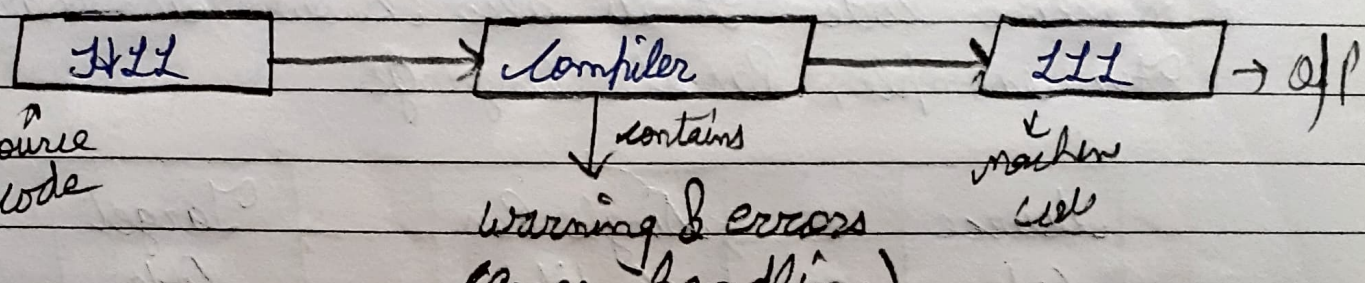
Compiler Construction: It is a process of ~~containing~~ constructing a compiler. It is also known as translator. It means, it converts the high level language into low level language.



Compiler: A compiler is a computer program which helps you transform source code written in HLL into LL machine language. It translate the code written in one programming language to some other language without changing the meaning of the code. The compiling process include basic translation mechanism and error detection. The compiler process goes through lexical (tokens), syntax and semantic analysis ~~and~~ at the front end and code generation and optimization at the back end.

Features of Compiler

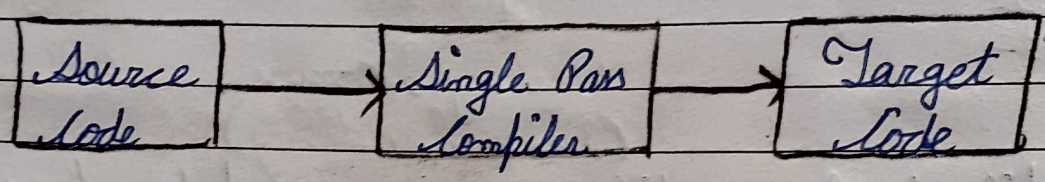
1) Correctness
2) Speed of compilation
3) Preserve the correct meaning of the code.
4) Good error reporting or handling.
5) Compiler should recognize the legal and illegal program.
6) Compiler should help in debugging the error.

Types of Compiler

The compiler are divided into two categories :

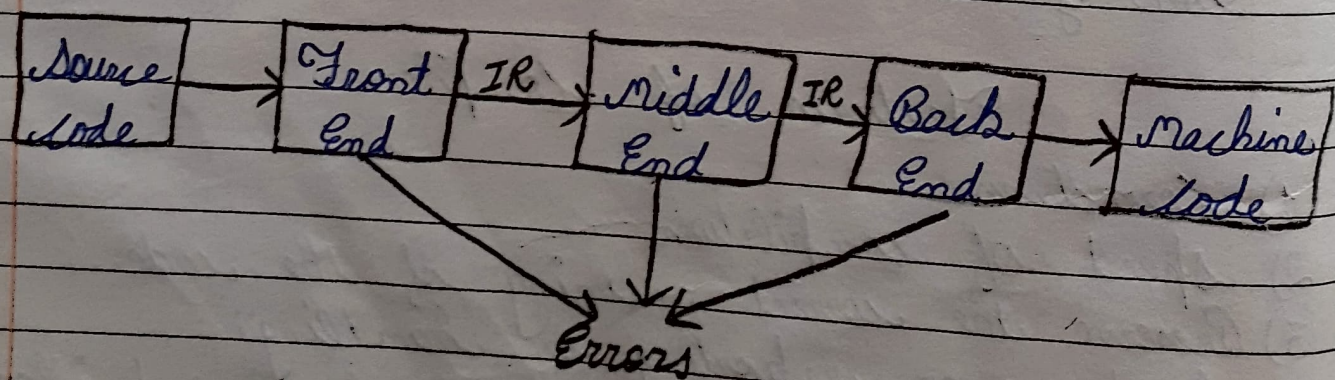
1) Single Pass Compiler
2) Multi Pass Compiler

