INDEX

NAME: Sindhu S Shastry STD.: ____ SEC.: ____ ROLL NO.: ____ SUB.: Cloud Computing Compiler Design notes

S. No.	Date	Title	Page No.	Teacher's Sign / Remarks
		PDA		
		Examples of all Phases		
		X Merits and demerits of Intermediate code forms		
		Left factoring and all.		
		Quadruples		
		Triples		
		Indirect triples		
		Operator Precedence ~~Parser~~ Grammar		

Write & learn for

classmate
Date
Page

Compiler Design

- Introduction : phases of Compilation and Overview. Lexical Analysis (Scanner):
Regular Languages, finite Automata, Regular Expressions, relating regular Expressions and Finite Automata, Scanner generator (lex, flex) Definitions

- Syntax Analysis (Parser) : Context-Free Languages and Grammars, Push-down Automata, LL(1) Grammars and top-down parsing, operator Grammars, LR(0), SLR(1), LR(1), LALR(1) grammars and bottom-up parsing, ambiguity and LR Parsing, yacc LALR(1) parser generation (yacc, bison).

- Semantic Analysis : Attribute grammars, syntax directed definition, evaluation and flow of attribute in a syntax.

- Symbol Table : Basic Structure, Symbol attributes and managment
Run-time environment : Procedure activation, Parameter Passing, value return, memory allocation, scope.

- Intermediate Code Generation : Translation of different language features, different types of intermediate forms.

- Code optimization : Control flow, data-flow dependence etc. local optimization, global optimization, loop optimization, peep-hole optimization.

Architecture dependent Code optimization : instruction Scheduling (for pipeline), loop optimization (for cache memory), Register allocation & Target Code generation.

Introduction

Compiler: It translates the code written in one language (higher level) to some other language without changing the meaning of the Program. (Machine level instructions)

It is also expected that a compiler should make the target code efficient and optimized in terms of time & space.

Compiler Design:

The principles of CD provides an in depth view of translation & optimization process. It covers basic translation mechanism and error detection & Recovery.

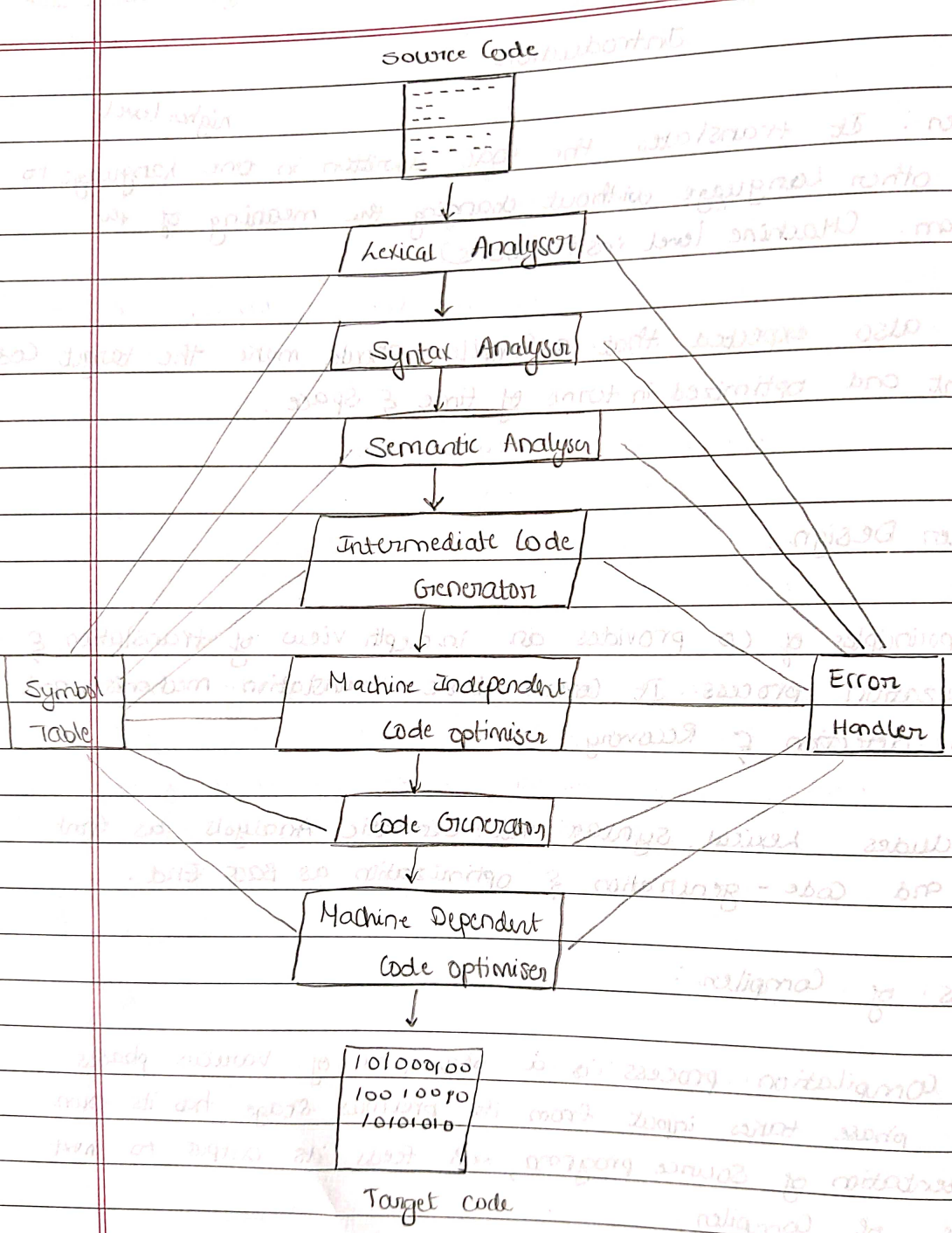
It includes Lexical, Syntax and Semantic Analysis as front End and code-generation & optimization as Back End.

phases of Compiler:

The Compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to next phase of Compiler.

Analysis Phases – Lexical, Syntax & Semantic

Synthesis Phases – code optimizer, code generation, intermediate code generator



∴ phases of Compilation

Lexical Analysis:

→ The First phase of Scanner works as a Text Scanner.
→ This phase scans the Source Code as a Stream of characters and converts it into Meaningful lexemes. Lexical Analyser represents lexemes as tokens in the form:

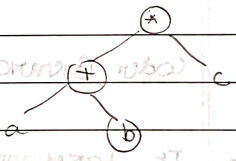
< token-name, attribute-value >

Ex: x = y + 10 Tokens x - identifier y - identifier
= - assigment operator + - addition operator
10 - Number

Syntax Analysis:

→ It is also called as Parsing. It takes token produced by lexical
→ Analysis as input and generates a parse tree (Syntax tree).
→ Token arrangements are checked against the source code Grammar
i.e., parser checks if the expression made by the tokens is syntactically correct.

(a + b) * c



Semantic Analysis

→ It checks whether the parse tree constructed follows the rules of Language. Ex: Assignment of values is b/w Compatible data types, and adding string to an integer

→ It also keeps track of identifiers, their types & expressions; whether identifiers are declared before use or not etc.
→ It produces an annotated Syntax tree as an output.

float x = 20.2;
float y = x * 30;

typecast the integer 30 to float 30.0.

Intermediate Code Generation

- After Semantic Analysis, the compiler generates an intermediate code of the target machine.
- It Represents a program for some abstract machine. It is in b/w the high-level language & machine language. This intermediate code should be Generated in such a way that it makes it easier to be translated into the target machine code.

```
total = count + rate * 5

t1 := int_to_float(5)      t3 := count + t2
t2 := rate * t1            total := t3
```

Code optimization

- Code optimization of the intermediate code is done.
- It Removes unnecessary code lines, and arranges the sequence of statements in order to speed up the Program execution without wasting Resources (CPU, Memory)

Code Generation

- It takes the optimized representation of intermediate code & maps it to the target machine language. The code Generator translates the intermediate code into a Sequence of re-locatable machine code.
- Sequence of instructions of machine code performs the task as the intermediate code would do.

```
a = b + 60.0

MOV  B, R1          MOV #60.0, R2
MOV  #60.0, R2
ADD  R1, R2
MOV  R2, A
```

Symbol table:

→ It is a data-structure maintained throughout all the phases of a Compiler.

→ All the identifier's name along with their types are stored here.

→ It makes it easier for compiler to quickly search the identifier record and retrieve it.

→ It is also used for scope Managment.

Regular Language:

It is a Language that can be expressed with a Regular Expression / a DFA or NFA.

It is a set of Strings which are made up of characters from a specified alphabet or set of symbols.

Finite Automata

FA is the simplest machine to recognize patterns.

It is an abstract machine with 5 tuples.

- Start State

- End State

$[Q, \Sigma, q, F, \delta]$

Q: Finite set of states

Σ: set of Input Symbols

q: Initial state

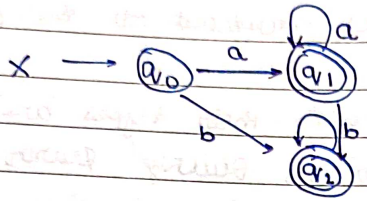
F: Set of Final States

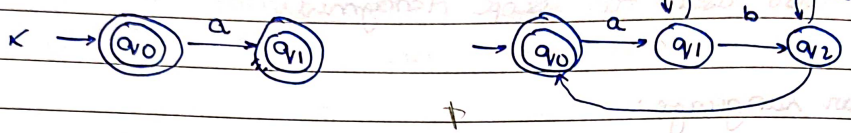
δ: Transition Function

$D = [Q, \Sigma, q, F, \delta]$

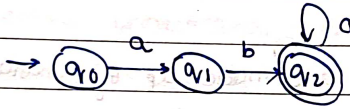
1. Draw a FA to input the string $a^n b^n$, $n > 0$

$a^0 b^0$ - $n \geq 0$





2. Draw a FA to input string $(ab)^n$



Regular Expression:

It can be recursively defined as follows:

→ ϕ is a regular expresion denoting a empty language.

→ ε is a regular expression indicating the language containing an empty string.

→ a is a regular expression containing {a}

→ If R is a regular expression denoting the language L_R and S is a regular expression denoting language L_S.

- R+S Corresponds to Language ~~$L_R \cup L_S$~~ $L_R \cup L_S$.
- R.S Corresponds to Language $L_R . L_S$
- R^* Corresponds to Language to L_R^*.

26/10/21

CFG - 4 tuple $G = (V, T, P, S)$ T → lower case

V → Upper case

Derivations → Leftmost

→ Rightmost

1) $S \to AB$

$A \to aaA \mid \varepsilon$

$B \to bB \mid \varepsilon$

Obtain string aaaabb

LMD :

$S \Rightarrow AB$

$\Rightarrow aaAB$

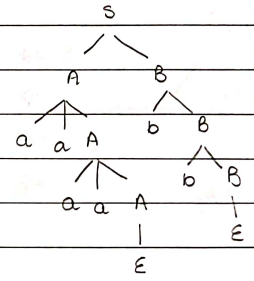
$\Rightarrow aaaaAB$

$\Rightarrow aaaa\varepsilon B$

$\Rightarrow aaaabB$

$\Rightarrow aaaabbB$

$\Rightarrow aaaabb\varepsilon$ //



RMD :

$S \Rightarrow AB$

$\Rightarrow AbB$

$\Rightarrow Abb B$

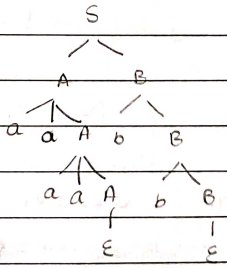
$\Rightarrow Abb\varepsilon$

$\Rightarrow aaAbb$

$\Rightarrow aaaaAbb$

$\Rightarrow aaaa\varepsilon bb$

$\Rightarrow aaaabb$



Ambiguous Grammar. Not Ambiguous.

