POORNIMA

Date	Unit No.	Lecture No.	Faculty	Subject Name	Subject Code	Main Topics:-

Unit-01 : Introduction & Lexical Analyzer

1) Assembler : It is used to convert the Assembly language to machine language.
2) Interpreter : It will read the message line by line.
3) Compiler : It will read the whole message, document and fatch error, show error details.

★ Analysis of compiler design

[1] Lexical Analysis
[2] Syntax Analysis
[3] Semantic Analysis
[4] Intermediate code generator
[5] Code optimization
[6] Machine code generator

★ Definition of Compiler

⇒ A compiler is software utility that translate code & retain in higher programming language.
The background of compiler is DFA & NDFA, formal languages.
CNF, GNF.

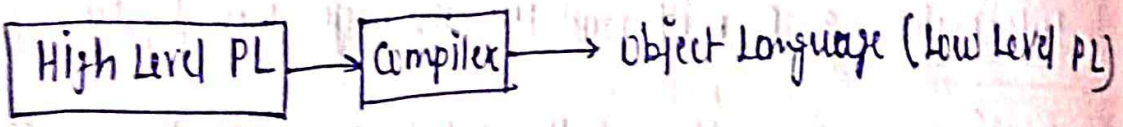
Main Ideas, Questions & Summary:

Library / Website Ref.:-

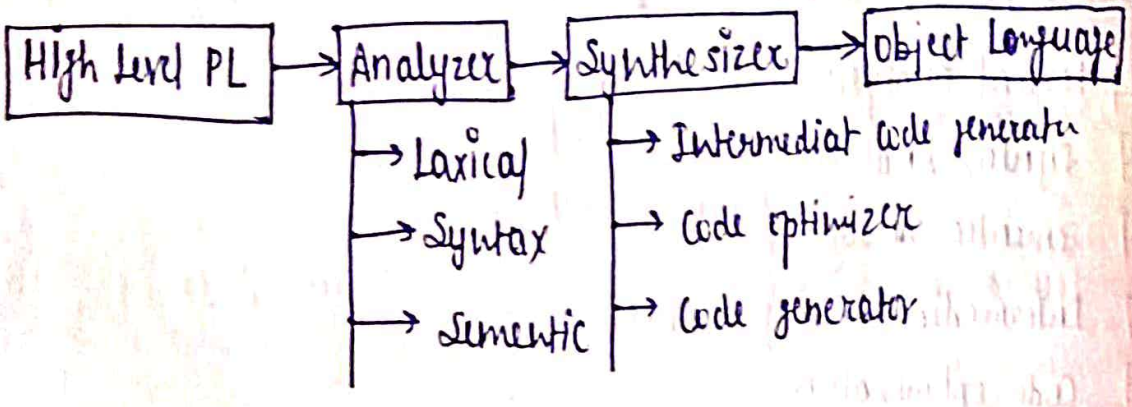
Types of Compiler

1. Single Pass Compiler
2. Multi Pass Compiler

→ In Single Pass Compiler the I/P is High Level programming Language and O/P is object language.



→ In multiPass Compiler High level language is passes to more than one pass to generate the target language



★ Programming Language Process

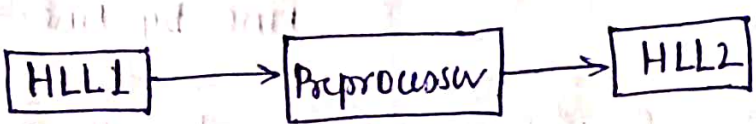
⇒ In compiler design we write the program in HLL. which is easy to understand. These program are feed into a series of tools operating system compile to get, target code. that can be use by the machine. This is known as language processing System.

→ Steps of language processing

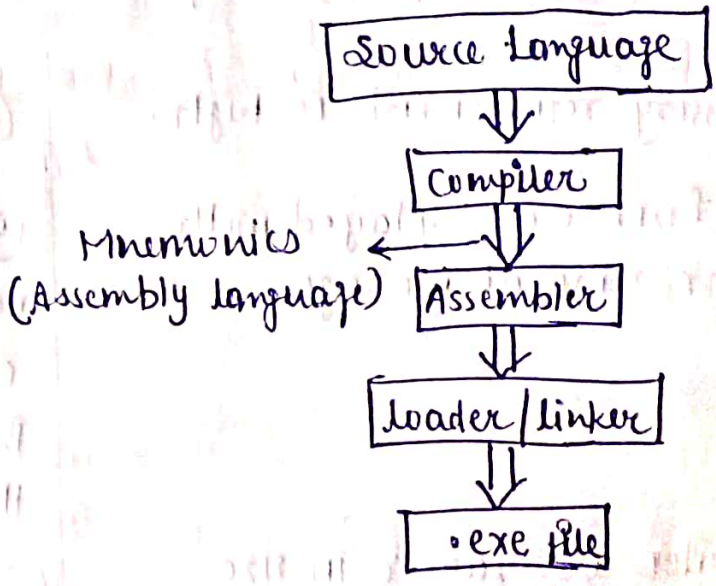
POORNIMA

Date	Unit No.	Lecture No.	Faculty	Subject Name	Subject Code	Main Topics:-

[1] Preprocessor : Preprocessor is a type of translater which converts one HLL to another HLL.



Eg: #include <stdio.h>



[2] Loader/Linker : A Linker links different objects files into a single executable file.

→ A Loader Loads that Executable files into the memory and Execute it.

[3] .exe file ⇒ Executable code can be produced by compiling and linking in one step. An executable file has a filename extension of .exe (windows) or no filename extension (UNIX).

Main Ideas, Questions & Summary:

Library / Website Ref.:-

★ Difference between compiler & Interpreter

Compiler	Interpreter
① compiler is reads the the whole program, message.	① Interpreter reads the program Line by Line.
② Speed of the compiler is high.	② Speed of the Interpreter is Low.
③ Memory requirement is high.	③ Memory requirement is Low.
④ All Errors are displayed with Error removal messages.	④ continuous translating the message until the 1st error is not remove. But Error detection is easier than Compiler.
⑤ Compiler are Larger in size.	⑤ Interpreter are smaller in size.
⑥ Compiler based languages are C, C++ etc.	⑥ Interpreter based languages are : Python, Perl, Ruby, etc.

Note: Java is compile as well as interpreter language.

POORNIMA

Date	Unit No.	Lecture No.	Faculty	Subject Name	Subject Code	Main Topics:-

⊛ Bootstraping

⇒ Bootstrapping is a process in which simple language is used to translate more complicated program which is turn may handle more complicated programs.

These complicated program can further handle even more complicated program and so on.

⊛ Finite automata

→ Finite automata are used to recognize patterns.

→ It takes the string of symbol as I/P & changes its state accordingly. when the desired symbol is found, then the transition occurs.

→ A finite automaton is a collection of 5 tuples $(Q, \Sigma, \delta, q_0, F)$

Q : Finite set of states

Σ : Finite set of input symbol

q_0 : Initial state

F : Final state

δ : Transition function

Main Ideas, Questions & Summary:

Library / Website Ref.:-

★ Finite automatan : Finite automatan are used to recognise pattern.

→ It takes the string of symbol as I/P & changes its state accordingly. when the desired symbol is found than the transition occurs.

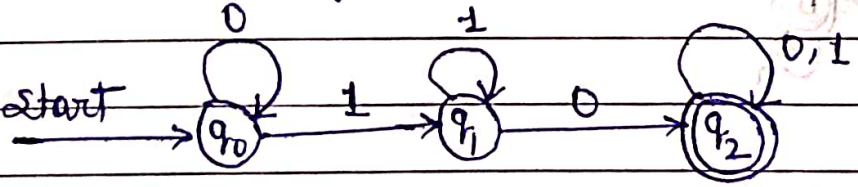
→ A finite automaton is collection of 5 tupples $(Q, \Sigma, \delta, q_0, F)$

Q : Finite set of state
Σ : Finite set of Input symbol
δ : Transition function
q_0 : Initial state
F : Final state

★ Designing of DFA

$Q = \{q_0, q_1, q_2\}$
$\Sigma = \{0, 1\}$
$q_0 = \{q_0\}$
$F = \{q_2\}$

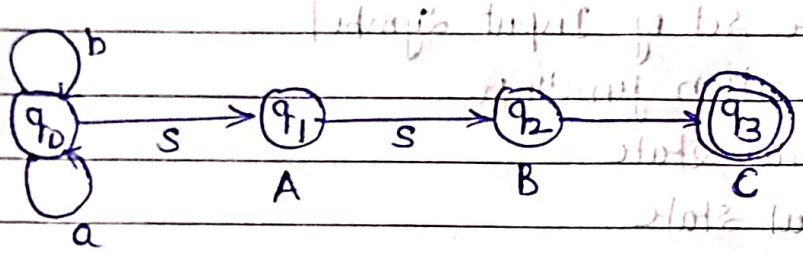
Sol^n ⇒ Transition diagram :



Transition Table

Present state	Next state for Input 0	Next state for Input 1
q_0	q_0	q_1
q_1	q_2	q_1
q_2	q_2	q_2

* Regular Expression to Regular Grammer
(DFA to Regular Expression)



Sol^n ⇒

$S \rightarrow aS \mid bS$

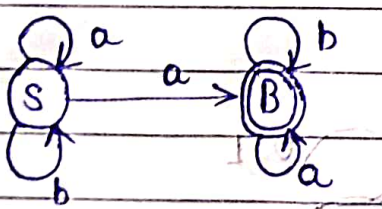
$S \rightarrow aA$

$S \rightarrow bB$

$B \rightarrow bC \mid b$

$C \rightarrow \Lambda$

Ex.



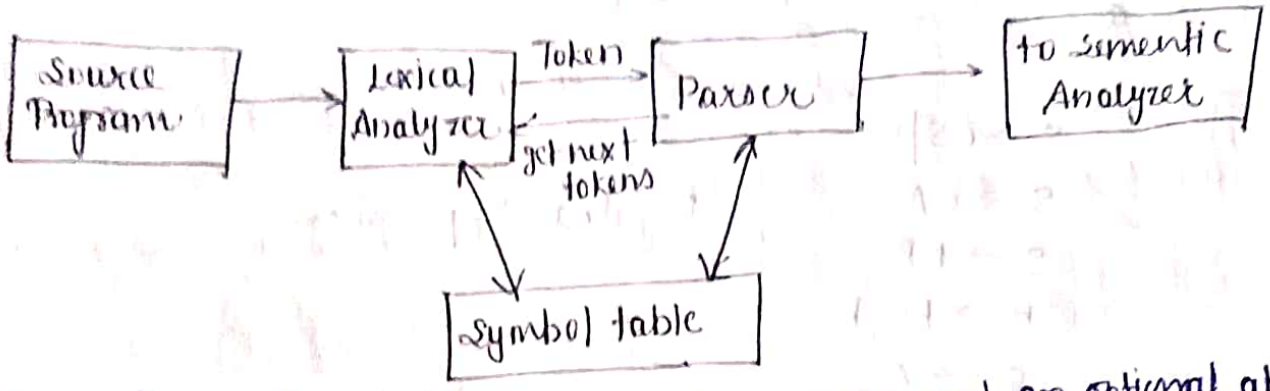
Sol^n ⇒

$S \rightarrow aS \mid bS \mid aB \mid a$

$B \rightarrow aB \mid bB \mid a \mid b$

* Lexical Analyzer.

⇒ the main task of Lexical Analyzer is to read the I/P character from the source Program, group them into lexims & produce as o/p as a sequence of token in each laxim in the source program.



* A token is a pair consisting of a token name and an optional attribute value.

* A pattern is a discription of the form that the lexims made take.

* A leximo is a sequence of character in the source program that matches the pattern for a token.

* Regular expressions are algebric representation of language. for example

R.E for identifiers.

$$\boxed{\text{letter}(\text{letter}/\text{digit})^*}$$

R.E for all string ending with abb.

$$\boxed{(a,b)^* abb}$$

* Regular definition : If Σ is an alphabet of basic symbols then

$$\text{digit} \rightarrow 0|1|2|\cdots 9 \qquad \begin{array}{l} d_1 \rightarrow x_1 \\ d_2 \rightarrow x_2 \\ \vdots \\ d_n \rightarrow x_n \end{array}$$

POORNIMA

Date	Unit No.	Lecture No.	Faculty	Subject Name	Subject Code	Main Topics:-

① Regular definition for identifiers

id → letter / ~~o~~ digit

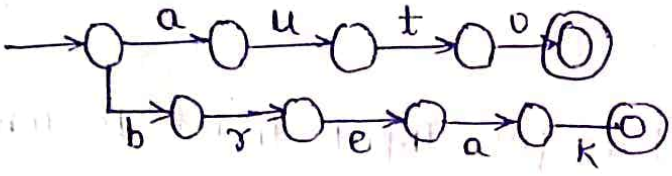
② Regular definition for letter

letter → [A-Z] / [a-z]

★ Regular diagram : It is a graphic representation of Regular expression.

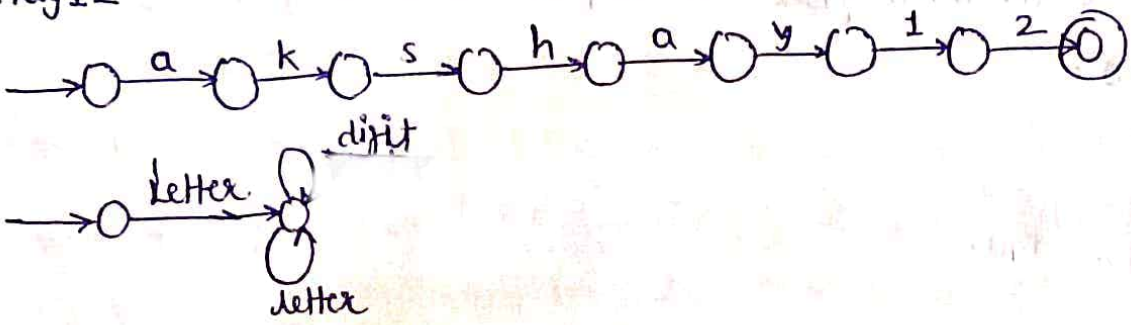
R.E : (a|b|c| . . . |z)*

Ex. Auto, break etc.