1) **Single Pass Compiler** : In single pass compiler source code directly transforms into machine code. It is also known as narrow compiler. For example, Pascal language.



actual code ie machine language / native code

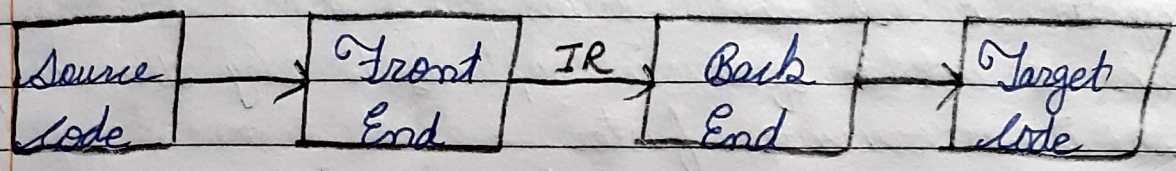
2) **Multipass Compiler** : The multipass compiler processes the source code or syntax tree of a program several times. It divided a large program into multiple small programs and process them. It develops multiple intermediate codes. All of these multipass take the output of the previous phase as an input. So, it requires less memory. It is also known as 'Wide Compiler'.



3) **Two pass Compiler** : Two pass Compiler is divided into two sections,

a) Front end: It maps the legal code into Intermediate Representation (IR).

b) Back end: It maps IR onto the target machine.



The two pass compiler method also simplifies the retargeting process. It also allows multiple front ends.

Pass: It is an implementation of compiler, one or more phases are combined into a module called a pass. The number of passes and grouping of phases into passes depends on two features:

1). Structure of the source language.

2). The environment in which compiler operates.

Tasks of Compiler

The main tasks performed by the compiler are:

a). Breaks up the source program into pieces and impose grammatical structure on them.

b) Allows you to construct the desired target program from the IR and also create the symbol table.

c) Compiles source code and detects errors in it.

d) Manage storage of all variables and codes.

e) Translating the source code into object code (target code) depending upon the type of machine.

Why use a Compiler?

- We use compiler because,

1) Compiler verifies entire program, so there are no syntax or semantic errors.
2) The executable file is optimized by the compiler so it is executes faster.
3) Allows you to create internal structure in memory
4) There is no need to execute the program on the same machine it was built.
5) Translate entire program in other language.
6) Generate files on disk.
7) Check for syntax errors and data types.
8) Helps you to enhance your understanding of language semantics.
9) Helps to handle language performance issues.
10) Link the files into an executable format.