2) Obtain CFG & tree for $3 * 2 + 5$ using

Date
Page

S → ASB | + | - | / | * | % | (A) | ε

A → C | 0 | 1 | ... | 9

B → 0 | 1 | ... | 9 | 1)

OR

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

E → ε | 0 | 1 | 2 | ... | 9

LMD : 3 * 2 + 5

S ⇒ ASBS

⇒ 3SBS

⇒ ~~3*B~~ ~~3BSB~~

⇒ 3 * 2 ⇒ 3 * BS

⇒ 3 * 2 S

⇒ 3 * 2 ASBS

⇒ 3 * 2 ε S BS

⇒ 3 * 2 + BS

⇒ 3 * 2 + 5 S

⇒ 3 * 2 + 5 ε

OR

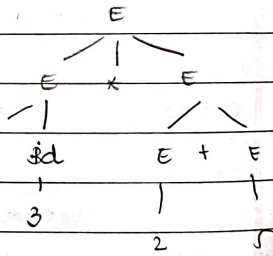
E ⇒ E * E

⇒ 3 * E

⇒ 3 * E + E

⇒ 3 * 2 + E

⇒ 3 * 2 + 5



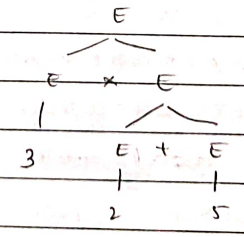
RMD : E ⇒ E * E

⇒ E * E + E

⇒ E * E + 5

⇒ E * 2 + 5

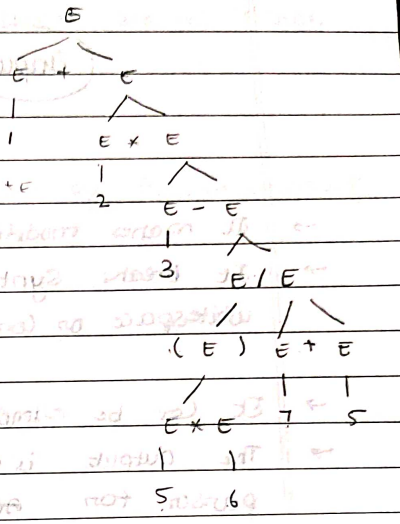
⇒ 3 * 2 + 5



1 + 2 * 3 - (5 * 6) / 7 + 5

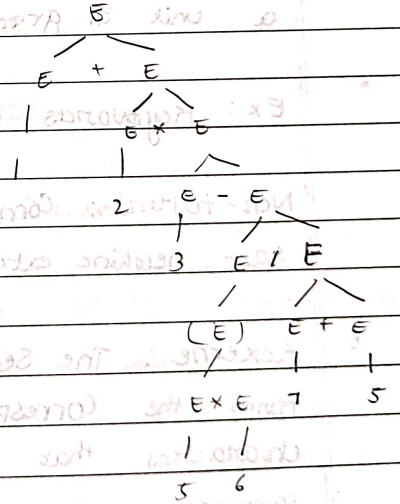
LMD:

$E \Rightarrow E + E$
$\Rightarrow 1 + E$
$\Rightarrow 1 + E * E$
$\Rightarrow 1 + 2 * E$
$\Rightarrow 1 + 2 * E - E$
$\Rightarrow 1 + 2 * 3 - E$
$\Rightarrow 1 + 2 * 3 - E / E$
$\Rightarrow 1 + 2 * 3 - (E) / E$
$\Rightarrow 1 + 2 * 3 - (E * E) / E$
$\Rightarrow 1 + 2 * 3 - (5 * E) / E$
$\Rightarrow 1 + 2 * 3 - (5 * 6) / E$
$\Rightarrow 1 + 2 * 3 - (5 * 6) / 7 + E$
$\Rightarrow 1 + 2 * 3 - (5 * 6) / 7 + 5$

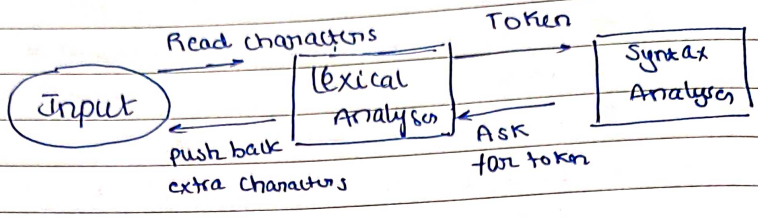


RMD:

$E \Rightarrow E + E$
$\Rightarrow E + E * E$
$\Rightarrow E + E * E - E$
$\Rightarrow E + E * E - E / E$
$\Rightarrow E + E * E - (E) / E$
$\Rightarrow E + E * E - (E) / E + E$
$\Rightarrow E + E * E - (E) / E + 5$
$\Rightarrow E + E * E - (E) / 7 + 5$
$\Rightarrow E + E * E - (E * E) / 7 + 5$
$\Rightarrow E + E * E - (E * 6) / 7 + 5$
$\Rightarrow E + E * E - (5 * 6) / 7 + 5$
$\Rightarrow E + E * 3 - (5 * 6) / 7 + 5$
$\Rightarrow E + 2 * 3 - (5 * 6) / 7 + 5$
$\Rightarrow 1 + 2 * 3 - (5 * 6) / 7 + 5$ //



Lexical Analyser (scanner)



→ It takes modified source code from Language Preprocessors.

→ It breaks Syntaxes into series of tokens, removing any whitespace or comments in the source code.

→ It can be implemented using DFA.

→ The output is a sequence of tokens that is sent to parser for Syntax Analysis.

- Token: A sequence of characters that can be treated as a unit in grammar of Programming Languages.

  Ex: Keywords, identifier, operators, seperators.

  Non-tokens: Comments, preprocessor directive, macros, blanks, tabs, newline etc.

- Lexeme: The sequence of characters matched by a pattern to form the corresponding token or a sequence of input characters that comprises a single token is called lexeme.

  Ex: "float", "=", "_", "273" etc.

How it works

→ It tokenizes, that is Tokenization takes place to divide the program into valid tokens
→ Remove white space characters
→ Remove comments
→ It also provides help in generating error by providing row & column numbers.

Suppose we pass a statement,

a = b + c through lexical Analyser.

It will generate token sequence like

id = id + id where id refers to it's variable in Symbol table

Ex: int main()
{
// 2 variables
int a, b;
a = 10;
return 0;
}

valid tokens are:
'int', 'main', '(', ')', '{', 'int', 'a', ',', 'b', ';', 'a', '=', '10', ';', 'return', '0', ';', '}'

100/100

Representing valid tokens of a language in regular expression

- x^* – zero / more occurrence of x
- x^+ – one / more occurrence of x

- x? means at most one occurrence of x. {ε, x}
- [a-z] - Lower case alphabets
- [A-Z] - Upper case alphabets
- [0-9] - all digits

Decimal : (sign)? (digit)$^+$ '.' (digit)*

identifier : (letter) (letter| digit)

→ Lex

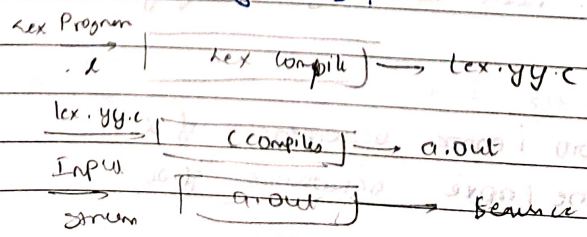
- Lex is a program that generates lexical Analyser. It is used with YACC parser generator
- The lexical Analyser is a program that converts input stream into sequence of tokens.

The Function of Lex

→ Lexical Analyser creates a program in lex language. Then Lex compiler runs the program and produces a C Program lex.yy.c

→ Finally C compiler runs lex.yy.c program and produces an object program a.out

→ a.out is a lexical Analyser that transforms an input stream into sequence of tokens.



Date
Page

Lex file Format

It is seperated by three sections by '.'. delimiters.

```
'/{
    definition section
'.}

'.'.
{ rules }

'.'.
{
  $User Subroutines }
}
```

→ Definitions include declarations of constant, variable and headerfiles.

→ Rules : define the statement of form

```
p1  { action 1 }
p2  { action 2 }
pn  { action n }
```

where pá describes regular expression and action1 describes the actions what lexical analyser should take when pattern pi matches a lexeme.

→ User Subroutine : are auxiliary procedures needed by the actions.

Special characters

.	any character except newline
\.	literal '.'
\n	newline
\t	tab
^	beginning of line
$	End of line

operations

?	zero / one copy of the expression
*	zero / more copies
+	one / more copies
a|b	a or b
(ab)+	one / more copies of ab
abc	abc
(abc)*	ε, abc, abcabc ....
abc*	ab, abc, abcc, abccc, ...
"abc"	literal abc
"abc*"	literal abc*
abc+	abc, abcc, abccc ...
a(bc)+	abc, abcbc, abcbcbc ...
a(bc)?	a or abc

Character class

[abc]	one of : a b c
[a-z]	any of a to z
[a\-z]	one of : a - z
[-az]	one of :- a z

[a ^b] a b

[A-Z a-z 0-9]+	one / more alphanumerics
[\t\n]+	whitespace
[^ab]	except : a b
[a^b]	one of : a ^ b
[a|b]	one of : a | b

Predefined Variables in Lex

int yylex (void)	Call to invoke lexer, returns token
char *yytext	pointer to Matched string
yyleng	length of Matched string
yylval	value associated with token
int yywrap(void)	wrapup, return 1 if done, 0 if not done
FILE *yyout	Output file
FILE *yyin	Input file
ECHO	write Matched string.

Syntax Analysis | Parsing

→ Limitations of lexical analyser is that it cannot check the syntax of a given sentence due to limitations of Regular expressions such as balancing parenthesis.

→ This phase uses Context-Free Grammar recognized by Push down Automata.

CFG is super set of Regular grammar.

→ CFG is helpful tool in describing the syntax of programming languages.

Context - Free Grammar:

A Grammar $G = (V, T, P, S)$ is said to be Context-Free Grammar if all the Productions are of the form

$A \rightarrow \alpha$ where $\alpha \in (V \cup T)^*$ and A is a non-terminal.

ϵ can appear on right hand side of any Production.

It is a 4 tuple denoted by

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

V = Set of Variables (non-terminals)
T = Set of Terminals
P = Set of Productions
S = start Symbol.

Derivation: The Process of obtaining a string of terminals ~~or non-terminals~~ from Start Symbol by applying some set of Productions is called derivation.

→ At each step, only one variable is replacing using a production
→ It starts with start Symbol S and ends in string of terminals

Right-most Derivation: The process of obtaining a string of terminals from a sequence of replacements such that only rightmost variables is replaced at each and every step.

Left-most Derivation: The process of obtaining a string of terminals by from a sequence of replacements such that only left-most non-terminal is replaced at each and every step.

Parse tree (Derivation Tree)

Let $G = (V, T, P, S)$ be a Context-Free grammar. The tree is derivation tree if:

- The Root has label S
- Every Vertex has label which is in $(V \cup T \cup \varepsilon)$
- Every leaf node has label from T and interior vertex has label from V
- If a vertex is labelled A and $X_1, X_2 \ldots X_n$ are all children of A from left,
  then $A \rightarrow X_1 X_2 \ldots X_n$ is the Production.