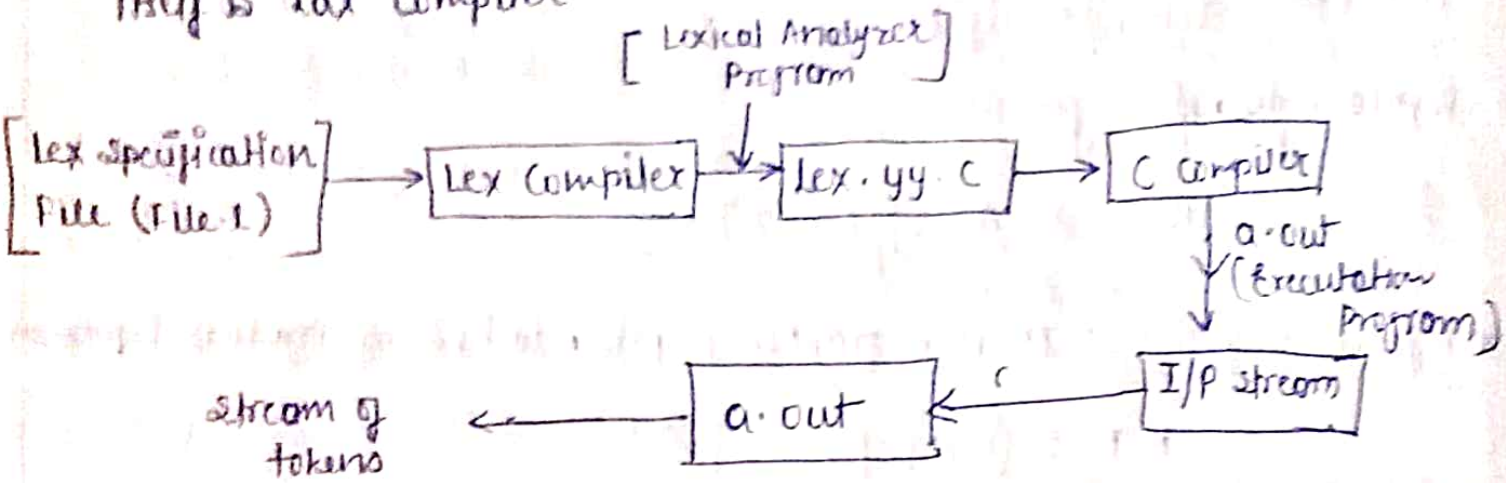
Ex. transition diagram for identifiers

Ex. akshay12



★ Lexical Analyzer generator tools (Lex)

⇒ Lex is a tool that allow one to specify a lexical analyzing by regular expression for describing for tokens. The I/p notation for lex tools is referred to as lex language and the tool itself is lex compiler



Given, some Lex specification file, Input to the Lex compiler & send to Lexical Analyzer. It's generate 'C' file & send to C compiler, it generate Executable file to input stream & generates stream of tokens.

[1] Declaration Rule

```
%{
  ===
%}
```

In this section declaration of variable & constant, it also consists of some regular Expression.

POORNIMA

Date	Unit No.	Lecture No.	Faculty	Subject Name	Subject Code	Main Topics:-

Eg:
```
%{
#include<stdio.h>
#define Pi 3.14
int a;
number [0-9]+
op[+/*/%/^]
%}
```

★ Rule Section :- In this section consist of regular expression ~~whic~~ with corresponding action. the transition rule can be given

Ex.
```
%%
R1 {action}
R2 {action}
.
.
Rn {action}
number {printf("This is Number");
%%
```

★ Procedure section

The procedure section contain the definition of all the procedure. the procedure may be required in action part of the rule section

Ex:
```
int main(){
    yylex();
    return;
}
```

Main Ideas, Questions & Summary:

Library / Website Ref.:-

→ The Auxiliary declaration or functions are copied ~~to~~ such to the lex.yy.c file.

→ yyin is a variable of file *fp and points to the input file. If programmer assign the then yyin is said to the points to that file.

→ yytext : It is a type of character *. It contains the lexims currently found.

→ yyleng : It is variable of type int Pointed to by yytext

→ yyrape : less declaration of function & return type lex.

★ Character Class

Pattern	Matches
[abc]	one of abc
[a-z]	any letter from a-z
[a|-z]	one of a, -z
[\t\n]+	white space

POORNIMA

Date	Unit No.	Lecture No.	Faculty	Subject Name	Subject Code	Main Topics:-

★ Error handling in Lexical Analyser

⇒ When the token pattern does not match the prefix of remaining input, the lexical analyzer gets stuck and has recover from this state to analyze the remaing input.

In simple words, a lexical error occurs when a sequence of character does not match the pattern of any token.
It typically happens during the execution of program

→ Types of Lexical Error

1. Exceeding length of identifier or numeric constant.
2. Appearance of illegal characters.
3. Unmatch string
4. Spelling error
5. Replacing a character with an incorrect character.
6. Removal of character that should be present
7. transposition of two ~~string~~ characters.

Main Ideas, Questions & Summary:

POORNIMA

Unit-02 : Review of CFG Ambiguity of grammers

Date	Unit No.	Lecture No.	Faculty	Subject Name	Subject Code	Main Topics:-

★ Syntax Analyzer

⇒ A program is made up of function, ~~out~~ A function out of declaration and statement or statement out of operation and so on.

The syntax of programming Language construct context free grammer.

$$A \rightarrow \alpha$$

This Expression is called Baker-Norr form.

Grammer after significant

→ A Grammer give precise, yet easy to understand syntetic specification for a programming Language.

★ **Classes of Grammer** : From certain classes of grammer

⇒ We can construct automatically an efficient parser that determines the syntactic structure of the source program.

A a site benefit, the parser construction process can ~~accept~~ reveal synthetic ambiguity & trouble discourse that might have slipped through the initial design phase of the Language.

The structure important to a language to be evolved or developed iteratively by adding new construct to perform new task.

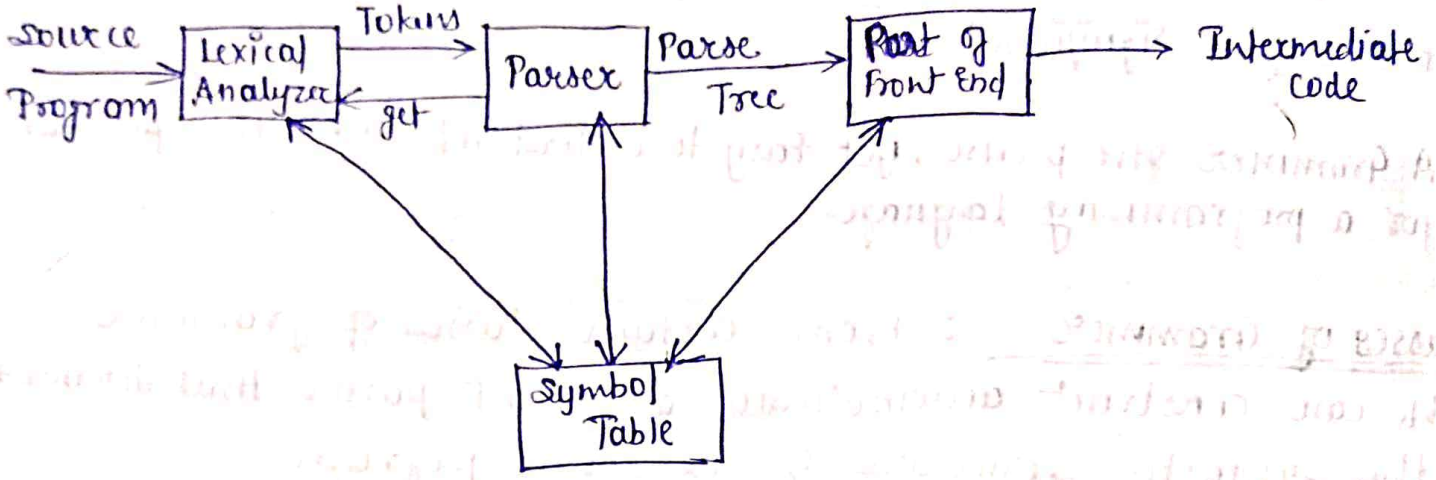
Main Ideas, Questions & Summary:

Library / Website Ref.:-

⇒ Parser is a compiler generating tool which is used for generating Parse tree which is input to the symentic analyzer.

★ Role of Parser

⇒ source program which is I/p to the Lexical Analyzer. The parser obtained a string of tokens from the Lexical analyzer and verifies that the string can be grammer ~~from~~ for the source Language



It detects and report any syntax errors and produces a parse tree from which intermediate code can be generated.

★ There are three types of Parser

① Universal Parser : Developed by Cocke - younger kasami (inefficient)

② Top-Down Parser
→ Recursive decent Parser
→ Non-Recursive Predictive parser
→ LL(1) Parser

POORNIMA

Date	Unit No.	Lecture No.	Faculty	Subject Name	Subject Code	Main Topics:-

[3] Bottom-Up Parser

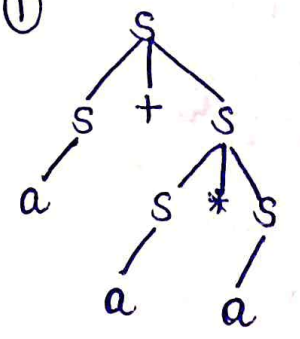
→ Shift Redux Parser
→ operater Procedure Parser
→ More efficient Parser
  - → SLR (simple Left Recursive)
  - → CLR (canonical Left Recursive)
  - → LALR (Look ahead Left Recursive)

★ Remove the Ambiguity in CFG

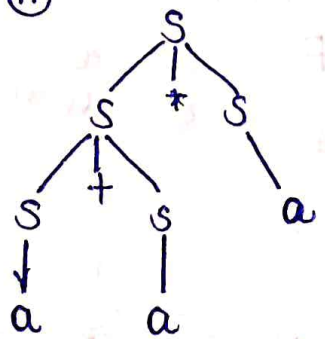
[1] Associativity Problem

$S \rightarrow S+S \mid S*S \mid a$

(I)



(II)



- If the associativity is left to right, then we have to prompt a left recursion in the production. The parse tree will also be left recursive and grow on the left side.

  +, −, *, / are left associative operators.

  ^ is a right associative operator.

Main Ideas, Questions & Summary:

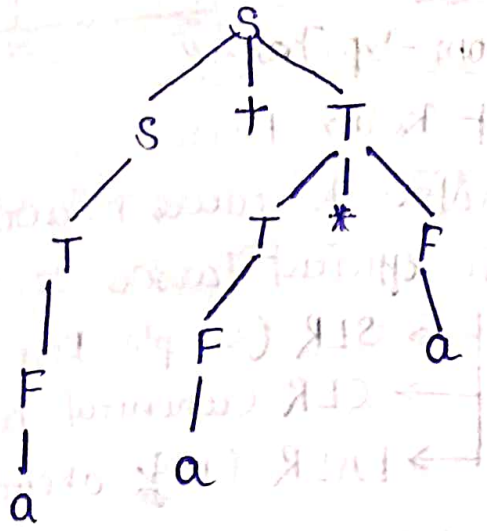
Library / Website Ref.:-

[2] Precedence Problem

$S \rightarrow S+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow a$



Ex. $E \rightarrow E+E \mid E-E \mid E*E \mid E/T \mid (E) \mid$ identifiers

Soln)

$E \rightarrow E+T \mid E-T \mid T$

$T \rightarrow T*F \mid T/F \mid F$

$E \rightarrow (E) \mid$ identifiers

★ Left Recursive Grammer

⇒ A Grammer is not recursive if it has a non-recursive α, such that their is a derivation

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

As top-down cannot handel left ~~recursion~~ recursive grammer, so that transformation is needed, to eliminate left recursion

A production of form $\boxed{A \rightarrow A\alpha} \overset{\alpha}{\Rightarrow}$ $A \rightarrow A\alpha$

$\Rightarrow A\alpha\alpha$

$\Rightarrow A\alpha\alpha\alpha$

POORNIMA

Date	Unit No.	Lecture No.	Faculty	Subject Name	Subject Code	Main Topics:-

Ex. $A \to A\alpha / \beta$

$\Downarrow$

$A \to \beta A'$

$A' \to \alpha A' / \varepsilon$

Ex.

Given $E \to E + T / T$ (A, A, α, β) $\Rightarrow$ $E \to TE'$, $E \to +TE' / \wedge$

$T \to T * F / F$ (A, A, α, β) $\Rightarrow$ $T \to FT'$, $T' \to *FT' / \wedge$

$F \to (E) / id$

Grammer after removal left recursion.

(*) General form of Grammer

$$[A \to A\alpha_1 / A\alpha_2 / A\alpha_3 / \ldots A\alpha_n / \beta_1 / \beta_2 / \beta_3 / \beta_4 \ldots \beta_n]$$

$\Downarrow$

$$[A \to \beta_1 A' / \beta_2 A' / \ldots \beta_n A'$$
$$A' \to \alpha_1 A' / \alpha_2 A' / \ldots \alpha_n A' / \wedge]$$

Main Ideas, Questions & Summary:

Library / Website Ref.:-

$S \to Aa \mid b \Rightarrow S \to Sda \mid b$

$A \to Ac \mid Sd \mid \wedge \Rightarrow A \to Ac \mid Aad \mid bd$

In above grammer their is indirect left recursion.