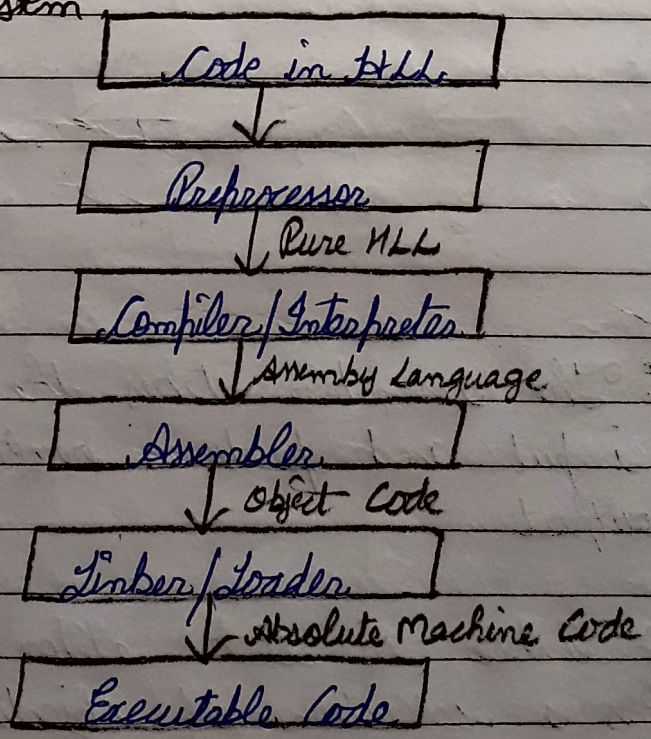
Application of Compilers

1) Compiler design helps full implementation of High-Level Programming Language.
2) Support optimization for Computer Architecture Parallelism.
3) Widely used for Translating Programs.
4) Design of New Memory Hierarchies of Machines.
5) Used with other software productivity tools.

Language Processing System

The compiler is an intelligent combination of software and hardware. Hardware is simply a piece of mechanical equipment and its functions are being compiled by the relevant software.

The hardware considers instructions as electronic charge, which is equivalent to the binary language in software programming. The binary language has only 0s and 1s. Writing such code would be an inconvenient and complicated task for computer programmers, so we write programs in a high-level language, which is convenient for us to comprehend and memorize. These programs are then fed into a series of devices and OS components to obtain the desired code that can be used by the machine. This is known as a language processing system.

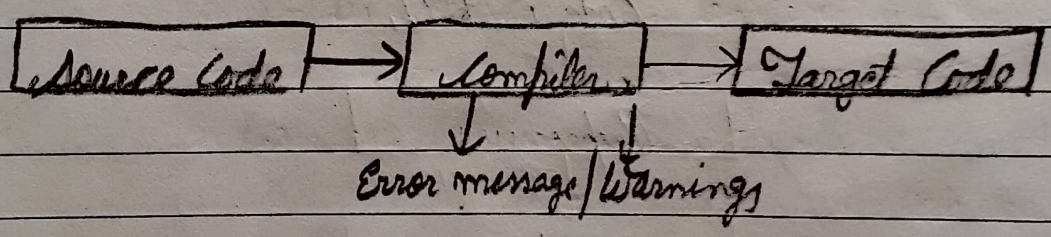


Components of Language processing system: As seen in the above diagram there are the following components:

- **Preprocessor**: The preprocessor includes all header files and also evaluates whether a macro is included. It takes source code as input and produces modified source code as output. The preprocessor is also known as a macro evaluator, processing is optional that does not support #include and macros processing is not required.

Macro: A macro is a piece of code that is given a name. Whenever the name is used, it is replaced by the contents of the macro by an interpreter or compiler. The purpose of macros is either to automate the frequency used for sequences or to enable more powerful abstraction.

- **Compiler**: The compiler takes the modified code as input and produces the target code as output.



- **Assembler**: The assembler takes the target code as input and produces real locatable machine code as output.

- **Linker**: A linker or link editor is a program that takes a collection of objects (created by assemblers and compilers) and combines them into an executable program.

- **Loader**: The loader keeps the linked program in the

main memory.

- **Executable Code**: It is low level and machine specific code and machine can easily understand. Once the job of linker and loader is done then object code finally converted into the executable code.

Difference between Linker and Loader

Linker	Loader
1) The linker is part of the library files.	The loader is part of an operating system.
2) The linker performs the linking operation.	The loader loads the program for execution.
3) It also converts user defined functions to user-defined libraries.	Loading a program involves reading the contents of an executable file in memory.
4) It uses an input of object code produced by the assembler and compiler.	It uses an input of executable files produced by the linker.
5 The foremost purpose of a linker is to produce executable files.	The foremost purpose of a loader is to load executable files to memory.
6. It is used to join all the modules.	It is used to allocate the address to executable files.