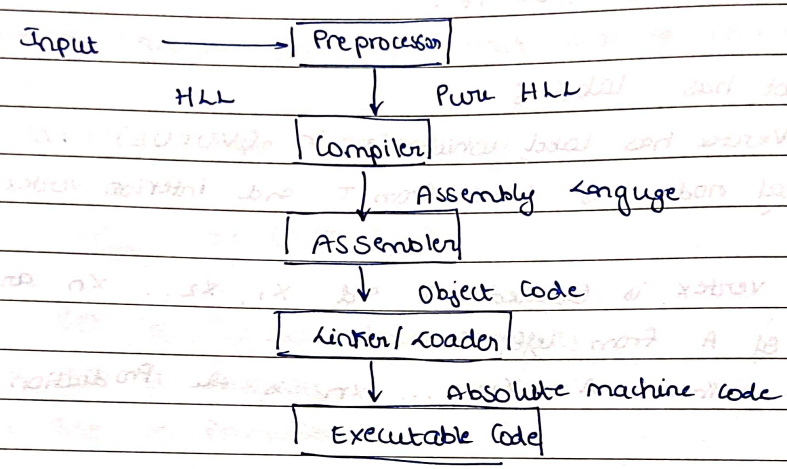
Ambigious Grammar

Let $G = (V, T, P, S)$ be a Context-Free Grammar. Grammar G is said to be ambigious
iff there exists one string $w \in T^*$ for which two / more different parse trees exist by applying either LMD or RMD.

Language Processing in Compiler Begin

The higher level language are fed into a series of devices and operating system (OS) components to obtain the desired code that can be used by the machine.

This is known as a Language Processing System.

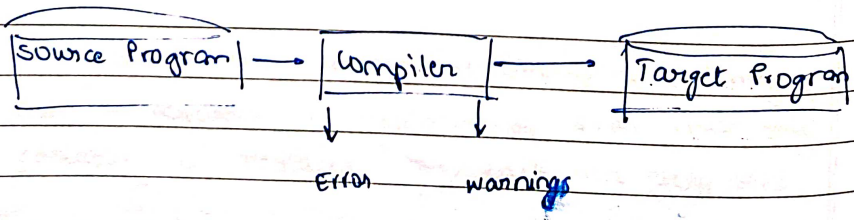


Preprocessor: It includes all the header files and also evaluates whether a macro is included. And expands the macro by its contents.

→ It takes source code as input and produces modified source code as output.

→ Also known as Macro evaluate.

Compiler:



→ It takes the modified source code as input and converts it into Assembly language.

Assembler

→ It takes Assembly language and converts it into machine code instructions as output.

** Linker :

A linker or Link editor is a program that takes a collection of objects and combines them into an executable program.

** Loader :

The Loader keeps the linked program in the main memory

o Executable code

It is the low level and machine Specific code that machine can easily understand.

Compiler	Interpreter
• Compiler takes Entire program as input	• Interpreter takes Single instruction as input
o Intermediate code Object code is generated	o No intermediate object code is genration
o Conditional control statements are executed first	o Conditional control statements are Executes slowes

○ Memory Requirement is More	○ Memory Requirement is slow.
○ Program need not be compiled every time	○ Every time higher level program is converted into lower level program.
○ Errors are displayed after entire program is checked	○ Errors are displayed for every instruction
Ex: C Compiler	Ex: BASIC

9/12/21

Context-Free Language:

- It is a language generated by Context-Free Grammar.
- They have many applications in programming languages, in particular, most arithmetic expressions are generated by Context-Free Grammar.

Ex: $\{a^n b^n \mid n \geq 0\}$

Context-Free Grammar:

A Grammar $G = (V, T, P, S)$ is said to be Context-Free if all the productions are of the form

$A \rightarrow \alpha$ where $\alpha \in (V \cup T)^*$ and A is a non-terminal.

ε can appear on RHS of any production.

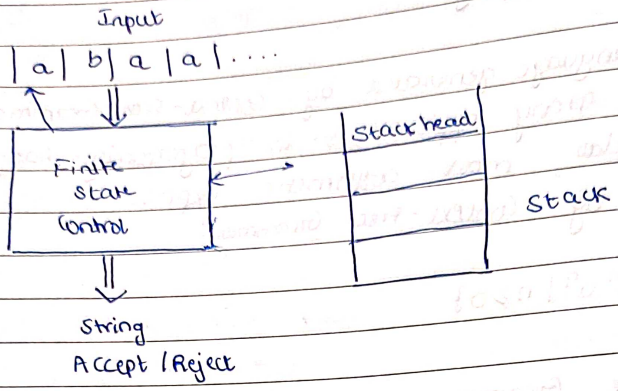
It can be implemented through Push-Down Automata.

Push Down Automata.

It is a finite automata with extra memory called stack which helps to recognize Context-Free Grammar.

$P = \{Q, \Sigma, \Gamma, q_0, Z, F, \delta\}$ 7-tuple

Q = Set of all states
Σ = Set of input Symbols
Γ = Set of pushdown symbols
q_0 = initial state
Z = initial Pushdown Symbol
F = Set of Final states
δ = Transition Function



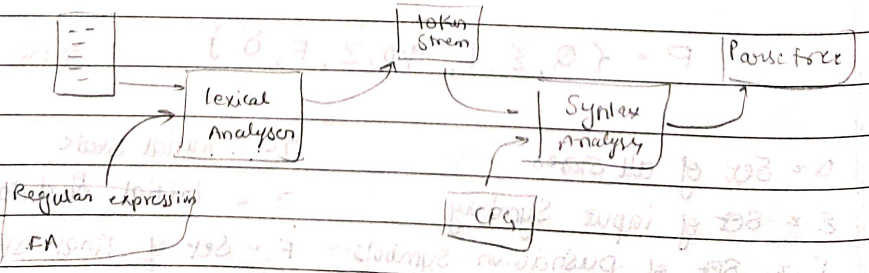
Syntax Analyser

→ It is the second phase of the compiler design process in which the given input string is checked for confirmation of rules and structure of formal grammar.

→ It analyses the syntactical structure and checks if the given input has correct syntax or not.

→ The parser analyzes the source code (token stream) against the production rules to detect any errors in the code. The output is a parse tree.

→ Parser parse through the whole code even if some errors exist in the program and use error recovering strategies.



First and Follow Sets:

These sets can provide the actual position of any terminal in the derivation. This is done to create the parsing Table.

First Set:

This set is created to know what terminal symbol is derived in the 1st Position by a Non-terminal.

Ex: $A \rightarrow tB$ $\quad t \in First(A)$ i.e., A derives t (terminal) in the 1st Position.

Algorithm for Calculating First Set.

→ If A is a terminal, then First(A) =

- $A \rightarrow \varepsilon$, then $First(A) = \{\varepsilon\}$
- $A \rightarrow a$, then $First(A) = \{a\}$
- $A \rightarrow XYZ$, then $First(A) = First(X) = \{a\}$
  $X \rightarrow a$ $\quad First(X) = \{a\}$
- $A \rightarrow abc$, then $First(A) = \{a\}$
- $A \rightarrow aA$, then $First(A) = \{a\}$

* Problems

1) $S \rightarrow abc \mid def \mid ghi$

$S \rightarrow abc$ $\quad First(S) = \{a, d, g\}$ //
$S \rightarrow def$
$S \rightarrow ghi$

2) $S \rightarrow ABC \mid ghi \mid jkl$

$A \rightarrow a \mid b \mid c$ First(A) = {a, b, c}

$B \rightarrow b$ First(B) = {b}

$C \rightarrow d$ First(C) = {d}

First(S) = {First(A), g, j}

= {a, b, c, g, j} //

3) $S \rightarrow A\,B\,C$

$A \rightarrow a \mid b \mid \varepsilon$ First(A) = {a, b, ε}

$B \rightarrow c \mid d \mid \varepsilon$ First(B) = {c, d, ε}

$C \rightarrow e \mid f \mid \varepsilon$ First(C) = {e, f, ε}

First(S) = First(A)

= {a, b, ε} First(B)

= {a, b, c, d, ε} First(C)

= {a, b, c, d, e, f, ε} //

4) $E \rightarrow TE'$

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

$T \rightarrow F.T'$

$T' \rightarrow +FT' \mid \varepsilon$

$F \rightarrow id \mid (E$

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

First(E') = {*, ε}

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

First(T') = {+, ε} First(E) = //

First(F) = {id, (}

5) $S \rightarrow aBDg$ — First(S) = {a}

$B \rightarrow cC$ — First(B) = {c}

$D \rightarrow EF$ — First(D) = First(E) = {g, ε, f}

$E \rightarrow g \mid \varepsilon$ — First(E) = {g, ε}

$F \rightarrow f \mid \varepsilon$ — First(F) = {f, ε}

Follow Sets

We Calculate what terminal symbol immediately follows a Non-terminal in Production Rules.

→ If S is start Symbol,

Follow(S) = {$}

→ $A \rightarrow \alpha CD$, Follow(A) = First(C), Follow(C) = First(D)

$A \rightarrow Aab$, Follow(A) = a

→ $S \rightarrow asbS$ Follow of S = {$, b} and Follow(S)

→ $A \rightarrow ACD$ Follow(D) = Follow(A)

1) $S \rightarrow ACD$ First(S) = First(A)

$C \rightarrow a \mid b$ First(C) = {a, b}

Follow(S) = {$}

Follow(A) = First(C) = {a, b}

Follow(C) = First(D)

Follow(D) = Follow(S) = {$}

2) $S \rightarrow asbS$

$S \rightarrow bSas$

$S \rightarrow \varepsilon$

Follow(S) = {$, b, a}

3) $S \rightarrow AaAb \mid BbBa$ Follow(S) = {$}

$A \rightarrow \varepsilon$ Follow(A) = {a, b}

$B \rightarrow \varepsilon$ Follow(B) = {b, a}

4) $S \rightarrow aBDh$ First(S) = {a}

$B \rightarrow CC$ First(B) = {c}

$C \rightarrow bC \mid \varepsilon$ First(C) = {b, ε}

$D \rightarrow EF$ First(D) = {g, f, ε}

$E \rightarrow g \mid \varepsilon$ First(E) = {g, ε}

$F \rightarrow f \mid \varepsilon$ First(F) = {f, ε}

Follow(S) = {$}

Follow(B) = First(D) = {g, f, h} [ε] Check

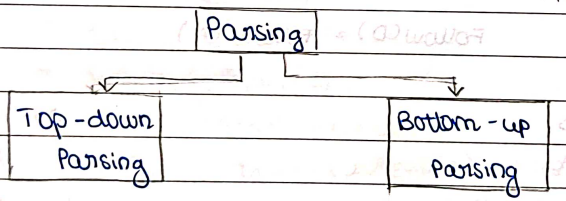
Follow(D) = {h}

Follow(C) = Follow(B) = {g, f, h}

Follow(E) = First(F) = {f} with ε = Follow(D) = {f, h}

Follow(F) = Follow(D) = {h}

Types of Parser



Top-Down:
- Recursive descent Parser
- Predictive Parser (LL(1))

Bottom up
- LR Parser (operator
- LR(0) Precedence
- SLR(1) parser
- LLR(1)
- LALR(1)

Top-Down Parser:

→ It generates parse for the given input string with the help of grammar Productions by expanding the non-terminals i.e., it starts from the start symbol and ends on the terminals.

It uses left-most Derivation.

Top-down Parser is further classified into

- Recursive descent descent Parser.
- Predictive / non-Recursive descent Parser.

Recursive Descent Parser (RDP):

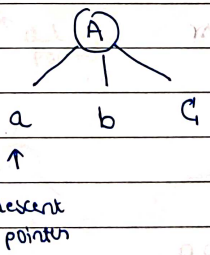
→ It is also known as Brute force parser or the backtracking Parser.
It basically generates parse tree by using brute force and backtracking.

1) $A \rightarrow abC \mid aBd \mid aAD$
$B \rightarrow bB \mid \epsilon$
$C \rightarrow d \mid \epsilon$
$D \rightarrow a \mid b \mid \epsilon$

Given input string : aaba

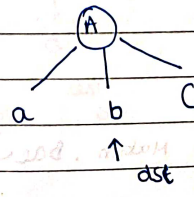
We consider 2 pointers : input pointer and descent pointer.
Construct the Parse tree.

a a b a
↑
input pointer



(They match). Increment i/p ptr and dst pointer by 1

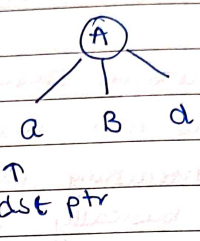
a a b a
↑
i/p ptr



Doesn't Match. Backtrack.

