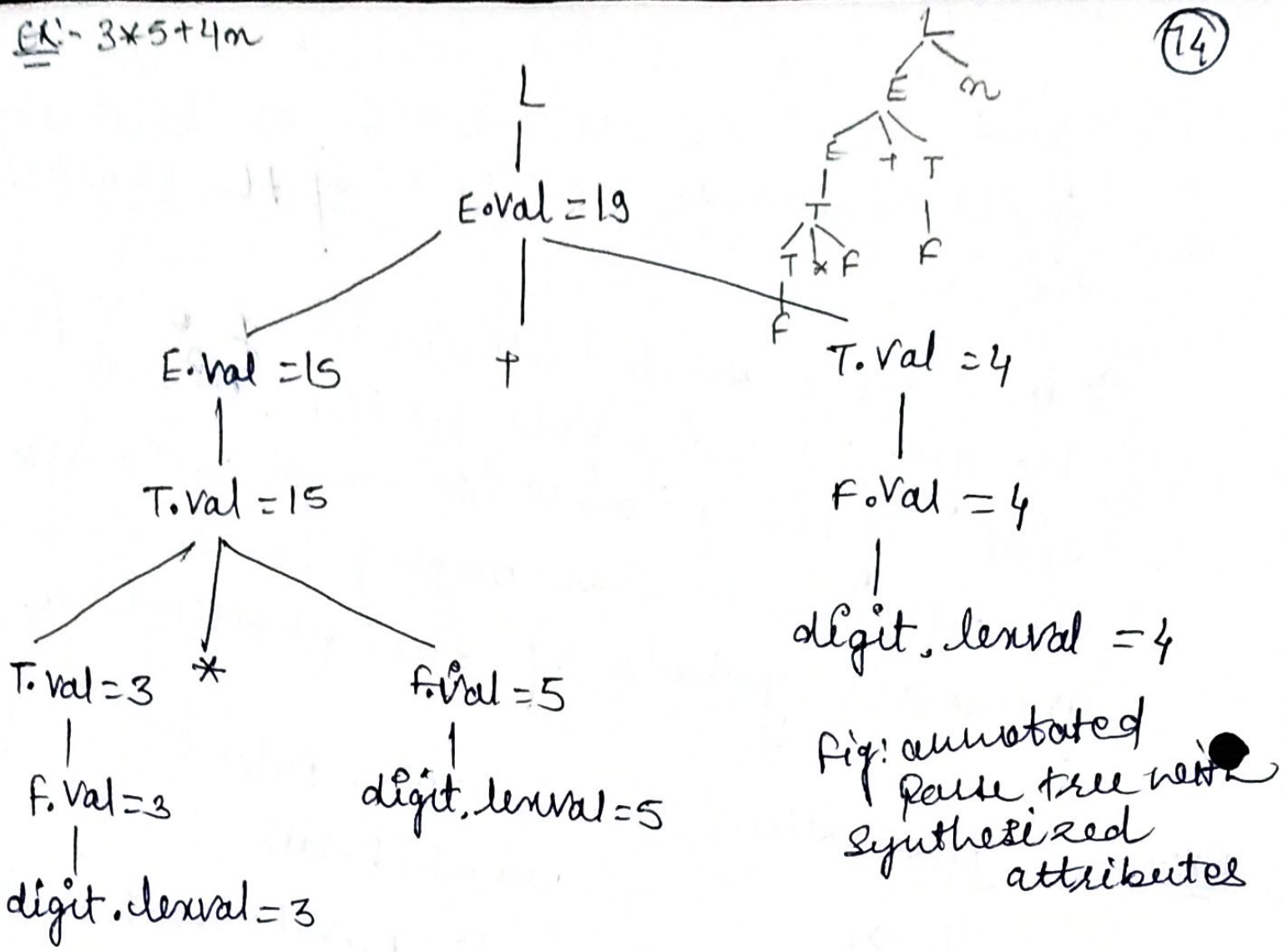


Fig: annotated parse tree with synthesized attributes

be annotated by evaluating the semantic rules for the attributes at each node bottom up, from the leaves to the root.

Inherited attributes:-

- Inherited attributes are convenient for expressing the dependence of a programming language construct on the context in which it appears for ex, we can use it to keep track whether an identifier appears on the left or right side of an assignment in order to decide whether the address or the value of identifier is needed.

Ex:-

Production	Semantic Rules
$D \rightarrow TL$	$L.in := T.type$
$T \rightarrow int$	$T.type := integer$
$T \rightarrow real$	$T.type := real$
$L \rightarrow L_1, id$	$L_1.in := L.in$ & addtype (id.entry, L.in)
$L \rightarrow id$	addtype (id.entry, L.in)

Synthesized attribute

addtype (id.entry, L.in) — add the type of each identifier to ids entry in the Symbol table.

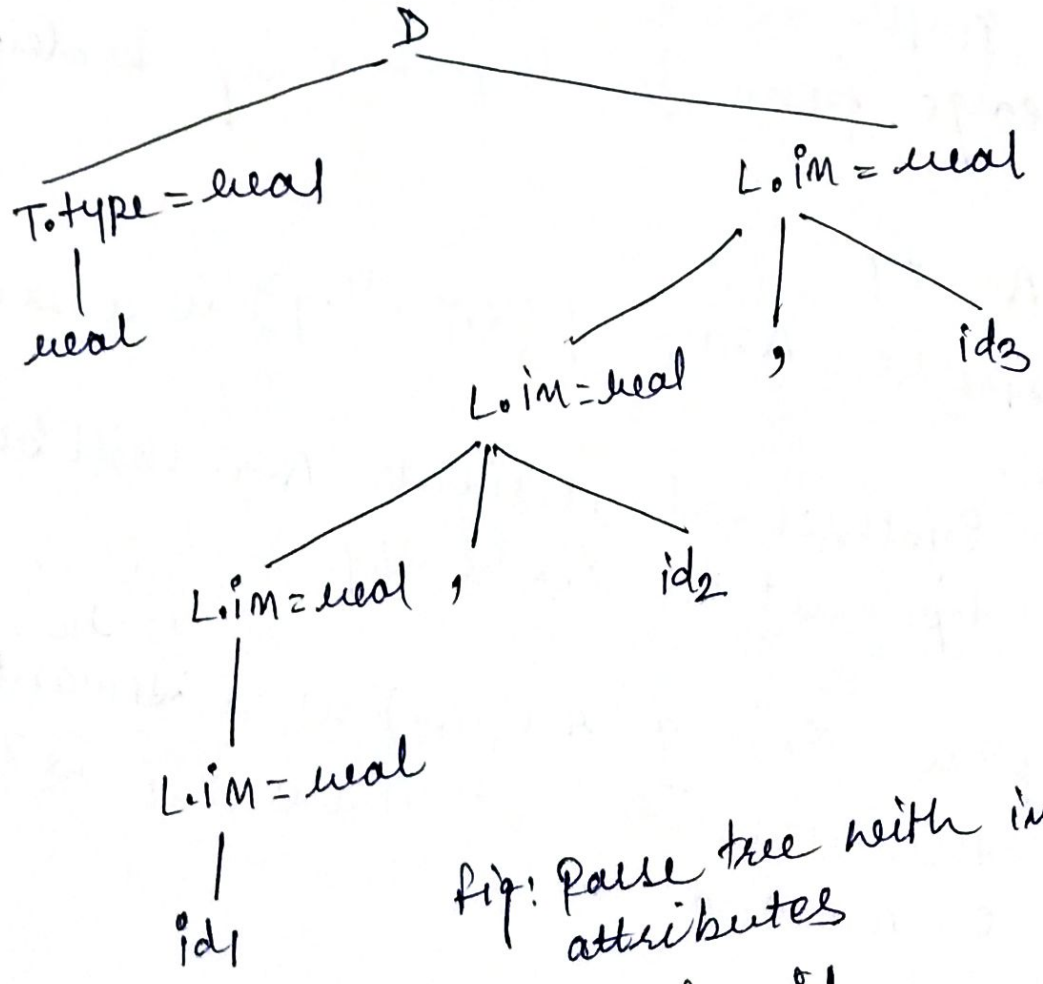


fig: Parse tree with inherited attributes

Ex:- id_1, id_2, id_3

Dependency Graphs:-

- The interdependencies among the inherited and synthesized attributes at the nodes in a parse tree can be depicted by a directed graph called a dependency graph.
- Before constructing a dependency graph for a parse tree, we put each semantic rule into the form $b := f(c_1, c_2 \ldots c_k)$, by introducing a dummy synthesized attribute b for each semantic rule that consists of a procedure call.
- The graph has a node for each attribute and an edge for b from c if b depends on c.

Ex:- $A \rightarrow XY$

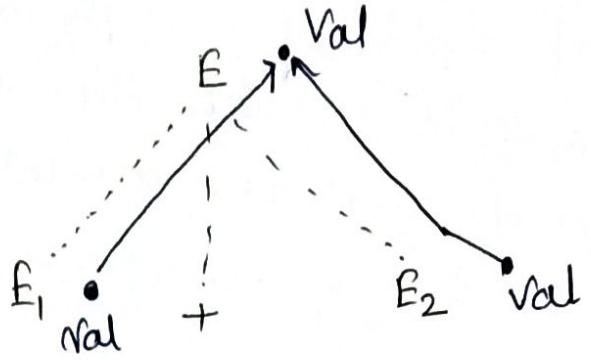
Suppose $A.a := f(X.x, Y.y)$ is a semantic rule.

Synthesized attribute A.a will be dependent on X.x & Y.y.

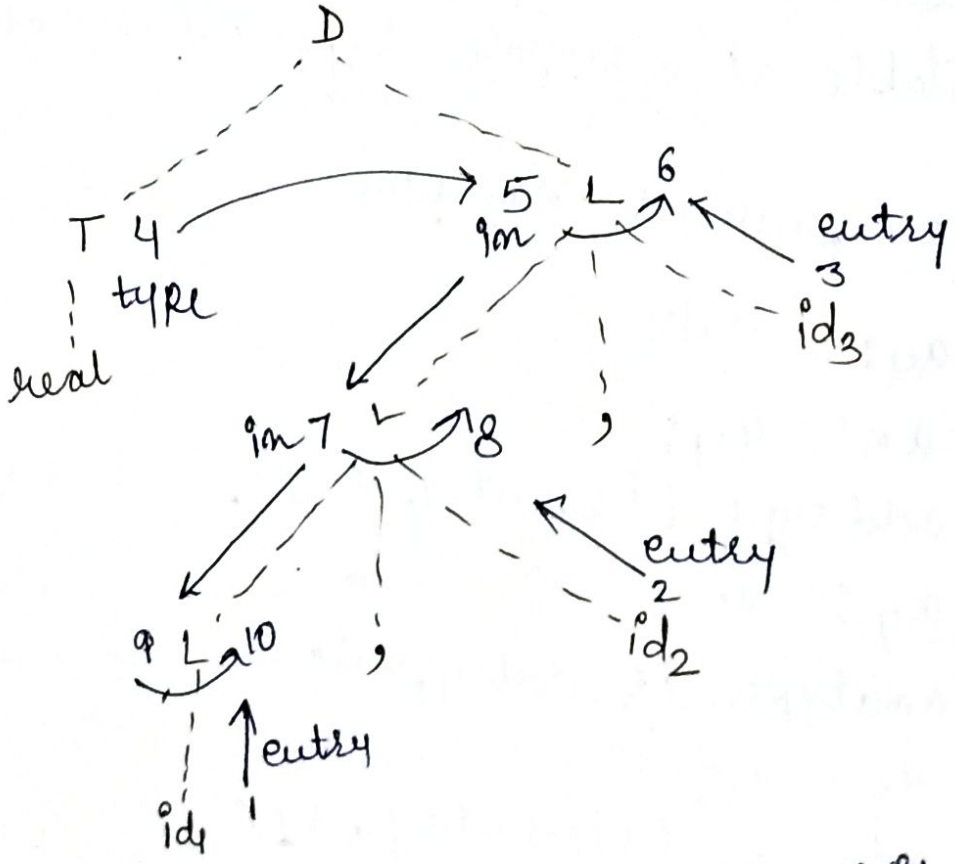
Suppose $X.i := g(A.a, Y.y)$ is a semantic rule then inherited attribute X.i will depend on A.a and Y.y

Ex:- $E \rightarrow E_1 + E_2$

Semantic rule $E.val := E_1.val + E_2.val$



Ex:- id_1, id_2, id_3



- Each of the semantic rules {addtype (id.entry, L.in) associated with the L-productions leads to the creation of a dummy attribute. Nodes 6, 8, 10 are constructed for these dummy attributes.

Evaluation Order:-

- Topological sort is used for this.
- Any topological sort of a dependency graph gives a valid order in which the semantic rules associated with the nodes in a parse tree can be evaluated. That is the dependent attributes $c_1, c_2 \cdots c_k$ in a semantic rule $b := f(c_1, c_2 \cdots c_k)$ are available at a node before f is evaluated.

Ex:- From previous example,

```
a4 := real;
a5 := a4;
addtype (id3.entry, a5);
a7 := a5;
addtype (id2.entry, a7);
a9 = a7;
addtype (id1.entry, a9);
```

Methods for evaluating semantic rules:-

1) Parse-tree methods:- These methods obtain an evaluation order from a topological sort of the dependency graph constructed from the parse tree for each input. This method will fail, if dependency graph contain cycle.

2) Rule-based methods:- At compiler construction time, the semantic rules associated with productions are analyzed, either by hand, or by a special tool.

3) Oblivious methods:- an evaluation order is chosen without considering the semantic rules.

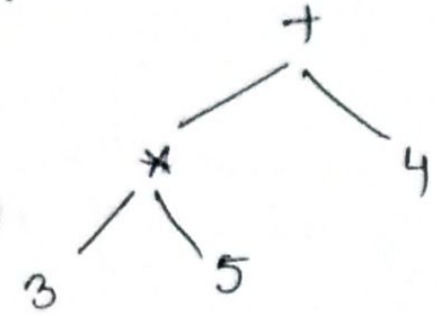
■ construction of syntax Trees:-

- The use of syntax trees as an intermediate representation allows translation to be decoupled from parsing.
- Translation routines that are invoked during parsing have two restrictions:
  1) grammar that is suitable for parsing may not reflect natural hierarchical structure of the constructs in the language.
  2) Parsing method constrains the order in which nodes in a parse tree are considered. This order may not match the order in which info about a construct becomes available.

Syntax Tree:-

- A syntax tree is a condensed form of parse tree useful for representing language constructs.

Ex:- Syntax tree for the tree given in Synthesized attribute is,



Constructing Syntax trees for expressions:-

- Similar to translation of expression into postfix form.

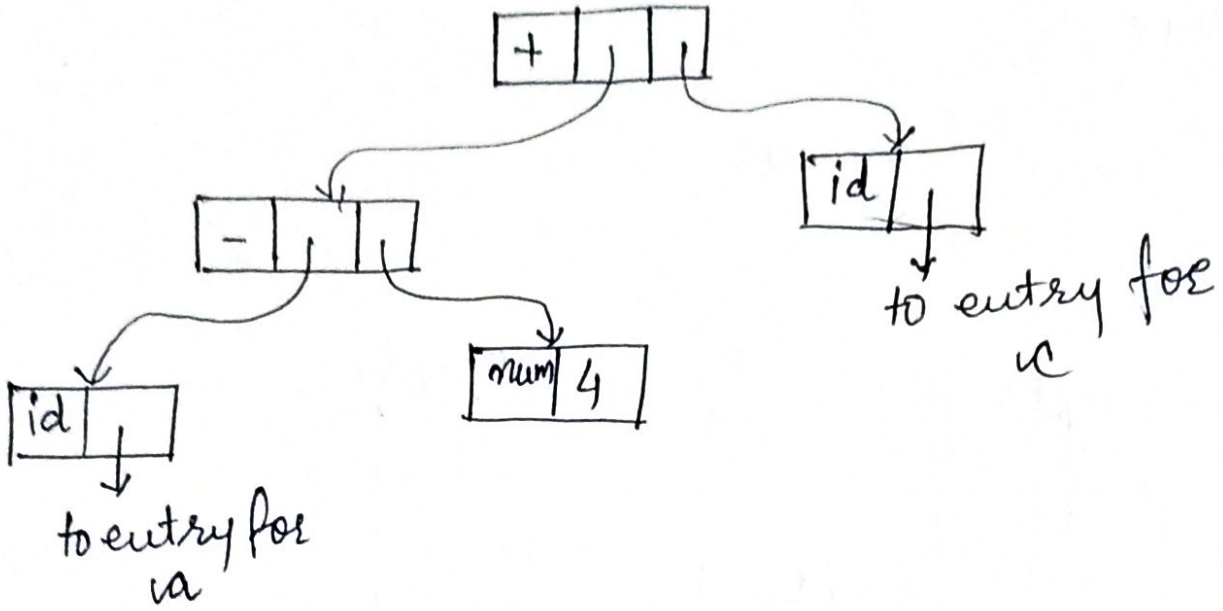
a) mknode (op, left, right): creates an operator node with label op and two fields containing pointers to left and right.

b) mkleaf (id, entry): identifier node with label id, and a field containing a pointer to symbol-table entry for identifier.

c) mkleaf (num, val): number node with label num and a field containing value.

Ex:- exp:- a-4+c

1) P_1 := mkleaf (id, entry a);
2) P_2 := mkleaf (num, 4);
3) P_3 := mknode ('-', P_1, P_2);
4) P_4 := mkleaf (id, entry c);
5) P_5 := mknode ('+', P_3, P_4);

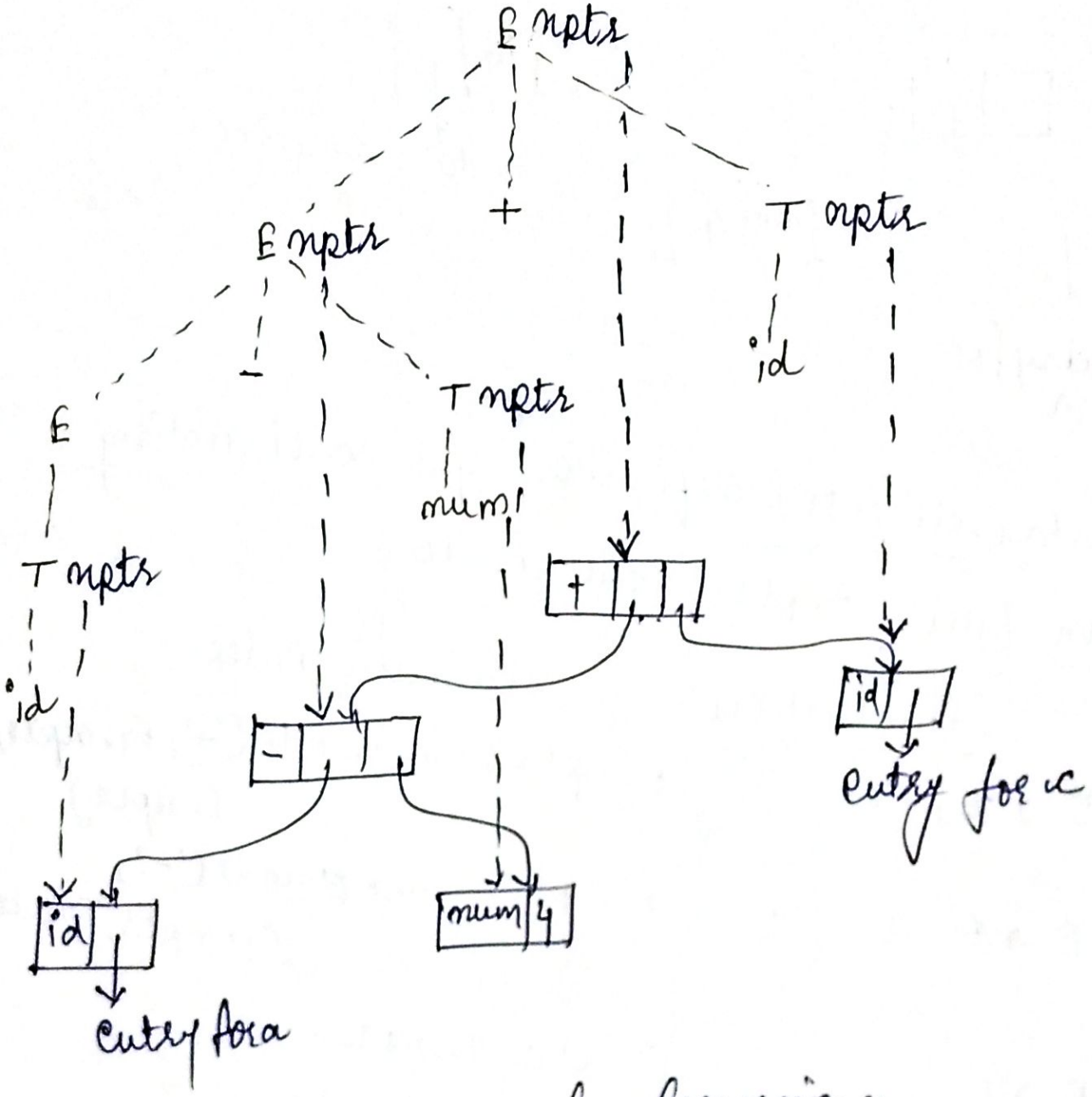


A syntax-directed definition for constructing Syntax trees:- (nptr = node pointer)

Production	Semantic Rules
$E \rightarrow E_1 + T$	E.nptr := mknode ('+', E_1.nptr, T.nptr)
$E \rightarrow E_1 - T$	E.nptr := mknode ('-', E_1.nptr, T.nptr)
$E \rightarrow T$	E.nptr := T.nptr
$T \rightarrow (E)$	T.nptr := E.nptr
$T \rightarrow$ id	T.nptr := mkleaf (id, id.entry)
$T \rightarrow$ num	T.nptr := mkleaf (num, num.val)

Fig:- S-attributed definition

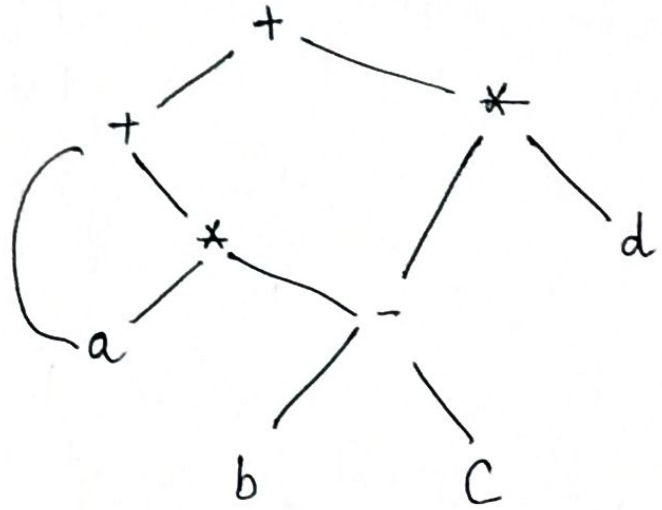
Ex:- a-4+c



Directed Acyclic Graphs for Expressions:-

- A dag for an expression identifies the common subexpressions in the expression. A dag has a node for every subexpression; an interior node represents an operator and its children represent its operands.
- A node in a dag representing a common sub-expression has more than one parent.

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



- 1) $P_1 := mkleaf(id, a)$;
  2) $P_2 := mkleaf(id, a)$;
  3) $P_3 := mkleaf(id, b)$;
  4) $P_4 := mkleaf(id, c)$;
  5) $P_5 := mknode('-', P_3, P_4)$;
  6) $P_6 := mknode('*', P_2, P_5)$;
  7) $P_7 := mknode('+', P_1, P_6)$;
  8) $P_8 := mkleaf(id, b)$;
  9) $P_9 := mkleaf(id, c)$;
  10) $P_{10} := mknode('-', P_8, P_9)$;
  11) $P_{11} := mkleaf(id, d)$;
  12) $P_{12} := mknode('*', P_{10}, P_{11})$;
  13) $P_{13} := mknode('+', P_7, P_{12})$;

- In case of dag
  - when the call $mkleaf(id, a)$ is repeated in line 2, the node constructed by the previous call $mkleaf(id, a)$ is returned, so $P_1 = P_2$
  - lines 8 & 9 are same as 3 & 4
  - line 10 is equal to line 5.

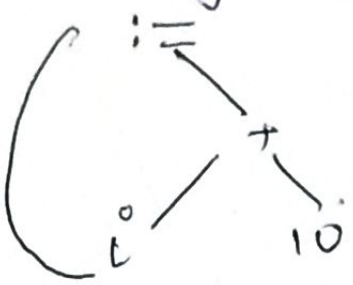
→ In many applications dag nodes are implemented as records also.

Ex:-

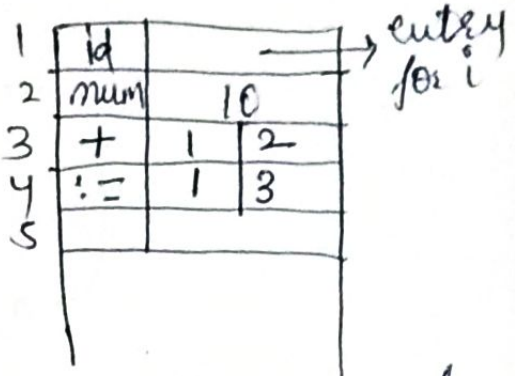
Assignment

i := i + 10

Dag



Representation



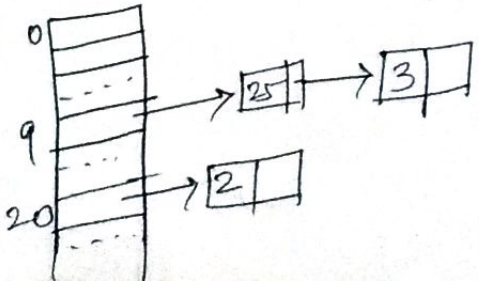
Value-number method for constructing a node in a dag:-

Input :- Label op, node l, node r

Output :- A node with signature ⟨op, l, r⟩

Method :- Search the array for same node m and return it. If same node does not exist then create a new one.