Difference between Compiler and Interpreter

Different Style	Compiler	Interpreter
	• Write a program in source code.	• Write a program in source code.
1. Programming steps	• Compiler will analyze your program statements and check their correctness if an error is found in a program, it shows an error message. If the program contains no error then compiler will convert the source code program into machine code.	• Its linking of files happens so one machine code will generate separately
	• The compiler links all the code files into a single runable program which is known as the exe file. Finally, it runs the program and generate output.	• The source code programming statements are executed line by line during the execution. If an error is found at any specific statement interpreter stops further execution until the error gets removed.
2. Translation Type	A compiler translates complete HLL programming code into machine code at once.	An interpreter translates one statement of programming code at a time into machine code
3. Running Time	A compiler takes more time to analyze source code. However, overall compiled programming code runs faster as compared to an interpreter.	An interpreter takes less time to analyze source code. However, overall interpreted programming code runs slower as compared to an compiler.

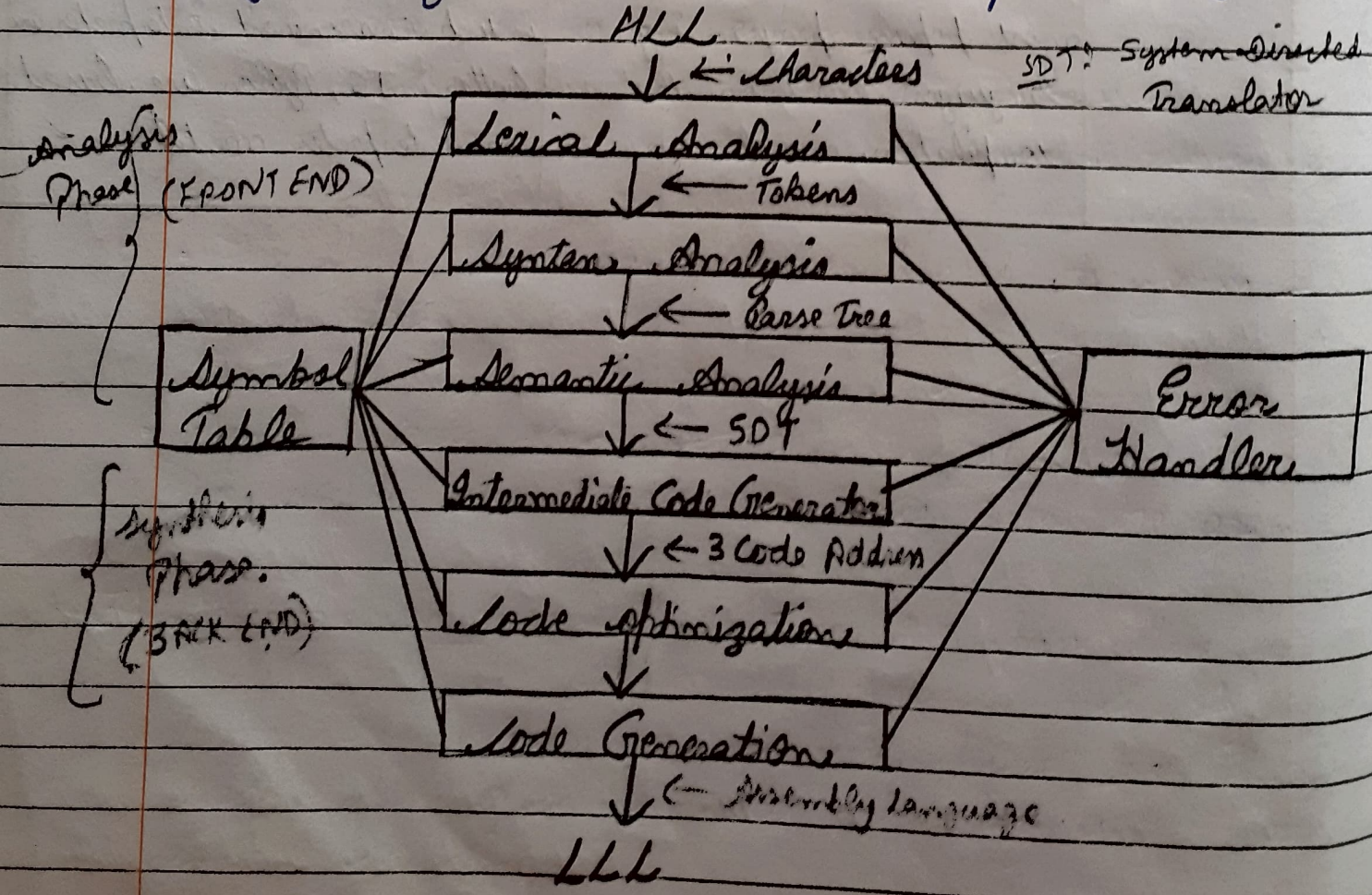
Different Style	Compiler	Interpreter
4. Program Generation	It generates an output of a program in the form of an exe file that can run separately from source code program.	It does not generates a separate machine code as an output program. So, it checks the source code every time during the execution.
5. Memory Management	A compiled program is generated into an intermediate object code, and it further required linking. So, there is a requirement for more memory	An interpreted program does not generate an intermediate code. So, there is no requirement for extra memory.
6 Best suited for	The compiled program is bounded to the specific target machine. It requires the same compiler on the machine to execute. C and C++ are the most popular programming languages based on the compilation model.	In web environments, compiling takes relatively more time to run even small code which may not run multiple times. As load time is essential in web environment interpreter are better. JS, Python are based on interpreter model.