$S \to bS'$
$S' \to daS' \mid \wedge$

$A \to bdA' \mid A'$
$A' \to cA' \mid adA' \mid \wedge$

★ Left Factoring

⇒ It is a grammer that is useful for producing a grammer suitable for predictive or top-down parsers. A Grammer of form:

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

⇓

$A \to \alpha A'$
$A' \to \beta_1 \mid \beta_2$ ← Removal to Left factoring

General form:

~~$A \to \alpha A' \mid \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \mid \ldots \ldots \alpha_n$~~

$A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3 \mid \ldots\ldots \alpha\beta_n \mid \gamma$

⇓

$A \to \alpha A' \mid \gamma$
$A' \to \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots \beta_n$

POORNIMA

Date	Unit No.	Lecture No.	Faculty	Subject Name	Subject Code	Main Topics:-

Ex: Given $S \to icts / icts e s / a$, $C \to b$ { if Condition then Statement else Statement

α_1 β_1 α_2 β_2 γ

$\Downarrow$

$S \to icts\, S' / a$

$S' \to es / \wedge$

$C \to b$

Q Remove the left factoring & left recursion in following Grammer.

[I] $S \to S+S / SS / (S) / S^* / a$

[II] $S \to s(s)s / \wedge$

(III) $S \to asbs / bsas / \wedge$

★ **Top-down Parsing**

Top down parsers can be viewed as the problem of constructing a parse tree starting form the root & creating the Node of the parse tree in preorder, equivalently top down parsing can be viewed as finding a left most derivation for an input string.

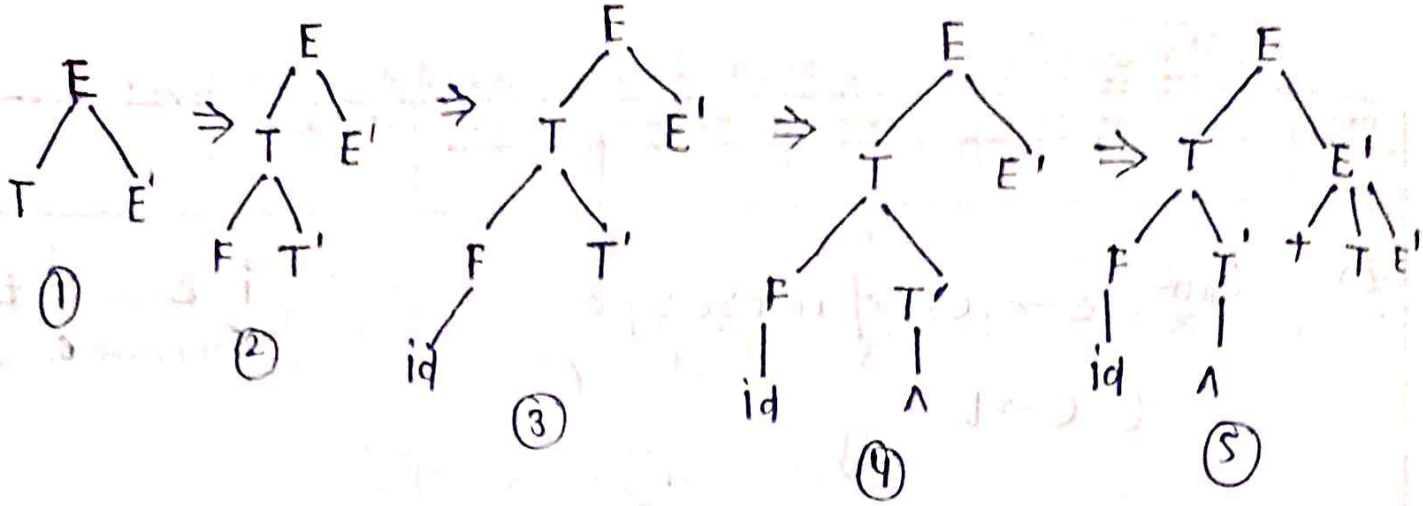
$E \to TE'$, $E' \to \pm TE' / \wedge$, $T \to FT'$, $T' \to *FT' / \wedge$

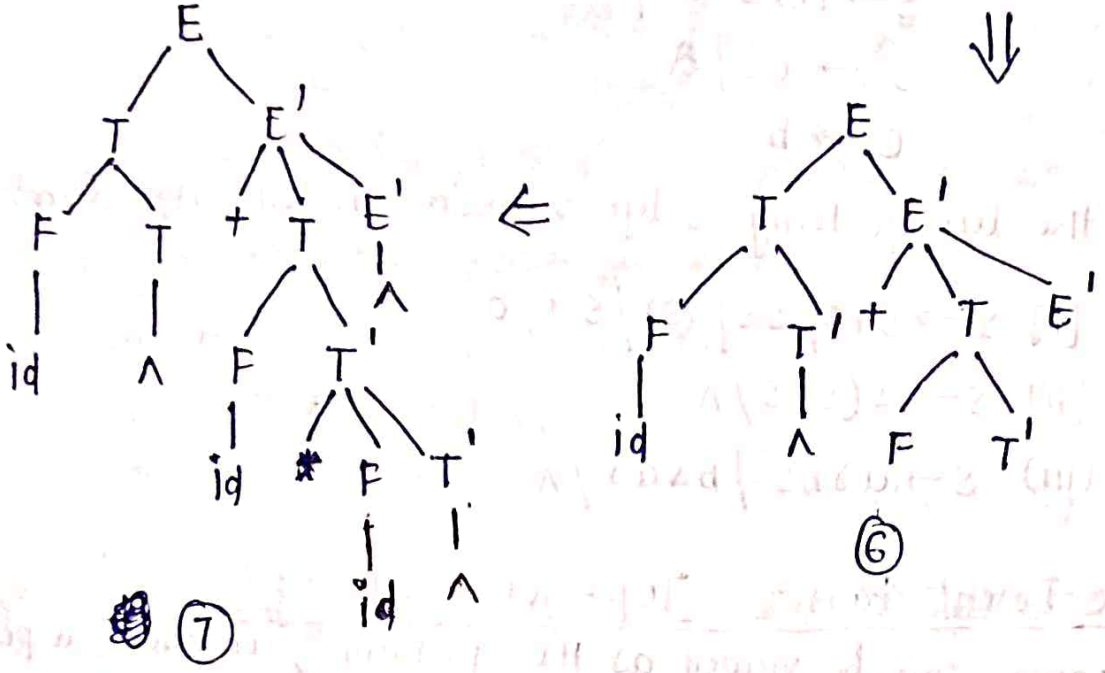
$F \to (E) / id$

[fwm = $id \pm id * id$]

Main Ideas, Questions & Summary:

Library / Website Ref.:-





Algorithm

```
void A()
① choose a A production A → X1 X2 ..... Xk
②     for (i ← 1 to k)
③         if ( Xi is a non-terminal )
④             call procedure Xi();
⑤         else
⑥             if (Xi Equals the convert I/P symbol a)
⑦                 advance the input to the next symbol
⑧             else /* an error has occured */
```

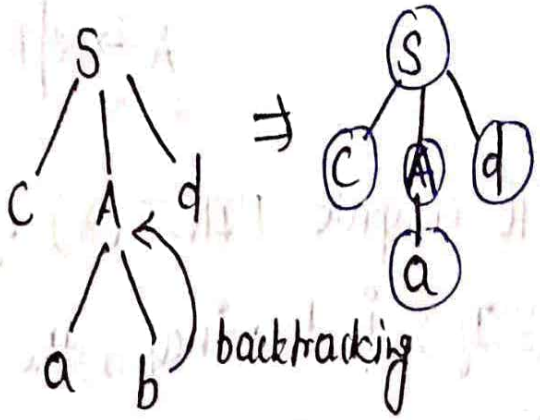
⇒ At Each step of a top-down parser, the key problem is that of determining the prediction to be applied for a non-terminal say A.

⊛ Backtracking

$S \rightarrow cAd$
$A \rightarrow ab/a$

$w = cad$

$S \rightarrow cAd$
$A \rightarrow ab/a$



This process is known as backtracking.
Backtracking is not efficient.

★ Predictive Parsing

⇒ A special case of Parsing when no backtracking are required. Predicting parsing chose the correct "A" production Looking ahead at the I/P of fixed. no. of symbol.

⇒ The class of grammer for which we can construct looking A symbol ahead put the I/P is sometimes called LL(k) grammer * for constructing predictive parser or LL(1) parser *.

⇒ For constructing predictive parser or LL(1) parser or some bottom up parser we have to compute the first and parser.

How to calculate FIRST & FOLLOW of all grammers

step (1): First (α) ⇒ where α is any string of any grammer to the set of terminal that begin string derived α.

$$A \rightarrow cY$$

$$FIRST(\alpha) = FIRST(cY) = c$$

for a previous how first can be used during

$A \rightarrow \alpha | \beta$ where First(α) at First(β) are disting set

→ To compute FIRST(X) of all Grammer, Applying the following rule.

① If X is terminal, the FIRST(X) = {X}.

Ex $A \rightarrow a$

FIRST(a) = a

② If X is a non terminal and given $X \rightarrow Y_1 Y_2 \cdots Y_k$

$$FIRST(X) = FIRST(Y_i)$$

③ If $X \rightarrow \wedge$ then add first(X). then FIRST(X) = ∧.

Ex: $E \rightarrow TE'$ $E' \rightarrow +TE' | \wedge$ $T \rightarrow FT'$, $T' \rightarrow *FT' | \wedge$

$F \rightarrow (E) | id$

Sol^n → FIRST(E) = FIRST(T) = FIRST(F) = { (, id }

FIRST(E') = { +, ∧ }

FIRST(T') = { *, ∧ }

POORNIMA

Date	Unit No.	Lecture No.	Faculty	Subject Name	Subject Code	Main Topics:-

⇒ For follow (A) to be set of terminals 'a' that can appear immediately to the right of a for some sentential form.

$$\alpha \Rightarrow \gamma$$

$$S^* \Rightarrow \alpha A a \beta$$

To compute follow (A)

→ If S is start symbol then include {'$'}

→ If their is a production $A \to \alpha B \beta$ then everything in first (B) accept NULL. Follow (B) ← First (B)

→ If there is a production $A \to \alpha B$ then Follow (B) ← Follow (A)

Ex.

Sol^n ⇒

Follow (E) = { $,) }

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

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

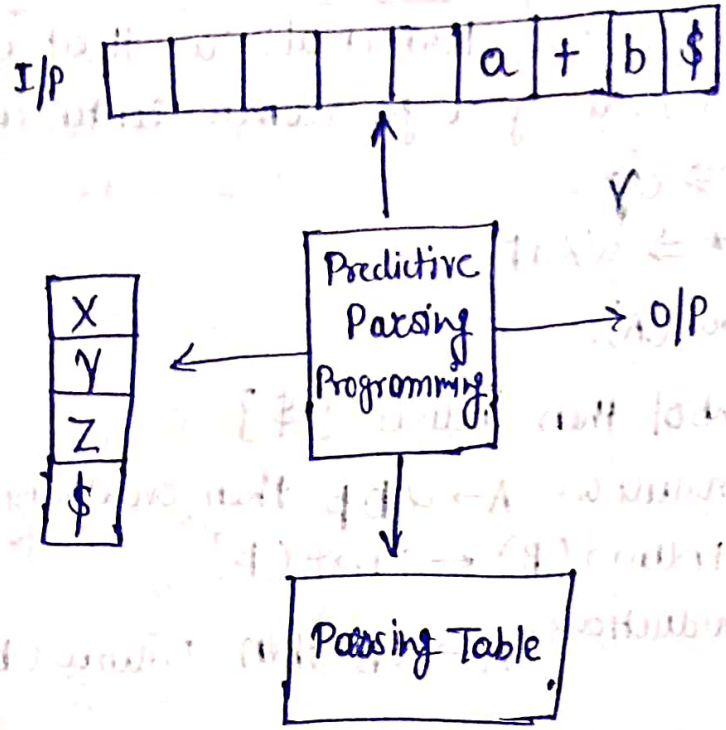
Follow (T') = { + ,) , $ }

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

Main Ideas, Questions & Summary:

Library / Website Ref.:-

Model of Predictive Parser



⊛ How to make a parsing table.?

Based on previous example.

I/P Symbol

I/P / Terminal	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \Lambda$	$E' \to \Lambda$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \Lambda$	$T' \to *FT'$		$T' \to \Lambda$	$T' \to \Lambda$
F	$F \to (E)$			$F \to (E)$		

POORNIMA

Date	Unit No.	Lecture No.	Faculty	Subject Name	Subject Code	Main Topics:-

① For each production $A \to \alpha$ of the grammer do the following
For each terminal A in FIRST(α), add $A \to \alpha$ to M[A, a].

② If $\wedge$ is in FIRST(α), then for each terminal b in FOLLOW(A) add $A \to \alpha$ to M[A, b]. If $\wedge$ is in FIRST(α) and \$ is in FOLLOW(A), add $A \to \alpha$ to M[A, \$] as well.

Ex. $S \to iCtSS' / a$
$S' \to eS / \wedge$
$C \to b$

Soln ⇒ FIRST(S) = {i, a}, FIRST(S') = {e, $\wedge$}
FIRST(C) = {b}
FOLLOW(S) = {\$, e, $\wedge$}

I/P / Terminal	i	t	a	b	e	\$
S	$S \to iCtSS'$		$S \to a$			
S'					$S \to eS$ $S \to \wedge$	$S \to \wedge$
C				$C \to b$		

Reduce-Reduce conflict

Main Ideas, Questions & Summary:

Library / Website Ref.:-

★ LL(1) Grammer

⇒ The first L in LL(1) grammer stands for scanning the I/P from left to right.