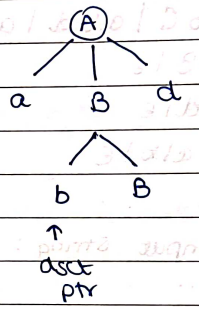
Using Backtracking,

a a b a
↑
i/p ptr



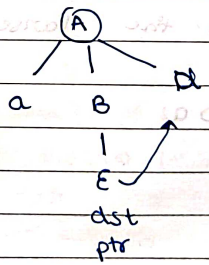
(They Match). Increment the pointers

a a b a
↑
i/p ptr

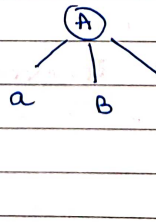


Doesn't Match. Backtrack.

a a b a
↑
i/p ptr



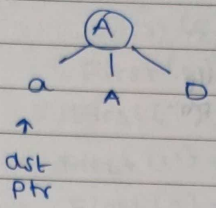
a a b a
↑
i/p ptr



Doesn't Match. Backtrack.

a a b a

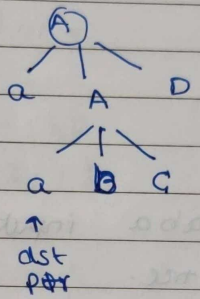
↑
i/p
ptr



Matches. increment ptrs.

a a b a

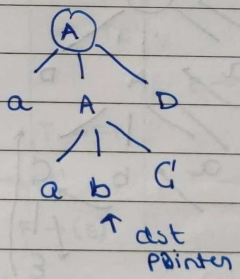
↑
i/p
ptr



Matches. Increment ptrs

a a b a

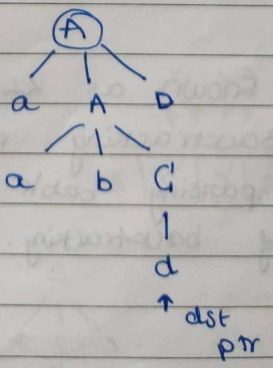
↑
i/p
ptr



Matches. Increment pointers

a a b a

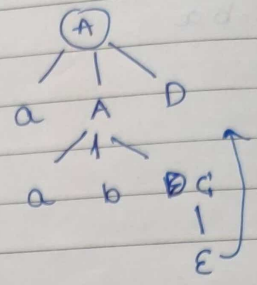
↑
i/p
ptr



Doesn't Match. Backtrack.

a a b a

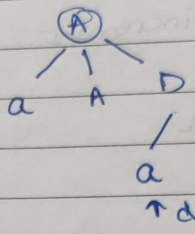
↑ i/p ptr



G → ε

a a b a

↑ i/p ptr

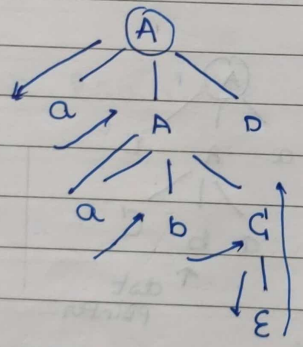


↑ dst ptr

Matches.

⟶

Hence, aaba input string is accepted using the below parse tree.



A → aAD

A → abC

C → ε

D → a

Non-Recursive Descent Parser: L-Left to Right, L-Left Derivation (1) - using 1 input symbol

It is also known as LL(1) Parser, Predictive parser or without Backtracking parser and Dynamic Parser. It uses parsing table to generate the Parse tree instead of backtracking.

1) $E \to TE'$ — First(E) = First(T) = { id, (}

$E' \to +TE' \mid \varepsilon$ — First(E') = { +, ε }

$T \to FT'$ — First(T) = First(F) = { id, (}

$T' \to *FT' \mid \varepsilon$ — First(T') = { *, ε }

$F \to id \mid (E)$ — First(F) = { id, (}

Follow(E) = { $,) }

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

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

Follow(F) = First(T') = { *, +, $,) }

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

follow first OF to fill table

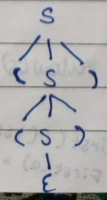
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to id$			$F \to (E)$		

2) $S \to (S) \mid \varepsilon$ — First(S) = { (, ε }

Follow(S) = {), $ }

(( )) $

	(	)	$
S	$S \to (S)$	$S \to \varepsilon$	$S \to \varepsilon$



Stack Implementation

$w = id * id + id \$$

Stack	I/P	O/P
$\$E$	$id*id+id\$$	$E \to TE'$
$\$E'T$	$id*id+id\$$	$T \to FT'$
$\$E'T'F$	$id*id+id\$$	$F \to id$
$\$E'T'id$	$id*id+id\$$	pop(id)
$\$E'T'$	$*id+id\$$	$T' \to *FT'$
$\$E'T'F*$	$*id+id\$$	Pop(*)
$\$E'T'F$	$id+id\$$	$F \to id$
$\$E'T'id$	$id+id\$$	pop(id)
$\$E'T'$	$+id\$$	$T' \to \varepsilon$
$\$E'$	$+id\$$	$E' \to +TE'$
$\$E'T+$	$+id\$$	pop(+)
$\$E'T$	$id\$$	$T \to FT'$
$\$E'T'F$	$id\$$	$F \to id$
$\$E'T'id$	$id\$$	pop(id)
$\$E'T'$	$\$$	$T' \to \varepsilon$
$\$E'$	$\$$	$E' \to \varepsilon$
$\$$	$\$$	pop($)

TO check LL(1) or not.

Ex: $S \to asa \mid bS \mid c$

$S \to \alpha_1 \mid \alpha_2 \mid \alpha_3$ - No ε production

First(asa) = {a}
First(bS) = {b}
First(c) = {c}

$First(\alpha_1) \cap F(\alpha_2) \cap F(\alpha_3) = \phi$ - LL(1).

$\neq \phi$ - Not LL(1)

$\neq \phi$ ∴ LL(1)

$S \to \alpha_1 \mid \alpha_2 \mid \varepsilon$

$F(\alpha_1) \cap F(\alpha_2) \cap Follow(S) = \phi$ LL(1)

$S \to iCtSS_1 \mid a$

$S_1 \to eS \mid \varepsilon$

$C \to b$

First(iCtSS1) = {i}
First(a) = {a} $= \phi$

First(S1) → {e, ε}
Follow{S, } = Follow(S) = First(ε)

$= \{e\}$
$\neq \phi$

2) $A \to abC \mid aBd \mid aAD$ — First(A) = {a}

$B \to bB \mid \varepsilon$ — First(B) = {b, ε}

$C \to d \mid \varepsilon$ — First(C) = {d, ε}

$D \to a \mid b \mid \varepsilon$ — First(D) = {a, b, ε}

Follow(A) = {$, First(D)} - {$, a, b}

Follow(C) = Follow(A) = {$, a, b}

Follow(B) = {d, Follow(B)} = {d}

Follow(D) = Follow(A) = {$, a, b}

	a	b	d	$
	aAD			
A	A→abC aBd	B→bB	B→ε	
B		B→bB	B→ε	
C	C→ε	C→ε	C→d	C→ε
D	D→a D→ε	D→b D→ε		D→ε

w = aaba

Stack	I/P	O/P
$A	aaba	A→aAD
$DAa	aaba	pop(a)
$DA	aba	A→aBd (A→aBC)
$DdBa	aba	pop(a)
$DdB	ba	B→bB
$DBb	ba	pop(b)
$DB	a	

Doesn't accept.

Bottom-up Parser

→ Bottom up Parser is the Parser which generates the parse tree for given input string with the help of grammar productions by compressing the non-terminals i.e., it starts from non-terminals and ends on the start symbol.
It uses reverse of Right most derivation.

There are 2 classifications

- LR parser:
- operator Precedence Parser

It generates parse tree for given string using unambiguous grammar. It follows reverse of Right-most derivation.

LR Parser is of 4 types:

a) LR(0) b) SLR(1) c) LALR(1) d) CLR(1)

- LR(0) - Left to Right and Reverse of Rightmost Derivation.
  (0) - doesn't care about lookahead items.

Rules for Construction.

LR(0) items are any item with '.' infront of them.

1) Construct Context-Free Grammar.
2) check for ambiguity
3) Add a Augment Production
4) Draw the DFA / DFD
5) create a Collection of Canonical LR(0) items
6) Construct LR(0) Parsing Table
7) check for validity with sample string.

$S \to AA$

$A \to aA \mid b$

$S' \to S$

$S \to A.A$

$A \to aA$

$A \to b$

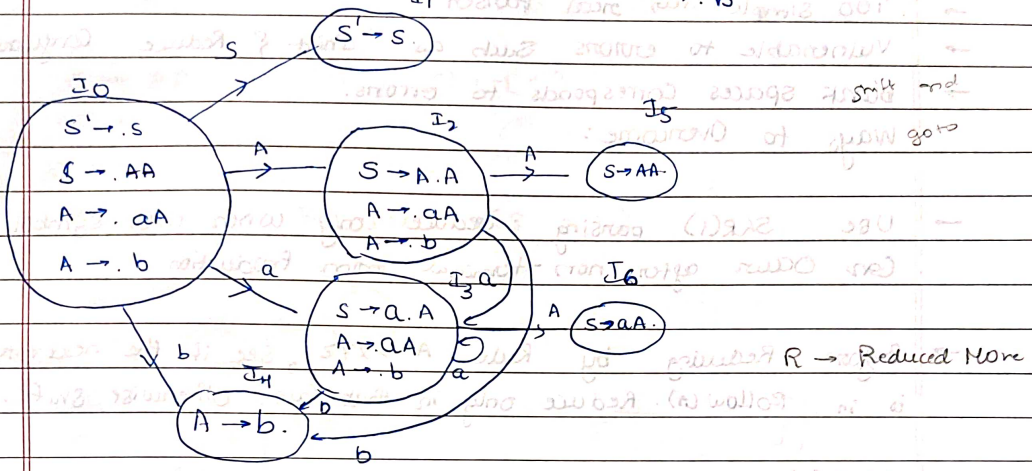
Canonical LR(0) items.

$S' \to .S$

$S \to .AA$

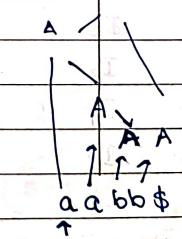
$A \to .aA$

$A \to .b$



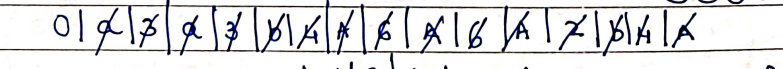
Parsing Table

States	Actions			Go to	
	a	b	$	A	S
I_0	S_3	S_4		2	1
I_1			Accepts		
I_2	S_3	S_4		5	
I_3	S_3	S_4		6	
I_4	R_3	R_3	R_3		
I_5	R_1	R_1	R_1		
I_6	R_2	R_2	R_2		



See stack implementation

$A \to b$ (3)



0	a a b b $		
0 a 3	a b b $		
0 a 3 a 3	b b $		
0 a 3 a 3 b 4	b $		
0 a 3 a 3 A 6	b $		
0 a 3 A 6	b $		
0 a 3 A 6 b 4	$		
0 a 3 A 6 A 5	$		
0 S 1	$	Accept.	

$A \to aA$ (2)

$A \to b$ (3)

$S \to AA$ (1)

Disadvantages of LR(0) Parser

- It doesn't lookahead of characters
- Too simple for real Parser
- Vulnerable to errors such as Shift & Reduce Conflicts.
- Blank spaces corresponds to errors.

Ways to Overcome:

- Use SLR(1) parsing : reduce only when next symbol can occur after non-terminal from Production.
- Before Reducing by Rule $A \rightarrow XYZ$, see if the next token is in Follow(A). Reduce only in that case. Otherwise, Shift.