Phases of Compiler

The compilation process contains the sequence of various phases. Each phase takes source program in one representation and produces output in another representation. Each phase takes input from its previous stage.

We have basically two phases of compilers, namely
1) Analysis Phase
2) Synthesis Phase

- The Analysis phase creates an intermediate representation from the given source code.
- The Synthesis phase creates an equivalent target program from the intermediate representation.



Symbol Table: It is a data structure being used and maintained by the compiler, consisting of all the identifier's names along with their types. It helps the compiler to function smoothly by finding the identifiers quickly.

The analysis of a source program is divided into mainly three phases:

1) **Linear Analysis**: This involves a scanning phase where the stream of characters is read from left to right. It is then grouped into various tokens having a collective meaning.
2) **Hierarchical Analysis**: In this analysis phase, based on a collective meaning, the tokens are categorized hierarchically into nested groups.
3) **Semantic Analysis**: This phase is used to check whether the components of the source program are meaningful or not.

The compiler has two modules namely

- Front-end that constitutes the Lexical Analysis, syntax analysis, semantic analysis and intermediate code generator.
- Back-end constitutes the code optimization and code generation.

1) **Lexical Analyzer**: It is also called scanner. It takes the output of the preprocessor (which performs file inclusion and macro expansion) as the input which is in a pure HLL. It reads

the characters from the source program and groups them into lexemes (sequence of characters that "go together"). Each lexeme corresponds to a token.

Tokens are defined by regular expressions which are understood by the lexical analyzer. It also removes lexical errors like comments, and white space.

2) Syntax Analyzer: It is sometimes called a parser. It constructs the parse tree.

- It takes all the tokens one by one and uses Context-Free Grammar to construct the parse tree.
- Parse tree are generally constructed to check for ambiguity in the grammar. There are certain rules associated with a derivation tree.
  - Any identifier is an expression.
  - Any number can be called an expression.
  - Performing any operations in the given expression will always result in an expression.
  - The parse tree can be compressed to form a syntax tree.

- Syntax error can be detected at this level if the input is not in accordance with the grammar.

3) Semantic Analyzer: It varifies the parse tree, whether it's meaningful or not. It furthermore produces a verified parse tree. It also does type checking, Label checking and Flow control checking.

4) **Intermediate Code Analyzer**: It generates intermediate code, which is a form that can be readily executed by a machine. We have many popular intermediate codes.

- Example: Three address codes etc.
- Intermediate code is converted to machine language using the last two phases which are platform dependent.
- Till intermediate code, it is the same for every compiler out there, but after that, it depends on the platform.
- To built a new compiler we don't need to build it from scratch.
- We can take the intermediate code from the already existing compiler and build the last two parts.

5) **Code optimization**: It transforms the code so that it consumes fewer resources and produced more speed. The meaning of the code being transformed is not altered.

- Optimization can be categorized into two types: machine-dependent and machine-independent.

6) **Code Generator**: The main purpose of the Target Code generator is to write a code that the machine can understand and also register allocation, instruction selection, etc.