- The search for m can be made more efficient by using k lists, called buckets and using a hashing function h to determine which bucket to search.
- Hash function h computes the number of a bucket from the value of op, l and r. If m is not in the bucket ~~op~~, then a new node n is created and added to this bucket.



Bottom-up Evalution of S-attributed definitions:-

→ we have already seen how to use SDD to specify translations, now we will see how to implement translators for them.

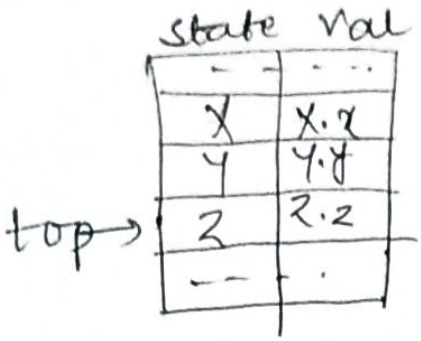
→ In this section, we will study only a class of SDD: the S-attributed definitions, means SDD with only synthesized attributes.

→ Synthesized attributes can be evaluated by a bottom-up parser as i/p is being parsed. The parser can keep the values of synthesized attributes associated with the grammar symbols on its stack. whenever a reduction is made, the values of the new synthesized attributes are computed from the attributes appearing on the stack for the grammar symbols on the right side of the reducing production.

Synthesized attributes on the parser stack:-

- Implemented using LR-parser generator
- A bottom-up parser uses a stack to hold info about subtrees that have been parsed. we can use extra fields in the parser stack to hold the values of

Synthesized attributes.



- if the ith state symbol is A, then val[i] will hold the value of attribute associated with the parse tree node corresponding to this A.

Suppose $A \to XYZ$
then Semantic rule $A.a := f(X.x, Y.y, Z.z)$

- Before XYZ is reduced to A, the value of the attribute Z.z is in val[top], Y.y in val[top-1] & X.x in val[top-2]. After reduction, top is decremented by 2 and A is put in state[top].

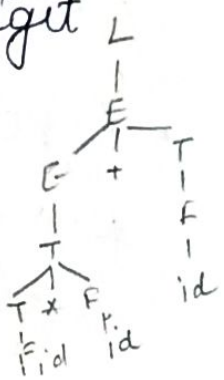
Example:-

Production	Code fragement
	print(val[top])
$L \to E\,n$	
$E \to E_1 + T$	val[ntop] := val[top-2] + val[top]
$E \to T$	
$T \to T_1 * F$	val[ntop] := val[top-2] * val[top]
$T \to F$	
$F \to (E)$	val[ntop] := val[top-1]
$F \to digit$	

- When a production with r symbols on the right side is reduced, the value of ntop is set to top - r+1. After each code fragment is executed, top is set to ntop.

Ex:- 3*5+4n

Input	state	val	Production Used
3*5+4n	—	—	
*5+4n	3	3	
*5+4n	F	3	F→digit
*5+4n	T	3	T→F
5+4n	T*	3-	
+4n	T*5	3-5	
+4n	T*F	3-5	F→digit
+4n	T	15	T→T*F
+4n	E	15	E→T
4n	E+	15-	
n	E+4	15-4	
n	E+F	15-4	F→digit
n	E+T	15-4	T→F
n	E	19	E→E+T
	En	19-	
	L	19	L→En



L - Attributed definitions:-

- When translation takes place during parsing, the order of evaluation of attributes is linked to order in which nodes of a parse tree are created by the parsing method. dfvisit is a natural order for many top to down and bottom to up translation.

- Depth-first evaluation order:-

```
procedure dfvisit (n: node);
begin
    for each child m of n, from left to
        right do begin
        evaluate inherited attributes of m;
        dfvisit (m)
    end;
    evaluate synthesized attributes of
        n
end
```

- L-attributed definitions:
  A syntax-directed definition is L-attributed if each inherited attribute of X_j, $1 \le j \le n$, on the right side of $A \to X_1 X_2 \cdots X_n$, depends only on
  1) the attributes of the symbols $X_1, X_2, \ldots,$

x_{j-1} to the left of x_j in the production and

2. the inherited attributes of A.

Ex: $A \rightarrow LM$

Here L's inherited attributes will depend on A and M's ——— A & L.

- Translation Schemes:-

→ A translation scheme is a context-free grammar in which attributes are associated with grammar symbols and semantic actions enclosed between braces {} are inserted within the right side of production.

→ For every production $A \rightarrow x_1 \ldots x_n$, the semantic rule formed the string for A by concatenating the strings for $x_1 \ldots x_n$. Now we could perform translation by simply printing the literal strings in the order they appeared in the semantic rules.

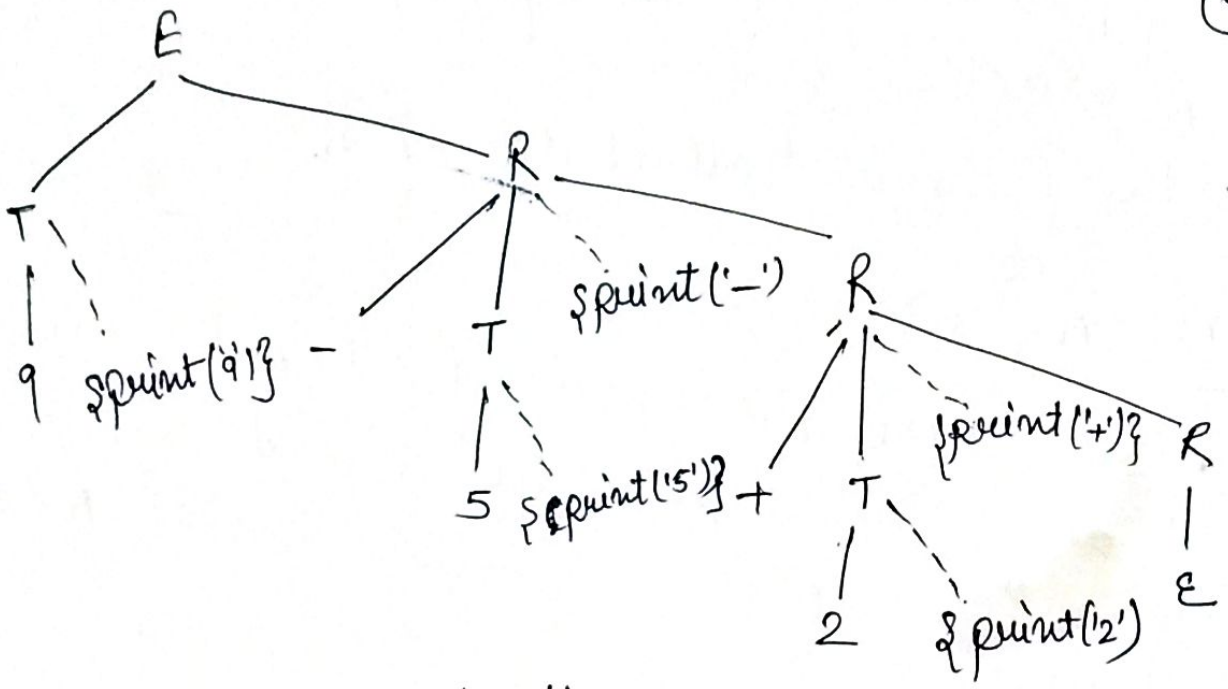
Ex:- A simple translation scheme that maps infix exp into postfix exp.

$E \rightarrow TR$

$R \rightarrow$ addop T {print(addop.lexeme)} $R \mid \varepsilon$

$T \rightarrow$ num {print(num.val)}

E-> TR
R-> +TR/ -TR/ε
T->num



Input → 9-5+2 (infix)
output → 95-2+ (postfix)

- When designing a translation scheme, we must ensure that an attribute value must available when an action refers to it.
- The easiest case occur when only synthesized attributes are needed. We place each semantic rule at the end of the right side of the associated production.

Ex:-

Production	Semantic Rule
$T \to T_1 * F$	$T.val := T_1.val \times F.val$

production & semantic rule

$T \to T_1 * F$ { $T.val := T_1.val \times F.val$ }

- If we have both inherited & synthesized attributes, we must be careful:

1.) An inherited attribute for a symbol on the right side of a production must be computed in an action before that symbol

2.) An action must not refer to a synthesized attribute of a symbol to the right of the action.

3.) A synthesized attribute for the nonterminal on the left can only be computed after all attributes it references have been computed. The action computing such attributes can usually be placed at the end of the right side of the production.

Ex! L-attributed SDD ~~for~~ which following above three rules,

Production	Semantic Rules
$S \to B$	$B.ps := 10$ inherited att. (point size) $S.ht := B.ht$ synthesized, height
$B \to B_1 B_2$	$B_1.ps := B.ps$ $B_2.ps := B.ps$ $B.ht := max(B_1.ht, B_2.ht)$
$B \to text$	$B.ht := text.h \times B.ps$

Fig:- SDD for size and height of boxes.

$S \rightarrow \{B.ps := 10\}$
$B \{s.ht := B.ht\}$

$B \rightarrow \{B_1.ps := B.ps\}$
$B_1 \{B_2.ps := B.ps\}$
$B_2 \{B.ht := max(B_1.ht, B_2.ht)\}$

$B \rightarrow text \{B.ht := text.h \times B.ps\}$

Top-Down Translation :-

→ Here we implement L-attributed definitions during predictive parsing. Here we work with translation schemes rather than SDD.

Eliminating left recursion from a translation scheme:-

• Here we apply transformations to translation schemes with synthesized attributes.

Ex:-

$E \rightarrow E_1 + T$	$\{E.val := E_1.val + T.val\}$
$E \rightarrow E_1 - T$	$\{E.val := E_1.val - T.val\}$
$E \rightarrow T$	$\{E.val := T.val\}$
$T \rightarrow (E)$	$\{T.val := E.val\}$
$T \rightarrow num$	$\{T.val := num.val\}$

fig:- Translation scheme with left recursive grammar (synthesized attributes)

E →

After Removing left Recursion.

$E \rightarrow TR$

$R \rightarrow +TR \mid -TR \mid \varepsilon$

$T \rightarrow (E)$

$T \rightarrow num$

Fig:- Transformed translation scheme with right-recursive grammar.

$E \rightarrow T \quad \{R.i := T.val\}$

$R \quad \{E.val := R.s\}$

$R \rightarrow +$

$T \quad \{R_1.i := R.i + T.val\}$

$R_1 \quad \{R.s := R_1.s\}$

$R \rightarrow -$

$T \quad \{R_1.i := R.i - T.val\}$

$R_1 \quad \{R.s := R_1.s\}$

$R \rightarrow \varepsilon \quad \{R.s := R.i\}$

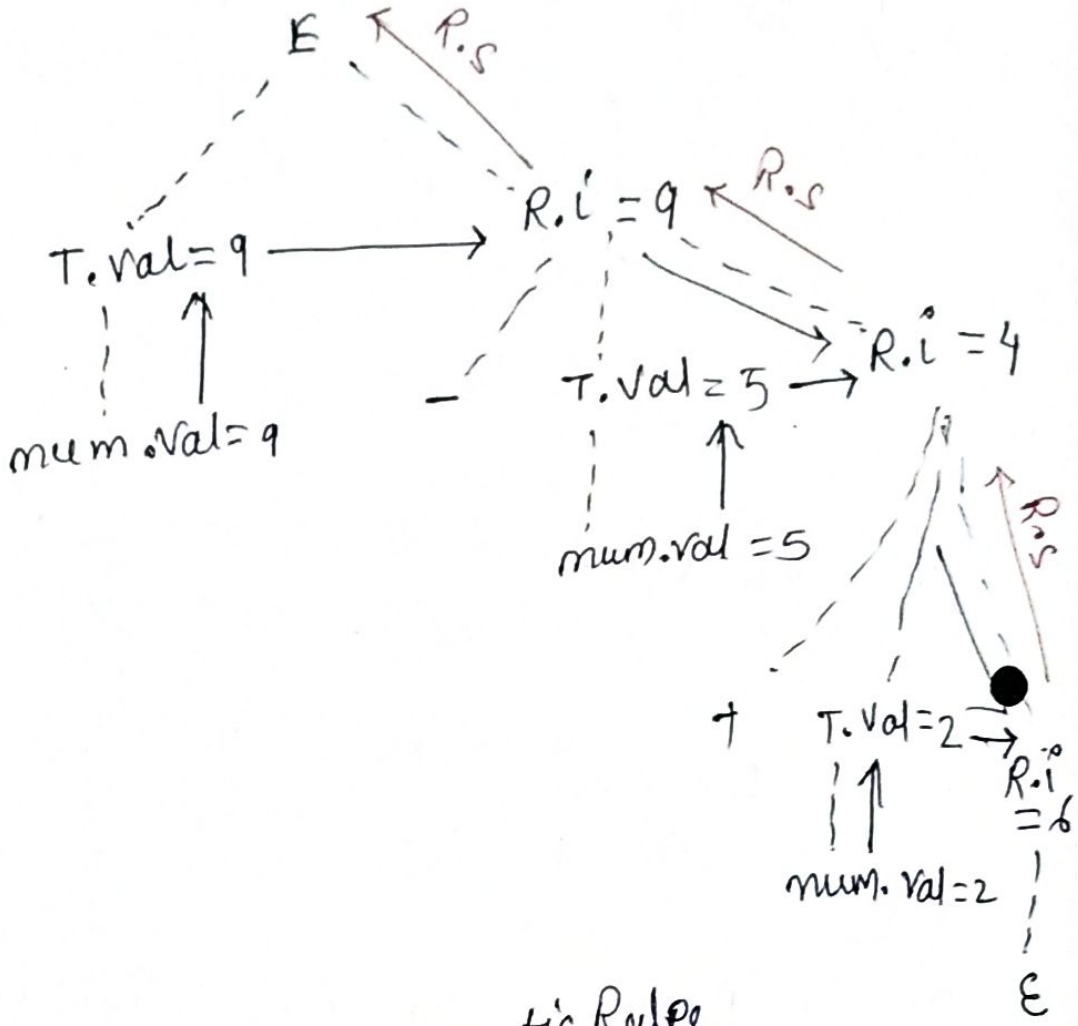
$T \rightarrow ($

E

$) \quad \{T.val := E.val\}$

$T \rightarrow num \quad \{T.val := num.val\}$

Fig:- Evaluation of exp 9-5+2



Ex:- Production

$E \rightarrow E_1 + T$

$E \rightarrow E_1 - T$

$E \rightarrow T$

$T \rightarrow (E)$

$T \rightarrow id$

$T \rightarrow num$

Semantic Rules

E.nptr = mknode('+', E_1.nptr, T.nptr)

E.nptr = mknode('-', E_1.nptr, T.nptr)

E.nptr = T.nptr

T.nptr = E.nptr

T.nptr = mkleaf(id, id.entry)

T.nptr = mkleaf(num, num.val)

After removing left recursion.

$E \rightarrow TR$

$R \rightarrow +TR \mid -TR \mid \varepsilon$

$T \rightarrow (E)$

$T \rightarrow id$

$T \rightarrow num$

Fig:- Transformed translation scheme for constructing syntax trees.

$E \rightarrow T \quad \{R.i = T.nptr\}$
$\quad R \quad \{E.nptr = R.s\}$

$R \rightarrow +$
$\quad T \quad \{R_1.i = mknode('+', R.i, T.nptr)\}$
$\quad R_1 \quad \{R.s = R_1.s\}$

$R \rightarrow -$
$\quad T \quad \{R_1.i = mknode('-', R.i, T.nptr)\}$
$\quad R_1 \quad \{R.s = R_1.s\}$

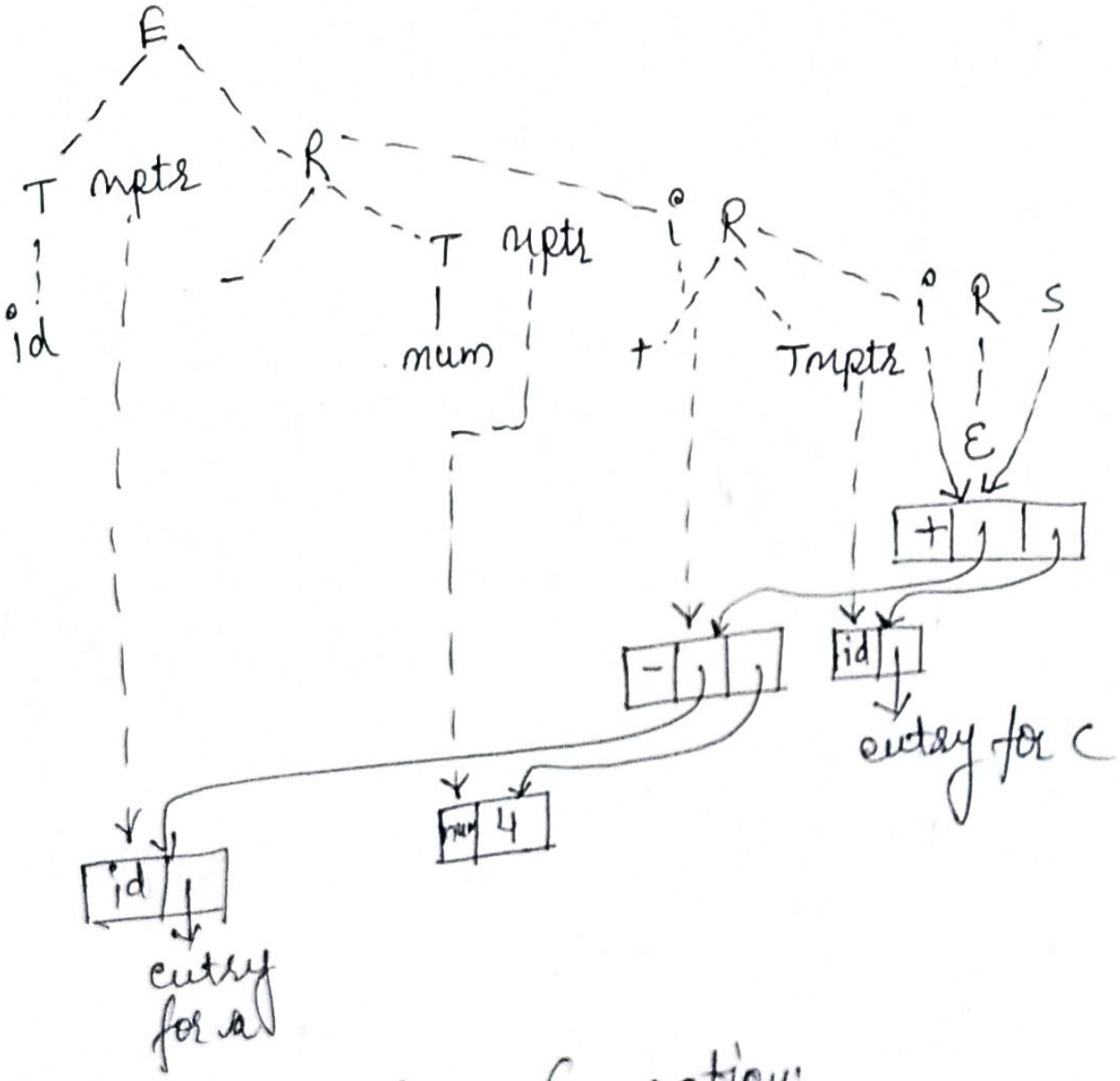
$R \rightarrow \varepsilon \quad \{R.s = R.i\}$

$T \rightarrow ($
$\quad E$
$\quad) \quad \{T.nptr = E.nptr\}$

$T \rightarrow id \quad \{T.nptr = mkleaf(id, id.entry)\}$

$T \rightarrow num \quad \{T.nptr = mkleaf(num, num.Val)\}$

Fig:- Use of inherited attributes to construct syntax trees. (9-4+C)



Intermediate Code Generation:-

- Although a source program can be translated directly into the target language, but there are some benefits of using a machine-independent intermediate form are:
  a) A compiler for a different machine can be created by attaching a back end.
  b) A machine-independent code optimizer can be applied to the intermediate representation.

Three-Address Code:-

- General form

$$x := y \; op \; z$$

where x, y and z are names, constants or compiler-generated temporaries. Op is an operator like arithmetic, logical or boolean operator.

- No built-up arithmetic expressions are permitted in right side, as there is only one operator on the right side of a statement.

Ex:- $x + y * z$

$t_1 := y * z$

$t_2 := x + t_1$

- complicated arithmetic expressions and of nested flow-of-control statements makes three-address code desirable for target code generation and optimization.
- Three-address code is a linearized representation of a syntax tree or a dag.

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

$t_1 := -c$
$t_2 := b * t_1$
$t_3 := -c$
$t_4 := b * t_3$
$t_5 := t_2 + t_4$ (code for syntax tree)
$a := t_5$

$t_1 := -c$
$t_2 := b * t_1$
$t_5 := t_2 + t_2$
$a := t_5$
code for the dag.

Types of Three-Address statements:-

1) Assignment statement → $x := y \text{ op } z$
2) Assignment statement → $x := \text{op } y$ (op can be unary operator like unary minus, logical negation)
3) Copy stmts → $x := y$
4) Unconditional jump goto L.
5) conditional jumps like if x relop y goto L.
   (<, =, >= etc.)
6) param x and call p, n

```
param x1
param x2
---
param xn
call p, n
```

} parameters

} Call to procedure p with n parameters

7) Indexed assignments → $x := y[i]$ or $x[i] := y$
8) Address and pointer assignments → $x := \&y$, $x := *y$, $*x := y$

Implementation of Three-address statements:-

- Abstract form of intermediate code and Implemented as records with fields for operator and operands.

1) Quadruples:-

- A record structure with four fields, op, arg1, arg2 & result.

Ex:- x := y op z

Put y in arg1, z in arg2 and x in result

Ex:- Quadruples for the previous ex in TAC

	op	arg1	arg2	result
0	uminus	c		t_1
1	*	b	t_1	t_2
2	uminus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	:=	t_5		a

- Statements ~~like~~ with unary operators like x := -y or x := y do not use arg2
- Operators like param use neither arg2 nor result.
- Conditional/unconditional jumps put the target label in result.

Triples:-

- In this, three-address are represented by records with only three fields: op, arg1 and arg2. The fields arg1 & arg2, for an op, are either pointers to the Symbol table or pointers into the triple structure.

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

pointers to the triple structure

Symbol-table pointers

Ex:- x[i] := y

	op	arg1	arg2
(0)	[]=	x	i
(1)	assign	(0)	y

Ex:- x := y[i]

	op	arg1	arg2
(0)	=[]	y	i
(1)	assign	x	(0)

Indirect Triples:-

- In this approach we list pointers to triples, rather than listing the triples themselves.

statement		op	arg1	arg2
(0)	(14)	(14) uminus	c	
(1)	(15)	(15) *	b	(14)
(2)	(16)	(16) uminus	c	
(3)	(17)	(17) *	b	(16)
(4)	(18)	(18) +	(15)	(17)
(5)	(19)	(19) assign	a	18)

Comparison of Representations: The use of indirection:-

* Quadruple → • using temporary can immediately access the location for that temp via the symbol table.
• In an optimizing compiler, statements are often moved around for doing optimization. If we move a stmt computing x, the stmt does not require changes. But in triples notation, moving a stmt that defines a temp value requires us to change all its references.

* Triples:- Save space
* Indirect triples:- a stmt can be moved by reordering the statement list.

Boolean Expressions:-

- Boolean expressions have two primary purposes,
  - → They used to compute logical values.
  - → Also used in conditional expressions in stmts that alter the flow of control such as if-then, if-then-else, while-do.

E_1 relop E_2

↓

arithmetic exp

- Boolean exp are composed of boolean operators (and, or, not) applied to boolean variables or relational exp.

Ex:- $E \rightarrow E$ or $E \mid E$ and $E \mid$ not $E \mid (E) \mid$ id relop id / true / false

- relop – $<, \leq, =, \neq, >, \geq$

Methods of translating boolean expressions:-

↗ Encode true and false numerically and to evaluate a boolean exp analogously to an arithmetic exp. 1 represent true / 0 represent false.

d) The 2nd method of implementing boolean exp is by flow-of-control, that is, representing the value of a boolean exp by a position reached in a program. For ex, given exp E_1 or E_2, if we determine E_1 is true, then we can conclude that the entire exp is true without having to evaluate E_2.

Numerical Representation:-

- 1 denote true and 0 denote false.
- Exp will be evaluated completely, from left to right.

Ex:- a or b and not c

Three address sequence:-

$t_1 := not\ c$
$t_2 := b\ and\ t_1$
$t_3 := a\ or\ t_2$

Ex:- if a<b then 1 else 0

Three address code sequence:-

100: if a<b goto 103
101: t:=0
102: goto 104
103: t:=1
104:

Short-Circuit Code:-

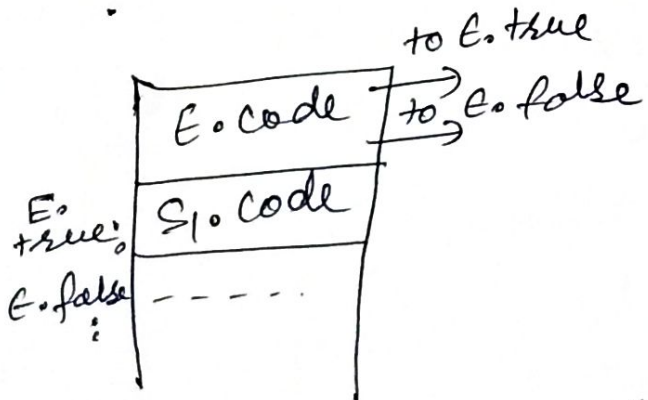
- In this it is not necessary to evaluate the entire expression, this is called "short circuit" or "jumping" code.

Ex:- 100: if a<b goto 103
101: t_1 := 0
102: goto 104
103: t_1 := 1
104: - ...