SLR(1):

The main difference is the Parsing Table only.

States	Actions			goto	
	a	b	$	A	S
I_0	S_3	S_4		2	1
I_1			Accept		
I_2	S_3	S_4		5	
I_3	S_3	S_4		6	
I_4	R_3	R_3	R_3		
I_5			R_1		
I_6	R_2	R_2	R_2		

I_4 - R_3 Follow(LHS)

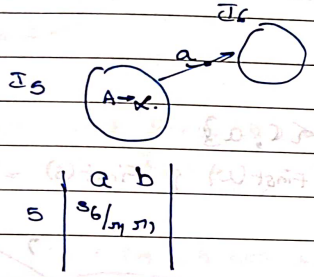
$A \to b.$ Follow(A) = First A & Follow(S) = {$, a, b}

I_5 - R_1

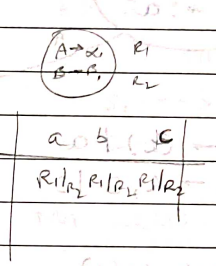
$S \to AA.$ Follow(S) = {$}

I_6 - R_2

$A \to aA$ Follow(A) = {$, a, b}



	a	b
5	s_6/r_1	r_1



1) $S \to dA \mid aB$ First(S) = {d, a} i) LL(1) ✓ Follow of all

$A \to bA \mid c$ First(A) = {b, c} ii) LR(0) ✓

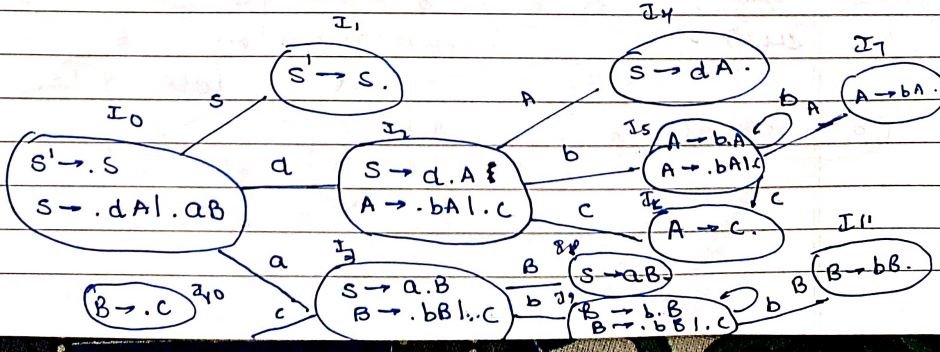
$B \to bB \mid c$ First(B) = {b, c} 3) SLR(1) ✓

$S' \to .S$

$S \to .dA \mid .aB$

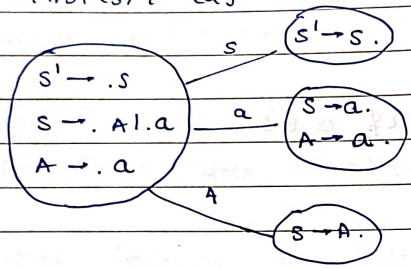
$A \to .bA \mid .c$

$B \to .bB \mid .c$



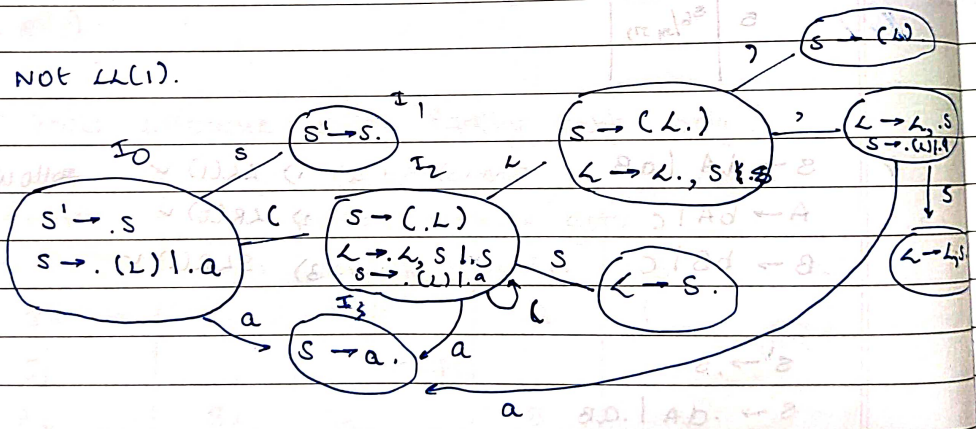
2) $S \to A \mid a$ First(S) = First A ∩ a i) LL(1) ✗
$A \to a$ ii) LR(0) ✗
iii) SLR(1) ✗

Follow (A) = Follow (S) = {$}
First (A) = {a}
First (S) = {a}
Both Same. Not LL(1)



3) $S \to (L) \mid a$ First(S) = {(, a} Same
$L \to L, S \mid S$ First (L) = First(L) & First(S) = {(, a}

i) Not LL(1).



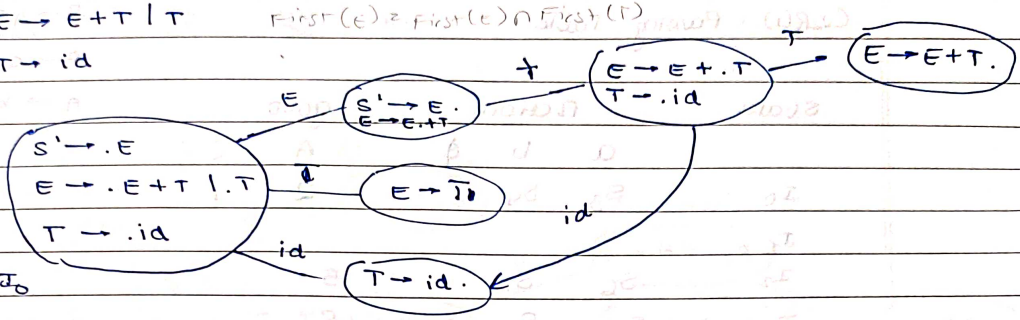
LR(0) ✓ S → First(S) = (∩ a

SLR(1) ✓ First (L) = First (L) ∩ First S
= (& .

Look ahead of the Symbol left

4) $E \to E+T \mid T$ First(E) = First(E) ∩ First(T)

$T \to id$



5) $S \to T+E \mid T$ LL(1)

$T \to id$

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

First(T) = {id}

id ∩ id ≠ ϕ NOT LL(1)

6) $S \to iCtSS_1 \mid a$

$S_1 \to eS \mid \varepsilon$

$C \to b$

NOT LL(1).

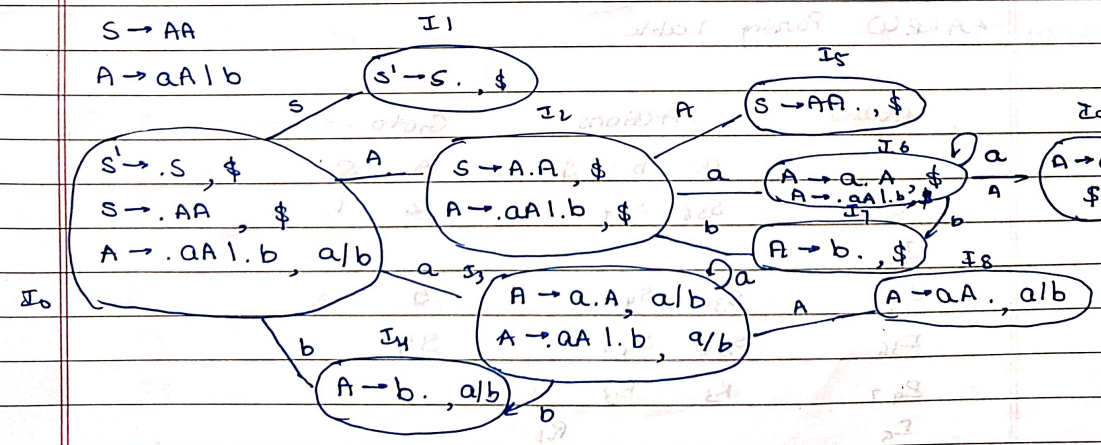
in a ≠ ϕ

e ∩ $ ≠ ϕ

Follow(S_1) = Follow(S) = $, e, ε

CLR Parser: Canonical Lookahead.

LALR(1) - CLR(1) = uses LR(1) item = LR(0) item + lookahead.



Date
Page

$S \rightarrow AA$ (1)
$A \rightarrow aA$ (2)
$A \rightarrow b$ (3)

CLR(1) Parsing Table

States	Actions			goto	
	a	b	$	A	S
I_0	S_3	S_4		2	1
I_1					
I_2	S_6	S_7		5	
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		

I_3 and I_6 - Combine
I_4 and I_7 - Combine
I_8 and I_9 - Combine

LALR(1) Parsing Table

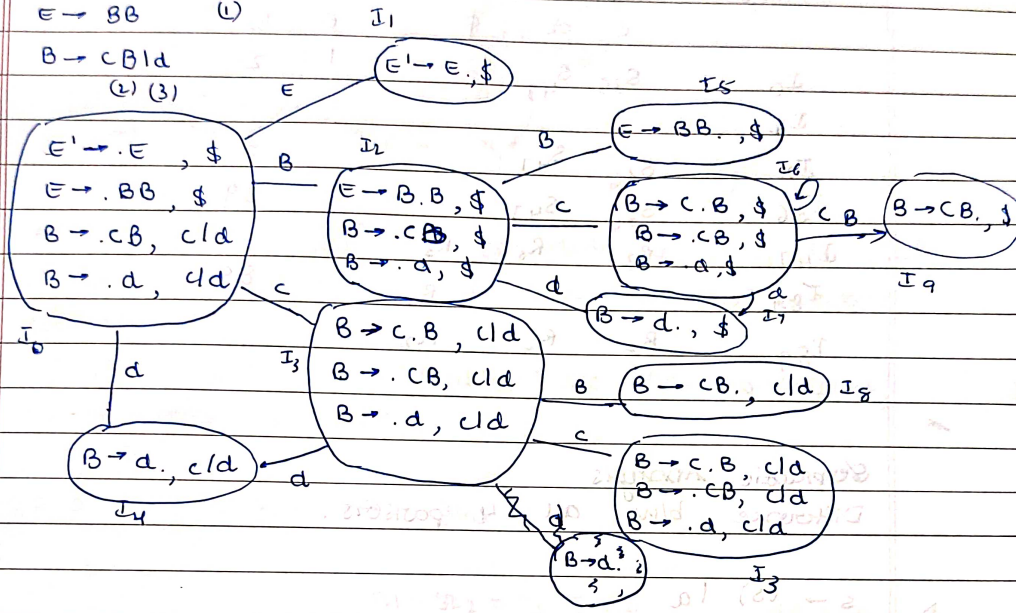
States	Actions			Goto	
	a	b	$	A	S
I_0	S_{36}	S_{47}		2	1
I_1					
I_2	S_{36}	S_{47}		5	
I_{36}	S_{36}	S_{47}		89	
I_{47}	R_3	R_3	R_3		
I_5			R_1		

$I_{8,9}$ R_1 R_2 R_2

4

2) $E \to BB$ (1)

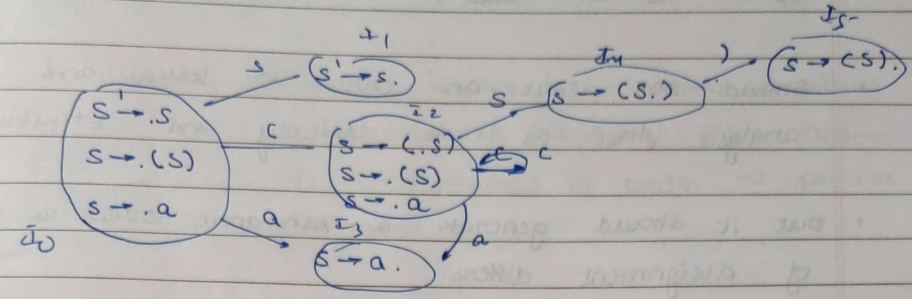
$B \to cB \mid d$ (2) (3)



CLR(1) Parsing Table

States	Actions			Goto		$I_4 - I_7$ = Combine
~~I_0~~	~~c~~	~~d~~	$	~~E~~	~~B~~	$I_3 - I_6$ = Combine
I_0	S_3	S_4		1	2	$I_8 - I_9$ = Combine
I_1						
I_2	S_6	S_7			5	
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			

2) $S \rightarrow (S) \mid a$



LALR : Same as CLR with one difference, i.e., in CLR if 2 states differ only in look ahead then combine those states.