- The O/P is dependent on the type of assembler. This is the final stage of compilation.
- The optimized code is converted into relocatable machine code which then forms the input to the linker and loader.

Lexical Analysis

Token: A lexical token is a sequence of characters that can be treated as a unit in the grammar of the programming language. For example, identifiers, keywords, operators, special symbols, constants and punctuations.

eg `int a = 2;`

id — keyword (variable) — operator — constant — punc

Tokens = 5 (spaces not consider)

eg
```
int main ()
{
    int a, b;
    a = 20;
    b = 10;
    c = a+b;
    return c;
}
```

Tokens = 28.

⇒ `printf("%d, c");`

⇒ `printf("%d, c", c);`

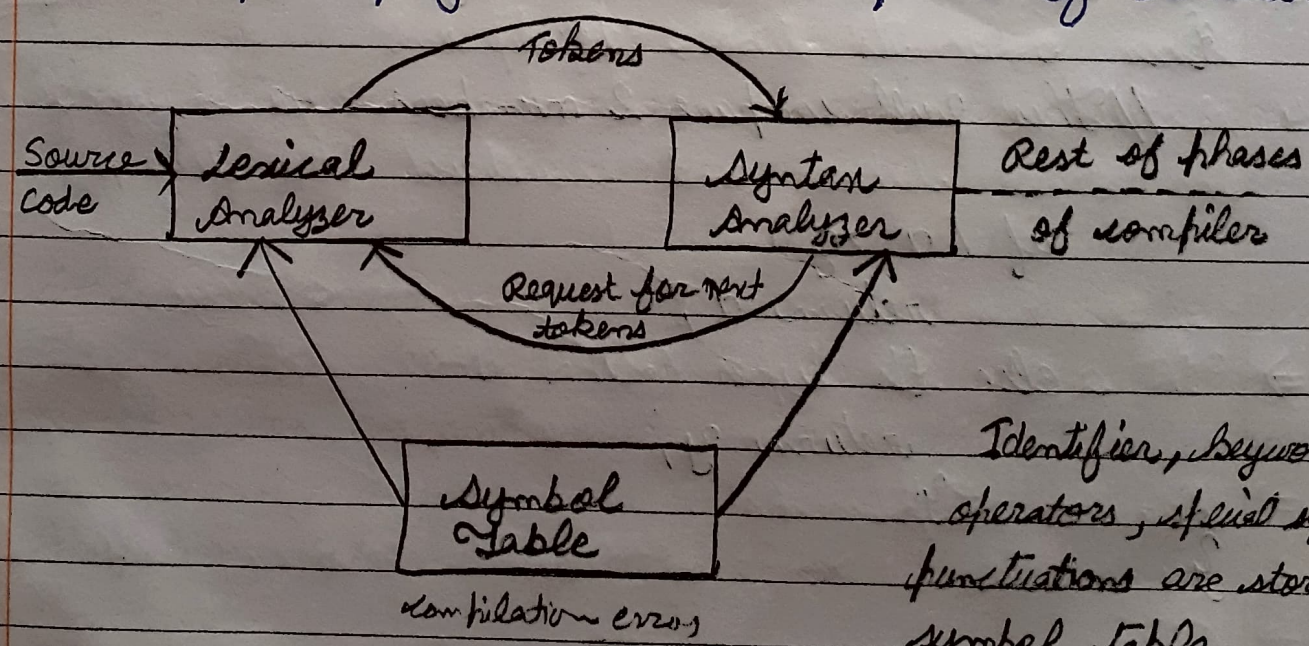
Lexemes: A lexemes is a sequence of characters in the source program i.e. matched by the pattern for a token.

Pattern: A set of strings in the I/P for which the same tokens is produced as O/P. This set of string is described by a rule called a pattern associated with the token.

e.g int a = 2 ;

Token	Lexeme
Identifier	int
Keyword	a
Operator	=
Constant	2
Punctuation	;

Lexical Analysis: It is the first phase of the compiler also known as scanner. It converts HLL input program into a sequence of tokens.