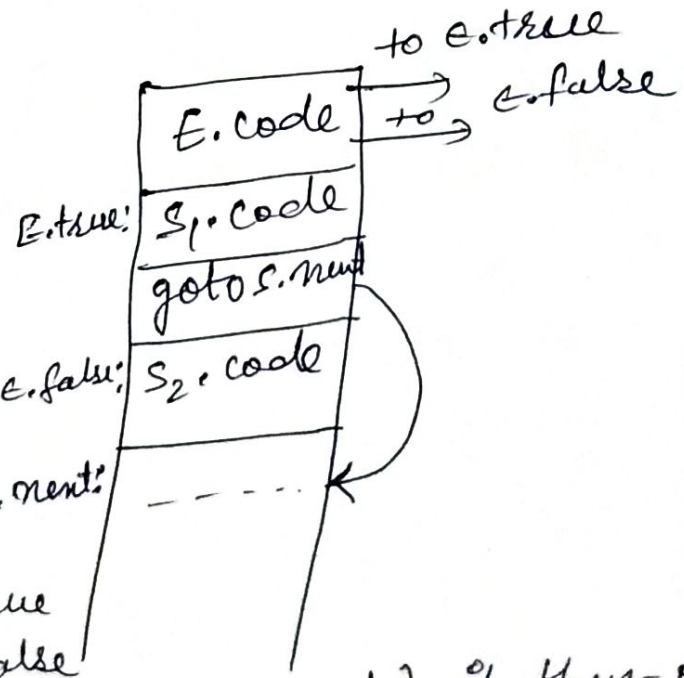
Here we can tell what value t_1 will have by whether we reach stmt 101 or 103.

control statements:-

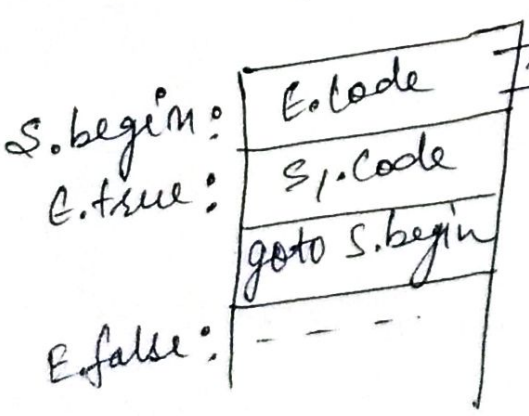
S → if E then S_1 | if E then S_1 else S_2
| while E do S_1



a) if-then



b) if-then-else



c) while-do

Examples of three address code:-

1) x = 1;
   if (x > 1)
      y = y + 3;

1. x = 1
2. if x > 1 goto (4)
3. goto (6)
4. t_1 = y + 3
5. y = t_1
6. stop

2.) x = 5;
   if (x < 5)
      z = z + 2;
   else
      y = y * 2;

1. x = 5
2. if x < 5 goto (4)
3. goto (7)
4. t_1 = z + 2
5. z = t_1
6. goto (9)
7. t_2 = y + 2
8. y = t_2
9. stop

3.) int i
   i = 1
   while i < 10 do
      if x > y then
         i = x + y
      else
         i = x - y

1. i = 1
2. if i < 10 goto (4)
3. goto (12)
4. if x > y goto (6)
5. goto (9)
6. t_1 = x + y
7. i = t_1
8. goto (2)
9. t_2 = x - y
10. i = t_2
11. goto (2)
12. Stop

Backpatching:- leaving the labels as empty and filling them later is called backpatching.

4.)
```
c=0
do
{
   if(a<b) then
       x++;
   else
       x--;

   c++;
}while(c<5);
```

```
1.  c = 0
2.  if(a<b) goto(4)
3.  goto (9)
4.  t1 = x+1
5.  x = t1
6.  t2 = c+1
7.  c = t2
8.  goto (14)
9.  t3 = x-1
10. x = t3
11. t4 = c+1
12. c = t4
13. goto (14)
14. if c<5 goto (2)
15. goto (16)
16. stop
```

5.) if (a<b & c<d or e<f)

```
101. if(a<b) goto(104)
102. t1 = 0
103. goto(105)
104. t1 = 1
105. if (c<d) goto (108)
106. t2 = 0
107. goto (109)
108. t2 = 1
109. if(e<f) goto (112)
110. t3 = 0
111. goto (113)
112. t3 = 1
113. t4 = t1 and t2
114. t5 = t4 or t3
```

6) for (i=0; i<10; i++)
    a = b+c;

```
    i = 0
L:  if (i < 10) goto L1
    goto Last
L1: t1 = b+c
    a = t1
    t2 = i+1
    i = t2
    goto L
Last: stop
```

7) Switch (i+j)
{
  case 1:
    a = b+c;
    break;
  case 2:
    P = Q+R;
    break;
  default:
    x = y+z;
    break;

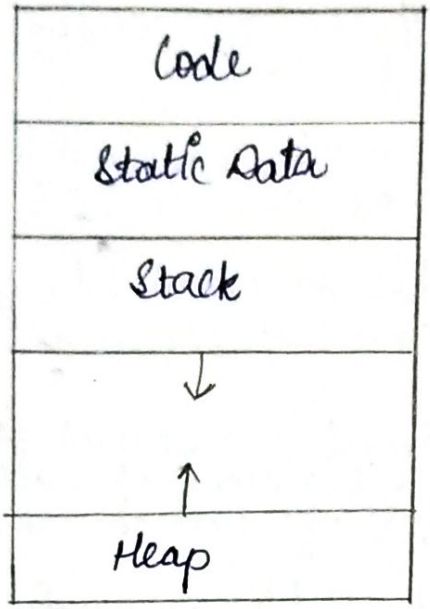
```
    t = i+j
    goto test
test: if (t==1) goto L1
      if (t==2) goto L2
      goto L3
L1: t1 = b+c
    a = t1
    goto Last
L2: t2 = Q+R
    P = t2
    goto Last
L3: t3 = y+z
    x = t3
    goto Last
Last: stop
```

8) for(i=1; i<=10; i++)

```
{
    a[i] = x * 5;
}
```

```
i=1
L: t1 = x * 5
   t2 = &a
   t3 = sizeof(int)
   t4 = t3 * i
   t5 = t2 + t4
   *t5 = t1
   i = i+1
   if (i<=10) goto L
   goto last
last: stop
```

Storage Organization:-



- Generated target code: Compiler can place it in a statically determined area.
- Static Data: which data whose size is known at compile time.
- Stack: when a call occurs, execution of an activation is interrupted and info about the status of the machine, such as the value of the program counter and machine registers, is saved on the stack. When control returns from the call, this activation can be restarted after restoring the values of relevant registers and setting the program counter to the point immediately after the call.
- Heap: used for dynamic memory allocation.
- Stack and heap grows in opposite side.

Activation Records: info needed by a single execution of a procedure is managed using a contiguous block of storage called an activation record or frame, consisting of the collection of fields.

returned value	used by called procedure to return a value to the calling procedure.
actual parameters	used by calling procedure to supply parameters to the called procedure
optional control link	points to the activation record of the caller.
optional access link	non local data held in other activation records.
saved machine status	info about the status (program counter & registers) of machine just before the procedure call.
local data	local data to a procedure.
temporaries	temporary values during exp evaluation.

Compile-Time layout of local data:-

- Run-time storage comes in blocks of contiguous bytes, (smallest unit of addressable memory).
- The amount of storage needed for a name is determined from its type. An elementary data type, such as a char, int or real can usually be stored in an integral number of bytes. For aggregate, such as array & record uses large space to hold its all components. For easy access, aggregates also use contiguous block of bytes.

- For local data memory locations are fixed at compile time. Variable-length data is kept outside this field. We keep a count of the memory locations that have been allocated for previous declarations from the count we determine a relative address, or offset is the difference b/w the address of the position and the data object.
- The storage layout for data objects is strongly influenced by the addressing constraints of the target machine. Like some machines have 16 bits for int and some have 32 bits for int.

Storage Allocation strategies:-

Static Allocation:-

- In static allocation, names are bound to storage as the program is compiled, so there is no need for a run-time support package. since the bindings do not change at run time, every time a procedure is activated, its name are bound to the same storage locations. This property allows the values of local names to be retained

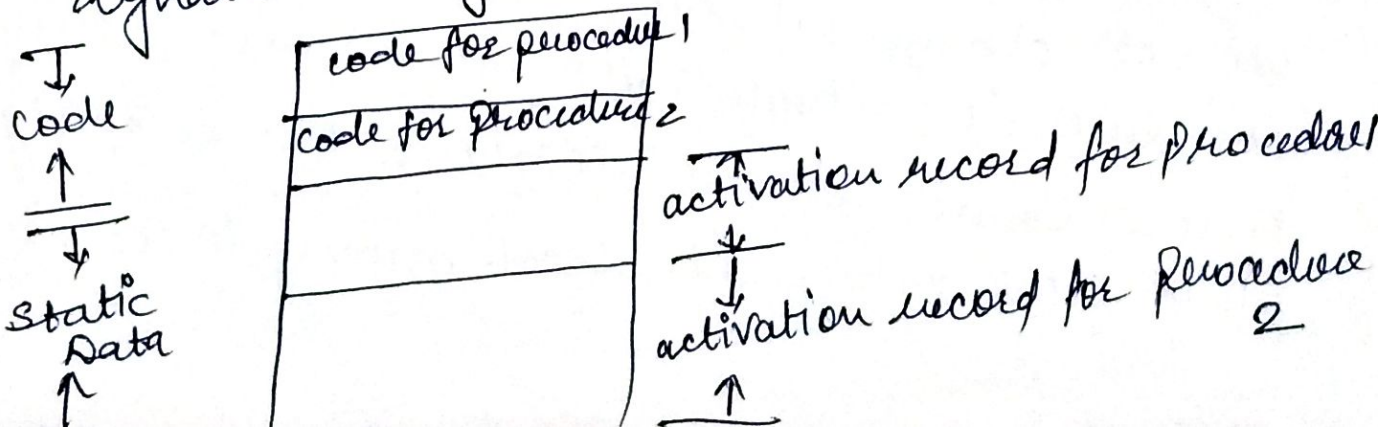
across activations of a procedure. That is, when control returns to a procedure, the values of the locals are the same as they were when control left the last time.

- From the type of Name, compiler determine the amount of storage to set aside for that name.
- In this scheme, at compile time we can therefore fill in addresses at which target code can find the data it operates on. similarly, compiler get to know on which addresses, which info is saved.

Limitations:-

1) Size of data object and constraints on its position in memory must known at compile time.
2) Recursive procedures are restricted, bcoz all activations of a procedure use the same bindings for local names.
3) Data structures cannot be created dynamically.

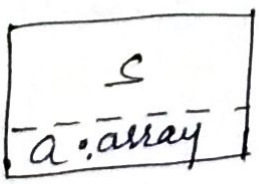
Ex:- Fortran 77

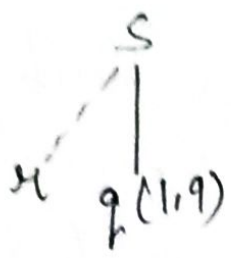


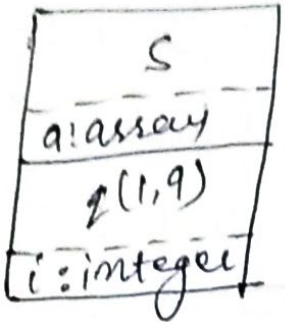
Stack Allocation:-

- Based on the idea of a control stack; storage is organized as a stack, and activation records are pushed and popped as activations begin and end respectively.
- storage for locals are freshly allocated at each time when a call made, bcoz the storage for locals disappears when the activation record is popped.
- Register top marks top of the stack.
- Suppose procedure q has an activation record of size a is pushed, top will be ↑ by a and after control returns from q, top is decremented by a.

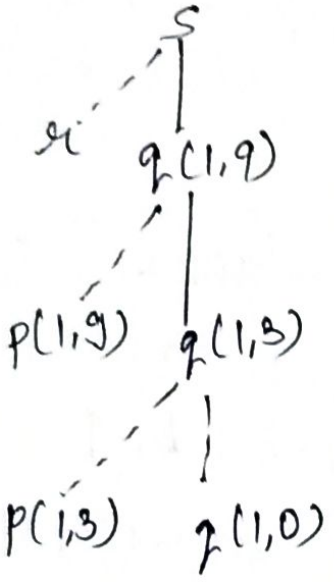
Ex:-

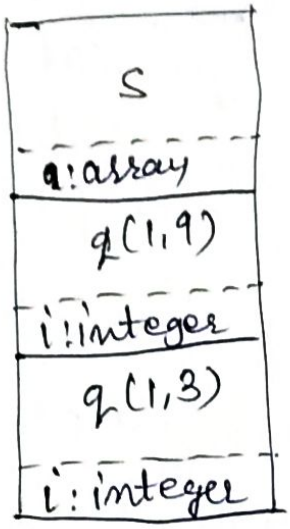
Position in activation tree	Activation Records on the stack	Remarks
s		frame for s
r / s		r is activated





frame for r has been popped and q(1,9) pushed



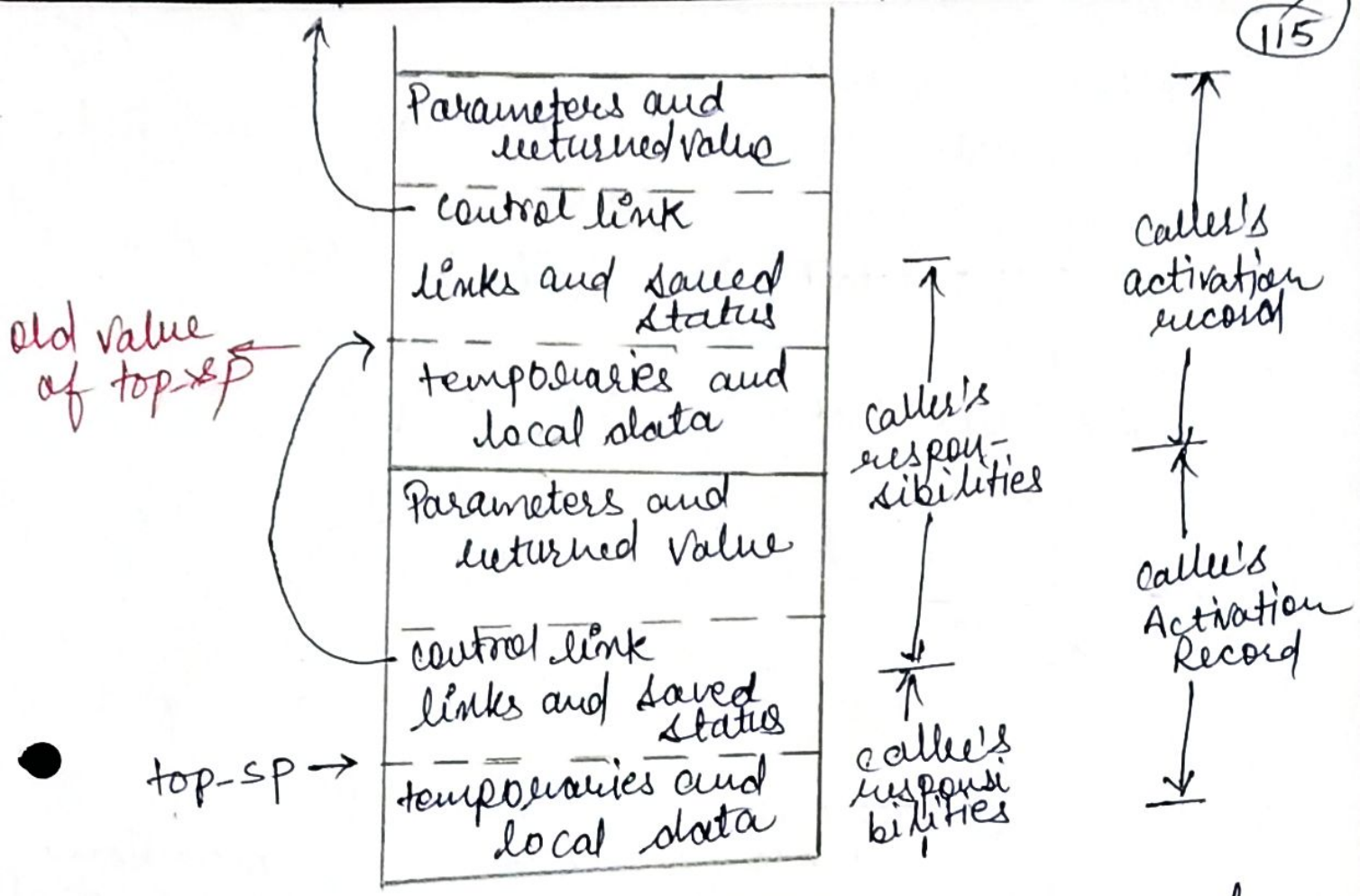


control has just returned to q(1,3)

Calling Sequences:-

- A call sequence allocates an activation record and enters info into its fields.
- A return sequence restores the state of the machine so the calling procedure can continue execution.

fixed-length data:-

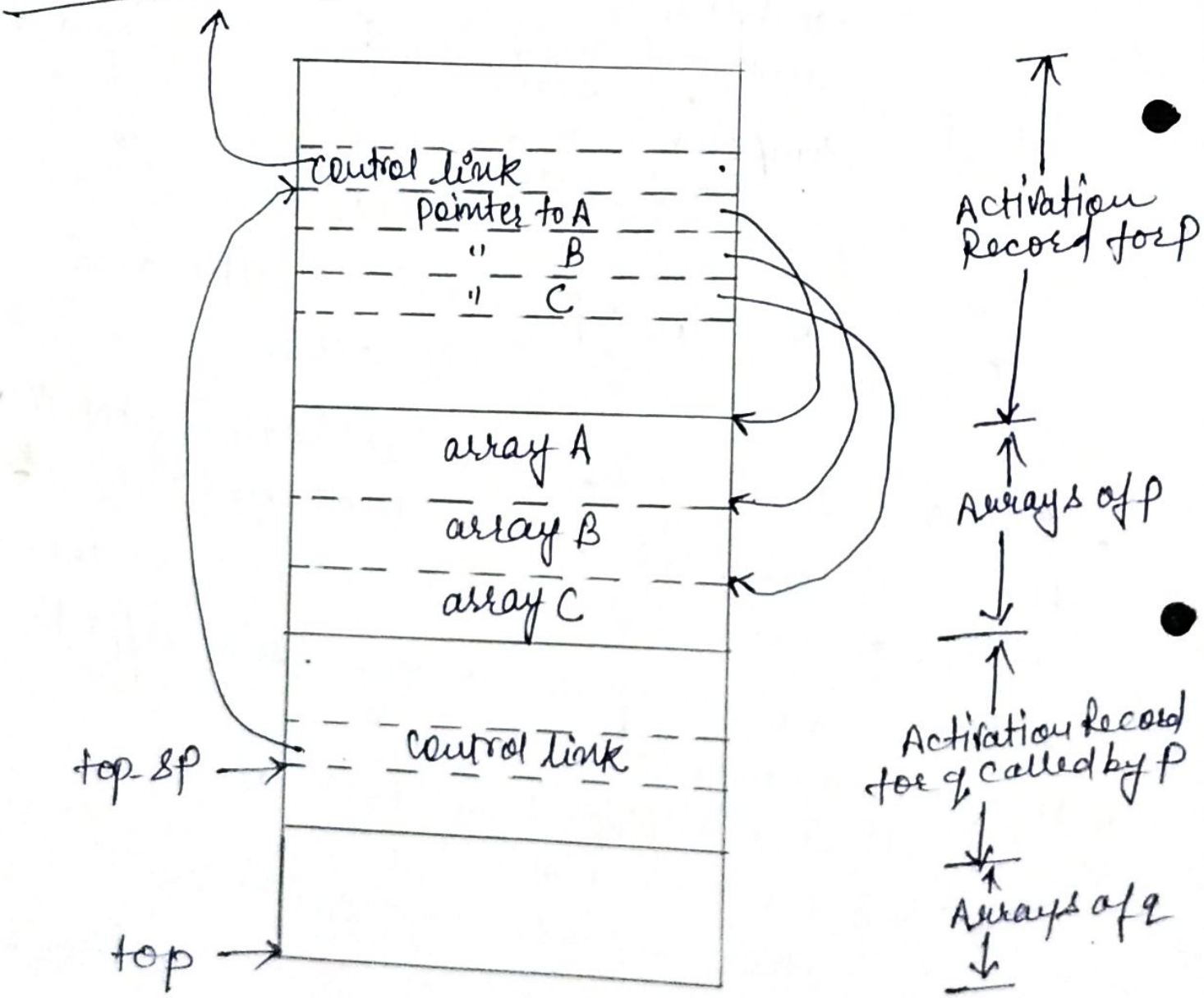


Division of tasks b/w caller and callee.

- In run-time stack, the activation record of the caller is just below that for the callee. So that callee can access parameters & returned values of callee by using offset.
- Sequence of tasks:-
  - * Caller evaluates actual parameters
  - * caller stores old value of top-sp into the activation record of callee.
  - * top-sp is moved to new position.
  - * Callee saves register values and other status info.
  - * callee initializes its local data and begins execution.

- Return sequence: -
  - * callee places a return value next to the activation record of caller.
  - * Callee restores top-sp and other registers
  - * Caller copy the value of top-sp

Variable-length Data:-



- P has three local arrays. storage for these arrays are not part of activation record, only a pointer to the beginning of each array appears in the AR.
- q is called by P. AR for q begins after array for P, and variable length arrays of q begin beyond that.
- Access data using — top (actual top of stack)
  — top_sp (used to find local data)

- when q returns,
  * top = top_sp — length of machine-status and parameter field s in q's activation record
  * top_sp — copied from the control link of q.

Dangling References:-

- A dangling reference occurs when there is a reference to storage that has been deallocated.

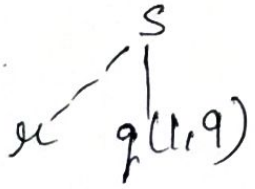
Ex:
```
main()
{ int *P;
  P = dangle();
}

int *dangle()
{ int i=23;
  return &i;
}
```

- Here when control returns to main from dangle, storage for local is freed. Since p in main refers to this storage, the use of p is a dangling reference.

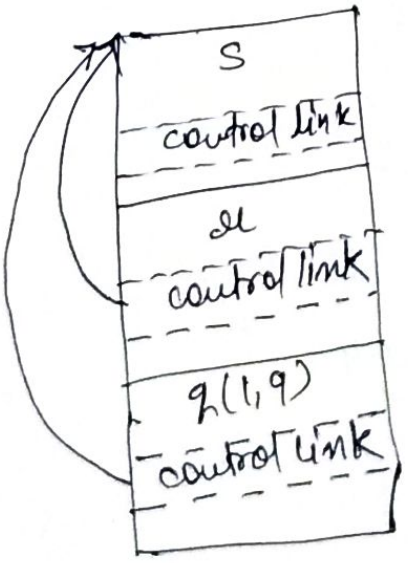
Heap Allocation:-

- Stack allocation will not work, if
  - * Values of local names must be retained when an activation ends.
  - * A called activation outlives the caller.
- Heap allocation can be used for these. It parcels out pieces of contiguous storage, as needed for activation records or other objects.

Eg:- Activation Tree



AR in the Heap



Remarks

Retained activation record for r.

- for each size of interest, keep a linked list of free blocks of that size.
- fill a request for size s with block of size s', where s' is the smallest size greater than or equal to s.
- For large blocks of storage use the heap manager.

Access to nonlocal names:-

Blocks:-

- A block is a statement containing its own local data declarations.

  { declarations statements }

Ex:-

```
main()
{   int a=0;
    int b=0;
    {   int b=1;
        {   int a=2;
            printf("%d%d\n", a, b);
        }
        {   int b=3;
            printf("%d%d\n", a, b);
        }
    }
}
```

B_0 B_1 B_2 B_3

```
    printf("%d %d \n", a, b);
}
printf("%d %d \n", a, b);
}
```

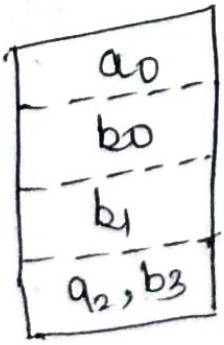
Fig: Blocks in a C program

Declaration	Scope
int a=0;	$B_0 - B_2$
int b=0;	$B_0 - B_1$
int b=1;	$B_1 - B_3$
int a=2;	B_2
int b=3;	B_3

output a, b

2 1
0 3
0 1
0 0

- can be implemented using stack. Space for the declared name can be allocated when the block is entered, and deallocated when control leaves the block.
- An alternative implementation is to allocate storage for a complete procedure body at one time.



- a_2 and b_3 may be assigned the same storage bcoz they are in blocks that are not alive at the same time.

Lexical Scope without nested procedures- (121)

Ex:-
```
int a[11];
readarray() { ... a ... }
int partition(y,z){ int y, z; ... a ... }
quicksort(m,n){ int m,n; ... a ... }
main() { ... a ... }
```

- Here nonlocal occurrences of a in readarray, partition and main refer to the array declared on line 1.
- In absence of nested procedures, the stack-allocation strategy for local names can be used directly for a lexically scoped language like C.
- Any non-local names can be allocated statically. Position for this storage is known at compile time, so we can use it directly.
- Local names to an activation at the top of stack, easily accessible through the top pointer.
- Benefit of static allocation for nonlocals is that declared procedures can freely be passed as parameters and returned as results.

Lexical scope with Nested Procedures:-

- A nonlocal occurrence of a name 'a' in a Pascal procedure is in the scope of the most closely nested declaration of 'a' in the static program text.