The second L for producing left most derivation & 1 stand for using the look ahead for no of I/P symbol to be predicted.

→ A Grammer is LL(1) iff whenever $A \to \alpha / \beta$

(1) For no terminal a to both α & β derive string beginning with a.
(2) At most 1 of α & β can derive the empty string.
(3) If $\beta \to \wedge$ the α doesn't derive any string beginning with the terminal in follow (a).

★ Bottom - up Parsing

⇒ A bottom-up ~~parser~~ path corresponds to the construction of the Reduction & Handle Pruning are the modulator, and working up towards the root (top).

(1) Reduction

⇒ At each 'reduction' step a 'specific' substring matching the body of a production is replaced by non-terminal at the head of that production.

(2) Handle

Pruning the children

⇒ Bottom-up parsing during ~~is~~ Pruning ^ construct a right most derivation in reverse can be obtained by handle Pruning.

POORNIMA

Date	Unit No.	Lecture No.	Faculty	Subject Name	Subject Code	Main Topics:-

$E \rightarrow E+T / T$

$T \rightarrow T*F / F$

$F \rightarrow (E) / id$

Right Sentential form	Handle	Reducing Production
$id * id_2$	$<id_1>$	$F \rightarrow id$
$F * id_2$	F	$T \rightarrow F$
$T * id_2$	id_2	$F \rightarrow id$
$T * F$	$T * F$	$T \rightarrow T * F$
T	T	$E \rightarrow T$
E	E	$

★ Shift Reduce Parser

⇒ It is the form of bottom up parsing in which stack hold grammer symbol and an input buffers holds the rest of the string to be parsed. Handle always appears at the top of the stack.

→ There are actually 4 possible actions.

① Shift ② Reduce ③ Accept ④ Error

[1] Shift the next I/P symbol into the top of the stack.

[2] Reduce : The right end of the string could be reduce must be add the top of the stack.

[3] Accept : Announce successful completion of parsing.

[4] Error : Discover a syntax error & call an error recovery routine.

Main Ideas, Questions & Summary:

Library / Website Ref.:-

There is conflict b/w when to shift or at when to reduce. This is known as "shift - reduce conflict".

So, there are LR Parser (simple LR) i.e SLR

Stack	Input	Action
$	$id_1 * id_2 $	shift
$id_1	* id_2 $	reduce F→id
$F	* id_2 $	reduce T→F
$T	* id_2 $	shift
$T*	id_2 $	shift
$T*id_2	$	reduce F→id
$T*F	$	reduce T→*F
$T	$	reduce E→T
$E	$	accept

(*) The most prevalent type of bottom up parser today is based on a concept called LR(K) parsing

'L' → for left to Right

'R' → for right most derived , K=0, K=1

★ Introduction to LR Parsing : Simple LR

⇒ The most prevalent type of bottom-up parser today is based on a concept called LR(K) parsing. the "L" is for left to right scanning of the input, the "R" for constructing a rightmost derivation in reverse, and the k for the number of input symbols of lookahead that are used in making parsing decisions.

(*) WHY LR Parsers ?

→ LR parsers can be constructed to recognize virtually all programming language constructs for which context-free grammars can be written.

POORNIMA

Date	Unit No.	Lecture No.	Faculty	Subject Name	Subject Code	Main Topics:-

→ The LR-parsing method is the most general nonbacktracking shift reduce parsing method known.

→ The LR parser can be detect a syntactic error as soon as it is possible to do so on a left to right scan of the input.

→ The class of grammers that can be parsed using LR methods is a proper superset of the class of grammers that can be parsed with prediction or LL methods.

The principal drawback of the LR method is that it is too much work to construct an LR parser by hand for a typical programming

★ LR and item(0) Automation

In LR(0) item for a grammer G is a production of G with a dot at some position of the body. Thus production

$A \rightarrow XYZ$

Produces 4 items.

$A \rightarrow \cdot XYZ$
$A \rightarrow X \cdot YZ$
$A \rightarrow XY \cdot Z$
$A \rightarrow XYZ \cdot$

$A \rightarrow \epsilon$
⇓
$A \rightarrow \cdot$

$A \rightarrow X \cdot YZ$ indicates that we have just seen on the input a string derivable item X and that we hope next to see a string derivable from YZ. Item $A \rightarrow XYZ \cdot$ indicates that we have seen the body XYZ and that it may be time to reduce XYZ to A.

Main Ideas, Questions & Summary:

Library / Website Ref.:-

* Augmented Grammer

⇒ If G is a grammer with start symbol S, then G' be the Augmented Grammer with start symbol will be S'.

$$S' \longrightarrow S$$

For creating LR(0) items we have to find

① Closure of the item
② Apply goto function

* closure of the item

⇒ If i is a set of items for a grammer G then closure i is a set of items constructed by following two rules.

① Initially Add every item I to closure (I).
② if there is a production $A \to \alpha B \beta$ is the closure of (I) and $B \to \gamma$ is a production. then Add the item $B \to .\gamma$ to closure i. Apply this rule until no more items can be added to closure i

* goto function

GOTO (I, X)

① If $A \to \alpha . X \beta$ in i, then call GOTO (I, X) in $A \to \alpha X . \beta$ is in I

Q. $E \to E + T \mid T$
$T \to T * F \mid F$
$F \to (E) \mid id$

POORNIMA

Date	Unit No.	Lecture No.	Faculty	Subject Name	Subject Code	Main Topics:-

Augmented Grammer

$E' \rightarrow E$

$E \rightarrow E+T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Closure of the item

$I \Rightarrow \{ E' \rightarrow \cdot E \}$

$I_0 : \begin{cases} E' \rightarrow \cdot E \\ E \rightarrow \cdot E+T \mid \cdot T \\ T \rightarrow \cdot T * F \mid \cdot F \\ F \rightarrow \cdot (E) \mid id \end{cases}$

$I_1 = GOTO(I_0, E)$

$E' \rightarrow E \cdot$

$E \rightarrow E \cdot + T$

$I_2 = GOTO(I_0, T)$

$E \rightarrow T \cdot$

$T \rightarrow T \cdot * F$

$I_3 = GOTO(I_0, F)$

$T \rightarrow F \cdot$

$I_4 : GOTO(I_0, (\ ))$

$F \rightarrow (\cdot E)$

$E \rightarrow \cdot E+T$

$E \rightarrow \cdot T$

$T \rightarrow \cdot T * F$

$T \rightarrow F \cdot$

$F \rightarrow \cdot (E) \mid \cdot id$

Main Ideas, Questions & Summary:

Library / Website Ref.:-

I_5
$F \to id \cdot$

I_6
$E \to E + \cdot T$
$T \to T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot id$

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

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

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

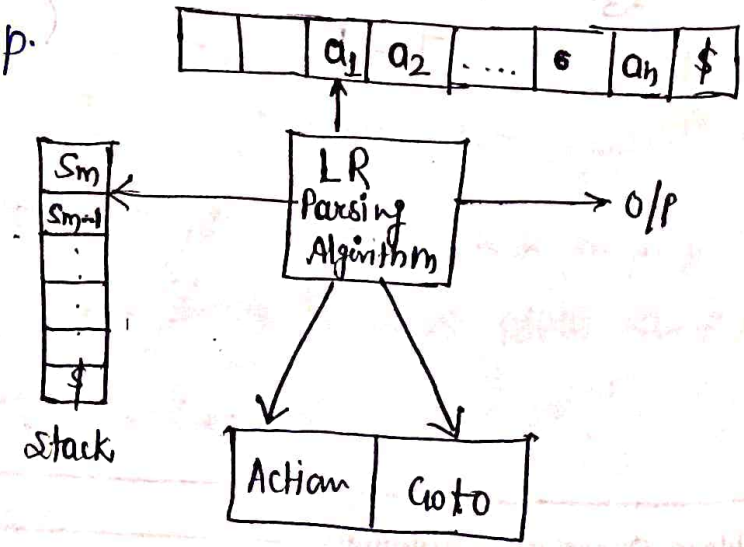
I_{10}
$T \to T * F \cdot$

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

★ The LR Parsing Algorithm

⇒ The schematic of an LR parser is shown below. It consists of an input, an output, a stack, a driver program, and a parsing table that has two parts (ACTION & GOTO). The driver program is the same for all LR parsers; only the parsing table changes from one parser to another

The stack holds a sequence of states $S_0 S_1 \ldots S_m$ where S_m is on top.
In this SLR method.
The stack holds states from the LR(0) automaton.
The canonical LR & LALR methods are similar.



Model of an LR parser

POORNIMA

Date	Unit No.	Lecture No.	Faculty	Subject Name	Subject Code	Main Topics:-

Construction of LR parsing table

1. Action() takes as arguments a state i and a terminal a
   ACTION[i, a]

   a) S_j (shift state S_j)
   b) If $A \to \beta \cdot$ (reduce)
   c) accept : the parser accept the i/p and finishes parsing.
   d) Error

2. If GOTO $(I_i, A] = I_j$ then go to maps a state i & and a nonterminal A to state j.

Method: ① Construct $C = \{I_0, I_1, \ldots I_n\}$

② a) If $A \to \alpha \cdot a\beta$ and Goto $(I_i, a) = I_j$ then set ACTION[i, a] to S_j

② b) If $A \to \alpha \cdot$ is in I_i then set ACTION[i, a] to $A \to \alpha$ when a is in FOLLOW(A)

c) If $S \to S \cdot$ is in I_i, then set ACTION(i, $) is accept.

Ex.

γ_1 : $E \to E+T$
γ_2 : $E \to T$
γ_3 : $T \to T*F$
γ_4 : $T \to F$
γ_5 : $F \to (E)$
γ_6 : $F \to id$

state	id	+	*	(	)	$	E	T	F
I_0	S_5			S_4			1	2	3
I_1		S_6				accept			
I_2		γ_2 ($E \to T$)	S_7		γ_2 ($E \to T$)	γ_2			
I_3		γ_4	γ_4		γ_4	γ_4			
I_4	S_5			S_4			8	2	3
I_5	γ_6	γ_6			γ_6	γ_6			

Main Ideas, Questions & Summary:

Library / Website Ref.:-

state	id	+	*	(	)	$	E	T	F
I_6	S_5			S_4				9	3
I_7	S_5			S_4				•	10
I_8		S_6			S_{11}				
I_9		γ_1	S_7		γ_1	γ_1			
I_{10}		γ_3	γ_3		γ_3	γ_3			
I_{11}		γ_5	γ_5		γ_5	γ_5			

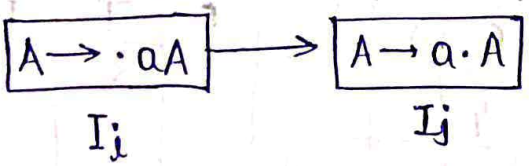
★ Constructing SLR-parsing Tables : SLR(1)

- Ⓐ SLR(1) refers to simple LR Parsing.
- Ⓐ Same as LR(0) Parsing. The only diff is in the parsing table.
- Ⓐ To construct SLR table, use canonical of LR(0) item.
- Ⓐ In SLR table, place the reduce move the follow of left hand side.

• SLR(1) table

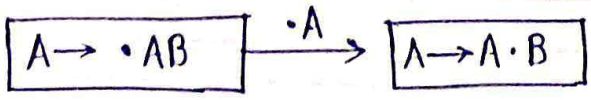
Ⓐ If a state I_i is going to some other state on a terminal then if corresponding to a move in the active part.

Ex:



States	Action		GO
	a	$	
I_i	S_j		
I_j			

Ⓐ If a state I_i is going to some other state on a variable (Non-terminal) then if corresponding to go to move in the GOTO panel.



States	ACTION	GO
	$	
I_i		
I_j		

POORNIMA

Date	Unit No.	Lecture No.	Faculty	Subject Name	Subject Code	Main Topics:-

If a state (I_i) contains the final state (item) like $A \rightarrow ab\cdot$ which has no transitions to the next state then the production is known as reduce production for all terminals X in FOLLOW(A). Write the reduce entry along with their production numbers.

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

step-①: Add Augment production & insert '·' symbol at the first position for every productions in G.

$S' \rightarrow \cdot E$
$E \rightarrow \cdot E+T$
$E \rightarrow \cdot T$
$T \rightarrow \cdot T*F$
$T \rightarrow \cdot F$
$F \rightarrow \cdot id$

step-②: DFA ⇒ Draw the DFA same as LR(0)

step-③: Find FIRST & FOLLOW of respective Non-terminals.

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

Main Ideas, Questions & Summary:

Library / Website Ref.:-

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

Step-④: For accept

I_1 Contains the final item which done $S \rightarrow E \cdot$

and FOLLOW (S) = {$}

So action {I_1, $} = Accept

Step-⑤: Apply GOTO part and shift & action part.

(Both are same as LR(0))

Step-⑥: For reduce in ACTION part

* States I_2, I_3, I_4, I_7 & I_8 contain

* Define a numbers to the productions

$E \rightarrow E+T$ —①

$E \rightarrow T$ —②

$T \rightarrow T*F$ —③

$T \rightarrow F$ —④

$F \rightarrow id$ —⑤

States	Action				Go to		
	id	+	*	$	E	T	F
I_0	S_4				1	2	3
I_1		S_5		Accept			
I_2		R_2	S_6	R_2			
I_3		R_4	R_4	R_4			
I_4		R_5	R_5	R_5			
I_5	S_4					7	3
I_6	S_4						8
I_7		R_1	S_6	R_1			
I_8		R_3	R_3	R_3			

★ CLR Parsing

CLR Reduce to canonical Lookahead.