After minimisation if Parsing Table has no conflict, the grammar is LALR.

LALR may contain RR conflict.

LR(0) = n_1

SLR = n_2

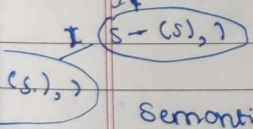
LALR = n_3

CLR = n_4

$n_1 = n_2 = n_3 \leq n_4$

Operator Precedence Parser:

It generates the Parse tree for from given grammar & string but the only condition is 2 consecutive non-terminals & (ε) never appear on RHS of Production



Semantic Analysis

Semantics of a language provide meaning to its constructs, like tokens and syntax structure. Semantics help interpret symbols, their types and their relations with each other.

CFG + Semantic Rules = Syntax Directed Definitions.

Ex: int a = "value";

- Should not issue an error in lexical and Syntax analysis phase, as it is lexically and structurally correct.
- but, it should generate a semantic error as the type of assignment differs.

These rules are set by the Grammar of the Language and evaluated in semantic Analysis.

Tasks of Semantic Analysis:

→ Scope Resolution: Checks the scope of the variables and also checks the scope Resolution operator.
→ Type checking: It checks the ~~typ~~ compatiable types in variables.
→ Array-bound checking
→ Flow control check: Keeps a check that control structures are used in a proper Manner (Ex: no break statement are outside a loop)

Semantic Errors

Some of semantic Errors that Semantic Analyzer recognizes are:

- Type Mismatch
- Undeclared variable
- Reserved identifier misuse
- Multiple declaration of variable in a Scope
- Accessing an out of Scope variable
- Actual & Formal parameter Mismatch.

Attribute Grammar :

It is a special form of Context-Free Grammar where some additional information (Attributes) are appended to one / more of its non-terminals in order to provide context-sensitive information.

Each attribute has well-defined Domain of values, such as integer, float, character, string and expressions.

Attribute grammar is a medium to provide semantics to the Context-Free grammar and it can help specify the Syntax and semantics of programming language.

Ex: $E \longrightarrow E + T$ { E.value = E.value + T.value }

- The RHS of CFG contains the semantic Rules that specify how the Grammar should be interpreted.

Here, the values of Non-terminals E & T are added toget and result is copied to the Non-terminal E.

Semantic attributes may be assigned to their values from their domain at the time of Parsing and evaluated at the time of Assignment.

Based on the way Attributes get their values, there are 2 Categories:

- Synthesized attributes
- Inherited attributes

Synthesized Attributes:

These attributes get values from the attribute values of their child nodes.

Ex: $S \rightarrow ABC$

If S is taking values from its child nodes (A, B, C), then it is said to be Synthesized attribute, as the values of ABC are Synthesized to S.

- Synthesized attributes never take values from their parent nodes or Any ~~and~~ sibling nodes.

Inherited Attributes:

Inherited attributes can take value from parent (or) siblings.

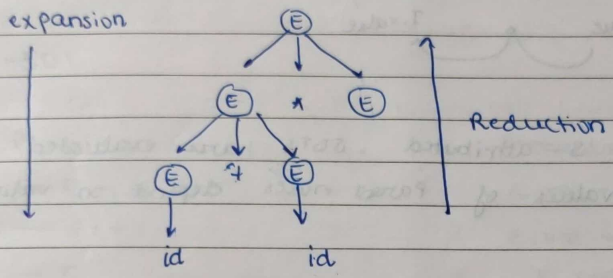
Ex: $S \rightarrow ABC$

'A' can get value from S, B and C.
B can take values from S, A, and C
C can get values from S, A and B.

→ Expansion: when a non-terminal is expanded to terminals as per a Grammatical Rule

→ Reduction: when a terminal is Reduced to its corresponding non-terminal according to Grammar Rules.

NOTE: Syntax trees are parsed Top-down and left to right.
Whenever Reduction occurs, we apply its corresponding Semantic Rules.



Semantic Analysis uses Syntax Directed Translations that facilitate in Semantic Analysis.
Expansion & Reduction

Semantic Analyser receives AST (Abstract Syntax tree) from Syntax Analysis and attached attribute information with AST, which are called Attributed AST.

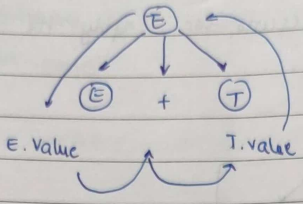
Attributes are 2 tuple values <attribute name, Attribute value>

Ex: int value = 5;
<type, "integer"> <name, 5>
<present value, "5">

S-Attributed SDT

If an SDT uses only synthesized attributes, it is called as S-attributed SDT.

E.value = E.value + T.value



Attributes in S-attributed SDTs are evaluated in Bottom-up Parsing, as values of Parent nodes depend on values of child node.

L-Attributed SDT:

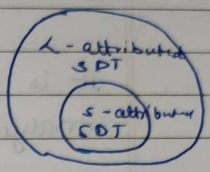
This form of SDT uses both Synthesized and inherited attributes with restriction of not taking values from Right siblings.

In L-attributed SDTs, a Non-terminal can get values from its parent, child and left sibling nodes only.

Ex: $S \rightarrow ABC$

- S can take values from A, B and C (Synthesized)
- A can take values from only S.
- B can take values from S and A
- C can get values from S, A, B.

Attributes in L-attributed SDTs are evaluated by Depth-First and left to Right Parsing.
Top-down



S-SDT

Production	Semantic Rules
L → E n	L.val = E.val
E → E1 + T	E.val = E1.value + T.value
E → T	E.val = T.value
T → T1 * F	T.value = T1.value * F.value
T → F	T.value = F.value
F → (E)	F.value = E.value
F → digit	F.val = digit.lexval.

L-SDT

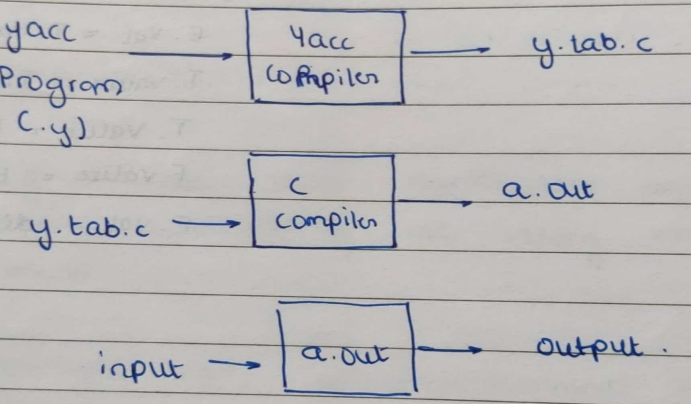
Production	Semantic Rules
D → TL	L.in = T.Type
T → int	T.type = integer
T → real	T.type = real
L → L id	L.in = L.in
L → id	addtype (identity, L.in)

Symbol Table.

Yacc - Yet another Compiler - Compiler

It is used to produce the source code of the Syntax Analyser of the Language Produced by LALR(1) grammar.

It is a Parser Generation. The input of Yacc is the rules / grammar along with lexemes / tokens. And the output is a C program.



Structure of Yacc Program:

It consists of 3 sections:

→ Declaration Section
→ Rules Section
→ Auxiliary Functions

Declaration %{ }%

%%. declaration / definitions

Rules %{ }%

%%. %token

%%.

Rules

%%.

Auxilary Function

→ declaration Section: This part of YACC has 2 sections; both are optional. The first section has ordinary C declarations delimited by %{ and %}. Any temporary variable used by the 2nd & 3rd section will be kept in this part.

Ex: %{

#include <stdio.h>

%}

%token digit

Declaration of grammar tokens also takes part in this ~~dig~~ section. This part defines the tokens that can be used in later parts.

→ Rules section: After the first %%, we place the translation Rules. Every rule has a grammar production and associated semantic action.

<head> : <Body 1> {<semantic action> 1}

| <body 2> {<semantic action> 2}

:

| <body n> {<semantic action> n}

;

Ex: S : A S B { printf("valid"); }
| ε
;

In semantic action, the symbol $$ is considered to be an attribute value associated with head Non-terminal.

→ Auxiliary Functions:

It is the last section of the YACC program and it includes the necessary C functions needed for the program to be executed along with few yacc functions such as yyparse(), yyerror etc.

Meta characters of Yacc

→ $$ - is the attribute value of the left hand Production
→ $i - is the attribute value of the RHS of Production.
→ y.tab.h - is the header file that has to be included, so that the yacc program gets to know the tokens used and accessed from the lex program.

Symbol Table

It is an important data structure created and maintained by the compiler in order to keep track of semantics of variables i.e., it stores information about the scope and binding information about ~~names, information about~~ instances of various entities such as variable and function names, classes, objects etc.

→ It is built-in lexical and syntax analysis phase.

→ The information is collected by the analysis phases of the compiler and is used by the synthesis phases of the compiler to generate code.

Analysis phases : lexical analysis, syntax analyser, semantic analyser.

Synthesis phases : intermediate code generator, code optimiser, target code generator.

→ It is used by the compiler to achieve compile-time efficiency.

→ It is used by various phases of compiler as :

- Lexical Analysis : Creates new table entries in the table, for ex : like entries ~~for~~ about tokens.
- Syntax Analysis : Adds information regarding attribute type, scope, dimension, line of reference, use in the table.
- Semantic Analysis : Uses available information in the table to check for semantics i.e., to verify that expressions and assignments are semantically correct and update it.

- Intermediate Code generation: Refers Symbol Table for knowing how much and what type of run-time is allocated and Table helps in adding temporary variable information.
- Code optimization: Uses information present in the Symbol Table for Machine-dependent optimization.
- Target Code Generator: Generates code by using address information of identifier present in the Table.

Symbol Table Entries:

Each Entry in the Symbol Table is associated with attributes that support the compiler in different phases.

Items Stored in Symbol Table:

→ Variables names and Constants
→ Procedure and function names
→ Literal Constants and Strings
→ Compiler generated temporaries
→ Labels in source languages

Information used by the compiler from Symbol Table:

→ Data type and name
→ Declaring Procedures
→ Offset in storage
→ If structure or record then, a pointer to structure Table.
→ For parameters, whether parameter passing by value or by Reference
→ Number and Type of arguments passed to Function
→ Base address.

operations of Symbol Table - The basic operations defined on a Symbol Table includes:

Operation	Function
allocate	- to allocate a new empty Symbol table.
Free	- to remove all entries and free storage of symbol table
look up	- to search for a name & return pointer to its entry.
Insert	- to insert a name in a Symbol Table and return a pointer to its entry
Set_attribute	- to associate an attribute with a given entry
get_attribute	- to get an attribute associated with a given entry.

Implementation of Symbol Table

Commonly used data structures for implementing Symbol ~~on~~ Table are:

→ Ordered / unordered List:

- In this method, an array is used to store names and associated information.
- A pointer "available" is maintained at end of all stored records and new names are added in the order as they arrive.
- Insertion is fast O(1), but lookup is slow for large tables - O(n) on Average.

- The Advantage is that it takes minimum amount of space.