Ex:-

```
Program sort (input, output);
  var a: array [0..10] of integer;
      x:
  !
  procedure readarray;
    var i ---
    - - -
  procedure exchange (- - );
    - - -
    - - -
  procedure quicksort(- - -);
    var k, lv - - -
    function partition (- - -)
      var i, j - - -
      begin - - - a - - -
        - - -
        - - - exchange (- - )
      end { - - - }
  begin - - - end {quicksort};
begin - - - end {sort}
```

This 'a' is first search in partition, then in quicksort, after that in Sort.

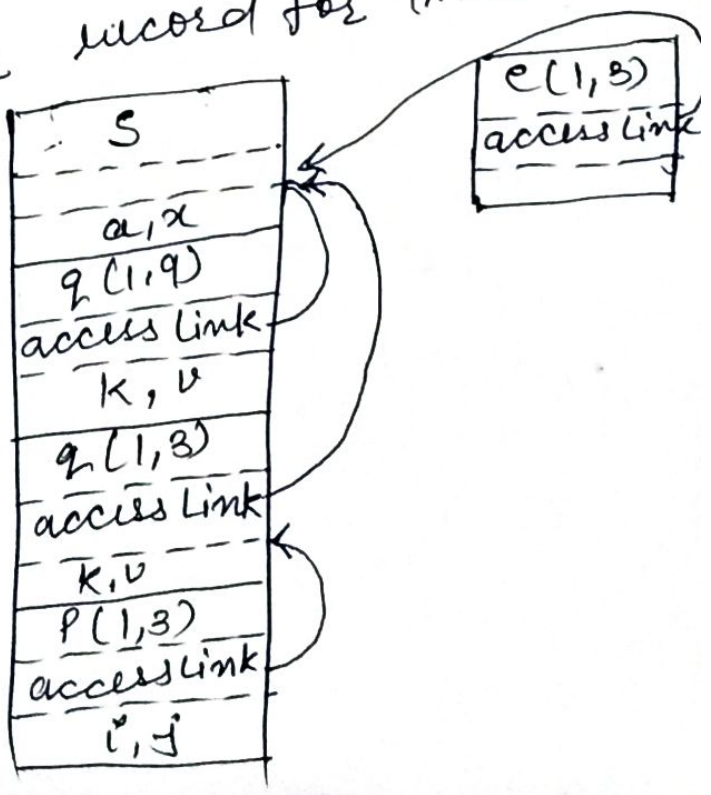
This exchange is first searched in quicksort, then in Sort.

1) Nesting Depth:-

- Nesting depth of a procedure is used to implement lexical scope.
- Let the name of the main program be at nesting depth 1; we add 1 to the nesting depth as we go from an enclosing to an enclosed procedure.
- In the above example 'quicksort' is at nesting depth 2, while partition ~~is~~ ~~at~~ ~~the~~ ~~is~~ is at nesting depth 3.

2) Access Links:-

- A direct implementation of lexical scope for nested procedures is obtained by adding a pointer called an access link to each activation record. If procedure p is nested immediately within q in the source text, then the access link in an activation record for p points to the access link in the record for most recent activation of q.

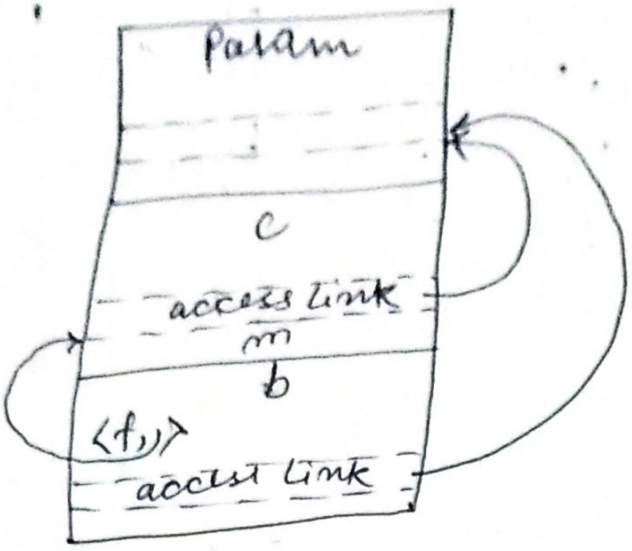


3) Procedure Parameters:-

Ex:-1. Program param (input, output)

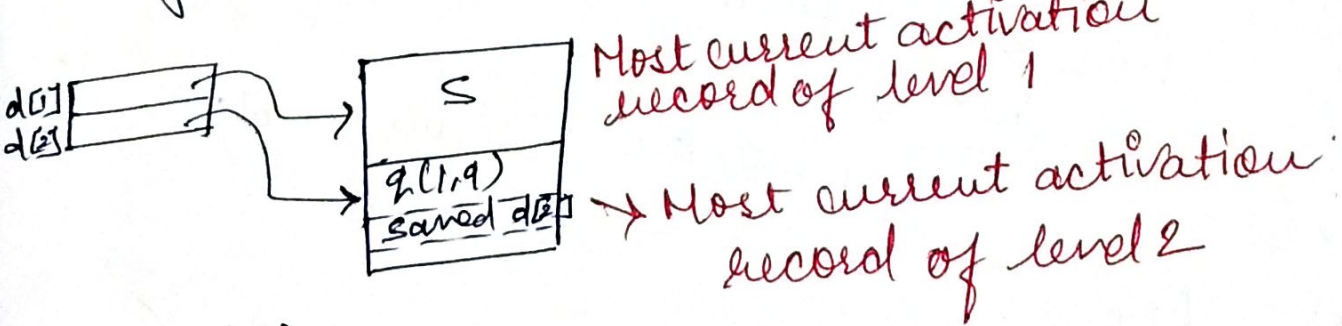
```
2.  procedure b (function h (n: integer): integer);
3.      begin ---
4.  procedure c;
5.      var m: integer;
6.      function f (n: integer): ——
7.          begin f: ---
8.      begin m:=0; b(f) end {c};
9.  begin
10. c
11. end
```

- Lexical scope rules apply even when a nested procedure is passed as a parameter.
- In procedure c at line 8, f (function) is passed as parameter to procedure b. So when a nested procedure is passed as a parameter must take its access link along with it.
- When procedure c passes f, it will also pass access link of f to b.

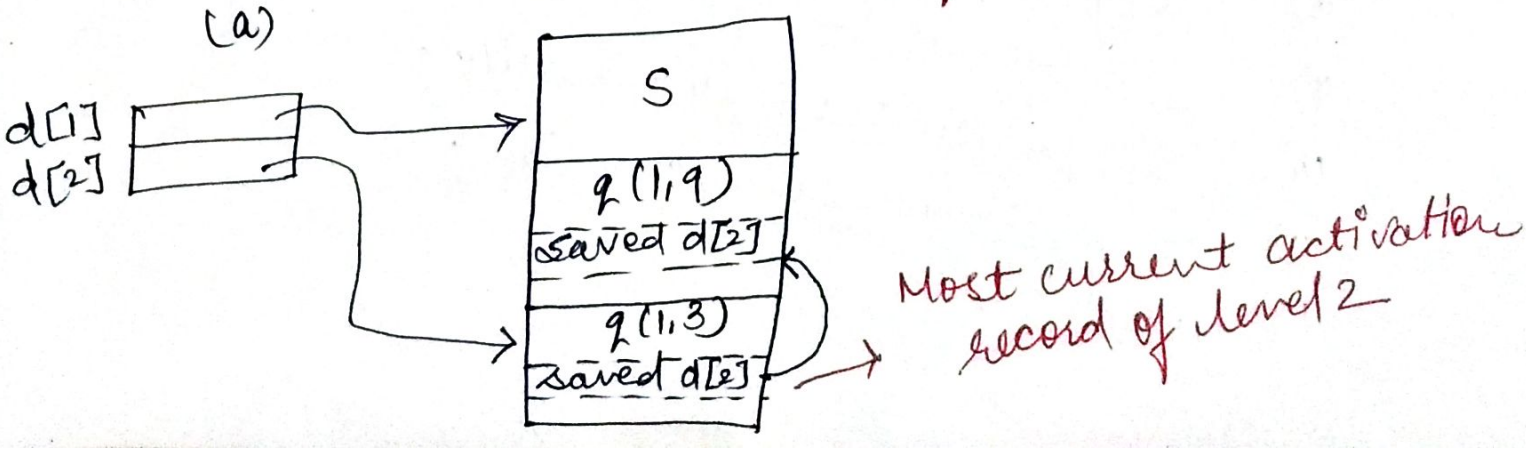


4) Displays:-

- Faster access to nonlocals than with access links can be obtained using an array d of pointers to activation records, called a display.
- Suppose control is in an activation of a procedure p at nesting depth j. Then, the first j-1 elements of the display point to the most recent activations of the procedures at depth j-1.



(a)



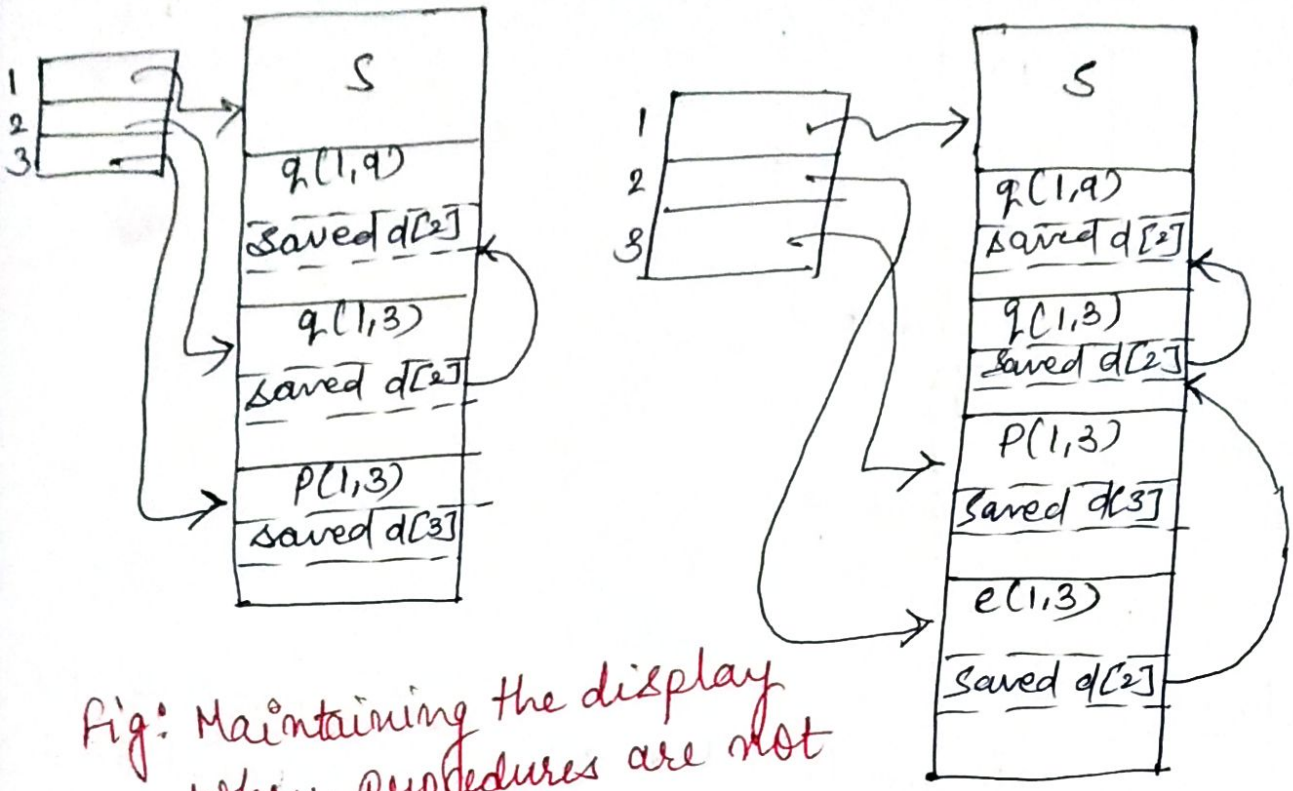


Fig: Maintaining the display when procedures are not passed as parameters

Dynamic Scope:-

Ex:-
```
1) Program dynamic( ——);
2)     Var x ——
3)     Procedure show;
4)         —— write(x) ——
5)     Procedure small;
6)         Var x ——
7)             —— x = 0.125; show end;
8)     begin
9)         x = 0.25
10)        show; small; write;
11)        show; small; write;
12) end
```

- In above example, when show is called on lines 10-11 in main program, 0.250 is written bcoz x is local to main function. But when show is called x = 0.125 is used as this x is local to ~~show~~ small.
- This is called dynamic scope.
- Two approaches for this,

1) Deep access:- in this control links are used rather than access links. In this search for a nonlocal can go deep into the stack, may be for the first activation record.

2) Shallow access:- hold the current value of each name in statically allocated storage. when a new activation of a procedure p occurs, a local name n in p takes over the storage statically allocated for n.

Parameter Passing:-

- Both nonlocals and parameters are used by the procedure below,

```
ex:- procedure exchange (i, j: integer);
     var x: integer;
     begin
          x := a[i]; a[i] = a[j]; a[j] = x
     end
```

- Here i and j are parameters and a is nonlocal name.

- Different parameter passing methods exists like call by value, call by reference, copy restore etc.
- Diff b/w parameter-passing methods are based primarily on whether an actual parameter represents an r-value, an l-value or the text of the actual parameter itself.

Call-by-Value:-

- Simplest method of passing parameters.
- Formal parameters are send from caller procedure. And the storage for formals is in the activation record of the called procedure.
- Called uses these parameters, modify them but these modifications are not seen at caller procedure.

Ex:-
```
swap (x, y)
{ int x, y;
  int temp;
  temp = x;
  x = y;
  y = temp;
}
```

```
main()
{ int a=1, b=2;
  swap (a, b);
  print ("a=%d, b=%d",
         a, b);
}
```

a and b will not be modified at main function.

Call-by-Reference:-

- when parameters are passed by reference (call-by-address or call-by-location)

Ex:-
```
swap(x,y)
{ int *x, *y;
  int temp;
  temp = *x
  *x = *y
  *y = temp
}
main()
{  int a=1, b=2;
   swap(&a, &b);
   printf("a is %d, b is %d", a, b);
}
```

Copy-Restore:- (Fortran uses this)

- Hybrid between call-by-value and call-by-reference.
- Also known as copy-in copy-out or value-result.
- Before control flows to the called procedure, the actual parameters are evaluated. The r-values of the actuals are passed to the called procedure as in call by value. In addition, the l-values of those actual parameters having l-values are determined before the call.
- when control returns, the current r-values of the formal parameters are copied back into the l-values of the actuals, using the l-values computed before the call.

Ex:- program copyout (input, output);

```
var a : integer;
procedure unsafe (var x : integer);
    begin x := 2; a := 0 end;
begin
    a := 1; unsafe(a); writeln(a)
end.
```

Here a's value is passed to x as 1. But in procedure unsafe x will be assign 2 and this value 2 is assign to a by restoring the value of a while returning from function.

Call-by-Name :-

- It is traditionally defined by the copy-rule of Algol, which is:
  * The procedure is treated as if it were a macro that is, its body is substituted for the call in the caller, with the actual parameters literally substituted for the formals. Such a literal substitution is called macro-expansion or in-line expansion.

Ex:-

```
#define f(var)
    do {
        var = 3;
        i += 1;
        var = 4;
    } while(0)
```

```
void g(void) {
    int a[2];
    int i;
    i = 0;
    F(a[i]);
}
```

- Here is we define f as a function then it will work like call-by-value. But macro will replace it's calling in a function and works like call-by-name and give correct output.

Symbol Tables:-

- A compiler uses a symbol table to keep track of scope and binding information about names.
- The symbol table is searched every time a name is encountered in the source text. Changes to the table occur if a new name or new info about an existing name is discovered.
- It is useful for a compiler to be able to grow the symbol table dynamically if necessary at compile time, rather than fixing the size of symbol table at compile time.

Symbol-Table Entries:-

- Each entry in the symbol table is for the declaration of a name. The format of entries does not have to be uniform, bcoz the info saved about a name depends on the usage of the name.
- Each entry can be implemented as a record.

- For uniformity, it can be possible to kept max infor outside the table entry, with only a pointer to this infor stored in the record.

- Infor about a name can be entered into the symbol table at various times. like keywords are entered into the symbol table initially. Further entries are done by lexical analyzer. Attributes values are filled as infor being available.

Ex:- int x;
struct x {float y, z; };

Here two entries will be done in Symbol table for x, one as int and one as struct.

- Attributes of a name are entered in response to declarations.

Characters in a Name:-

- When we are doing entry for first time it is considered as lexeme, but when a lexeme in input found in symbol table it is considered as a name. A common representation of a name is a pointer to a symbol-table entry for it.

- If there is a modest upper bound on the length of a name, then the characters in the name can be stored in the symbol-table entry. If there is no limit on the length of a name, the indirect scheme can be used.

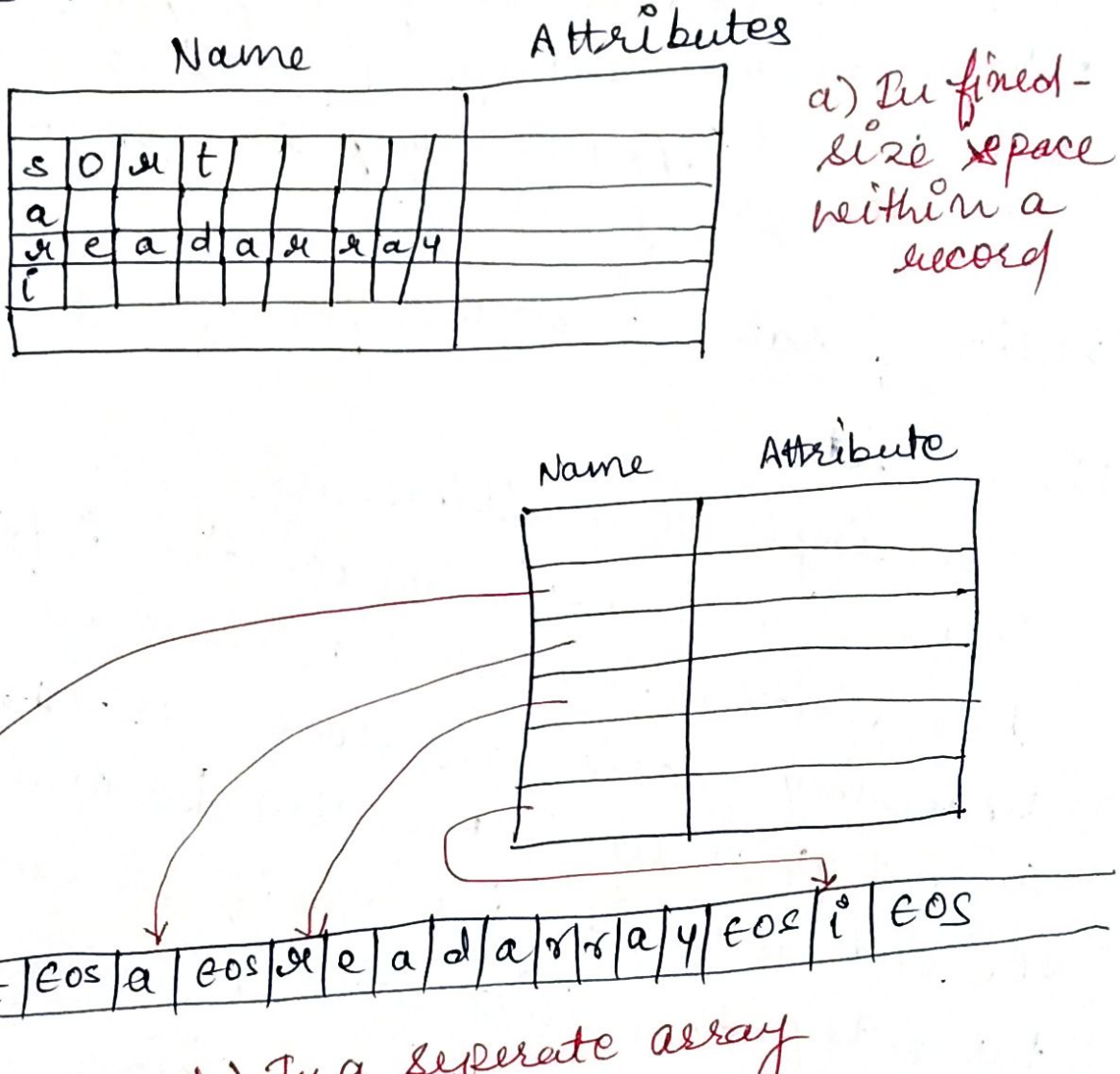


Fig:- Storing the characters of a name.

Storage Allocation Information:-

- Info about the storage locations that will be bound to names at run time is kept in

the Symbol table.

- Static Storage:- If target code is in assembly language then assembler will take care of storage. If target code is in machine language, then the position of each data object is relative to fixed origin.
- Names whose storage is allocated on a stack or heap, the compiler does not allocate storage at all.

The List of Data Structure for Symbol Tables:

1) Linear List:-

- Use single or multiple arrays to store names and their associated infor. Array's last position is marked as available and upcoming entries are done here. For searching an entry, array is processed backwards from the end to the beginning. If we don't find the entry then fault occurs.
- Making an entry and searching a name are independent operations.
- In general, an occurrence of a name is in the scope of most closely nested declaration of the name. It can be implemented using list by making a fresh entry for a name every time it is declared.

- By searching from available towards the beginning of the array, we are sure to find the most recently created entry.

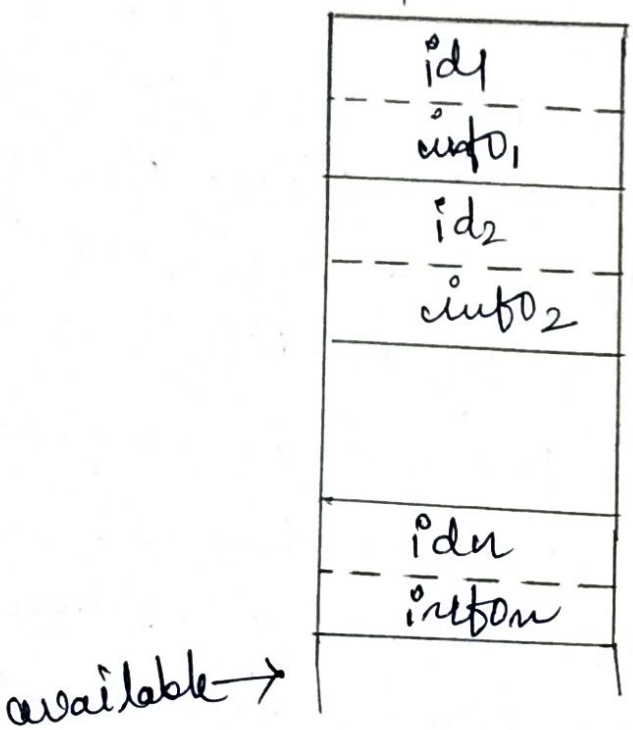


fig: linear list of records.

- Symbol table contains → n names
  - * if we do not check for multiple entries in symbol table and make a new entry, it will require constant work
  - * if multiple entries are not allowed, n - require work to check whether entry already available or not.

n - also cost for an inquiry

So Total work for inserting n names and making e inquiries = $c n(n+e)$

where c = constant time requires for machine operations.

- As n and e increases too much time is spent for performing operations.

8) Hash Tables:-

- Hashing is another data structure used for symbol table creation. Here we are considering "open hashing", where "open" refers to the property that there need be no limit on the number of entries that can be made in the table.
- To perform e inquiries on n names requires = $\frac{n(n+e)}{m}$ time, for constant m.
- As m increases, the space requirement grows, so a time-space tradeoff is involved.

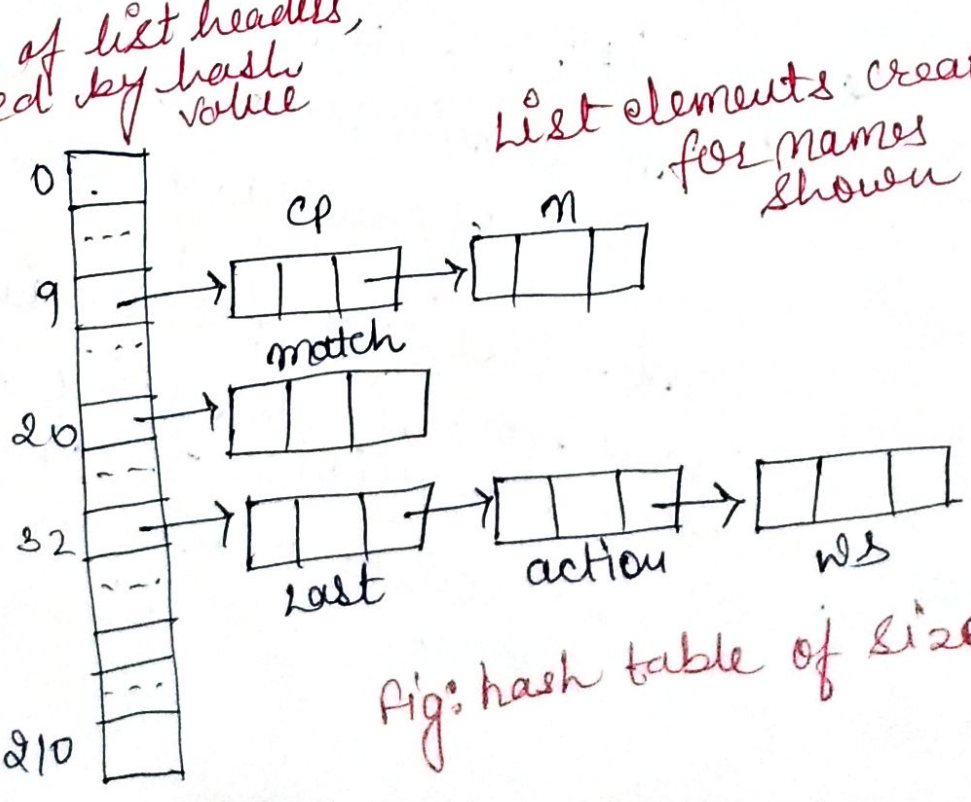


Fig: hash table of size 211.