CLR parsing use the canonical collection of LR(1) items to build the CLR(1) Parsing table.

CLR(1) Parsing produce the more number of states as compare to SLR(1)

We Here the reduce node only in the lookahead symbols.

LR(1) item is a collection of LR(0) items and a lookahead symbol

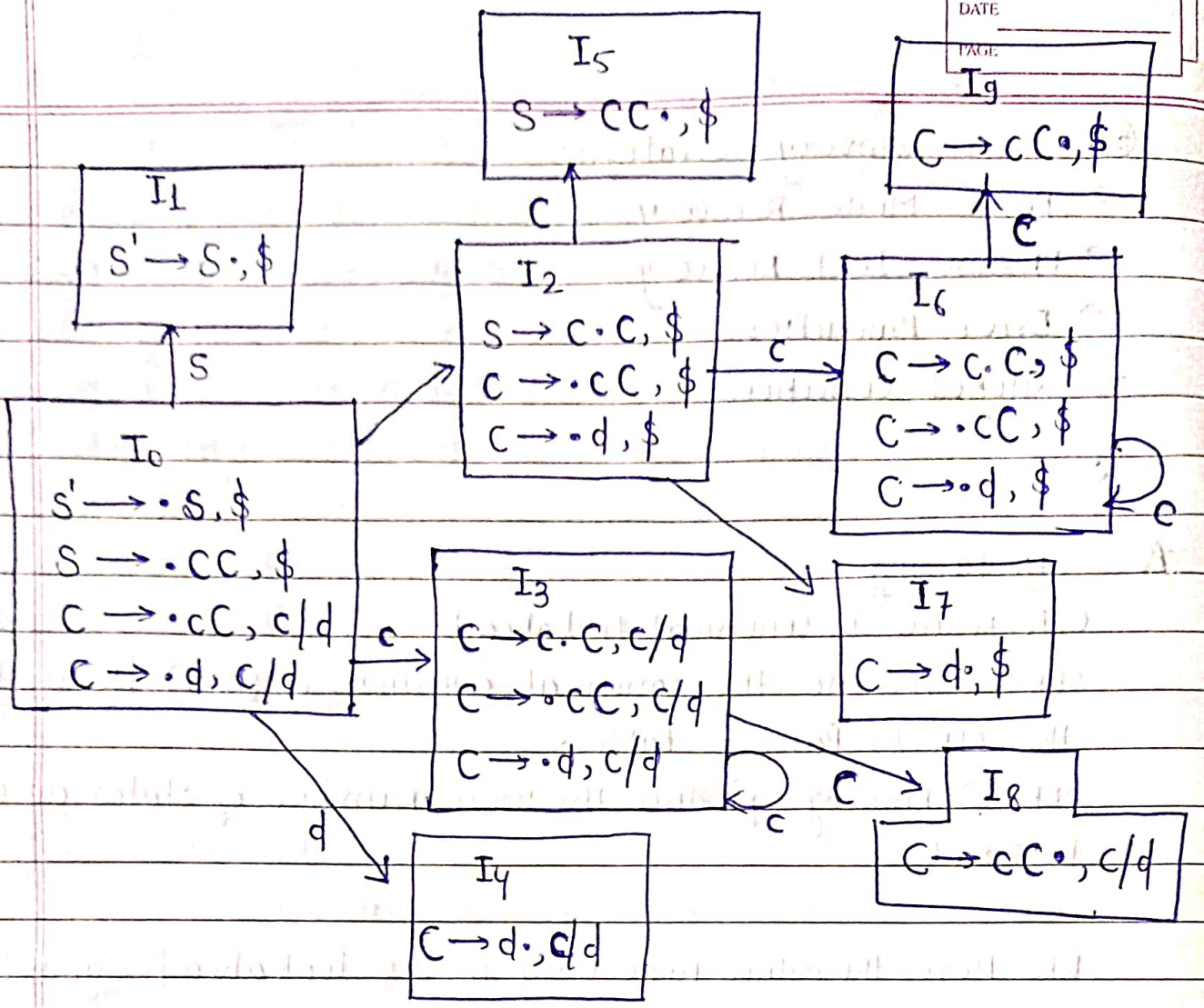
LR(1) item = LR(0) item + lookahead

The look ahead is used to determine that where we place the find item. the look ahead always add $ symbol for the argument production.

Ex. $S \rightarrow CC$

$C \rightarrow cC/d$

step (1): Add Argument production, insert at the first position for every production and also lookahead.



The GOTO graph for grammer

step-③ : Construct CLR Table

① Apply shift and accept of Action part

② Apply goto part

Note : Both are same as SLR and LR(0)

③ For Reduce

I_4 contain's the final item which drives

$A \rightarrow d\cdot, c/d$

So, action $\{I_4, c\} = \{I_4, d\} = R_3$

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

I_5 contains final item

$S \rightarrow CC\cdot, \$$

So, action $(I_5, \$) = R_1$

I_7 contains final item

$S \rightarrow d\cdot, \$$, so action $(I_7, \$) = R_3$

I_8 contains final item

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

So action $\{I_8, a\} = \{I_8, b\} = R_2$

I_9 contains the final item

$A \rightarrow aA\cdot \$$

$\{I_9, \$\} = R_2$

	ACTION			GOTO	
states	c	d	$	S	C
I_0	S3	S4		1	2
I_1			Accept		
I_2	S6	S7			5
I_3	S3	S4			8
I_4	R3	R3			
I_5			R1		
I_6	S6	S7			9
I_7			R3		
I_8	R2	R2			
I_9			R2		

LALR (1)

- LALR refers to the lookahead LR
- In LALR(1) Parsing, the LR(1) items which have same productions but different lookahead are combined to form a single set of items.
- LALR(1) Parsing is same as the CLR(1) parsing, only difference in the parsing table.

LALR states

$I_0 \rightarrow$

$S' \rightarrow .S, \$$
$S \rightarrow .AA, \$$
$A \rightarrow .aA, a/b$
$A \rightarrow .b, a/b$

$I_1 \rightarrow$ Goto (I_0, S)

$S' \rightarrow S\cdot, \$$

$I_2 \rightarrow$ Goto (I_0, A)

~~$S' \rightarrow a\cdot A$~~ $S \rightarrow A\cdot A, \$$

$A \rightarrow \cdot aA, \$$

$A \rightarrow \cdot b, \$$

$I_3 \rightarrow$ Goto (I_0, a)

$A \rightarrow a\cdot A, a/b$

$A \rightarrow \cdot aA, a/b$

$A \rightarrow \cdot b, a/b$

$I_4 \rightarrow$ Goto (I_0, b)

$A \rightarrow b\cdot, a/b$

$I_5 \rightarrow$ Goto (I_2, A)

$S \rightarrow AA\cdot, \$$

$I_6 \rightarrow$ Goto (I_2, a)

$A \rightarrow a\cdot A, \$$

$A \rightarrow \cdot aA, \$$

$A \rightarrow \cdot b, \$$

$I_7 \rightarrow$ Goto (I_2, b)

$A \rightarrow b\cdot, \$$

$I_8 \rightarrow$ Goto (I_3, A)

$A \rightarrow aA\cdot, a/b$

$I_9 \rightarrow$ Goto (I_6, A)

$A \rightarrow aA\cdot \$$

Now. State I3 and I6 are same in their LR(0) but differ in their lookahead, so we can combine them and called as I36

I36 =>

$A \to a \cdot A, a/b/\$$

$A \to \cdot aA, a/b/\$$

$A \to \cdot b, a/b/\$$

State I4 and I7 are same in their LR(0) but differ in their lookahead, so we can combine them and called as I47.

I47 →

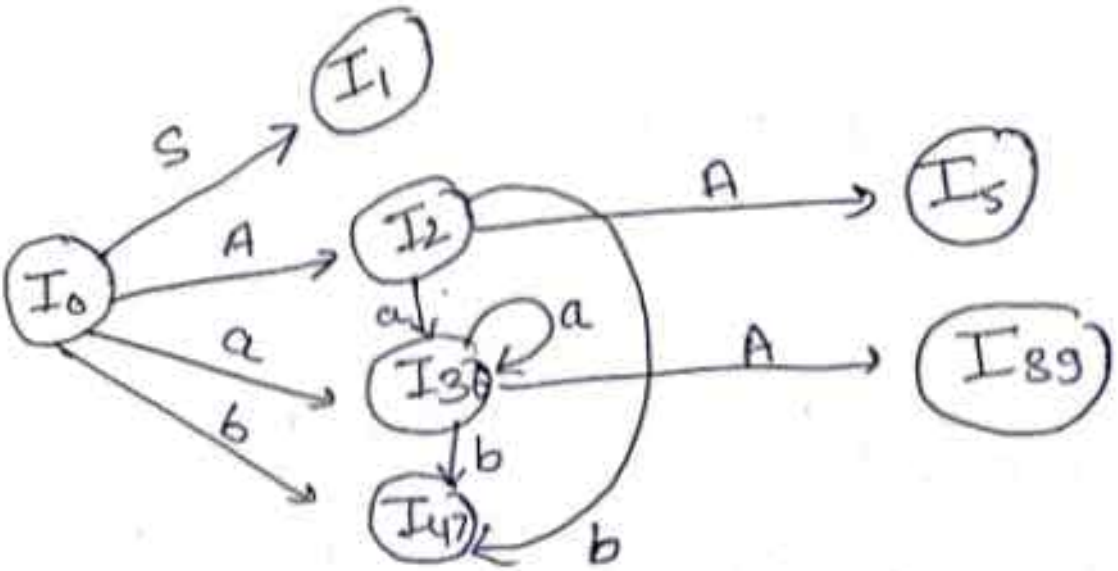
$A \to b \cdot, a/b/\$$

State I8 and I9 are same in their LR(0) but differ in their lookahead, so we can combine them and called as I89.

I89 →

$A \to aA \cdot \; a/b/\$$

DFA



LALR(1) Parsing table :

States	Action a	Action b	Action $	Goto S	Goto A
I0	S36	S47		1	2
I1			accept		
I2	S36	S47			5
I36	S36	S47			89
I47	R3	R3	R3		
I5			R1		
I89	R2	R2	R2		

Construction of Predictive parsing table:

① I/p is grammar G.

② O/p is parsing table M

Method: For each production $A \to \alpha$ of the grammar, do the following:

1. For each terminal a in FIRST(α), add $A \to \alpha$ to M[A, a].
2. If ε is in FIRST(α), then for each terminal b in FOLLOW(A), add $A \to \alpha$ to M[A, b]. If ε is in FIRST(α) and $ is in FOLLOW(A), add $A \to \alpha$ to M[A, $] as well.

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

Unit-03 : Syntax directed definitions

⊛ Construction of Syntax trees

→ We shall implement the nodes of a syntax tree by objects with a suitable number of fields. Each object will have one op field that is the label of the node.

→ If the ~~node~~ node is a leaf, an additional field holds the lexical value for the leaf. A constructor function Leaf(op, val) creates a leaf objects. Alternatively, if nodes are viewed as records, then Leaf returns a pointer to a new record for a leaf.

→ If the node is an interior node. there are many additional fields as the node has children in the syntax tree. A constructor function Node takes two or more arguments Node(op, $c_1, c_2, \ldots, c_k$) creates an object with first field op and k additional fields for the k children $c_1, \ldots, c_k$.

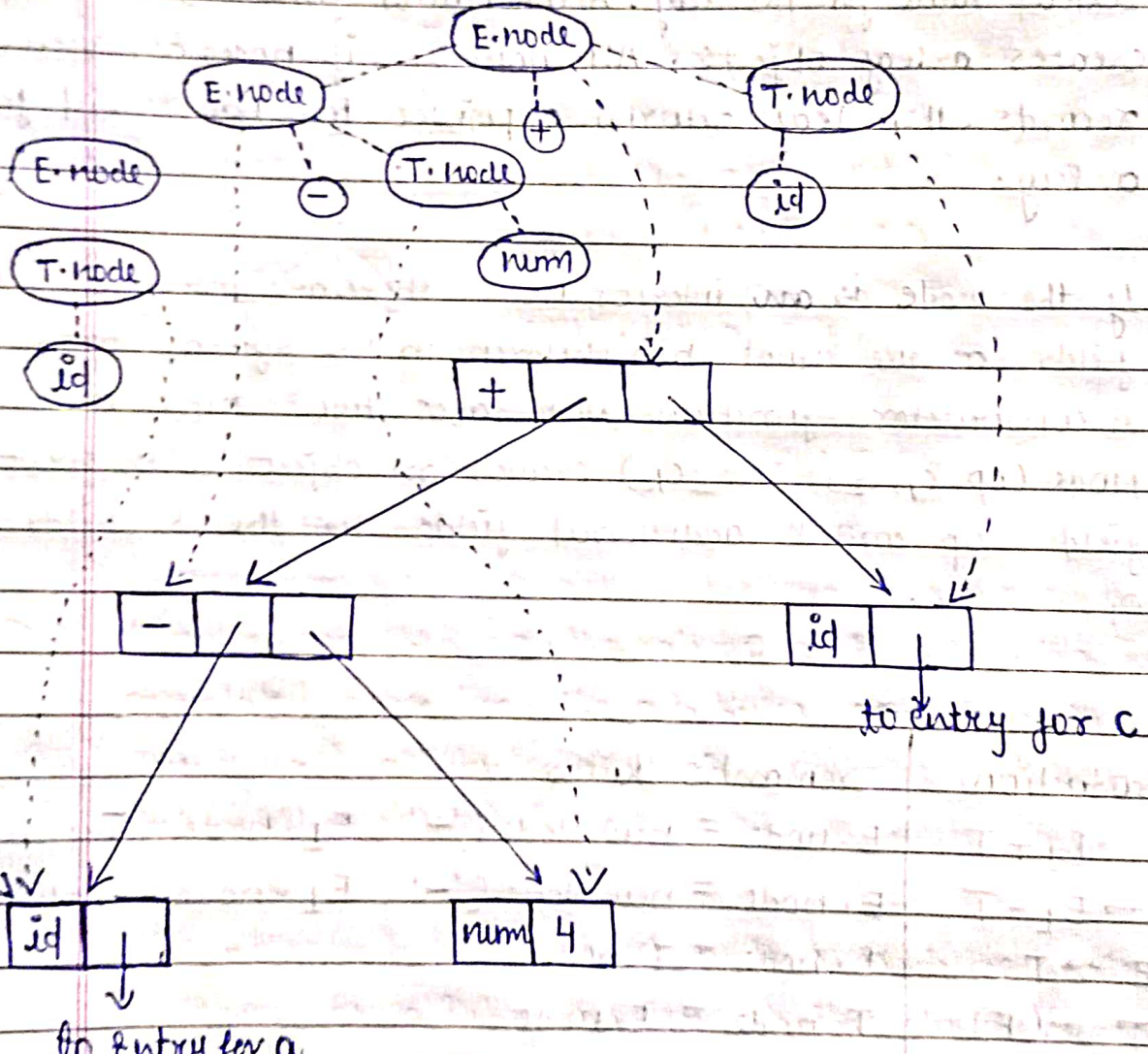
	Production	Semantic Rules
①	$E \rightarrow E_1 + T$	E.node = new Node('+', E_1.node, T.node)
②	$E \rightarrow E_1 - T$	E.node = new Node('-', E_1.node, T.node)
③	$E \rightarrow T$	E.node = T.node
④	$T \rightarrow (E)$	T.node = E.node
⑤	$T \rightarrow id$	T.node = new Leaf(id, id.entry)
⑥	$T \rightarrow num$	T.node = new Leaf(num, num.val)