→ Linked List:

- A link field is added to each data entry/record. A pointer "First" is maintained to point to the First record of the Symbol Table.
- Insertion is fast O(1), but lookup is slow - O(n)

→ Hash Table:

- In Hashing Scheme, 2 tables are maintained - a hash table and a Symbol Table.
- A Hash Table is an array with an Index Range: 0 to table_size -1. These entries are pointers pointing to the names of the Symbol Table.
- Insertion and Lookup can be very fast - O(1).

→ Binary search Tree:

- Another approach to implement is binary Search Tree.
- We add 2 link fields i.e., left and right child.
- Insertion and lookup are $O(\log_2 n)$ on average.

A Symbol Table stores its entry in :

< Symbol name, type, attribute>

Ex: Static int stats ;

< stats, int, static>

Operations

→ insert() : This operation is more frequently used by analysis phase where tokens are identified and names are stored in the table.

This operation is used to add information in the Symbol table about unique names occuring along with their attribute, value, state and scope.

Ex : int a; is processed as

insert(a, int);

→ lookup() : It is used to search a name in Symbol Table to see :

- if the Symbol exists in the Table
- if it is declared before its use
- if the name is used in Scope
- if Symbol is initialised
- if Symbol is declared multiple times.

lookup(Symbol)

If Symbol exists in Table, it returns its attributes stored in the Table.

Scope Management

A Compiler maintains 2 types of Symbol Tables:

- a global Symbol Table which can be accessed by all Procedures
- a scope Symbol Table that are Created for each scope in the Program.

Ex:

```
int value = 10;
void pro_one()
{
    int one_1;
    int one_2;
    {
        int one_3;      } inner
        int one_4;
    }
    int one_5;
}
void pro_two()
{
    int two_1;
    int two_2;
    {
        int two_3;
        int two_4;
    }
    int two_5;
}
```

Symbol	Type	Scope
Pro_one	Proc	global
Pro_two	Proc	global
value	int Proc	global

Symbol	Type	scope
one_1	int	Proc para
one_2	int	proc para
one_5	int	pro paree

Symbol	Type	Scope
two_1	int	Proc para
two_2	int	proc para
two_5	int	proc para

Symbol	Type	Scope
one_3	int	inner
one_4	int	inner

Scope	Type	Scope
two_3	int	inner
two_4	int	inner

Runtime Environment and Storage Management

By Runtime, we mean a program in execution. Runtime Environment is a State of the Target machine, which may include software libraries, environment variables to Provide Services to the processes running in the System.

Runtime support System is a Package, mostly generated with the executable program itself and facilitates the Process of Communication b/w the Process and the runtime environment. It takes care of the memory allocation and de-allocation while the program is being executed.

Date
Page

Procedure activation

A Program is a sequence of instructions combined into a number of procedures.

Instructions in a procedure are executed sequentially. A procedure has a start and an end delimiter and everything inside is called body of the Procedure.

The execution of Procedure is called its activation.

A activation record contains all the necessary information required to call a Procedure.
They contain

Temporaries : stores temporary and intermediate values of an exp

Local Data : stores local data of the called Procedure.

Machine Status : stores machine status such as Registers, Program Counter etc before the procedure is called.

Control Link : stores the address of activation record of the caller procedure

Access Link : stores the information of data which is outside the local scope.

Actual Parameters : stores actual Parameters i.e., parameters which are used to send input to called Procedure.

Return value : stores return value.

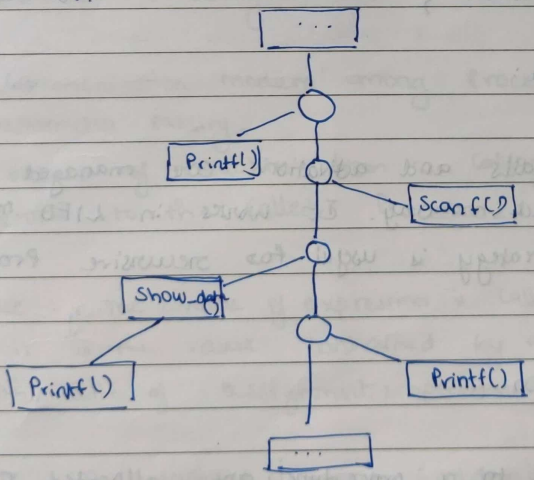
Return value
Actual Parameters
Control Link
Access Link
Machine Status
Local Data
Temporaries

whenever a procedure is executed, its activation Record is stored on the stack, also known as control stack.

A series of activations are easier to represent in form of a tree, known as activation Tree.

```
...
Ex :    printf("Enter Name\n");
        scanf("%s", Name);
        show_data(Name);
        printf("Press any key");
...
        int show_data(char *s)
        {
            printf("Your Name is %s", Name);
            return 0;
        }
...
```

Return value
Actual Parameters
Control link
Access link
Machine Status
Local Data
Temporaries



Activation Tree

Storage Allocation

Runtime environment manages Runtime memory requirements for:

- Code
- Procedures
- Variables

Static Allocation:

In this Allocation Scheme, the compilation data is bound to a fixed location in the memory and it does not change when the program executes.
As the memory requirement and storage locations are known in advance, run-time support Package for memory allocation & de-allocation is not required.

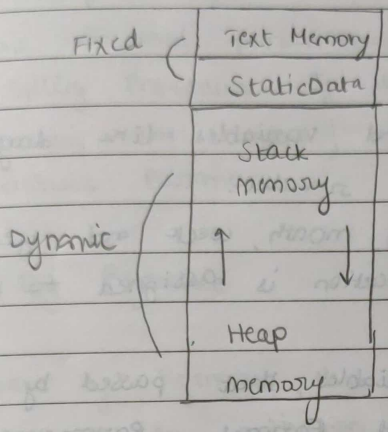
Stack Allocation:

Procedure calls and activations are managed by means of stack memory. It works in LIFO method and this strategy is useful for recursive Procedure calls.

Heap Allocation

Variables local to a procedure are allocated & de-allocated only at runtime. Heap memory is used to dynamically allocate memory to the variables and claim it back when not Required.

Except static allocation memory area, both stack and heap memory can grow / shrink dynamically.



The text part of code is also allocated a fixed amt of memory.

Parameter Passing

→ The communication medium among procedures is known as parameter passing.

→ The values of variables from a calling procedure are transferred to the called procedure.

r-value : The value of expression is called its r-value, and it is the value contained by a single variable on the RHS of assignment operator.

L-value : The location of memory (address) where an expression is stored is known as l-value of that expression. It appears on the LHS of assignment operator.

Ex:
```
day = 1;
week = day * 7;
month = 1;
year = month * 12;
```

Constants like 1, 7, 12 and variables like day, week, month and year all have r-value.
But, only variables day, month, week and year have l-values as memory location is assigned to them.

Formal Parameters: Variables that passed by the Caller procedure are called Formal Parameters. These variables are declared in the definition of the called fn.

Actual Parameters: Variables whose values are being passed to the called procedure are called actual parameters and are specified in the function call as arguments.

Ex:
```
main()
{
    int x = 10, y = 20;
    add(x, y);      → actual, calling
}
void add(int a, int b)   → Formal, called
{
    printf("sum = %d", (a+b));
}
```

Pass by value

The calling procedure passes the r-value of the actual parameters and compiler puts that into the called procedure's Activation Record. Formal parameters then hold the value passed by the calling procedure. If the values held by the Formal Parameters are changed; it should ~~not~~ have no impact on the actual parameters.

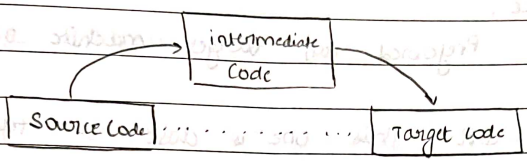
Pass By Reference

In pass by Reference, the l-value of the actual parameter is copied to the activation record of the called procedure. The called procedure now has the address of the actual parameter and formal parameters refer to the same memory location. If the value pointed by Formal Parameter is changed, then it impacts on Actual Parameter as well.

Pass By Copy - Restore

Intermediate Code Generation

Reason why intermediate Code Generator is required.



→ If Compiler translates the Source language to its target machine language without having the option for generating intermediate code, then for each new machine, a full native Compiler is required.

→ Intermediate code eliminates the need of a new full Compiler for every unique Machine by keeping the analysis Portion same for all Compilers.

→ The second part of Compiler Synthesis, is changed according to the target machine.

→ It becomes easier to apply the Source code modifications to improve code Performance by applying code optimization techniques on intermediate code.

Intermediate Representation

They can be represented in a variety of ways and they have their own benefits.

IR - Intermediate Representation

→ High level IR : This is very close to source language itself. They can be easily generated from source code and we can easily apply code modifications to enhance performance.
It is less preferred for target machine optimization.

→ Low level IR : This one is close to target machine, which makes it suitable for register and memory allocation, Instruction set selection etc.
It is good for Machine-dependent optimizations.

Intermediate code can be either language specific (Byte code for Java) or language independent (3-address code).

The following are commonly used intermediate code Representation:

→ Postfix Notation :

The expressions are written in infix notations are

a+b

The Postfix Notation for the same expression placed the operator after the operands

ab+

No paranthesis are needed in Postfix Notation because the position and no of arguments of operators permit only one way to decode postfix expression.

ab- cd+* ab-

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

(a-b) (c+d)

T1 T2 T3 T4

$T_1 = ab-$ $T_2 = cd+$ $T_3 = T_1 T_2 *$ $T_4 = ab-$

ab- cd+ * ab- : Post fix Notation. ab- cd+ * ab- +

$T_3 + T_4$

$T_3 T_4 +$

Three - Code Address

A Statment that has 3 refernces ie., 2 for operands and one for result is known as 3-address statemnt.

A sequence of 3-address statemnt is known as 3-address code.

They are of the form $x = y\ op\ z$ here x, y and z will have address.

Ex: 3-address code for a+b*c+d:

T1 = b*c

T2 = a+b*c T2

T3 = T2+d

T1, T2 and T3 are temporary variables

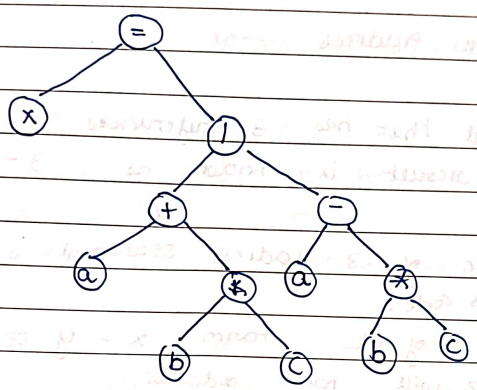
Syntax tree:

It is nothing more than condensed form of a Parse tree. The operator and keyword nodes of the parse tree are moved to their parents and a chain of single productions is replaced by single link in syntax tree.

The internal nodes are operators, child nodes are operands.

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

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



Code optimization

The code optimization in the synthesis phase is a program transformation technique, which tries to improve the intermediate code by making it consume fewer resources (CPU, memory etc) so that faster running machine code will Result.

Objectives of code optimization :

→ The optimization should be correct, it must not, in any way, change the meaning of the Program

→ It should increase speed and Performance of the Program.

→ The compilation time must be kept Reasonable.

→ The optimization Process should not delay the overall Compiling Process.

When to optimize ?

Optimization of code is often performed at the end of the development stage since it reduces readability and adds code to increase performance.

Why optimize ?

Optimizing an algorithm is beyond the scope of the code optimization phase.
So the Program is optimized, it may involve reducing the size of code.

Optimization helps to :

- Reduce the space consumed and increases the speed of compilation.
- Manually performing the optimization is tedious, hence we use a code optimizer.
- An optimised code often promotes re-usuability of code.

Types of code optimization

- Machine Independent optimization :
It attempts to improve the intermediate code to get a better target code as the output. The part of intermediate code which is transformed does not involve any CPU registers or absolute memory locations.