- Two parts of hash table
  * A hash table consisting of a fixed array of m pointers to table entries.
  * Table entries organized into m separate linked lists, called buckets.

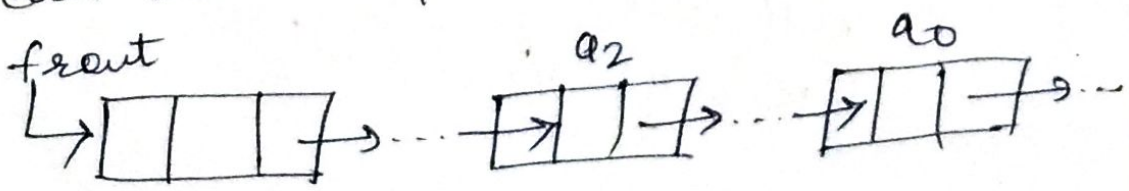
- To determine whether there is an entry for string s in the symbol table, we apply a hash function h to s, such that h(s) returns an integer b/w 0 and m-1. If s is in the symbol table, then it is on the list numbered h(s). If s is not yet in the symbol table, it is entered by creating a record for s that is linked at the front of the list numbered h(s).
- The choice of m depends on the intended application for a symbol table. Choosing m to be large enough, make table lookup a negligible time.
- Approach for designing hash function,
  * Determine a +ve integer k from the characters $c_1, c_2 \cdots c_k$ in string s by adding ascii values of its characters
  * convert k into a b/w 0 and m-1. (Divide by m and taking remainder. Generally m is taken as prime no.

- Variations for calculating h,
  * add ittle characters values.
  * $h_i = d h_{i-1} + c_i$ (d is a constant)

*

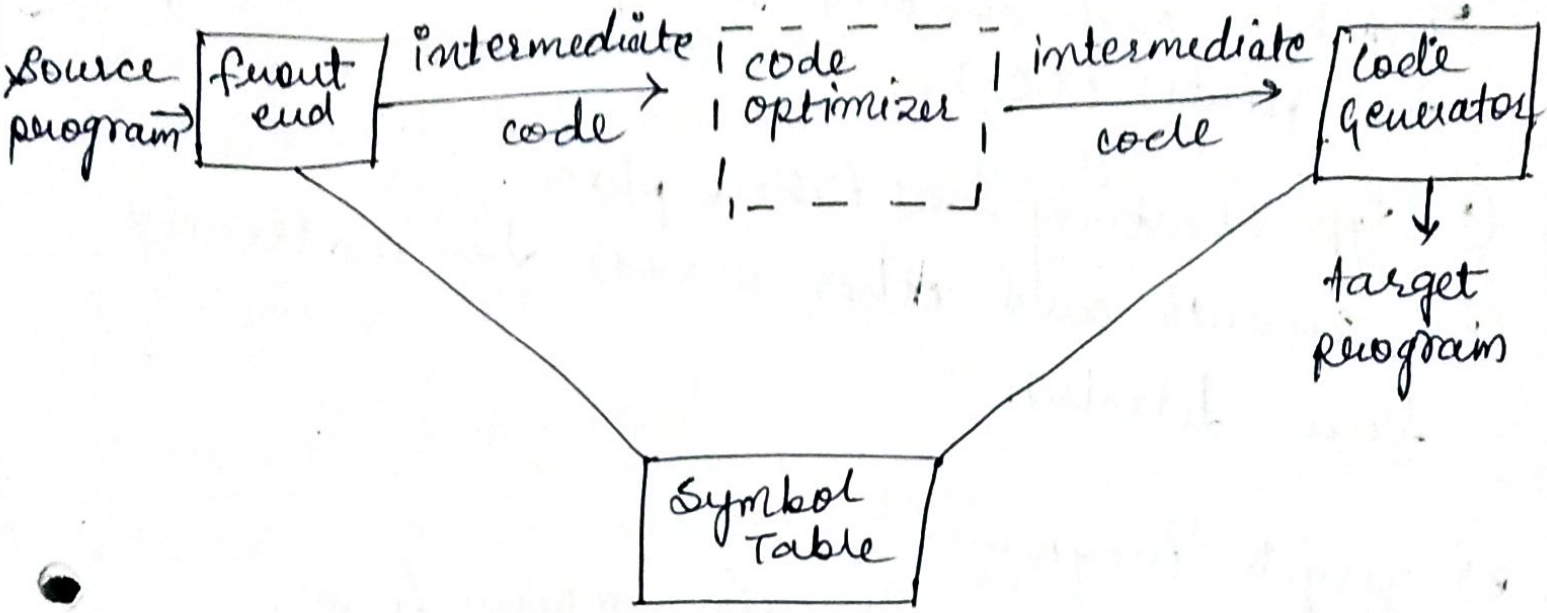
Representing Scope Information:-

- When an occurrence of a name in the source text is looked up in the symbol table, the entry for the appropriate declaration of that name must be determined. (Means name from correct scope must be returned.
- Simple approach:- Maintain separate symbol table for each scope.
- 2nd Approach:- every procedure must provide a unique number. And this no will be appear with declaration of that name.
- 3rd Approach:- a pointer "front" points to the most recently created entry in the list. Entries will be made from front and will be searched from front to last. So that we can access most recent name (local name)
- Deletion can also be performed in same way.



- 4th Approach:- A hash table consists of m lists accessed through an array. Since a name always hashes to the same list, individual lists are maintained as previous example.
- Deletion of entries from the ~~list~~ hash table must be done with care. If we delete ith entry, then i-1 st entry must point to i+1 st entry. For this hash lists must be circular linked list, so that i-1 st entry can be accessed.

Code Generation



- Code optimizer only exists in "optimizing compilers."
- Requirements imposed on a code generator,
  - * output code must be correct and of high quality
  - * it should make effective use of resources
  - * code generator itself should run efficiently.

Issues in the design of a code generator:-

1) Input to the code generator:-

- consists of intermediate representation of the source program and info' in Symbol table that is used to determine run time address.
- code generation assumes that, ① front end has scanned, parsed and translated the source program into a reasonably detailed intermediate

representation. So the values of names can be represented by quantities that the target machine can directly manipulate (bit, integer, real, pointer etc.)

(a) Type checking has taken place.
(b) Semantic and other errors have already been detected.

c) Target Programs:-

- Target code can be in many forms,
  (1) Absolute machine language program:- it can be placed in a fixed location in memory and immediately executed.
  Ex:- PL/C
  (2) Relocatable machine language program:- it allows subprograms to be compiled separately. A set of relocatable object modules can be linked together and loaded for execution by a linking loader. For this we will have added expense. It provides great flexibility in being able to compile subroutines separately and to call other previously compiled programs from an object module.

(3) Assembly language program:- makes code generation easier. We can generate symbolic instructions and use the macro facilities of assembler to help generate code. Price paid is the assemble step after code generation.

3) Memory Management:-

- Mapping names in the source program to addresses of data objects in run-time memory is done co-operatively by the front end and the code generator.
- A name in a three-address refers to a Symbol-table entry for the name. Symbol-table entries were created as the declaration in a procedure were examined. The type in a declaration determines the width, i.e., the amount of storage, needed for the declared name. From the Symbol-table info, a relative address can be determined for the name in a data area for the procedure.
- If machine code is being generated, labels in three-address statements have to be converted to address of instruction.

4) Instruction Selection:-

- The nature of the instruction set of the target machine determines the difficulty of instruction selection. The uniformity and completeness of the instruction set are imp factors.
- Instruction speeds and machine idioms are other imp factors.
- Instructions must be choosen like this that the target program is efficient.

Ex:- $a := b + c$
$d := a + e$

```
Mov  b, R0
add  c, R0
Mov  R0, a
Mov  a, R0
add  e, R0
Mov  R0, d
```

Here stmt 4 is redundant, and R0 is the third if a is not subsequently used.

- Different instruction sets leads to different implementation of given operation and may have significant cost difference.

- A naive (lack of experience) translation may lead to correct but unacceptable target code.

Ex:- a := a + 1

```
Mov a, R0
Add #1, R0
Mov R0, a
```

This must be like,

```
Inc a
```

5) Register Allocation:-

- Instructions involving register operands are usually shorter and faster than those involving operands in memory. Efficient utilization of registers is particularly imp in generating good code. The use of registers is often subdivided into two subproblems:

(1) During "register allocation", we select the set of variables that will reside in registers at a point in the program

(2) During a subsequent "register assignment" phase, we pick the specific register that a variable will reside in.

- finding an optimal assignment of registers to variables is difficult.
- certain machines require register-pairs (even-odd) for some operands and results.

Ex:- IBM System/370 multiplication operation

M x, y (odd) x will be in a register and (even) y will be in another register.

(6) Choice of Evaluation Order:-

- The order in which computations are performed can affect the efficiency of the target code. Some computation orders require fewer registers.

7) Approaches to Code Generation:-

- The most imp criterion for a code generator is that it produce correct code.

Basic Blocks and Flow Graphs:-

- Graphical representation of three-address statements is called flow graph. Nodes in the flow graph represent computations and the edges represent the flow of control.

Basic Blocks:- a basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end.

Ex: $a^2 + 2ab + b^2$

$t_1 := a * a$
$t_2 := a * b$
$t_3 := 2 * t_2$
$t_4 := t_1 + t_3$
$t_5 := b * b$
$t_6 := t_4 + t_5$

- Algo: Partition into basic blocks
  I/P: Sequence of 3-address stmts.
  O/P: A list of basic blocks
  Method:
  1) A stmt will be a leader:
     i) A first stmt is a leader
     ii) Any stmt that is the target of a conditional/unconditional goto is a leader
     iii) Any stmt that immediately follows a goto or conditional goto stmt is a leader.
  2) For each leader, its basic block consists of the leader and all stmts up to but not including the next leader or the end of program.

Ex:-
```
begin
    prod:=0;
    i:=1;
    do begin
        prod:= prod+ a[i] * b[i];
        i:=i+1
    end
    while i<=20
end
```

first block
1. prod=0 → leader by rule 1
2. i=1

2nd block
3. $t_1 = 4 * i$ → leader by rule 2
4. $t_2 = a[t_1]$
5. $t_3 = 4 * i$
6. $t_4 = b[t_3]$
7. $t_5 = t_2 * t_4$
8. $t_6 = prod + t_5$
9. $prod = t_6$
10. $t_7 = i + 1$
11. $i = t_7$
12. if $i <= 20$ goto 3
13. → leader by rule 3

Transformations on Basic Blocks:-

- A basic block computes a set of expressions. Two basic blocks are said to be equivalent if they compute the same set of expressions.
- A number of transformations can be applied to a basic block without changing the set of expressions computed by the block. These transformations can improve the quality of code.

1) Structure - Preserving Transformations:-

(i) Common subexpression elimination:-

These can not be eliminated as 2nd stmt redefine b.

```
a := b+c          a := b+c
b := a-d    ──→   b := a-d
c := b+c          c := b+c
d := a-d          d := b
```

(ii) Dead-code elimination:- suppose x is dead, that is, never subsequently used, at the point where stmt $x = y + z$ appears. So will can eliminate this stmt.

(iii) Renaming temporary variables:-
$t = b + c$ we can replace all occurence of t by u, this will not create a differenc

(iv) Interchange of statements:-

Ex:- $t_1 = b + c$
$t_2 = x + y$

} can be interchanged as independent of each-other.

2) Algebraic Transformations:-

- countless algebraic transformations can be used to change the set of expressions computed by a basic block into an algebraically equivalent set.

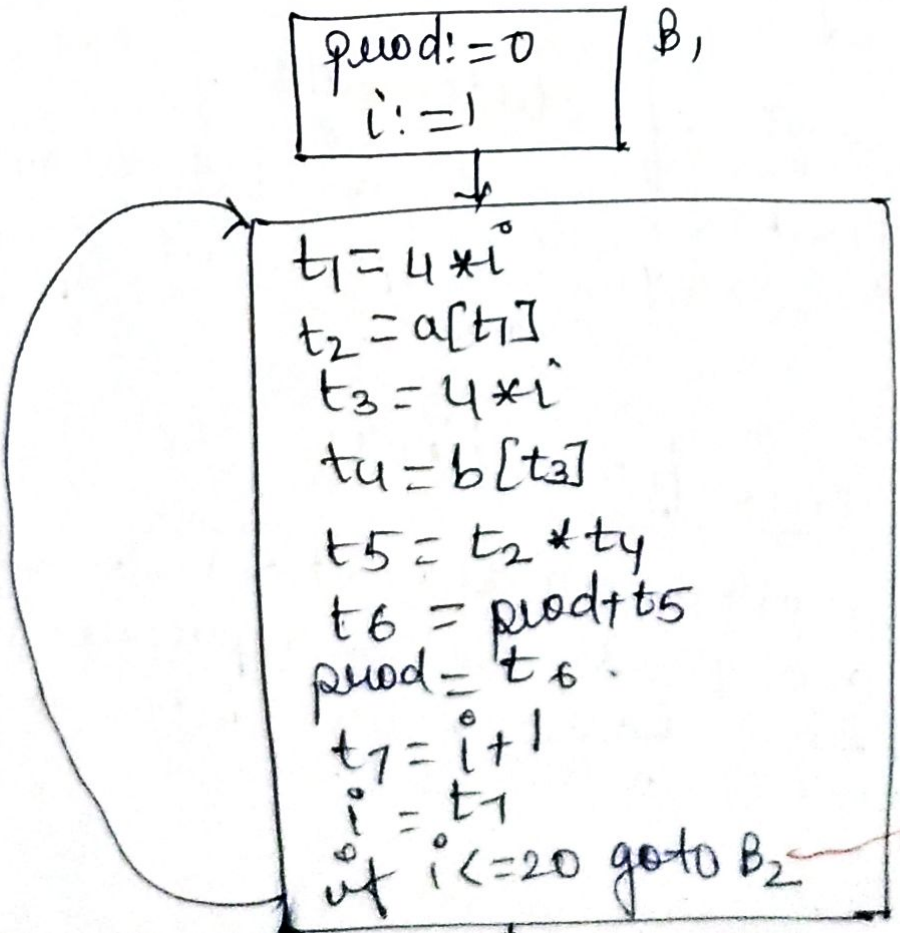
Ex:- $x = x + 0$ } can be eliminated
$x = x * 1$ }

Ex:- $x = y ** 2$
can be replaced with,
$x = y * y$

Flow Graphs:-

- We can add the flow-of-control info to the set of basic blocks making up a program by constructing a directed graph called a "flow graph".
- The nodes of the flow graph are the basic blocks.

Ex:- Flow graph for the prog given in basic block, (vector product)

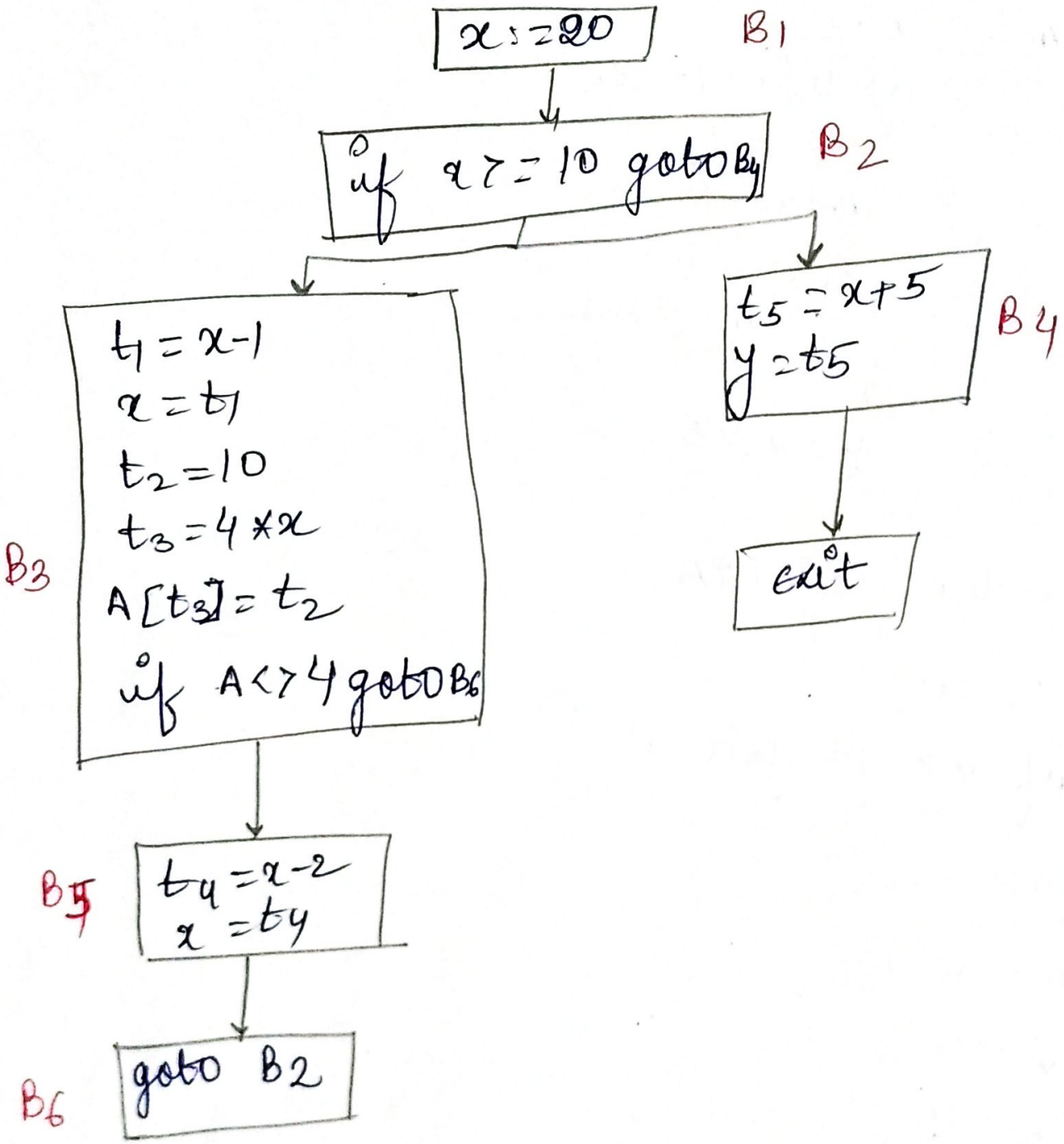


Ex:-

```
x := 20
while(x<10) do
{ x := x-1;
  A[x] = 10;
  if(x==4)
      x = x-2
}
y = x+5
```

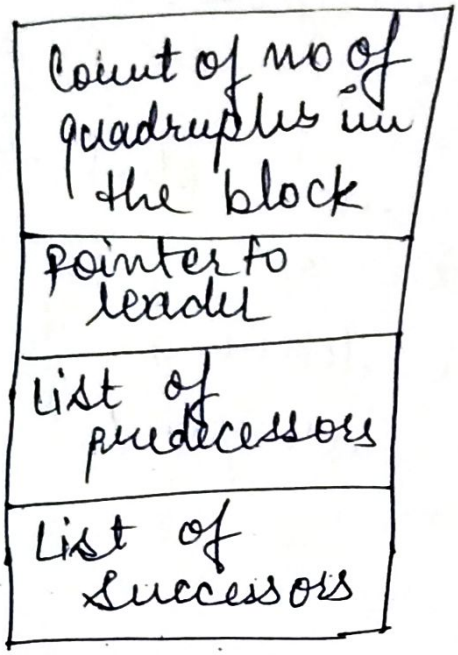
Intermediate Code (TAC):-

1) x := 20
2) if x7=10 goto 12
3) ~~goto~~
3) $t_1 = x - 1$
4) $x = t_1$
5) $t_2 = 10$
6) $t_3 = 4 * x$
7) $A[t_3] = t_2$
8) if x<>4 goto 11
9) $t_4 = x - 2$
10) $x = t_4$
11) goto 2
12) $t_5 = x + 5$
13) $y = t_5$


x := 20
B1
if x >= 10 goto B4
B2
t1 = x - 1
x = t1
t2 = 10
t3 = 4 * x
A[t3] = t2
if A <> 4 goto B6
B3
t5 = x + 5
y = t5
B4
exit
t4 = x - 2
x = t4
B5
goto B2
B6

Representation of Basic Blocks:-

- Can be represented by variety of data structures.
- By Record:- (In this each basic block is represented by following record)



- By linked list :- make linked list of the quadruples in each block.

Loops:- Ex- In previous ex there is a loop consisting of B2.

- Loop is a collection of nodes such that,
  (i) All nodes in the collection are strongly connected. (There is a path from each node to other)
  (ii) The collection of nodes must have a unique entry.

A Simple Code Generation:-

- Code generation strategy consider each statement in turn, remembering if any of the operands of the stmt are currently in registers, and taking its advantage.
- Computed results are generally kept in registers as long as possible, store them only,
  (1) if register is needed for another computation or
  (2) just before leaving a basic block (Procedure call, jump or labeled stmt)

Ex: $a = b + c$

Add Rj, Ri
Result in Ri, cost = 1 (Both operands exists in registers)

Add c, Ri
Result in Ri, cost = 2 (One operand exist in memory)

Mov c, Rj
Add Rj, Ri
Result in Ri, cost = 3 (c can be used again & again, as it exist in Rj)

(b is not subsequently live)

- Many more cases to consider,
  - * where b, c are located.
  - * whether b's value will be subsequently used.
  - * either b or c is a constant (Both)
  - * whether + is commutative

we have to consider a large no of cases before code generation.

Register and Address Descriptors:-

- Uses descriptors to keep track of register contents and addresses for names,
  - (i) A register descriptor keeps track of what is currently in each register
  - (ii) An address descriptor keeps track of the location where the current value of the name can be found at run time. The location might be a register, a stack location, a memory address or some set of these.

A Code-generation algorithm:-

- Input:- sequence of 3-address stmts from a basic block
- For each stmt $x := y \text{ op } z$ do following,
  - (i) Invoke a func getreg to determine location L where result will be stored. (Register / Memory)
  - (ii) Get address of y from address descriptor. If y is at both register/memory. Then choose register.

(ii) if value of y is not already in L, then MOV y', L.

(iii) Generate instruction OP z', L (z' is location of z). Update address descriptor of x to indicate that x is in location L. Remove x from all other register descriptors.

(iv) If after the current uses, y and z are not live then alter registers, that they no longer will contain y or z.

- if stmt has a unary operator then same steps will be followed.
- if stmt x := y,
  - (i) if y is in a register, then simply change the register and address descriptor, and x will now be found only in the register containing y.
  - (ii) if y is in a memory location, then use getreg to find a register in which to load y and make that register location of x.

- After processing of all TAC in a basic block, we store, by MOV instructions, those names that are live on exit and not in their locations.

The function getreg :-

- This function returns the location L to hold the value of x in x := y op z. Scheme for selecting the location,
  (i) if y is in register that holds the value of no other names and y is not live after this stmt then return register of y for L.
- (ii) Failing (i), return an empty register for L.
  (iii) Failing (ii), if x has next use in the block, then find an occupied register R, store the value of R in memory and return R.
  (iv) if x is not used in the block, or no
- suitable occupied register then select memory location for x as L.

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

TAC

```
t := a-b
u := a-c
v := t+u
d := v+u
```

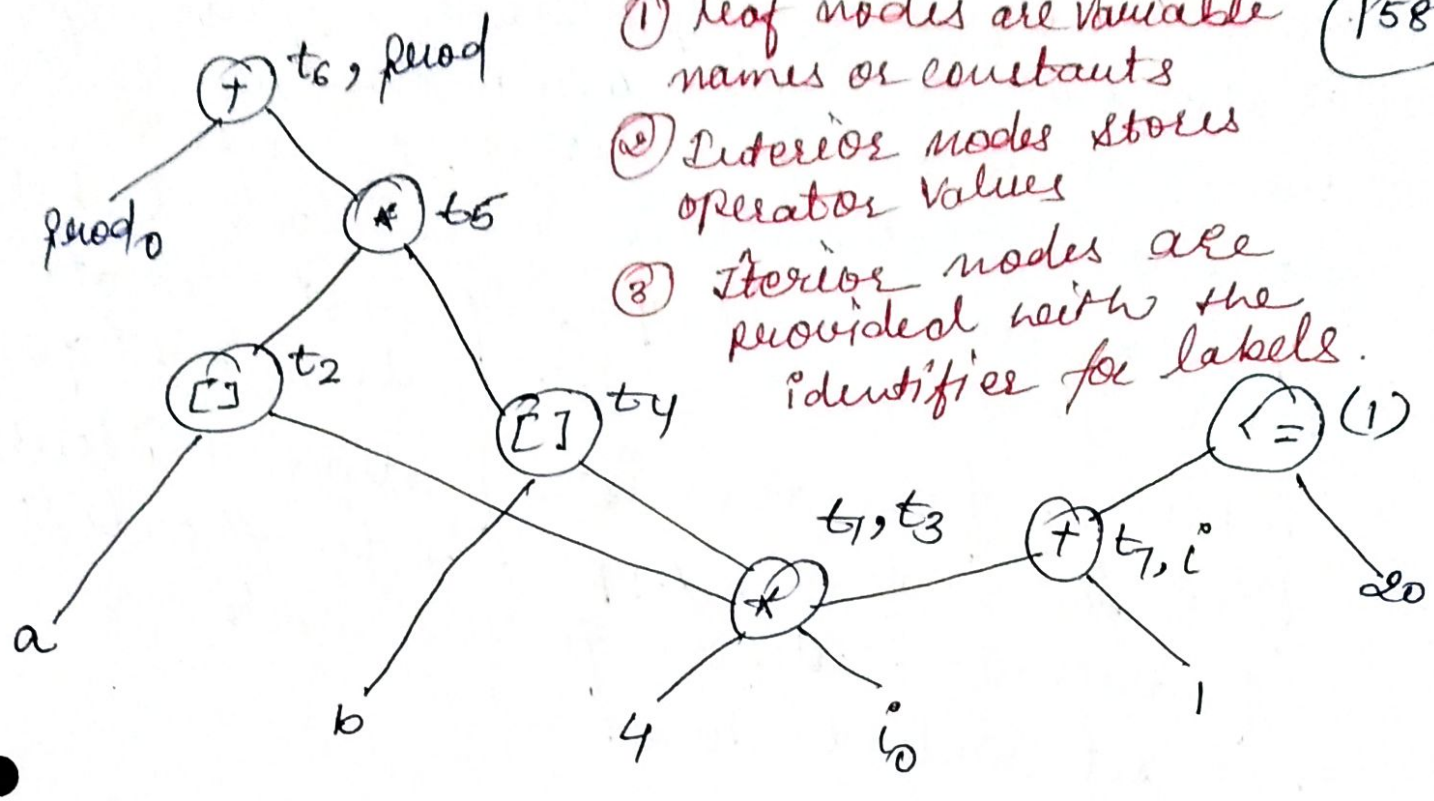
Statements	code Generated	Register Descriptor Registers empty	Address Descriptor
t := a-b	MOV a, R0 SUB b, R0	R0 Contains t	t in R0
u := a-c	MOV a, R1 SUB c, R1	R0 Contains t R1 contains u	t in R0 u in R1
v := t+u	ADD R1, R0	R0 contains v R1 contains v	u in R1 v in R0
d := v+u	ADD R1, R0 MOV R0, d	R0 contains d	d in R0 d in memory

Fig:- Code Sequence

The Dag Representation of Basic Blocks:-

- Directed acyclic graphs (dags) are useful data structures for implementing transformations on basic blocks. A dag gives a picture of how the value computed by each statement in a basic block is used in subsequent statements of the block.