constructing syntax trees for simple expression

Ex: construct syntax tree for a - 4 + c

→ steps for constructing syntax tree

① P_1 = new Leaf (id, entry - a);
② P_2 = new Leaf (num, 4);
③ P_3 = new Node ('-', P_1, P_2);
④ P_4 = new Leaf (id, entry - c);
⑤ P_5 = new Node ('+', P_3, P_4);



- **S-Attributed Definitions**

Syntax directed definitions (SDD) is a context free grammer together with attributes & rules.
The first class of SDD is S-attributed definitions.
An SDD is S-attributed if every attribute is synthesized.
S-attributed definitions can be implemented during bottom up parsing, since a bottom-up parse corresponds to a postorder traversals.

- **L-Attributed Definitions**

⇒ The second class of SDD is called L-attributed definitions. The idea behind this class is that, b/w the attributes associated with a production body, dependency-graph edges can go from left to right but not right to left.

Each attribute must be:

① Synthesized
② Inherited
→ Inherited attributes associated with the head A
→ Either inherited or synthesized attributes associated with the occurences of symbol $X_1, X_2, X_3, \ldots X_{i-1}$.

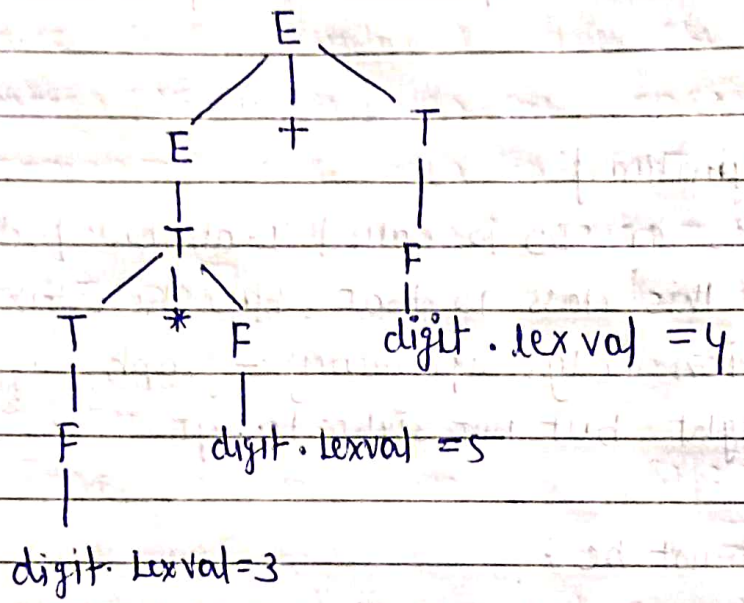
→ Inherited or synthesized attributes associated with this occurrences of X_i

✱ Implementation of SDT

⇒ SDT is implemented by constructing a parse tree and performing the action in left to right depth first order (DFS).

construct parse tree for the input string

3 * 5 + 4



A parse tree showing the values of its attributes is annotated parse tree. the attributes can be of two types

① Synthesized attribute
② Inherited attributes

A synthesized attributes at node N is defined only in terms of attributes value at the children of N and at N itself.

An inherited attribute at node N is designed only in terms of attributes value, at N's parent, and itself, N's sibling.

★ S. Attributed SDT

⇒ If every attribute is synthesized then an SDT is called S. attributed SDT.

The attribute is synthesized if the value of parent node depends upon the value of child node.

The S-attributed SDT is evaluating in bottom up parsing. The ~~weight~~ have most space of RHS holds the semantic action.

★ Directed Acyclic Graph (DAG)

⇒ DAG is used to represent the structure of basic block helps to see flow of values and offers optimization tool.

DAG provides easy transformation on basic blocks. The curve of DAG are labelled by unique identifier and that identifier can be variable, names or constants.

Interior node of the graph is Labelled by an operator symbol.

Nodes are also gives a sequence of identifiers for labels to store the computed value.

DAG's are our a type of Data structure, it's used to implement transformation on basic blocks.

→ DAG provides a good way to determine the common sub supervision

TAC (Three address code)

x := y operator z

Algo for construction of TAC

→ Apply TAC method for an expression.

① $x := y \text{ op } z$

② $x := \text{op } y$

③ $x := y$

eg: $si = (p * r * t)/100$

$t_1 = p * r$

$t_2 = t_1 * t$

$t_3 = t_2/100$

$si := t_3$

→ if operand y and z are not defined, create node (y) create node (z)

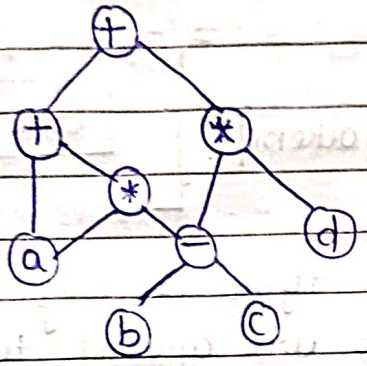
→ if case (1) : then create node (operator), whose right child is node (z) & left child is node (y)

→ In case (2)
check whether there is node (operator) with child node (y).

Case - 3

$x = y$, node(x) will be node(y)

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



$t_1 = b - c$

$t_2 = a * t_1$

$t_3 = a + t_2$

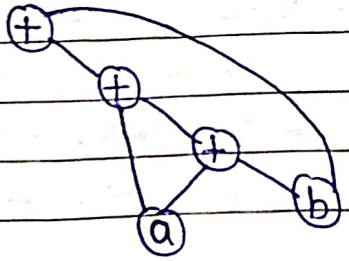
$t_4 = t_1 * d$

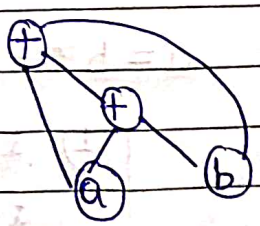
$t_5 = t_3 + t_5$

⊛ SDD (Syntax directed definition) to produce DAG

Production	Semantic Rule
$E \rightarrow E + T$	E.node = new Node ('+' E.node, T.node)
$E \rightarrow E - T$	E.node = new Node ('−', E.node, T.node)
$T \rightarrow (E)$	E.node = T.node
$T \rightarrow$ num	T.node = E.node
	T.node = new leaf (num, num.lexval)

Ex. ① $(a+b) + (a+b)$ ② $a+b+a+b$



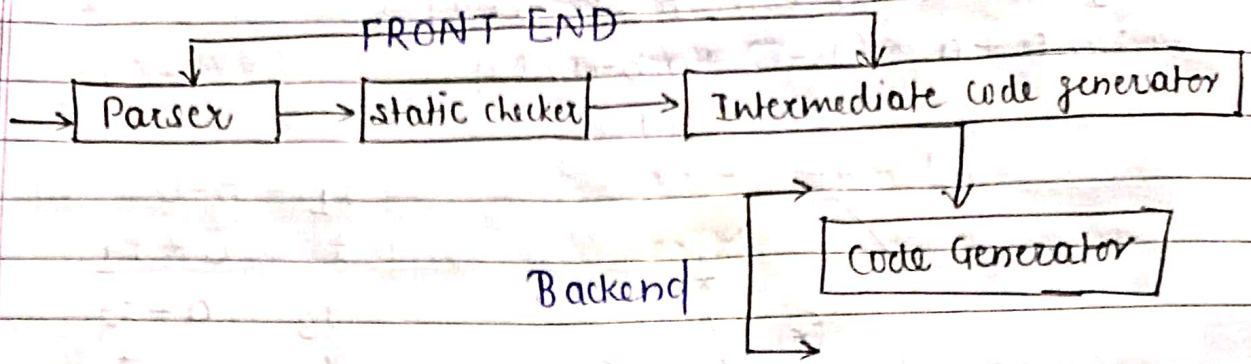


③ $a+a+(a+a+a+(a+a+a+a))$

④ $(a/b) + ((a/b) * (*d))$

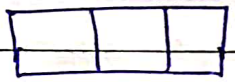
} try it ?

☆ Intermediate code Generator

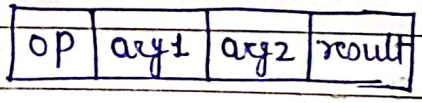


static checking includes type checking which ensures that operators are applied to compatible operands. It also includes any static checks that remain after parsing. These address code is used for generating intermediate code. & general form is $x = y \text{ op } z$.

Storage is in form



General form → ~~tree~~ triples



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

Triples
1) $t_1 = -c$
2) $t_2 = b * t_1$
3) $t_3 = -c$
4) $t_4 = b * t_3$
5) $t_5 = t_2 + t_4$
6) $a = t_5$

Quadruples

op	arg1	arg2	result
−	c		t_1
*	b	t_1	t_2
−	c		t_3
*	b	t_3	t_4
+	t_2	t_4	t_5
=	t_5		a

★ TAC for various control flow structure

① If (a<b) then
x++;

(TAC)
① if (a<b) goto 3
② goto 5
③ $t_1 = x+1$
④ $x := t_1$
⑤ exit

② If (a<b) then
x++
else
x--

(TAC)
① if (a<b) goto 3
② goto 5
③ $t_1 := x+1$
④ $x := t_1$, goto 7
⑤ $t_2 := x-1$
⑥ $x := t_2$, goto 7
⑦ exit

③ while (A<C)
do
A = A+B

(TAC)
① if (A<C) goto 3
② goto 5
③ $t_1 := A+B$
④ $A := t_1$ goto 1
⑤ exit

Exercise

[1] for (i=1; i<=10; i++)
do
x := x+10

→ Write the Tree address code (TAC) of the Expression.

Sol^n ⇒
1) i=1;
2) if (i<=10) goto 4
3) goto 8
4) $t_1 = x+10$
5) $x = t_1$
6) $t_2 = i+1$
7) $i = t_2$ goto 2
8) Exit

[2] c=0
do{
c++;
} while (c<10);

Sol^n ⇒ 1) c=0
2) $t_1 = c+1$
3) $c = t_1$
4) if (c<10) goto 2
5) Exit

[3] switch (ch){
case 1 : a++;
break;
case 2 : b++;
break;
}

Sol^n ⇒ ① if (ch==1) goto
② if (ch==2) goto
③ $t_1 = a+1$
④ $a = t_1$ goto 7
⑤ $t_2 = b+1$
⑥ $b = t_2$ goto 7
⑦ Exit

POORNIMA

Date	Unit No.	Lecture No.	Faculty	Subject Name	Subject Code	Main Topics:-

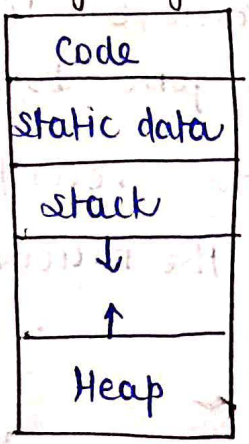
* Unit - 04 : Storage Organisation *

⇒ The executing target program run in its logical address space in which each program ~~value~~ has value. The management & organization of this logical space is shared with compiler, operating systems and target machines.

The OS match the logical add. to physical add. which are usually spread through memory.

The run time representation of an object in the logical address consist of data & program. Typical sub division of run time into code and data are as shown below.

Storage Organisation



(*) Strategies

→ Static Allocation : It is for the all data objects for compile time. the size of data objects is known for compiler time & the binding the data object name and amount are do not change at run time.

At compile time compiler can fill the the data its operate on.

Main Ideas, Questions & Summary:

Library / Website Ref.:-

Limitations

The static allocation can be done only if the size of data object is known as compile time. Recursive procedure are not supported

[2] Main Needs to memory wastage.

Advantage:

→ It leads to faster execution process

→ It provide the more efficiency.

To maximize of utilization of at run time, other two area stack, and heap are at the opposite are the opposite ends of remainder of the address space.