→ Machine Dependent Optimization:

It is done after the target code has been generated and the when the code is transformed, it involves CPU registers and absolute memory references.
It put efforts to take max advantage of memory hierachy.

Different ways of code optimization

1. Compile Time Evaluation:

i) A = 2 * (22.0/7.0) * r

Performs 2 * (22.0/7.0) * r during compile time

ii) x = 12.4
y = x / 2.3

Evaluate x/2.3 as 12.4/2.3 at compile time

2. Variable Propagation

```
c = a * b                    c = a * b
x = a          After →       x = a
till                         till
d = x * b + 4                d = a * b + 4
```

Hence, a * b and x * b will be identified as common sub-expression.

3. Dead Code elimination:

Variable Propagation often leads to making assignment statement into dead code.

```
c = a*b                          c = a*b
x = a          After elimination  till
till                              d = a*b+4
d = a * b+4
```

4. Code Motion:

- Reduce the evaluation frequency of expression.
- Bring loop invariant statements out of the loop.

```
a = 200;                         a = 200;
while (a>0)                      b = x+y;
{                                while (a>0)
    b = x+y;                     {
    if (a%. b==0)                    if(a%. b==0)
    printf("%.d", a);                printf("%.d", a);
}                                }
```

5. Induction variable and strength Reduction:

- An Induction variable is used in loop for following
  i = i + constant
- Strength reduction means replacing the high strength operator by the low strength.

```
i = 1;
while (i<10)
{
    y = i*4;
}
```

```
i=1
t=4
I
    while (t<40)
        y=t;
        t= t+4;
}
```

6. Constant folding :

→ It eliminates the expressions that calculate a value before code execution.

Ex : $i < 320 * 200 * 32$

$\downarrow$

$i < 2048000$

Basic Blocks

Source code generally have a no of instructions, which are always executed in sequence and are considered as the basic blocks of the code.

Basic Block Identification

They are important for code optimization and code generation.

Source code

```
w=0;
x = x+y;
y = 0;
if (x>z)
{
    y=x;
    x++;
}
else
{
    y=z;
    z++;
}
w= x+z;
```

Basic Blocks

B1
```
w=0;
x = x+y;
y = 0;
if (x>z)
```

B2
```
y=x;
x++;
```

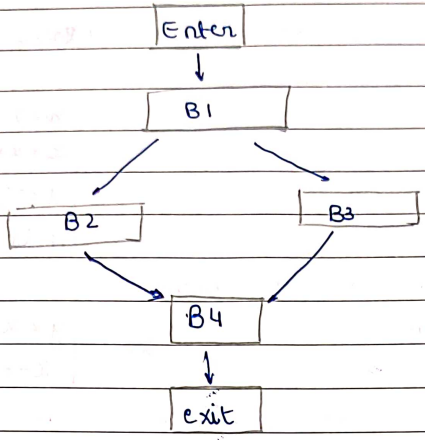
B3
```
y=z;
z++;
```

B4
```
w= x+z;
```

They play an imp role in identifying variables, which are being used more than once in a single block.

Control flow graph.

Basic blocks in a Program can be represented by means of control flow graphs. It depicts how the program control is being passed among the blocks. It helps in optimization by locating any unwanted loops.



Phases of optimization(scope of optimization)

Local optimization: The optimization which occurs within the basic block.
Techniques used in local optimization are:

1) Local Common Subexpression elimination

$x = y * z + k$
$A = y * z - m$
→
$t_1 = y \times z$
$x = t_1 + k$
$A = t_1 - m$

2) Dead code elimination within the basic block.

```
if (0)
{
    printf ("Dead");
}
```

Control would never execute

3) Constant Propagation

$x = 10$
$A = B + C$
$z = x + y$
→
$z = 10 + y$.

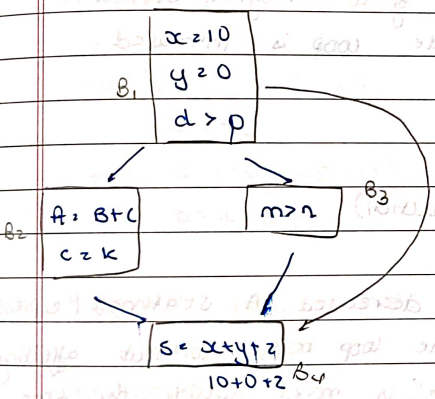
- Algebraic Laws (associative, Commutative)
- Reordering statements.

Global Optimization

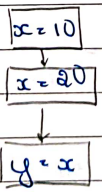
Transformations are applied to large Program segment that includes functions, Procedures and loops between the basic blocks.

The optimization are extended to entire Control flow graph :

1) Constant Propagation blw basic blocks



2) Dead code elimination



→ eliminate.

Loop optimization: Comes under Machine independent optimization

It is the process of increasing execution speed and reducing the overheads associated with loops.

It plays an imp role in improving cache performance and making effective use of parallel processing capabilities.

Most execution time of a scientific program is spent on loops.

Decreasing the no of instructions in an inner loop improves the running time of a program even if the amt of code outside loop is increased.

Techniques used are:

→ Code motion (Frequency Reduction)

The amt of code in loop is decreased. A statement / expression which can be moved outside the loop body without affecting the semantics of the program, is moved outside the loop

```
while(i<100)
{
    a = sin(x) / cos(x) + i;
    i++;
}
```

→

```
t = sin(x) / cos(x)
while (i<100)
{
    a = t+i;
    i++;
}
```

→ Loop Unrolling:

It helps to optimize the execution time of a Program. We basically remove or reduce iterations. It increases the Program's speed by eliminating loop control instruction and loop test instructions

```
for (int i=0; i<3; i++)
    printf(" Pankaj \n");
```

→

```
Printf(" Pankaj \n");
Printf(" Pankaj \n");
Printf(" Pankaj \n");
```

→ Loop Jamming (Fusion):

It is combining the 2 or more loops in a single loop. It reduces the time taken to compile many no of loops.

```
for(int i=0; i<5; i++)
    a=i+5;
for(int i=0; i<5; i++)
    b=i+10;
```

```
for(i=0; i<5; i++)
{
    a=i+5;
    b=i+10;
}
```

Peep-hole optimization

The small set of instructions or small part of code is called peephole or window.

Peep-hole optimization is performed on a small part of the code, very small set of instructions in a segment of code.

It basically works on the theory of replacement ~~in~~. part of code is replaced by shorter and faster code without change in output.

Peephole is a Machine Dependent optimization:

The objectives of Peephole Optimization is

→ To improve Performance
→ To reduce memory footprint
→ To Reduce Code size

Optimization Techniques:

1) Redundant load and Store elimination: In this, redundancy is eliminated

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

```
MOVE b, R0               MOVE b, R0
ADD  c, R0               ADD  c, R0
MOVE R0, a      →        MOVE R0, a
MOVE a, R0               ADD  e, R0
ADD  e, R0               MOVE R0, d
MOVE R0, d
```

2) Strength Reduction

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

$x^2 \Rightarrow x * x$

$x * 2 \rightarrow$ left shift $\quad x << 2$

$x / 2 \Rightarrow$ Right shift $\quad x >> 2$

3) Simplify Algebraic expressions

```
a = a+0
a = a*1
a = a/1
a = a-0
```

4) Unreachable code

```
int add (int x)
{
    return (x+10);
    printf(" ");
}
```

6) Flow of Control Optimization

```
MOV R1, R2          R1 = R2          MOV R1, R2
GOTO L1                              GOTO L2
                        →
                                     L2 : INCR R.

L1 : goto L2
L2 : INCR R
```

Instruction Scheduling (For Pipeline)

Instruction scheduling is one of the compiler optimization passes that occurs in the back end of the code generation.

→ Front End → optimization → Back end
- Instruction selection
- Instruction Scheduling → Pre-Pass
- Register Allocation
- Instruction Scheduling - Post-Pass

It is typically for a compiler to have 2 instruction Scheduling passes, one before & one after Register Allocation.

Instruction scheduling

The main purpose of this is to Reorder instructions, in such a way that there is less load when executing in Pipeline.
i.e., if a load is an example of an instruction that has long latency.

Ex: If a write instruction is followed by Read instruction then the CPU has to fetch the value before it writes into it.

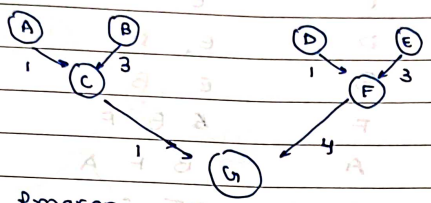
A load instruction has a typically a latency of two cycles of 3-4 cycles.

A : Load I 5 → r_1
B : Load a → r_2
C : add r_1, r_2 → r_3
D : Load I 7 → r_4
E : Load y → r_5
F : mult r_4, r_5 → r_6
G : add r_3, r_6 → r_7.

If we consider the load latency has 3 cycles and mult Latency is 4 cycles, and rest of latency of 1 cycle.

Assuming it is a single issue processor, the reordering of the instructions that done based on the target machine Model.

we construct a Data Dependency graph before reordering the instructions



C ∝ A, B
F ∝ D, E
G ∝ C, F

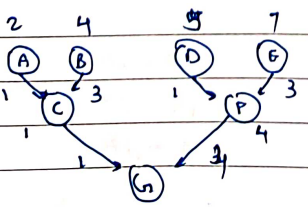
Original Program Order

1st cycle : A
2nd cycle : B (takes 3 cycles)
3rd cycle : stall
4th cycle : stall
5th cycle : C
6th cycle : D
7th cycle : E (takes 3 cycles)
8th cycle : stall
9th cycle : stall
10th cycle : F (takes 4 cycles)
11th : stall
12 : stall
13 : stall
14 : G

In total, it takes 14 cycles.

The code optimizer will minimize the no of cycles by reordering the instructions.
It can be done through a technique called List Scheduling which keeps track of Ready List.

It goes with the critical path, distance first of an instruction x is the length of the longest path from x to a leaf instruction



	Ready List	Instr	Active list
1 :	A B D E	E	E
2 :	A B D	D	E, D
3 :	A B	B	E B
4 :	A F	F	B F
5 :	A	A	B F A
6 :	C	C	F C
7 :	Stall		F
8 :	G	G	G

Hence the stall is just one after re-ordering and is executed by 8 cycles with the help of pipeling.

Register Allocation

Registers are the fastest locations in the memory hierarchy, but their resource is very limited.

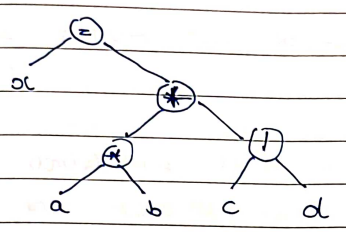
The Register Allocation of a Compiler maps the virtual registers to Physical registers for storage and reuse.

The Register allocation optimization takes care of the register Pressure as there are limited no of registers physically Present on the Machine.

Let us consider an example of an expression

$x = a * b + c / d$

Drawing a Abstract Syntax tree, we get.



The linear Assembly of instructions will be

Load a → r_1
Load b → r_2
Mul ~~load~~ r_1, r_2 → r_3
Load c → r_4
Load d → r_5
Div r_4, r_5 → r_6
Add r_3, r_6 → r_7
Store r_7 → x.

The virtual registers required are 7 registers. But if we are using a Compile which has only 3 Physical registers that can be used. Then we apply register allocation technique to use it in the optimized way.

Assuming that the Target machine has only 3 Physical registers P_1, P_2, P_3. we can see how the optimization technique works

For that we need to keep track of live-range is the time of definition and its usage instruction.

The Main aim of Register allocation is to use min no of Physical registers and to reuse registers for optimization.

$r_1 \rightarrow P_1$
$r_2 \rightarrow P_2$
$r_3 \rightarrow P_1$
$r_4 \rightarrow P_2$
$r_5 \rightarrow P_3$
$r_6 \rightarrow P_2$
$r_7 \rightarrow P_1$
$P_1 \rightarrow x$.