Ex:-
(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)



① leaf nodes are variable names or constants
② Interior nodes stores operator values
③ Interior nodes are provided with the identifier for labels.

Dag Construction:-

Algo: constructing a dag

I/p: A basic block

O/p: A dag

Method:

case 1:- x := y op z

case 2:- x := op y

case 3:- x := y

1. If node(y) is undefined, create a leaf labeled y, then set this to node(y). In case 1, if node(z) is also undefined, then create node(z)

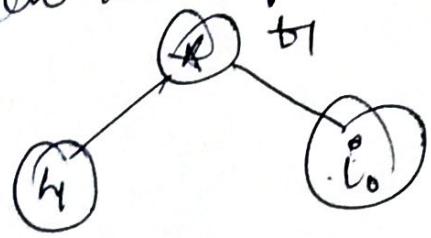
2. In case 1, if there is a node labeled op, with left child (y) and right child (z).

(to check common subexpressions.) If not, create such a node. Assume n is the node found or created. In case 2, determine whether there is a node labeled op, and alone child (y). If not, create node n. In case 3, let n be node(y).

3. Delete x from the list of attached identifiers for node(x). Append x to the list of attached identifiers for the node n and set node(x) to n.

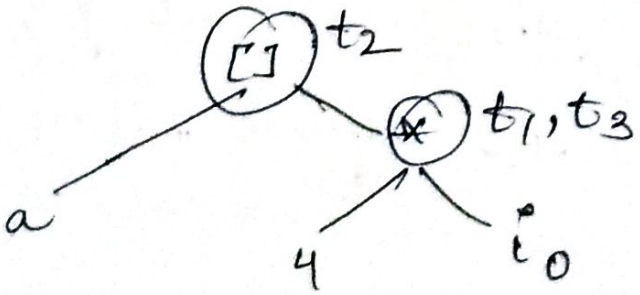
Ex:- $t_1 := 4 * i$

→ create leaves for 4 and i.
→ create a node labeled *.
→ attach identifier t_1 to it



(a)

$t_2 := a[t_1]$ and $t_3 = 4 * i$



Applications of Dag:-

1) We automatically detect common subexpressions.
2) We can determine which identifiers have their values used in the block. (those for which a leaf is created in step 1)
3) We can determine which statements compute values that could be used outside the block. (those statements & whose node n constructed or found in step 2)
4) List of quadruples are simplified by eliminating the common subexpressions. Also the assignment of the form a:=b cannot be performed ~~good code~~ until and unless it is necessary to do this.

Peephole Optimization:-

- The quality of target code can be improved by applying "optimizing" transformations to the target program.
- A effective technique is peephole optimization.
- It is a method for trying to improve the performance of the target program by examining a short sequence of target

instructions (called peephole) and replacing these instructions by a shorter or ~~faster~~ faster sequence, whenever possible.

- Can be applied directly after intermediate representation.
- Peephole is a small, moving window on the target program. The code in peephole need not be contiguous.

Examples of program transformations using peephole optimization:-

1) Redundant loads and stores:- (redundant instructions)

ex:- MOV R0, a
MOV a, R0 → can be deleted.

But if second stmt has a label then we can't remove it.

2) Unreachable code:-
- Remove unreachable instructions.

ex:- #define debug 0
...
if (debug) { print debugging info }

Intermediate code for this,

```
if debug = 1 goto L1
goto L2
L1: print debugging info
L2:
```

After eliminating jumps over jumps,

```
if debug ≠ 1 goto L2
print debugging info
L2:
```

Replacing 'debug' with its defined value,

```
if 0 ≠ 1 goto L2
print debugging info
L2:
```

As in above example, 1st statement always evaluated to true, it can be replaced by goto L2. Now all statements "printing debugging info" are unreachable and can be eliminated one at a time.

3) Flow-of-control optimization:-

- ~~Unnecessary~~ Unnecessary jumps are eliminated.

Ex:- goto L1 ... L1 : goto L2 → goto L2 ... L1 : goto L2

Ex:- if a<b goto L1 ... L1: goto L2 → if a<b goto L2 ... L1: goto L2

4) Algebric Simplification:-

- There is no end to the amount of algebric simplification that can be attempted through peephole optimization.

Ex:- x := x + 0
or x := x * 1
} can be eliminated directly

5) Reduction in Strength:-

- This replaces expensive operations by equivalent cheaper ones.

Ex:- x2 is invariably cheaper to implement as x * x than as a call to an exponentiation routine.

6) Use of machine Idioms:-

- The target machine may have h/w instructions to implement certain specific operations efficiently. Detecting situations that permit

the use of these instructions can reduce execution time significantly.

Ex:- Some machines have auto-increment and auto-decrement addressing modes.

Generating Code from Dags:-

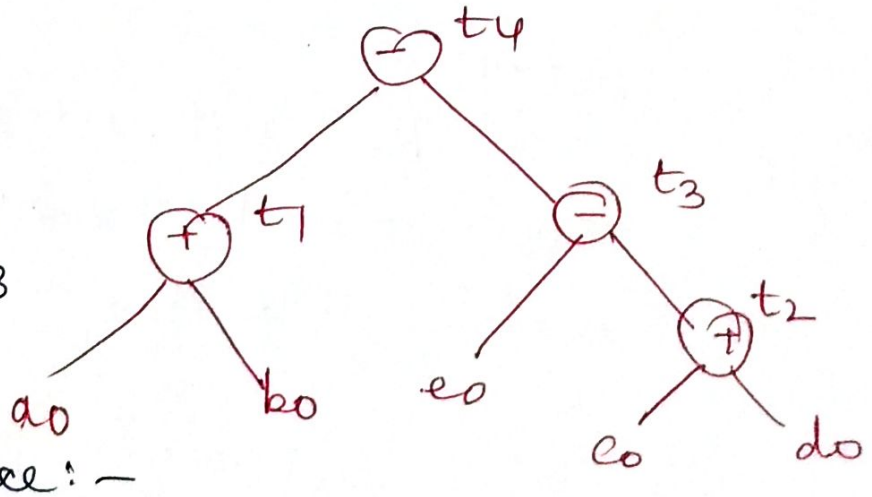
- This section shows, generating code for a basic block from its dag representation.

Rearranging the order:-

- In the following example, it is shown, how the order in which computations are done can affect the cost of resulting object code.

Ex:- stmt := (a+b)-(e-(c+d))

$t_1 := a+b$
$t_2 := c+d$
$t_3 := e-t_2$
$t_4 := t_1 - t_3$



Code Sequence:-

R0-a	MOV a, R0	R0 - e
R0-a+b	ADD b, R0	MOV e, R0
R1-c	MOV c, R1	SUB R1, R0 R0 - e-(c+d)
R1-c+d	ADD d, R1	MOV t1, R1 R1 - t1
t1-a+b	MOV R0, t1	SUB R0, R1 R1 - t1-(e-(c+d))
		MOV R1, t4 t4 -

After rearranging the order of stmts,

$t_2 := c + d$
$t_3 := e - t_2$
$t_1 := a + b$
$t_4 := t_1 - t_3$

Code sequence:-

MOV c, R0	MOV a, R0
ADD d, R0	ADD b, R0
MOV e, R1	SUB R1, R0
SUB R0, R1	MOV R0, t4

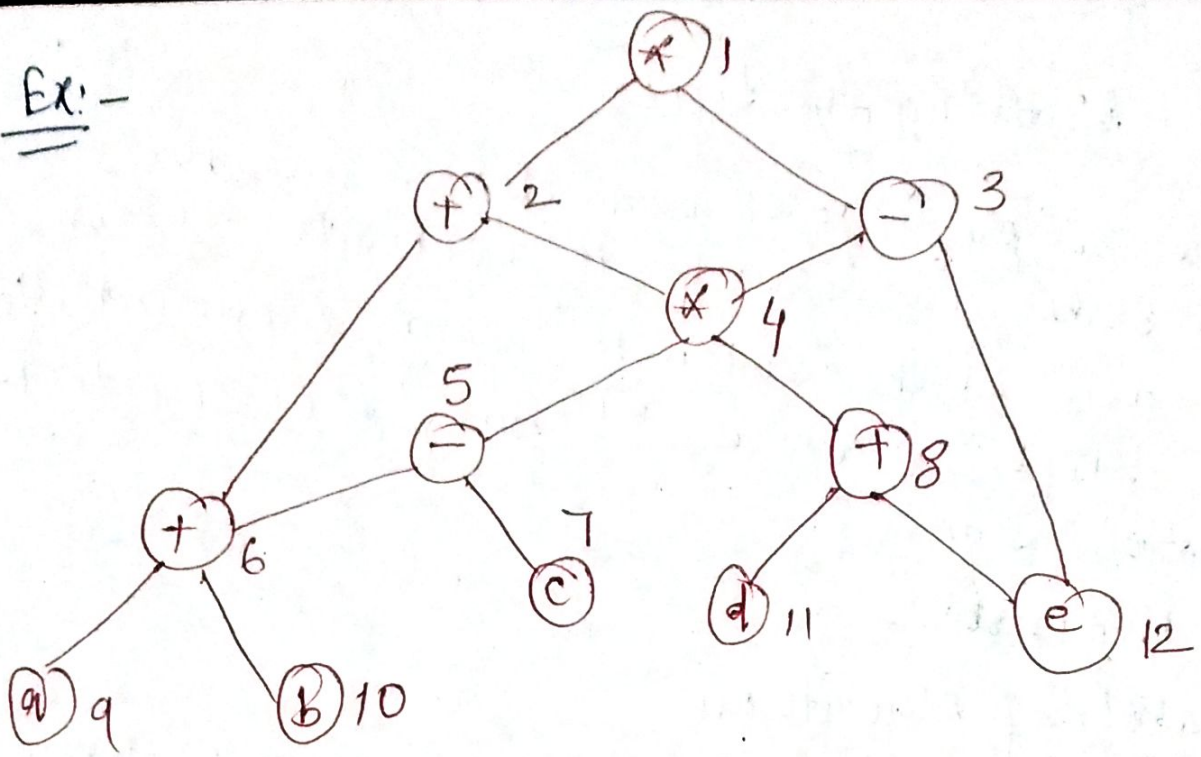
A Heuristic ordering for Dags:-

- A node listing algorithm is presented in this section, it gives the order of statements of three-address code.

(1) while unlisted interior nodes remain do
    begin
(2) Select an unlist node n, all of whose parents have been listed
(3) list n;
(4) while the leftmost child m of n has no unlisted parents and is not a leaf do,
        begin
(5) list m;
(6) n := m
        end
    end

Fig: Node listing Algorithm

Ex:-



- Initially only node 1 is there which has no unlisted parents, so n=1
- left child of 1 is 2, has its parent listed, so n=2
- Now left child of 2 is 6 which has an unlisted parent 5, so we select new n at line (2) of algo as 3. List 3
- Now proceed down its left chain 4, 5, 6.
- This leaves only 8. So list it.
- Resulting list is 1234568. So the suggested order of evaluation is 8654321.
- TAC:-

$t_8 = d + e$
$t_6 = a + b$
$t_5 = t_6 - c$
$t_4 = t_5 * t_8$
$t_3 = t_4 - e$
$t_2 = t_6 * t_4$
$t_1 = t_2 * t_3$

Optimal Ordering for trees:-

- It is a simple algorithm to determine the optimal order in which to evaluate statements in a basic block when the dag representation of the block is a tree. It gives the shortest instruction sequence.
- It has two parts:-

1) The Labeling Algorithm:-

- Labeling is done in bottom-up order.
- left leaf:- a node that is a leaf and leftmost descendant of its parent.
- Right leaves:- other leaves

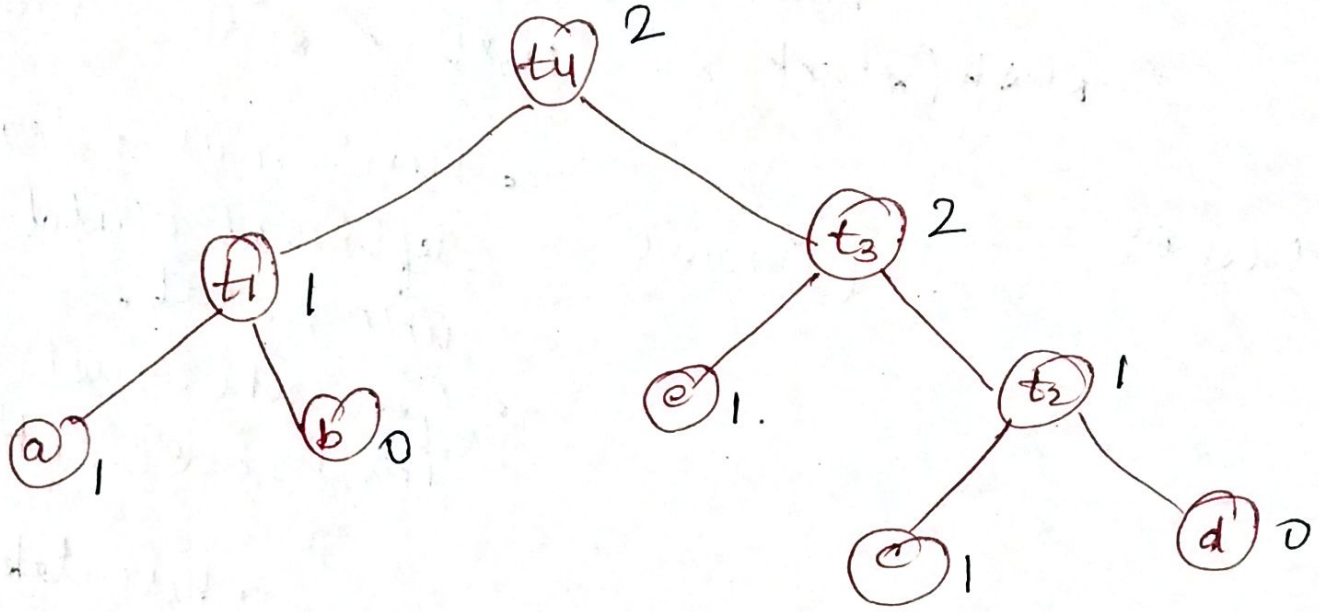
Algo:- Label computation

(1) if n is a leaf then
(2) if n is the leftmost child of its parent then
label(n) = 1
(3)
(4) else label(n) = 0
else begin /* n is an interior node */
(5) let $n_1, n_2 \ldots n_k$ be the children of n ordered by label
so label(n_1) ≥ label(n_2) ≥ ... ≥ label(n_k);
(6) label(n) = $\max_{1 \le i \le k}$ (label(n_i) + i - 1)
end

- If n is a binary node then its children have labels l_1 and l_2, the formula of line (6) reduces to,

$$label(n) = \begin{cases} \max(l_1, l_2) & \text{if } l_1 \neq l_2 \\ l_1 + 1 & \text{if } l_1 = l_2 \end{cases}$$

Ex:-



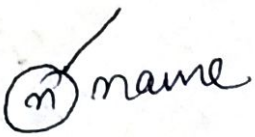
- post order visit = a b t_1 e c d t_2 t_3 t_4
(order for doing labeling)

2) Code Generation from a labeled Tree:-

- Takes as input a labeled tree T and produces as output a machine code sequence that evaluates T in R0.
- Procedures used by the algorithm,
  - * gencode(n) - to produce machine code evaluating the subtree of T with root n into a register.

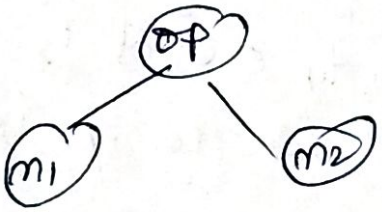
* Stack — rstack [To allocate registers, available register initially $R_0, R_1 \dots R(n-1)$]
* Swap (rstack) — interchanges the top two registers on rstack.
* tstack - temporary memory locations.
* x = pop(stack) - pop stack and assign value to x
* push(rstack, x) — push x onto stack.

Case 0:-



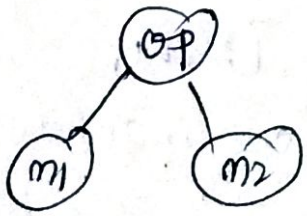
- Here n is a leaf & leftmost child of its parent.
- Generate just a load instruction into register R = top (rstack), followed by instruction op name, R.

Case 1:-



- Here label (n2) = 0
- Generate code to evaluate n1 into R = top (rstack), followed by instruction op name, R.

Case 2:-



- Here label (n2) > label (n1) Means n2 is harder to evaluate.
- Swap top two registers on rstack, evaluate n2 into R = top (rstack). Now remove R from stack & evaluate n1 into S = top (rstack).

- Then generate the instruction OP, R, S.

Case 3:- similar to case 2, except that here left tree is harder to evaluate. So its evaluated first. (label(n_2) ≤ label(n_1))

Case 4:- when both subtrees require n (total no of registers) or more registers to evaluate without stores.

- Use a temp location to evaluate right subtree, then left subtree and finally the root.

Ex:- Generate code for labeled tree in previous example,

```
gencode(t4)        [R1 R0]      // case 2   t3-2 t1-1
  gencode(t3)      [R0 R1]      // case 3   e & t2-1
    gencode(e)     [R0 R1]      // case 0
      print MOV e, R1
  gencode(t2)      [R0]         // case 1   c-1 d-0
    ~~print MOV c, R0~~          ~~// case 0~~
  ~~print ADD d, R0~~
    gencode(c)     [R0]         // case 0
      print MOV c, R0
  print ADD d, R0
print SUB R0, R1
gencode(t1)        [R0]         // case 1   a-1 b-0
```

```
gencode(a)      [R0] //case 0
    print MOV a, R0
print ADD b, R0
print SUB R1, R0
```

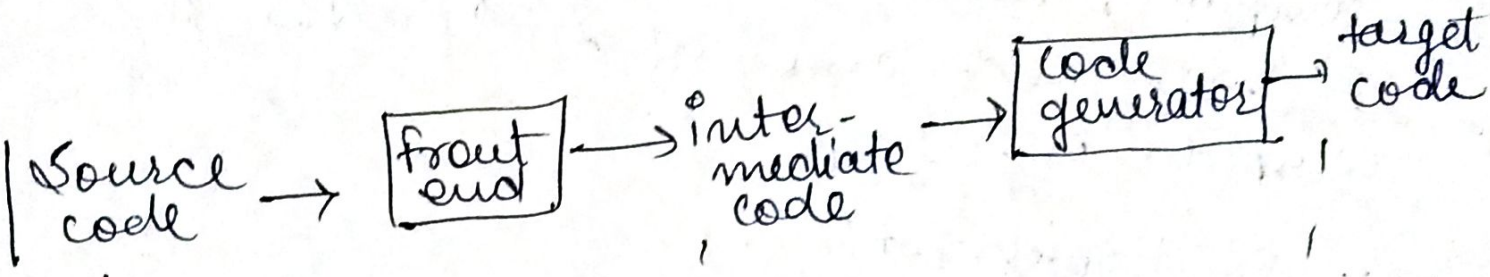
Common Subexpressions:

- When there are common subexpressions in a basic block, the corresponding dag will no longer be a tree. The common subexpressions will correspond to nodes with more than one parent, called shared nodes. We can no longer apply the labeling algorithm or gencode directly.

- In practice, we can obtain a reasonable solution if we partition the dag into a set of trees.

Code Optimization:-

- Compilers that apply code-improving transformations are called optimizing compilers.
  - ↳ Machine-independent optimization (program transformations that improve the target code without taking into consideration any properties of target machine)
  - ↳ Machine-dependent optimization (improves register allocation & utilization of special machine-instruction sequence.



- User can,
  1) Profile program
  2) change algo
  3) Transform loops

compiler can,
1) improve loops
2) procedure calls
3) Address calculations

compiler can,
1) Use registers
2) Select instructions
3) Peephole optimization

The principle Sources of optimization:-

- A transformation of a program is called local if it can be performed by looking only at the stmts in a basic block; otherwise it is called global.

Function-Preserving Transformations:-

- By using various ways the compiler can improve the program without changing the properties of the function it computes.

1) Common subexpressions:-

- An occurrence of an expression E is called a common subexpression if E was previously computed, and the values of variables in E have not changed since the previous computation.

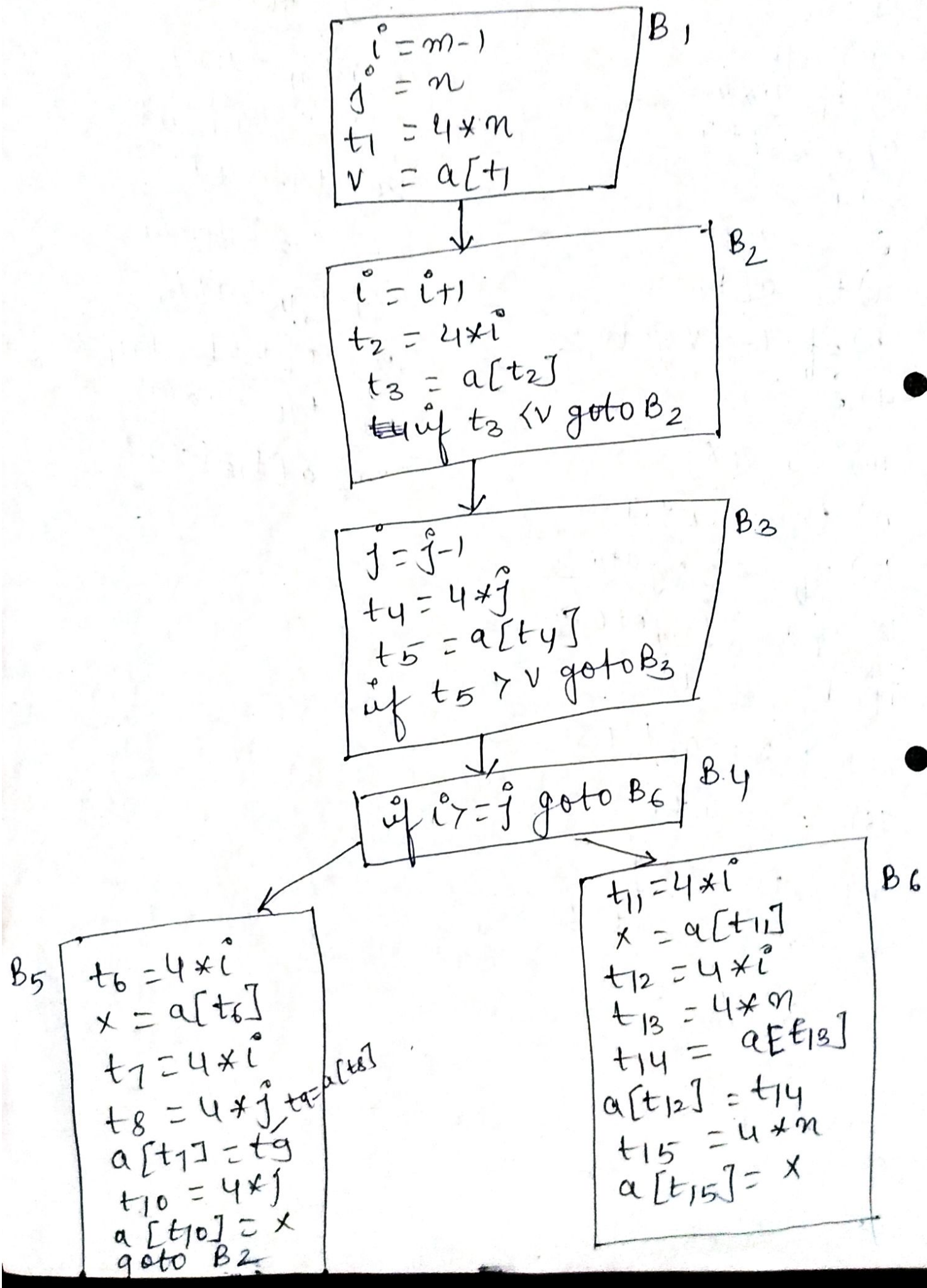
Ex:-

```
void quicksort(m,n)
int m,n;
{
    int i,j,v,x;
    if(n<=m) return;
    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;
    quicksort(m,j); quicksort(i+1,n);
}
```

TAC for above code:-

1) $i = m-1$
2) $j = n$
3) $t_1 = 4 * n$
4) $v = a[t_1]$
5) $i = i+1$
6) $t_2 = 4 * i$
7) $t_3 = a[t_2]$
8) if $t_3 < v$ goto 5
9) $j = j-1$
10) $t_4 = 4 * j$
11) $t_5 = a[t_4]$
12) if $t_5 > v$ goto 9
13) if $i >= j$ goto 23
14) $t_6 = 4 * i$
15) $x = a[t_6]$
16) $t_7 = 4 * i$
17) $t_8 = 4 * j$
18) $t_9 = a[t_8]$
19) $a[t_7] = t_9$
20) $t_{10} = 4 * j$
21) $a[t_{10}] = x$
22) goto 5
23) $t_{11} = 4 * i$
24) $x = a[t_{11}]$
25) $t_{12} = 4 * i$
26) $t_{13} = 4 * n$
27) $t_{14} = a[t_{13}]$
28) $a[t_{12}] = t_{14}$
29) $t_{15} = 4 * n$
30) $a[t_{15}] = x$

Flow graph:-


B1
i = m-1
j = n
t1 = 4*n
v = a[t1]
B2
i = i+1
t2 = 4*i
t3 = a[t2]
if t3 < v goto B2
B3
j = j-1
t4 = 4*j
t5 = a[t4]
if t5 > v goto B3
B4
if i >= j goto B6
B5
t6 = 4*i
x = a[t6]
t7 = 4*i
t8 = 4*j
t9 = a[t8]
a[t7] = t9
t10 = 4*j
a[t10] = x
goto B2
B6
t11 = 4*i
x = a[t11]
t12 = 4*i
t13 = 4*n
t14 = a[t13]
a[t12] = t14
t15 = 4*n
a[t15] = x

B_5 after local common subexpression elimination,

B_5

$t_6 = 4 * i$
$x = a[t_6]$
$t_8 = 4 * j$
$t_9 = a[t_8]$
$a[t_6] = t_9$
$a[t_8] = x$
goto B_2

Here t_6 is used in place of t_7 and t_8 in place of t_{10}.