These areas are dynamic and their size can change as the program execute. In practice, the stack gone to lower adderss and heap gone higher address.

An activation record is used to storage ~~ab~~ info. above, the Status of the machine.

When control return from the calls the activation of the calling procedure, can be restructured, after restarting the values of relavant registers an setting program counter to the parent immediate after the call.

Many programing allows, program to allows the program to allocate and deal data under program ended.

POORNIMA

Date	Unit No.	Lecture No.	Faculty	Subject Name	Subject Code	Main Topics:-

★ Static V/S Dynamic Allocation

⇒ The two adjective static & dynamic distinguisse b/w compile time & run time respectively. We say that storage allocation is static, if it can be made by compiler looking text of the program.

Not at what, program, the program does when is execute, conversely decision is dynamic, only while the program is running.

⇒ Many compiler use some combination, for following two strategies for dynamic store allocation.

① Stack storage : Names local to a procedure are allocated space on a stack. ~~The Name to~~ the stack support normal called return policy.

② Heap storage: Heap is an area of virtual memory & that allows objects & data element to obtained storage, and data element ant to return that storage when they are validated.

To support heap management garbage collection
to detect useless data element & Reuse there storage,

Main Ideas, Questions & Summary:

Library / Website Ref.:-

→ All most all compilers for language that use procedure, functions or methods, as unit of user defined action manage atleast part of their runtime memory as a stack.

→ The stack allocation of the spaces are managed by two way.

(A) Activation Tree
(B) Activation Record

[A] Activation Tree : stack allocation would not be fisible if procedure calls and activations of procedure does not nest in time.

Taking the Ex. of quicksort

```
Void Quicksort (arr, int p, int r){
        q ← PARTITION (A, P, r)
        Quicksort (A, P, q-1)
        Quicksort (A, q+1, r)
```

we there for can represent the activation of the procedure running entire program by tree & each node corresponds to 1 activation & root is # activation, such tree is called activation tree.

```
int a[11];
void readArray() { /* Reads 9 integers into a[1], ..., a[9]. */
    int i;
    ...
}
int partition(int m, int n) {
    /* Picks a separator value v, and partitions a[m..n] so that
       a[m..p-1] are less than v, a[p] = v, and a[p+1..n] are
       equal to or greater than v. Returns p. */
    ...
}
void quicksort(int m, int n) {
    int i;
    if (n > m) {
        i = partition(m, n);
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
}
main() {
    readArray();
    a[0] = -9999;
    a[10] = 9999;
    quicksort(1,9);
}
```

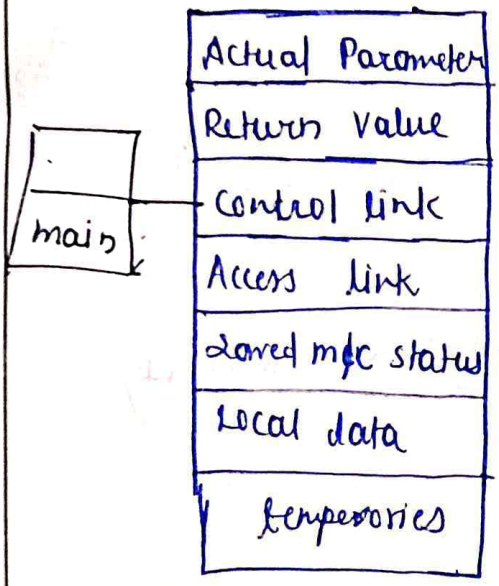
Figure 7.2: Sketch of a quicksort program

POORNIMA

Date	Unit No.	Lecture No.	Faculty	Subject Name	Subject Code	Main Topics:-

★ Activation Record

⇒ The procedure calls and returns are usually managed by run time stack called control stack. each live activation has an activation record (some time called as frame) on the control stack with the roots of activation tree at the bottom, and the entire sequence of activation records on the stack corresponding to the path in the activation tree to the activation where control currently resits.



(1) temporary

Temporary value such as those arrising from the evaluation of expressions in case where those temporary.

Local data belonging to the procedure whose activation record this is.

A saved machine

about the state of machine just before the call to the procedure. This info includes return address. Value of the program counter. To which the call proce

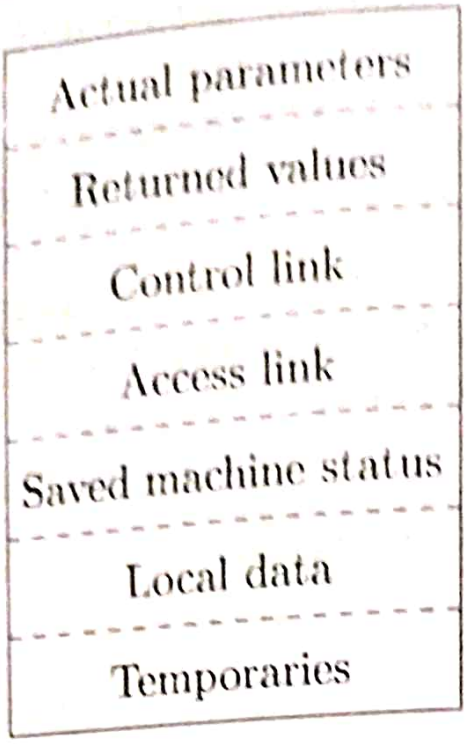


Figure 7.5: A general activation record

1. Temporary values, such as those arising from the evaluation of expressions, in cases where those temporaries cannot be held in registers.

2. Local data belonging to the procedure whose activation record this is.

3. A saved machine status, with information about the state of the machine just before the call to the procedure. This information typically includes the *return address* (value of the program counter, to which the called procedure must return) and the contents of registers that were used by the calling procedure and that must be restored when the return occurs.

4. An "access link" may be needed to locate data needed by the called procedure but found elsewhere, e.g., in another activation record. Access links are discussed in Section 7.3.5.

5. A *control link*, pointing to the activation record of the caller.

6. Space for the return value of the called function, if any. Again, not all called procedures return a value, and if one does, we may prefer to place that value in a register for efficiency.

7. The actual parameters used by the calling procedure. Commonly, these values are not placed in the activation record but rather in registers, when possible, for greater efficiency. However, we show a space for them to be completely general.

(*) Symbol table Organisation

⇒ Symbol table is an important data structure used in compiler.

→ Symbol table is used to store the information about the occurrance of various entities such as object, classes, variable name etc.

→ The symbol table used for following purpose.

(1) It is used to store the name of all entities in a structured form at one place.
(2) It is used to verify if a variable has been declared.
(3) It is used to determine the scope of the name.
(4) It's verify assignment & expression in source code are semantically correct.

< symbol name, type, attribute >
static int salary
< salary, int, static >

- Implementations

(1) Linear (sorted / unsorted list)
(2) Hash table
(3) Binary search tree

- operations :

(1) Insert() : insert (x, int);
(2) lookup() : lookup (symbol);
(3)

POORNIMA

Date	Unit No.	Lecture No.	Faculty	Subject Name	Subject Code	Main Topics:-

Unit-05 : Code Optimization

⇒ Optimization is a program transformation technique which tries to improve by making it consumless resources and deliver high speed. It is the 5th phase of the compiler, and this phase is optional.

A code optimization process follow the following rules:

[1] The output code not, in any way change the meaning of the program.

[2] The optimization should increase the speed of the program and if possible the program should demand less no. of resources.

[3] optimization should not delay in overall compiling process.

→ Optimization can be divide broadely into two types:

① Machin independent : In this optimization the compilers transform the code without involving any CPU register and absolute memory location.

Ex.

```
while (i<10) {
    k=10;
    i=i+k;
}
```

⇒ optimized code (code motion)

```
k=10;
while (i<10)
    i=i+k;
}
```

Main Ideas, Questions & Summary:

Library / Website Ref.:-

[2] Machin dependant : It is done after the target code has been generated & when the code is transformed according to the target machine architecture.
It involves CPU register & may have absolute memory reference rather than relative reference.

(*) Source of code optimization

1. Compile time evaluation
2. Common sub expression elimination
3. Dead code elimination
4. Loop optimization
5. Variable propagation
6. Constant propagation
7. Constant Folding
8. Copy propagation
9. Unreachable code elimination

8.5 Optimization of Basic Blocks

We can often obtain a substantial improvement in the running time of code merely by performing *local* optimization within each basic block by itself. More thorough *global* optimization, which looks at how information flows among the basic blocks of a program, is covered in later chapters, starting with Chapter 9. It is a complex subject, with many different techniques to consider.

8.5.1 The DAG Representation of Basic Blocks

Many important techniques for local optimization begin by transforming a basic block into a DAG (directed acyclic graph). In Section 6.1.1, we introduced the DAG as a representation for single expressions. The idea extends naturally to the collection of expressions that are created within one basic block. We construct a DAG for a basic block as follows:

1. There is a node in the DAG for each of the initial values of the variables appearing in the basic block.

2. There is a node N associated with each statement s within the block. The children of N are those nodes corresponding to statements that are the last definitions, prior to s, of the operands used by s.

3. Node N is labeled by the operator applied at s, and also attached to N is the list of variables for which it is the last definition within the block.

4. Certain nodes are designated *output nodes*. These are the nodes whose variables are *live on exit* from the block; that is, their values may be used later, in another block of the flow graph. Calculation of these "live variables" is a matter for global flow analysis, discussed in Section 9.2.5.

The DAG representation of a basic block lets us perform several code-improving transformations on the code represented by the block.

a) We can eliminate *local common subexpressions*, that is, instructions that compute a value that has already been computed.

b) We can eliminate *dead code*, that is, instructions that compute a value that is never used.

c) We can reorder statements that do not depend on one another; such reordering may reduce the time a temporary value needs to be preserved in a register.

d) We can apply algebraic laws to reorder operands of three-address instructions, and sometimes thereby simplify the computation.

8.5.2 Finding Local Common Subexpressions

Common subexpressions can be detected by noticing, as a new node M is about to be added, whether there is an existing node N with the same children, in the same order, and with the same operator. If so, N computes the same value as M and may be used in its place. This technique was introduced as the "value-number" method of detecting common subexpressions in Section 6.1.1.

Example 8.10: A DAG for the block

```
a = b + c
b = a - d
c = b + c
d = a - d
```

is shown in Fig. 8.12. When we construct the node for the third statement `c = b + c`, we know that the use of `b` in `b + c` refers to the node of Fig. 8.12 labeled $-$, because that is the most recent definition of `b`. Thus, we do not confuse the values computed at statements one and three.

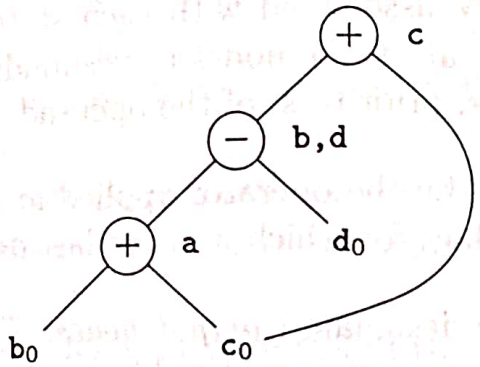


Figure 8.12: DAG for basic block in Example 8.10

However, the node corresponding to the fourth statement `d = a - d` has the operator $-$ and the nodes with attached variables `a` and d_0 as children. Since the operator and the children are the same as those for the node corresponding to statement two, we do not create this node, but add `d` to the list of definitions for the node labeled $-$. □

It might appear that, since there are only three nonleaf nodes in the DAG of Fig. 8.12, the basic block in Example 8.10 can be replaced by a block with only three statements. In fact, if `b` is not live on exit from the block, then we do not need to compute that variable, and can use `d` to receive the value represented by the node labeled $-$ in Fig. 8.12. The block then becomes

```
a = b + c
d = a - d
c = d + c
```

However, if both b and d are live on exit, then a fourth statement must be used to copy the value from one to the other.[1]

Example 8.11: When we look for common subexpressions, we really are looking for expressions that are guaranteed to compute the same value, no matter how that value is computed. Thus, the DAG method will miss the fact that the expression computed by the first and fourth statements in the sequence

```
a = b + c
b = b - d
c = c + d
e = b + c
```

is the same, namely $b_0 + c_0$. That is, even though b and c both change between the first and last statements, their sum remains the same, because $b + c = (b - d) + (c + d)$. The DAG for this sequence is shown in Fig. 8.13, but does not exhibit any common subexpressions. However, algebraic identities applied to the DAG, as discussed in Section 8.5.4, may expose the equivalence. □

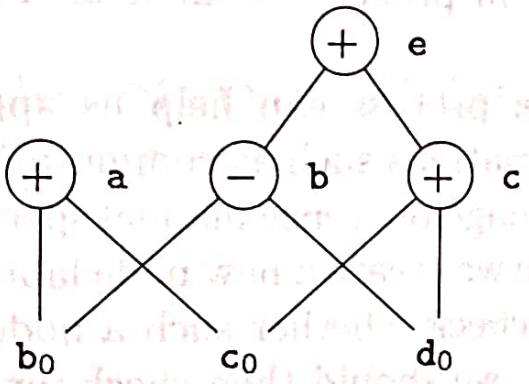


Figure 8.13: DAG for basic block in Example 8.11

8.5.3 Dead Code Elimination

The operation on DAG's that corresponds to dead-code elimination can be implemented as follows. We delete from a DAG any root (node with no ancestors) that has no live variables attached. Repeated application of this transformation will remove all nodes from the DAG that correspond to dead code.

Exam

8.5.4 The Use of Algebraic Identities

Algebraic identities represent another important class of optimizations on basic blocks. For example, we may apply arithmetic identities, such as

$$x + 0 = 0 + x = x \qquad x - 0 = x$$
$$x \times 1 = 1 \times x = x \qquad x/1 = x$$

to eliminate computations from a basic block.