B_5 and B_6 after global and local CSE,

B_5

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

B_6

$x = t_3$
$t_{14} = a[t_1]$
$a[t_2] = t_{14}$
$a[t_1] = x$

2) 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.

```
k=0
if(k=1)
{
   a=9+5;
}
```

} Dead code

3) Copy Propagation:-

- Assignment statement $f = g$ called copy statements or copies for short.
- The idea behind the copy-propagation transformation is to use g for f, wherever possible after the copy stmt $f = g$.

Ex:- B_5 block in previous ex

Read code now →

```
x = t3                    x = t3
a[t2] = t5      ⟶         a[t2] = t5
a[t4] = x                 a[t4] = t3
goto B2                   goto B2
```

Now we can eliminate the assignment to x.

4) Loop Optimization:-

- Loop is a very imp place for optimization.

a) Code Motion:- it takes an exp that yields the same result independent of the number of times a loop is executed and places the exp before the loop.

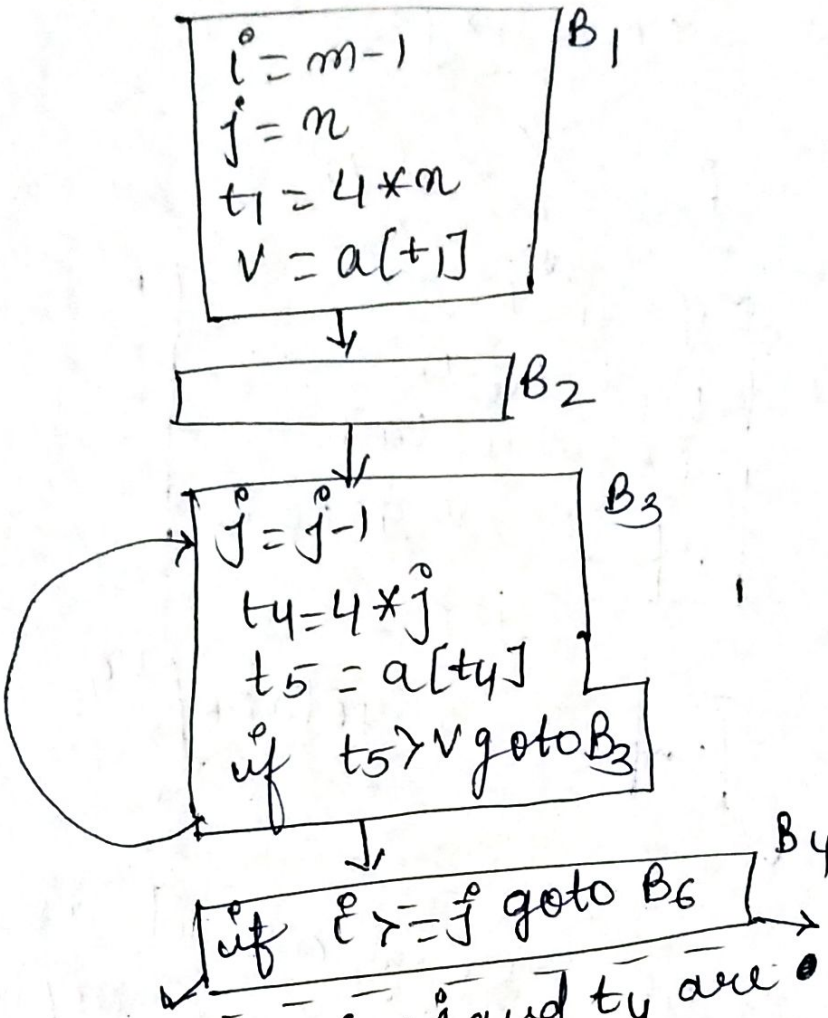
Ex:- while (i <= limit - 2)

↓

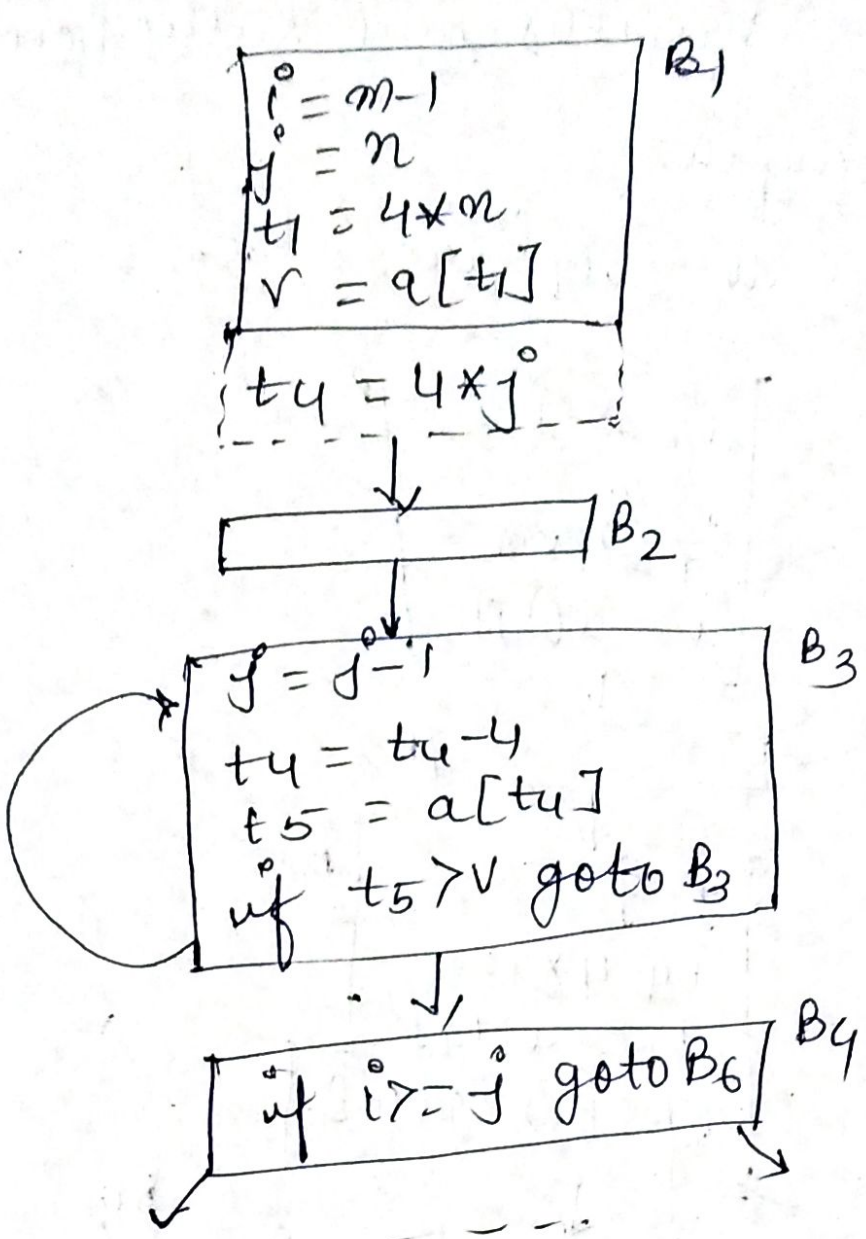
```
t = limit - 2;
while( i <= t)
```

b) Induction Variables and Reduction in Strength:-

Ex:- Consider the loop around B_3,



- In this, block B_3 contains j and t_4 are remain in lock-step; every time the value of j is decreased by 1, t_4 decreases by 4. Such identifiers are called induction variables.
- Now we can replace $t_4 = t_4 - 4$. But here we have to initialize t_4 too, otherwise it is not having a initial value, we place an initialization of t_4 at the end of block where j itself is initialized. (In Block B_1)



- Here we can't remove j, as it is using in block B4. But in t4 we have replaced as multiplication with subtraction. (Mul takes more time than Sub)

~~Loops in flow graph:-~~

- Now i and j are only in use B4 after B1. We know that the values of i and t_2 satisfy the relationship $t_2 = 4*i$. Same as with j and t4 ($t_4 = 4*j$).

So the test $t_2 >= t_4$ is equivalent to $i >= j$ in B_4. After eliminating these induction-variable,

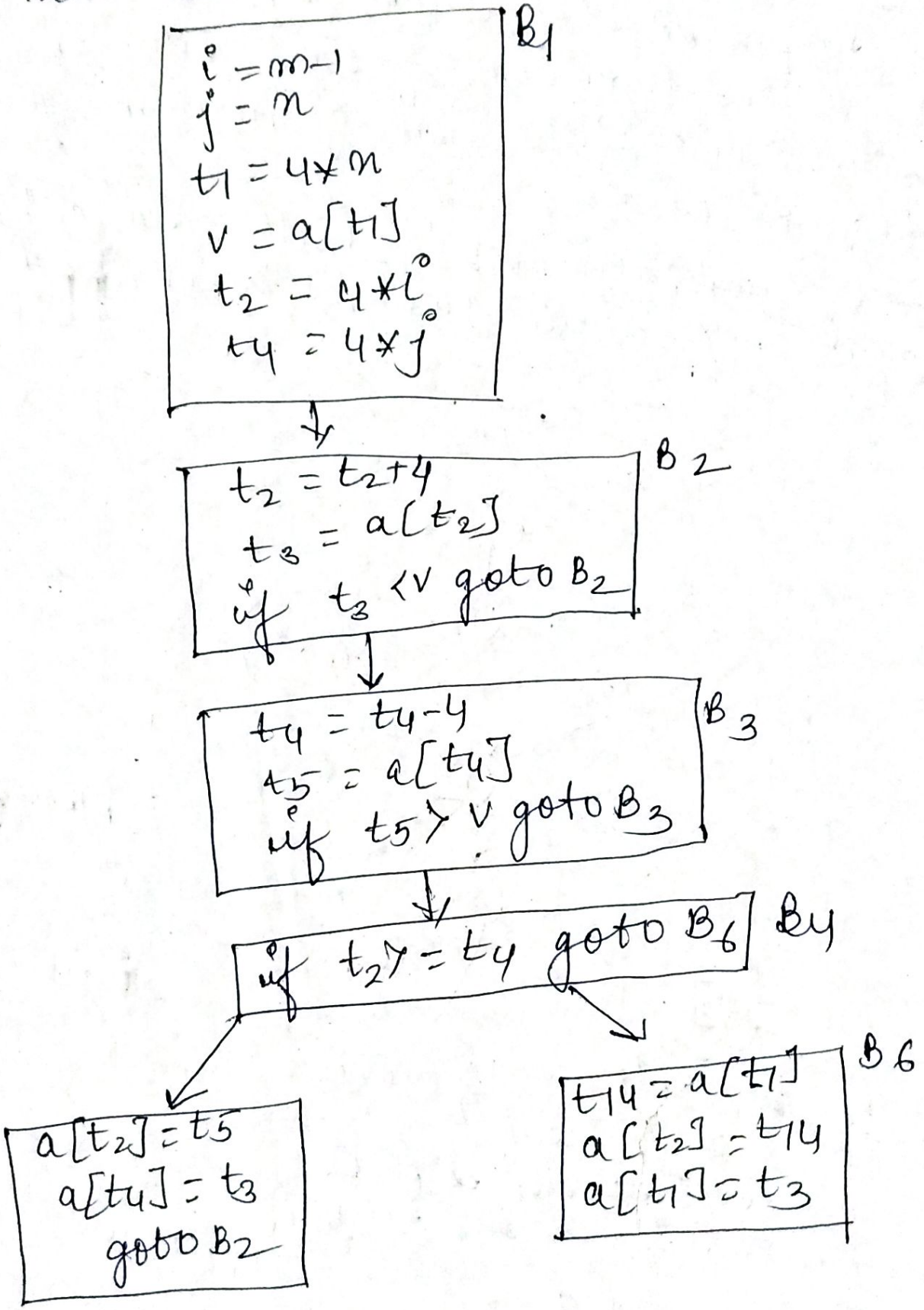
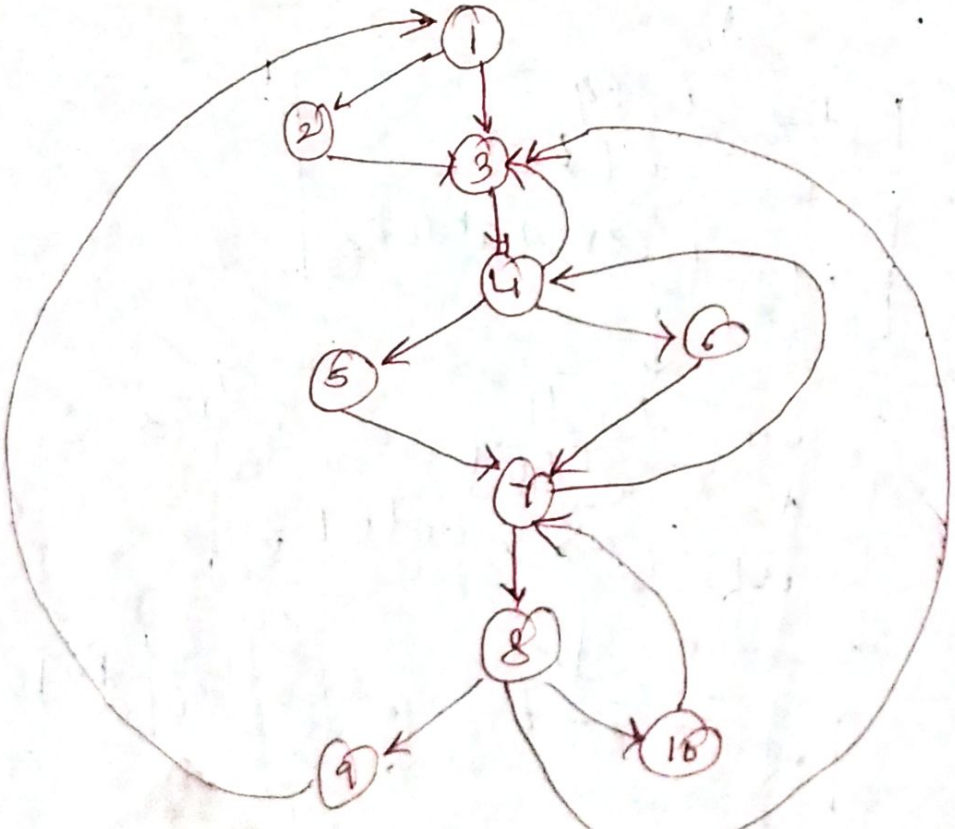


Fig: final optimized TAC

Loops in Flow Graphs:-

Dominators:-

- We say node d of a flow graph dominates node n, written d dom n, if every path from the initial node of the flow graph to n goes through d.
- Every node dominates itself and entry of a loop dominates all nodes in the loop.



Flow graph

1 — dominates every node
2 — " itself
3 — dominates all, except 1 & 2
4 — " " " 1, 2, 3
5, 6, 9, 10 — itself
7 — 7, 8, 9, 10
8 — 8, 9, 10

Natural Loops:-

- Two essential properties,
  - * A loop must have a single entry point called the header.
  - * There must be at least one way to iterate the loop, i.e. at least one path back to the header.

Ex:- • In previous example there is an edge $7 \to 4$ and 4 dom 7.

- Similarly $10 \to 7$ & 7 dom 10
  $4 \to 3$ & 3 dom 4
  $8 \to 3$ & 3 dom 8
  $9 \to 1$ & 1 dom 9

} natural loops

- Given a back edge $n \to d$, we define the natural loop of the edge to be d plus the set of nodes that can reach n without going through d.

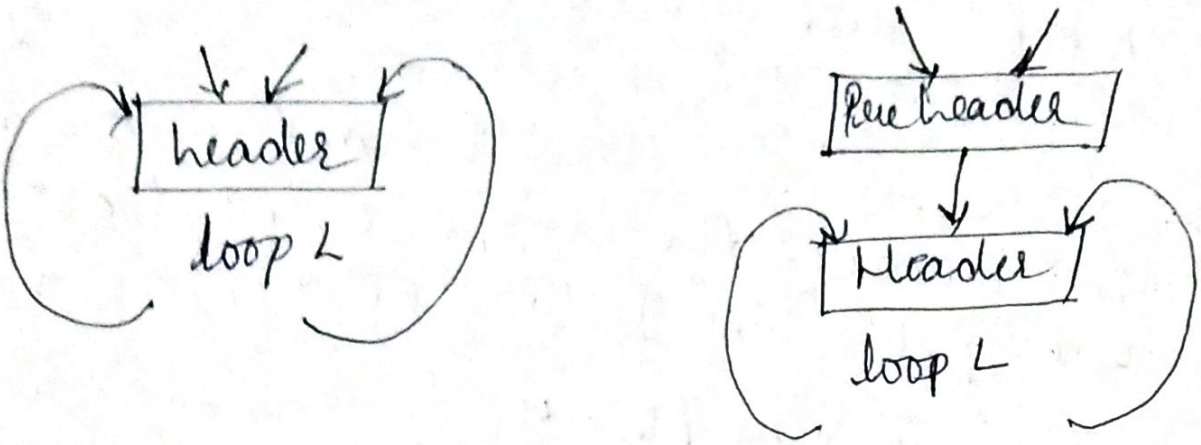
Inner loops:-

- A loop which contains no loop within it is called inner loop. In previous ex 7,8,10 is a inner loop

Pre-Headers:-

- when we move many stmts outside of the loop (before the header), we create a new

block, called preheader and move stmts inside this.



a) before b) after

Reducible flow Graphs :-

- Structured flow-of-control statements such as if-then-else, while-do, continue and break stmts produces programs whose flow graphs are always reducible.
- A flow graph G is reducible if and only if we can partition the edges into two disjoint groups, often called the forward edges and back edges with following two properties:
  1) The forward edges ~~form~~ form an acyclic graph in which every node can be reached from the initial node of G.

2) The back edges consist only of edges whose heads dominate their tails.

Ex:- • The graph in previous example is reducible. In general, we can find and remove all back edges. If all remaining forward edges make an acyclic graph, then it is reducible.

• Like if we remove five back edges 4→3, 7→4, 8→3, 9→1 and 10→7, whose heads dominate their tails, the remaining graph is acyclic. So it is reducible.

Loop Invariant Computations:-

• it can be obtained by moving some amount of code outside the loop and placing it just before entering the loop. Code which is moved outside the loop is the expression which is computed independent of the no of times.

Ex:- while (k<=l-1)
{ sub = sub - a[k]; }

→

temp = l-1;
while (k<=temp)
{ sub = sub - a[k]; }

Global Data-flow analysis:-

- The process of collecting data-flow info by an optimizing compiler is know as data-flow analysis. This will help in code optimization

Ex:- (1) To know what variables are live on exit from each block.
(2) knowledge of global common sub-expressions for elimination.
(3) knowledge of unreachable code for dead-code elimination.

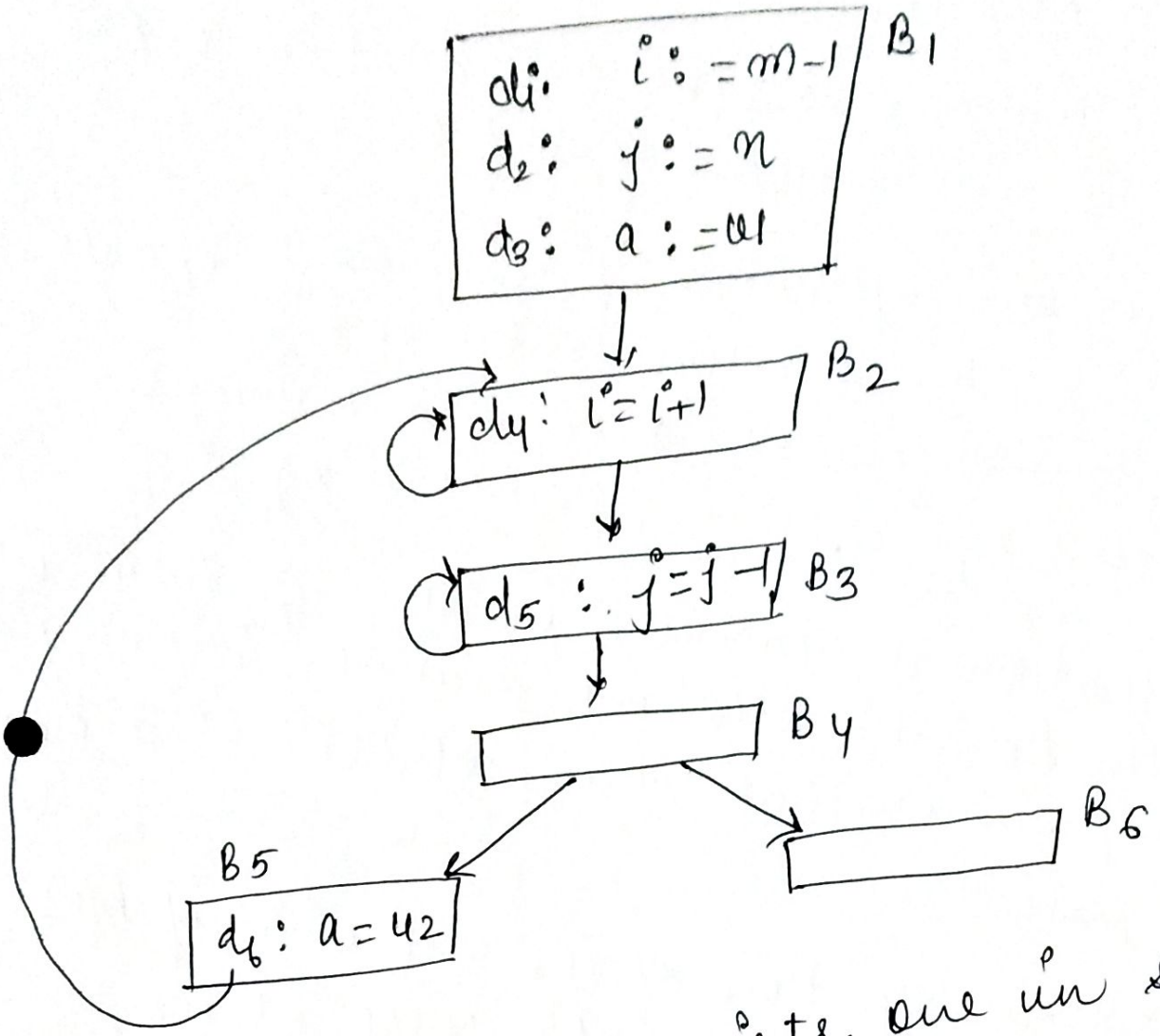
- Data-flow information can be collected by setting up and solving systems of equations that relate info at various points in a program.

Ex:- $out[s] = gen[s] \cup (in[s] - kill[s])$

the info at the end of stmt is either generated within the stmt or enters at the beginning and is not killed as the control flows through the statement. Such equations are called data-flow equations.

Points and Paths:-

- within a basic block, we talk of the point blw two adjacent stmts, as well as the point before the first stmt and after the last.



In block B_1 we have 4 points. one in starting and one after each of the three assignments.

- A path from P_1 to P_n is a sequence of points $P_1, P_2 \cdots P_n$ such that for each i b/w 1 & n-1 either,
  (1) P_i is the point immediately preceding a stmt & P_{i+1} is following the stmt in same block or
  (2) P_i is end of some block and P_{i+1} is beginning of a successor block.

Ex:- path → $B_5 - B_2 - B_3 - B_4 - B_6$
(from beginning of B_5 to beginning of B_6)

Reaching Definitions:-

- A definition of a variable x is a stmt that assigns, or may assign, a value to x.
- Unambiguous definition:- if assignment operator or I/O device assign value to x.
- Ambiguous definition:- assignment through pointer (*q=y, where q points to x)
- A definition d reaches a point p if there is a path from the point immediately following d to p, such that d is not "killed" along that path.
- We kill a definition of a variable 'a' if b/w two points along the path there is a definition of a.

Ex:- In last example, both definitions $i=m-1$ and $j=n$ in B_4 reach to the beginning of B_2, but after B_3 $j=j-1$ will reach to B_4, B_5 and B_6. So the assignment to j in B_3 kills the definition $j=n$, so the latter does not reach B_4, B_5 or B_6.

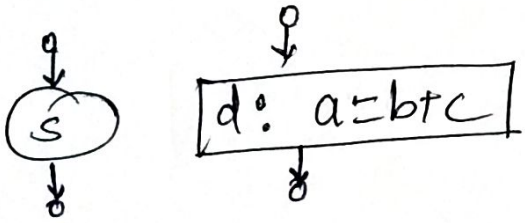
Data-Flow Analysis of Structured Programs:

- Flow graphs for control-flow constructs have a useful property: there is a single beginning point at which control enters and a single end point that control leaves from when execution of the stmt is over.

Ex:- $S \rightarrow id := E \mid S; S \mid$ if E then S else $S \mid$ do S while E

$E \rightarrow id + id \mid id$

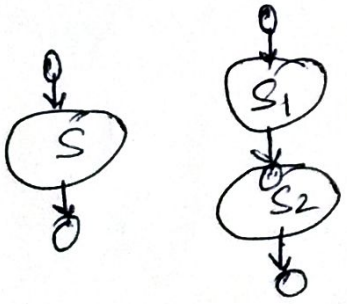
(a)



gen[s] = {d}
kill[s] = Da − {d}
out[s] = gen[s] ∪ (in[s] − kill[s])

d - definition
Da - all definition of a

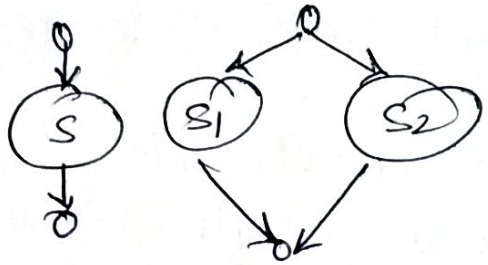
(b)



gen[s] = gen[s2] ∪ (gen[s1] − kill[s2])
kill[s] = kill[s2] ∪ (kill[s1] − gen[s2])

in[s1] = in[s]
in[s2] = out[s1]
out[s] = out[s2]

(c)



$gen[S] = gen[S_1] \cup gen[S_2]$

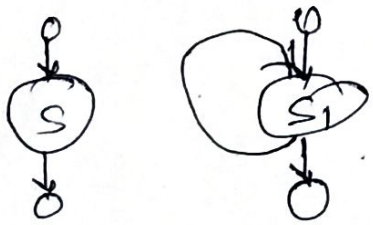
$kill[S] = kill[S_1] \cap kill[S_2]$

$in[S_1] = in[S]$

$in[S_2] = in[S]$

$out[S] = out[S_1] \cup out[S_2]$

(d)



$gen[S] = gen[S_1]$

$kill[S] = kill[S_1]$

$in[S_1] = in[S] \cup gen[S_1]$

$out[S] = out[S_1]$

Fig: Data-flow equations for reaching definitions.

Swami Keshvanand Institute of Technology, Management & Gramothan,
Ramnagaria, Jagatpura, Jaipur-302017, INDIA
Approved by AICTE, Ministry of HRD, Government of India
Recognized by UGC under Section 2(f) of the UGC Act, 1956
Tel. : +91-0141- 5160400Fax: +91-0141-2759555
E-mail: info@skit.ac.in Web: www.skit.ac.in

असतो मा सद्गमय

Tutorial Sheet -1

(Medium)

1. Remove left-recursion from following grammars,
   A->ABd / Aa / a
   B->Be / b

(Medium)

2. Remove left factoring from following grammars,
   S->iEtS / iEtSeS / a
   E->b

(Easy)

3. Remove left-recursion from following grammars,
   E-> E+E
   E-> E*E
   E->a

(Difficult)

4. Show that following grammar is LL(1) or not
   $S \rightarrow EF$
   $E \rightarrow a \mid \in$
   $F \rightarrow abF \mid ac$
   Note: Upper case letters are non-terminals and lower case letters are terminals.

Name :- Raghav Bhandari Section : D G1

Subject :- Compiler Design Branch : C.S.E

Mam : Mam, Deepa Modi

Tutorial Sheet 1

Answers :-

Ans 1

i) $A \rightarrow ABd / Aa / a$

$A \rightarrow ABd / a$

$A \rightarrow aA'$

$A' \rightarrow Bd A' / \varepsilon$

$A \rightarrow Aa / a$

$A \rightarrow aA'$

$A' \rightarrow aA' / \varepsilon$

So After removing left recursion we get.

$A \rightarrow aA'$

$A' \rightarrow Bd A' / aA' / \varepsilon$

ii) $B \rightarrow Be / b$

$B \rightarrow bB'$

$B' \rightarrow eB' / \varepsilon$

So After removing left recursion we get

$B \rightarrow bB'$

$B' \rightarrow eB' / \varepsilon.$

Ans 2

$S \rightarrow iEtS \mid iEtSeS \mid a$

$E \rightarrow b$

After removing Left factoring we get.

$S \rightarrow iEtSS' \mid a$

$S' \rightarrow eS \mid \varepsilon$

$E \rightarrow b.$

Ans 3.

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow a$

or

$E \rightarrow E + E \mid E * E \mid a$

After removing left recursion we get.

$E \rightarrow aE'$

$E' \rightarrow +EE' \mid \varepsilon$

$E' \rightarrow * EE' \mid \varepsilon$

or

$E \rightarrow aE'$

$E' \rightarrow + EE' \mid *EE' \mid \varepsilon$

Ans 4 LL(1)

First (S) : {a, ε}
First (E) : {a, ε}
First (F) : {a}
First (F') : {b, c}

Follow (S) = {$}
Follow (E) = {a}
Follow (F) = {$}
Follow (F') = {$}

Grammer :-

① S → EF
② E → a | ③ ε
④ F → a F'
⑤ F' → bF | ⑥ c

→ Parse Table :-

	a	b	c	$
S	①			
E	② ③			
F	④			
F'		⑤	⑥	

As we get production ②, ③ in one cell of E row and a column so the given grammer does not support LL(1) parsing.

Swami Keshvanand Institute of Technology, Management & Gramothan,
Ramnagaria, Jagatpura, Jaipur-302017, INDIA
Approved by AICTE, Ministry of HRD, Government of India
Recognized by UGC under Section 2(f) of the UGC Act, 1956
Tel. : +91-0141- 5160400Fax: +91-0141-2759555
E-mail: info@skit.ac.in Web: www.skit.ac.in

Tutorial Sheet -2

1. Remove left-recursion from following grammars, (Easy)
   S-> (L) / a
   L-> L,S / S
   S->A

2. Remove left factoring from following grammars, (Medium)
   A->aAB / aBc / aAc

3. Remove left-recursion from following grammars, (Medium)
   S->A
   A-> Ad / Ae / aB / ac
   B->bBC / f

4. Find the first and follow for the following grammar: (Difficult)
   $S \rightarrow ACB \mid CbB \mid Ba$
   $A \rightarrow da \mid BC$
   $B \rightarrow g \mid \varepsilon$
   $C \rightarrow h \mid \varepsilon$
   Note: Upper case letters are non-terminals and lower case letters are terminals.

Name- Sakshi Khandelwal
Roll No- 19CSKCS812

Tutorial Sheet - 2

(1) $S \rightarrow (L) / a$
$L \rightarrow L, S / S$
$S \rightarrow A$

Removing Left Recursion:

$S \rightarrow (L) / a$
$L \rightarrow SL'$
$L' \rightarrow ; SL' / \varepsilon$
$S \rightarrow A$

(2) Left factoring:

$A \rightarrow aAB / aBc / aAc$

$A \rightarrow aA'$
$A' \rightarrow AB / Bc / Ac$

(3)

$S \rightarrow A$
$A \rightarrow Ad / Ae / aB / ac$
$B \rightarrow bBC / f$

$S \rightarrow A$
$A \rightarrow Ad / aB$
$A \rightarrow Ae / aC$
$B \rightarrow bBc / f$

Now Removing left Recursion

$S \rightarrow A$

$A \rightarrow aBA'$

$A' \rightarrow dA' / \varepsilon$

$A \rightarrow aCA'$

$A' \rightarrow eA' / \varepsilon$

$B \rightarrow bBC / f$

4) $S \rightarrow ACB / CbB / Ba$

$A \rightarrow da / BC$

$B \rightarrow g / \varepsilon$

$C \rightarrow h / \varepsilon$

First (S) = { d, g, h, b, a, ε }

First (A) = { d, g, h, ε }

First (B) = { g, ε }

First (C) = { h, ε }

Follow (A) = { h, g, $ }

Follow (S) = { $ }

Follow (B) = { $, a, h, g }

Follow (C) = { g, $, b, h }

Swami Keshvanand Institute of Technology, Management & Gramothan,
Ramnagaria, Jagatpura, Jaipur-302017, INDIA
Approved by AICTE, Ministry of HRD, Government of India
Recognized by UGC under Section 2(f) of the UGC Act, 1956
Tel. : +91-0141- 5160400Fax: +91-0141-2759555
E-mail: info@skit.ac.in Web: www.skit.ac.in

असतो मा सद्गमय

Tutorial Sheet -3

1. Write three address code for following programming segments, (Medium)

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

   b) If A < B and C < D then t = 1 else t = 0

   c)
```
c = 0
do
{
    if (a < b) then
            x++;
    else
            x–;
    c++;
} while (c < 5)
```

2. Translate the arithmetic expression a + b * c / e ↑ f + b * c into, (Difficult)
   a) Three address code
   b) Quadruple
   c) Triples
   d) Indirect triples

Name: Shiv Maheshwari

Roll No: 19ESKCS819

Tutorial Sheet

1) Three Address code:

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

1. $t_1 = a * b$
2. $t_2 = -t_1$
3. $t_3 = c + d$
4. $t_4 = a + b$
5. $t_5 = c + d$
6. $t_6 = t_4 + t_5$
7. $t_7 = -t_6$
8. $t_8 = t_5 + t_7$
9. $t_9 = t_2 + t_8$

2) if A < B and C < D then t = 1 else

1. if A < B goto (3)
2. goto (7)
3. if C < D goto (5)
4. goto (7)
5. t = 1
6. goto (8)
7. t = 0
8. Stop

c) c = 0

do

{

if (a < b) then

x + 1 ;

else

x - ;

c + + ;

} while (c < 5)

1. c = 0
2. if a < b goto (4)
3. goto (7)
4. t_1 = x + 1
5. x = t_1
6. goto (9)
7. t_2 = x - 1
8. x = t_2
9. t_2 = c + 1
10. c = t_3
11. if c < 5 goto (2)
12. stop

3) $a + b \times c / e \uparrow f + b \times c$

a) Three Address Code :-

$t_1 = e \uparrow f$

$t_2 = b \times c$

$t_3 = t_2 / t_1$

$t_4 = t_3 + t_2$

$t_5 = a + t_4$

b) Quadruple :-

S.No	Operator	arg1	arg2	Result
0	$\uparrow$	e	f	t_1
1	*	b	c	t_2
2	/	t_2	t_1	t_3
3	+	t_3	t_2	t_4
4	:=	a	t_4	t

c) Triples :

S.No	Operator	arg1	arg2
(0)	$\uparrow$	e	f
(1)	*	b	c
(2)	/	(1)	(0)
(3)	+	(2)	(1)
(4)	+	a	(3)

c) Indirect Triples:-

Statement	
(0)	31
(1)	32
(2)	33
(3)	34
(4)	35

	operator	arg 1	arg 2
(31)	↑	e	d
(32)	*	b	c
(33)	/	(32)	(31)
(34)	+	(33)	(32)
(35)	+	a	(34)

Swami Keshvanand Institute of Technology, Management & Gramothan,
Ramnagaria, Jagatpura, Jaipur-302017, INDIA
Approved by AICTE, Ministry of HRD, Government of India
Recognized by UGC under Section 2(f) of the UGC Act, 1956
Tel. : +91-0141- 5160400Fax: +91-0141-2759555
E-mail: info@skit.ac.in Web: www.skit.ac.in

Question Bank:

Unit-I

Q1. What is a compiler?

A. A compiler does a conversion line by line as the program is run
B. A compiler converts the whole of a higher level program code into machine code in one step
C. A compiler is a general purpose language providing very efficient execution
D. All of the Above

Ans.: B

Q2. Users write the programs in which language?

A. Low-level Language
B. High-Level Language
C. Decimal-Format
D. Middle-Level Language

Ans: B

Q3. Which computer program accepts the high-level language and converts it into assembly language?

A. Interpreter
B. Linker
C. Assembler
D. Compiler

Ans: D

Q4. Does the compiler program translate the whole source code in one step?

A. No
B. Depends on the Compiler
C. Don't Know
D. Yes

Ans:D

Q5.Which tool is used for grouping of characters in tokens in the compiler?

A. Parser
B. Code optimizer
C. Code generator
D. Scanner

Ans: D

Q6.What is the linker?

A. It is always used before the program execution.
B. It is required to create the load module.
C. It is the same as the loader
D. None of the above

Ans: B

Unit-II

Q1. What are the stages in the compilation process?
- A. Feasibility study, system design, and testing
- B. Implementation and documentation
- C. Analysis Phase, Synthesis Phase
- D. None of These

Ans.: C

Q2. What is the definition of an interpreter?
- A. An interpreter does the conversion line by line as the program is run
- B. An interpreter is a representation of the system being designed
- C. An interpreter is a general purpose language providing very efficient execution
- D. All of the Above

Ans.: A

Q3. Symbol table can be used for
- A. Checking type compatibility
- B. Storage allocation
- C. Suppressing duplication of error massages
- D. All of the Above

Ans.: D

Q4. Assembly language
- A. is usually the primary user interface
- B. requires fixed format commands
- C. is a mnemonic form of machine language
- D. is a quite different from the SCL interpreter

Ans.: C

Q5. Every symbolic references to a memory operand has to be assembled as
- A. (offset, index base)
- B. (segment base, offset)
- C. (index base, offset)
- D. offset

Ans.: B

Q6. Which translator program converts assembly language program to object program
- A. Assembler
- B. Compiler
- C. Microprocessor
- D. Linker

Ans.: A

Q7. A compiler for a high level language that runs on one machine and produces code for a different machine is called

A. Optimizing compiler
B. One pass compiler
C. Cross compiler
D. Multipass Compiler

Ans.: C

Q8. The action of passing the source program into the proper syntactic classes is known as
A. syntax analysis
B. lexical analysis
C. interpretation analysis
D. general syntax analysis

Ans.: B

Q9. Grouping of characters into tokens is done by the
A. scanner
B. parser
C. code generator
D. code optimizer

Ans.: A

Q10. Lexical analysis phase uses
A. regular grammar
B. context free grammar
C. context sensitive grammar
D. none of the above

Ans.: A

Unit-III

Q1. Which one of the following is not a syntax error
- A. Semantic
- B. Lexical
- C. Arithmetic
- D. Logical

Ans.: A

Q2. CFG can be recognized by a
- A. Push-down automata
- B. 2-way linear bounded automata
- C. Both (a) and (b)
- D. None of these

Ans. C

Q3. The ambiguous grammar can have
- A. Only one parse tree
- B. More than one parse tree
- C. Parse trees with l-values
- D. None of the above

Ans. B

Q4. Which of the following suffices to convert an arbitrary CFG to an LL(1) grammar
- A. Removing left recursion alone
- B. Factoring the grammar alone
- C. Removing left recursion and factoring the grammar
- D. None of the above

Ans. C

Q5. The ‘k’ in LR(k) cannot be
- A. 0
- B. 1
- C. 2
- D. None of these

Ans. D

Q6. Non backtracking form of the top-down parser are called
- A. Recursive-descent parsers
- B. Predictive parsers
- C. Shift reduce parsers
- D. None of these

Ans. B

Q7. Parsing table in the LR parsing contains
- A. action, goto

B. state, action
C. input, action
D. state, goto

Ans. A

Q8. The condensed form of the parse tree is called
A. L-attributed
B. Inherited attributed
C. DAG
D. Syntax tree

Ans. D

Q9. Shift-reduce parsers are
A. top-down parsers
B. bottom-up parsers
C. both (a) and (b)
D. none of these

Ans. B

Q10. In some programming languages, an identifier is permitted to be a letter followed by any number of letters or digits. If L and D denote the sets of letter and digits respectively, which expression defines an identifier?
A. (LUD)+
B. L(LUD)*
C. (LD)*
D. L(LD)*

Note: Here U stands for Union

Ans. B

Q11. Backtracking is possible in
A. LR parsing
B. Predictive parsing
C. Recursive descent parsing
D. None of the above

Ans. C

Q12. Consider the following grammar
expr op expr
expr->id
op-> + | *

Which of the following is true?
A. op and expr are start symbols
B. op and id are terminals
C. expr is start symbol and op is terminal
D. none of these

Ans. C

Q13. To compute FOLLOW(A) for any grammar symbol A
- A. We must compute FIRST of some grammar symbols
- B. No need of computing FIRST of some symbol
- C. May compute FIRST of some symbol
- D. None of these

Ans. A

Q14. Which language is generated by the given grammar S-> 0S1 | 01
- A. 0011
- B. 00*11
- C. 001*1
- D. 00*1*1

Ans. D

Q15. The space consuming but easy parsing is

- A. LALR
- B. SLR
- C. LR
- D. Predictive parser

Ans. A

Q16. A top down parser generates
- A. left-most derivation
- B. right-most derivation
- C. right-most derivation in reverse
- D. left-most derivation in reverse

Ans. A

Q17. Choose the false statement
- A. LL(k) grammar has to be a CFG
- B. LL(k) grammar has to be unambiguous
- C. There are LL(k) grammars that are not Context Free
- D. LL(k) grammars cannot have left recursive non-terminals

Ans. C

Q18. If a grammar is unambiguous then it is surely be
- A. regular
- B. LL(1)
- C. Both (a) and (b)
- D. Cannot say

Ans. D

Q19. Predictive parsing is a special case of
- A. top down parsing

B. bottom up parsing
C. recursive descent parsing
D. none of the above

Ans. C

Unit-IV

Q1. Type checking is normally done during
- A. lexical analysis
- B. syntax analysis
- C. syntax directed translation
- D. code optimization

Ans.: C

Q2. An intermediate code form is
- A. Postfix notation
- B. Syntax trees
- C. Three address code
- D. All of these

Ans.: D

Q3. Semantic errors can be detected
- A. at compile time only
- B. at run time only
- C. both at compile and run time
- D. none of these

Ans.: C

Q4. Undeclared name is …… error
- A. syntax
- B. lexical
- C. semantic
- D. not an error

Ans. C

Q5. Intermediate code generator is used between
- A. symbol table and code generator
- B. symbol table and code optimizer
- C. code optimizer and code generator
- D. semantic analyzer and code optimizer

Ans.: D

Q6. 'Divide by 0' is a
- A. lexical error
- B. syntactic error
- C. semantic error
- D. internal error

Ans. C

Q7. In which of the following has attribute values at each node
- A. Associated parse trees

B. Postfix parse tree
C. Annotated parse tree
D. Prefix parse tree

Ans.: C

Q8. The post fix form of A-B/(C*D$E) is
A. ABCDE$*/-
B. AB/C*DE$
C. ABCDE$-/*
D. ABCDE/-*$

Ans. A

Q9. Synthesized attribute can easily be simulated by an
A. LL grammar
B. Ambiguous grammar
C. LR grammar
D. None of the above

Ans. C

Q10. The prefix form of (A+B)*(C-D) is
A. +-AB*C-D
B. *+-ABCD
C. *+AB-CD
D. *AB+CD

Ans. C

Q11. In a syntax directed translation scheme, if the value of an attribute of a node is a function of the values of the attributes of its children, then it is called a
A. Synthesized attribute
B. Inherited attribute
C. Canonical attribute
D. None of the above

Ans. A

Q12. Which of the following is not an intermediate code form?
A. Postfix notation
B. Syntax trees
C. Three address code
D. Quadruples

Ans. D

Q13. Three address codes can be implemented by
A. indirect triples
B. direct triples
C. both (a) and (b)

D. none of the above

Ans. A

Q14. In a bottom up evaluation of a syntax directed definition, inherited attributes can be

A. always be evaluated
B. be evaluated only if the definition is L-attributed
C. be evaluated only if the definition has synthesized attributes
D. none of the above

Ans. C

Q15. Three address code involves

A. at the most 3 address
B. exactly 3 address
C. no unary operators
D. none of the above

Ans. A

Q16. Inherited attribute is a natural choice in

A. keeping track of variable declaration
B. checking of the correct use of L-values and R-values
C. both (a) and (b)
D. none of the above

Ans. C

Unit-V

Q1. The idea of cache memory is based _____
A. On the property of locality of reference
B. On the heuristic 90-10 rule
C. On the fact that references generally tend to cluster
D. All of the mentioned
Ans. A

Q2.Which of the following known as the text part of a program that does not change at runtime. Its memory requirements are known at the compile time?
A. Code
B. Procedures
C. Variables
D. All of the above
Ans: A

Q3.A procedure has a start and an end delimiter and everything inside it is called the body of the procedure.
A. TRUE
B. FALSE
C. A procedure has a start with delimiter but not end with delimiter.
D. A procedure has a not start with delimiter but end with delimiter.
Ans: A

Q4.In activation record, Which of the following Stores the address of activation record of the caller procedure?
A. Access Link
B. Actual Parameters
C. Control Link
D. Temporaries
Ans: C

Q5. Whenever a procedure is executed, its activation record is stored on the stack, also known as?
A. Access Stack
B. Control stack
C. Formal Stack
D. Return Stack
Ans : B

Q6. _____ are known at the runtime only, unless they are global or constant.
A. Values
B. Object
C. Variables
D. All of the above
Ans : C

Q7.The location of memory (address) where an expression is stored is known?

A. r-value
B. k-value
C. l-value
D. t-value

Ans : C

Q8.What is true about Formal Parameters?

A. These variables are declared in the definition of the called function.
B. These variables are specified in the function call as arguments.
C. Variables whose values or addresses are being passed to the called procedure are called Formal Parameter.
D. All of the above

Ans: A

Q9.In which mechanism, the calling procedure passes the r-value of actual parameters and the compiler puts that into the called procedure's activation record?

A. Pass by Reference
B. Pass by Name
C. Pass by Copy-restore
D. Pass by Value

Ans: D

Q10.In which mechanism, the name of the procedure being called is replaced by its actual body?

A. Pass by Reference
B. Pass by Name
C. Pass by Copy-restore
D. Pass by Object

Ans: B

Q11.What will be error?

7 = x + y;

A. l-value error
B. r-value error
C. Infinite loop
D. Both A and B

Ans: A

Unit-VI

Q1. A basic block can be analyzed by
- A. A DAG
- B. A graph which may involve the cycles
- C. Flow Graph
- D. None of These

Ans.: A

Q2. Which of the following is true for the flow of control among procedures during execution of program?
- A. Control flows randomly
- B. Control flows line by line without jumping
- C. Control flows sequentially
- D. None of these

Ans. C

Q3. The identification of common sub-expression and replacement of run time computations by compile-time computations is------
- A. Local optimization
- B. Constant folding
- C. Loop optimization
- D. Data flow analysis

Ans: B

Q4. Which of the following class of statement usually produces no executable code when compiled?
- A. Declaration
- B. Assignment statements
- C. Input and output statements
- D. Structural statements

Ans: A

Q5. Code optimization is responsibility of--------
- A. Application programmer
- B. System programmer
- C. Operating system
- D. All of the above.

Ans: B

Q6. Dead-code elimination in machine code optimization refersto ----------
- A. Removal of all labels.
- B. Removal of values that never get used.
- C. Removal of function which are not involved.
- D. Removal of a module after its use.

Ans: B

Q.7. Which of the following statement is false?

A. Flow graph is used to represent DAG.
B. Three address code is the input to the code generator.
C. The first statement of three address code is always leader of the first basic block. Transformation of block is needed for code optimization.

Ans: A

Q.8. -------- is the final phase of compiler.

A. Semantic analysis
B. Code generation
C. Target code generation
D. Syntax analysis

Ans: B

Q.9.Some code optimizations are carried out on the intermediate code because------

A. They enhance the portability of the compiler to other target processors
B. Program analysis is more accurate on intermediate code than on machine code
C. The information from dataflow analysis cannot otherwise be used for optimization
D. The information from the front end cannot otherwise be used for optimization

Ans: A

Q.10 Peephole optimization is form of------

A. Loop optimization
B. Local optimization
C. Constant folding
D. Data flow analysis

Ans: B

Swami Keshvanand Institute of Technology, Management &Gramothan,
Ramnagaria, Jagatpura, Jaipur-302017, INDIA
Approved by AICTE, Ministry of HRD, Government of India
Recognized by UGC under Section 2(f) of the UGC Act, 1956
Tel. : +91-0141- 5160400Fax: +91-0141-2759555
E-mail: info@skit.ac.in Web: www.skit.ac.in

List of Text and Reference Books

Text Books:

1. Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. "Compilers, principles, techniques." Addison wesley 7.8 (1986): 9.
2. Ullman, Jeffrey D., and Alfred V. Aho. "Principles of compiler design." Reading: Addison Wesley (1977).

Reference Books:

1. Srikant, Y. N., and Priti Shankar, eds. The compiler design handbook: optimizations and machine code generation. CRC Press, 2018.
2. Mogensen, TorbenÆgidius. Introduction to compiler design. Springer, 2017.
3. Grune, Dick, et al. Modern compiler design. Springer Science & Business Media, 2012.
4. Aho, Alfred V. Compilers: principles, techniques and tools (for Anna University), 2/e. Pearson Education India, 2003.
5. Louden, Kenneth C. "Compiler construction." Cengage Learning (1997).
6. Wilhelm, Reinhard, Dieter Maurer, and Stephen S. Wilson. Compiler design. Reading: Addison-Wesley Publishing Company, 1995.