Another class of algebraic optimizations includes local *reduction in strength*, that is, replacing a more expensive operator by a cheaper one as in:

EXPENSIVE		CHEAPER
x^2	=	$x \times x$
$2 \times x$	=	$x + x$
$x/2$	=	$x \times 0.5$

A third class of related optimizations is *constant folding*. Here we evaluate constant expressions at compile time and replace the constant expressions by their values.[2] Thus the expression $2 * 3.14$ would be replaced by 6.28. Many constant expressions arise in practice because of the frequent use of symbolic constants in programs.

The DAG-construction process can help us apply these and other more general algebraic transformations such as commutativity and associativity. For example, suppose the language reference manual specifies that $*$ is commutative; that is, $x*y = y*x$. Before we create a new node labeled $*$ with left child M and right child N, we always check whether such a node already exists. However, because $*$ is commutative, we should then check for a node having operator $*$, left child N, and right child M.

The relational operators such as $<$ and $=$ sometimes generate unexpected common subexpressions. For example, the condition $x > y$ can also be tested by subtracting the arguments and performing a test on the condition code set by the subtraction.[3] Thus, only one node of the DAG may need to be generated for $x - y$ and $x > y$.

Associative laws might also be applicable to expose common subexpressions. For example, if the source code has the assignments

```
a = b + c;
e = c + d + b;
```

the following intermediate code might be generated:

[2] Arithmetic expressions should be
run time. K

```
a = b + c
t = c + d
e = t + b
```

If t is not needed outside this block, we can change this sequence to

```
a = b + c
e = a + d
```

using both the associativity and commutativity of $+$.

The compiler writer should examine the language reference manual carefully to determine what rearrangements of computations are permitted, since (because of possible overflows or underflows) computer arithmetic does not always obey the algebraic identities of mathematics. For example, the Fortran standard states that a compiler may evaluate any mathematically equivalent expression, provided that the integrity of parentheses is not violated. Thus, a compiler may evaluate $x * y - x * z$ as $x * (y - z)$, but it may not evaluate $a+(b-c)$ as $(a+b)-c$. A Fortran compiler must therefore keep track of where parentheses were present in the source language expressions if it is to optimize programs in accordance with the language definition.

8.5.5 Representation of Array References

At first glance, it might appear that the array-indexing instructions can be treated like any other operator. Consider for instance the sequence of three-address statements:

```
x = a[i]
a[j] = y
z = a[i]
```

If we think of `a[i]` as an operation involving `a` and `i`, similar to $a + i$, then it might appear as if the two uses of `a[i]` were a common subexpression. In that case, we might be tempted to "optimize" by replacing the third instruction `z = a[i]` by the simpler `z = x`. However, since `j` could equal `i`, the middle statement may in fact change the value of `a[i]`; thus, it is not legal to make this change.

The proper way to represent array accesses in a DAG is as follows.

1. An assignment from an array, like `x = a[i]`, is represented by creating a node with operator $=[]$ and two children representing the initial value of the array, a_0 in this case, and the index `i`. Variable `x` becomes a label of this new node.

2. An assignment to an array, like `a[j] = y`, is represented by a new node with operator $[]=$ and three children representing a_0, `j` and `y`. There is no variable labeling this node. What is different is that the creation of

Example 8.13: The DAG for the basic block

```
x = a[i]
a[j] = y
z = a[i]
```

is shown in Fig. 8.14. The node N for x is created first, but when the node labeled `[]=` is created, N is killed. Thus, when the node for z is created, it cannot be identified with N, and a new node with the same operands a_0 and i_0 must be created instead. □

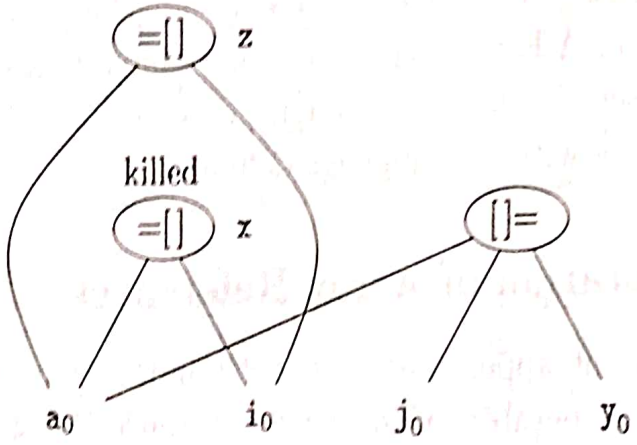


Figure 8.14: The DAG for a sequence of array assignments

Example 8.14: Sometimes, a node must be killed even though none of its children have an array like a_0 in Example 8.13 as attached variable. Likewise, a node can kill if it has a descendant that is an array, even though none of its children are array nodes. For instance, consider the three-address code

```
b = 12 + a
x = b[i]
b[j] = y
```

What is happening here is that, for efficiency reasons, `b` has been defined to be a position in an array `a`. For example, if the elements of `a` are four bytes long, then `b` represents the fourth element of `a`. If `j` and `i` represent the same value, then `b[i]` and `b[j]` represent the same location. Therefore it is important to have the third instruction, `b[j] = y`, kill the node with `x` as its attached variable. However, as we see in Fig. 8.15, both the killed node and the node that does the killing have a_0 as a grandchild, not as a child. □

★ Peephole Optimization

⇒ A simple but effective technique for locally improving the target code is peephole optimization, which is done by examining a sliding window of target instructions (called the peephole) and replacing instruction sequence within the peephole by a shorter or faster sequence.

The peephole is a small, sliding window on a program.

→ characteristic of peephole optimizations

① Redundant - instructions elimination

⇒ Redundant loads and stores of this nature would not be generated by the simple code generation algorithms of the previous section.

② Eliminating Unreachable code

Ex.

For debugging purpose, a large program may have within it certain code fragments that are executed only if a variable debug is equal to 1.

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

③ Flow of control optimization

⇒ Simple intermediate code generation algorithms frequently produce jumps to jumps to conditional jumps, or conditional jumps to jumps.

④ Algebric simplification and Reduction in strength

⑤ Reduction in strength transformation

⑥ Use of Machine Idioms

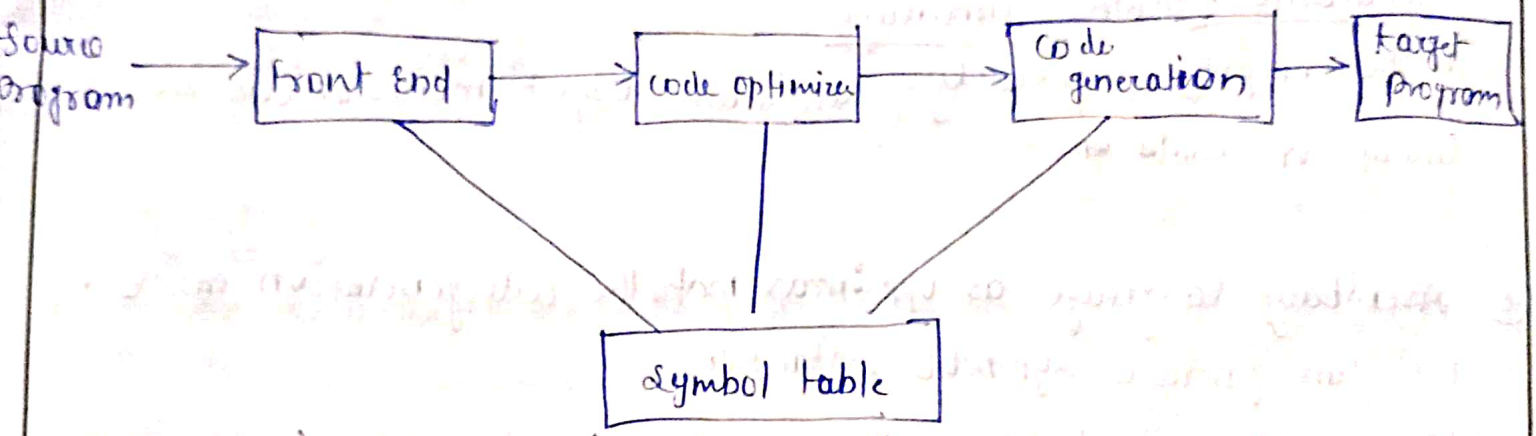
⑦ Register allocation & assignment

POORNIMA

Code - Generation

Date	Unit No.	Lecture No.	Faculty	Subject Name	Subject Code	Main Topics:-

The final state of compiler model is code generation. It takes as input a intermediate representation of the source code & produce an output a equivalent target program. The code generation techniques can be used whether or not an optimizing phase occurs before code generation.



→ Following issues rycrs in code generation

[1] I/p to the code generation consist of the intermediate code generated by front end, along with the info. of symbol table to determine rm the addresses of the data denoted by intermediate code be represented in

The code generation procen the provided Input data are error free.

Main Ideas, Questions & Summary:

[2] Target Program is the output of the code generator. The o/p may be absolute machine language. The language may facilitate some machine-specific instructions to help the compiler generate the code is more convenient way.

* Absolute machine as language o/p has advantage that this can be placed in fixed memory location, and can be immediately Executed

* Relocatable machine language

to be compile seperately. Relocatable Object modules can be linked or loade

* Asembaly language as o/p : make the code generation easier, we can generate symbolic instruction in generation code.

[3] Instruction selection : The code generator takes intermediate Representation as input & converts (maps) it into target machines instruction set. One representation can have may way (instructions) to convert it, so it becomes the responsibility of the code generater to choose the appropriate instructions wisely

[4] Register allocation

[5] Ordering of instruction

POORNIMA

Date	Unit No.	Lecture No.	Faculty	Subject Name	Subject Code	Main Topics:-

★ Basic blocks and flow graph

- Flow graph : Basic block is set of statement that always Execute in a sequence One after other.

→ It means that the flow of control enters at the beginning & it always leaves at the end without halt.

All the statement Execute in some manner as the after.

⊛ Algorithm

① determine leader : Following code are called as leader.

[A] First statement of the code

[B] Target code of the conditional & unconditional immediately after goto statement.

② determine basic blocks :

→ Input

→ Output

→ Method

③ Transformations of basic blocks

[1] structure-preserving Transformations

[2] Algebric transformation

Main Ideas, Questions & Summary:

Library / Website Ref.:-

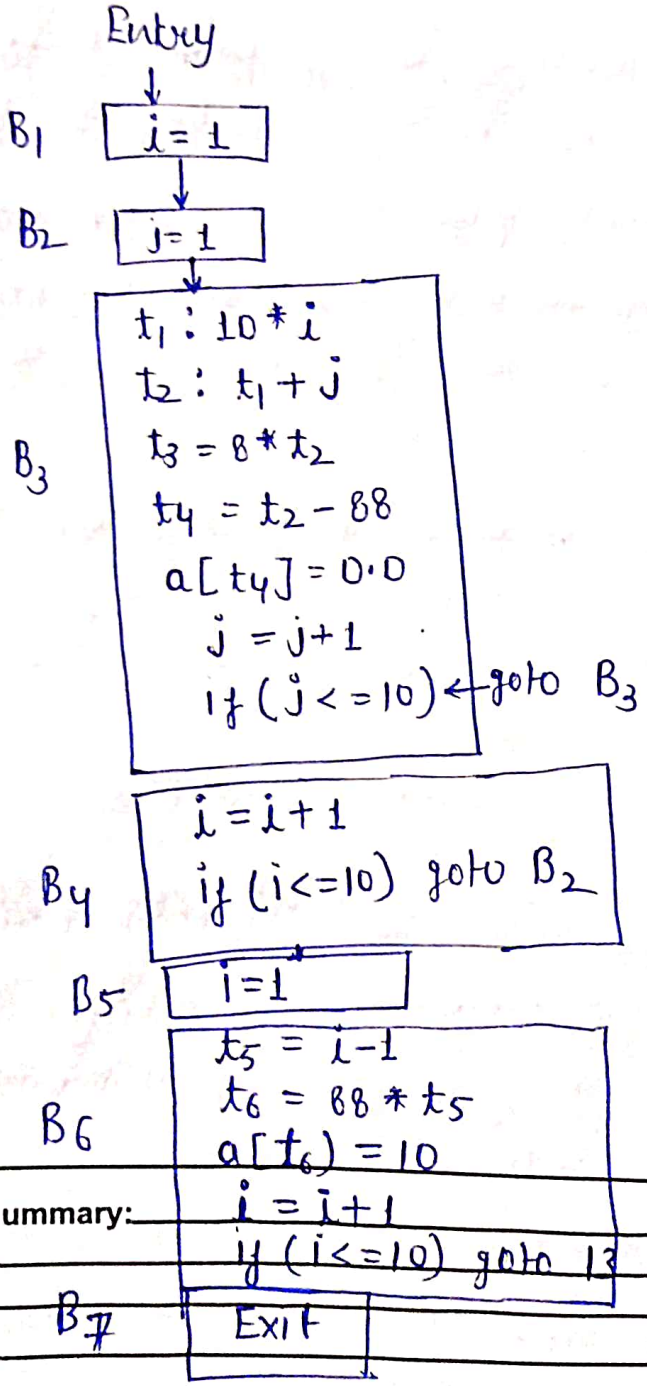
POORNIMA

Date	Unit No.	Lecture No.	Faculty	Subject Name	Subject Code	Main Topics:-

★ Flow graph is a directed graph with flow information added to the basic blocks.

① The basic block serve as node of the flow graph.

② There is a directed edge from block b_1 to block b_2. If b_2 appears, immediatly by b_1 in the code



Main Ideas, Questions & Summary:

Library / Website Ref.:-