Identifier, keyword, operators, special symbols, punctuations are stored in symbol table

It takes source code as input. It reads the source program one character at a time and converts it into meaningful lexemes. Lexical Analyzer represents these lexemes in the form of tokens.

Function of Lexical Analyzer

1. Read input character from source program.
2. Dividing the program into valid tokens. This process is known as tokenization.
3. Remove white space character (/, \)
4. Remove comments (#, //, /*, */)
5. Helps to identify tokens into the symbol table.
6. It generate lexical errors.

⇒ Lexical analyzer skips whitespaces and comments while creating the tokens. If any error is present, then Lexical analyzer will correlate that error with the source file and line number.

Example of Lexical Analysis, Tokens, Non-Tokens

```
# include <stdio.h>
    int maximum (int x, int y)
{
  //This will compare 2 numbers

      if (x > y)
          return x;
      else {
          return y;
      }
}
```

Tokens created? Lexeme	Token
int	Id Keyword
maximum	Id
(	operator
int	Id Keyword

Lexeme	Tokens
x	identifier
,	operator
int	keyword
y	identifier
)	operator
{	operator
if	keyword

Non Tokens:	Type	Example
1.	Comment	// This will compare 2 numbers
2.	Pre-processor Directive	# include <stdio.h>
3.	Macro	NUMS
4.	Whitespace	/n/b/t

Lexical Errors: A character sequence (sequence of characters) which is not possible to scan into any valid token is a lexical error.

Facts about Lexical error:

- Lexical errors are not very common, but it should be managed by a scanner.
- Misspelling of identifiers, operators, keyword are considered as lexical errors.
- Generally, a lexical error is caused by the appearance of some illegal character, mostly at the beginning of a token.

Advantages of Lexical analysis:

1. This method is used by programs like compilers which can use the parsed data from a programmer's code to create a compiled binary executable code.
2. It is used by web browsers to format and display web page with the help of parsed data from JS, HTML, CSS
3. It helps you to construct a specialized and potentially more efficient processor for the task.

Disadvantages:

1. You need to spend significant time reading the source program.
2. Some regular expressions are quite difficult to understand as compared to PEG or EBNF rules.
3. More effort is needed to develop and debug the lexer and its token description.
4. Additional runtime overhead is required to generate the lexer tables and construct the tokens.

⇒ FA and its types (DFA, NFA)

⇒ Regular Expressions

⇒ CFG.

BNF
(Backus Naur Form)

In 1960 John Backus and Peter Naur introduced formal ~~definition~~ notation method for describing syntax of programming language which is known as Backus Naur Form (BNF)

- BNF was basically design for ANGOL 60.
- BNF and CFG was nearly identical.
- BNF is a meta language ~~for~~ used to describe the grammar of a programming languages.

Meta Language: It is a language that is used to describe the other language.

BNF symbols to describe a syntax:

::= means 'is defined as'

< > means 'can be described as'

| means or.

Example: <assign> → <var> = <expression>

<assign> ↓ terminal (Abstraction) (LHS)

<expression> ↓ Non-terminals (tokens, lexemes and patterns) (RHS)

Terminal: Terminal are the symbols that can appear in the output of a language because of its rules. But cannot be changed by rules themselves. For example +, =, {, }, *, /, if, while, int, double, float, etc.

Non-Terminal: It can be replaced and they are string composition of terminals and non-terminals. It is a syntatic entity that defines

a part of grammar. For example: sentences, expressions, words, statements, programs, etc.

- How to write rules in BNF?

<Non-terminals> ⟶ {<Terminal>} {<Non-terminal>}
start symbol

OR

<NT> ::= {<T>} {<NT>}
one or more terminals — one or more non-terminals

If statement:

```
if (condition)
{
body
}
```

In BNF,

<if statement> ::= <if part>
<if part> ::= if (<condition>) <body>

OR

<if statement> ::= if {<condition>} <body>
Terminal — Non-terminal

If else statement:

<if statement> ::= if {<condition>} {body} | else <body>

OR

<if statement> ::= <if part> <else part>
<if part> ::= if {<condition>} {<body>}
<else part> ::= else {<body>}

Recursion in BNF

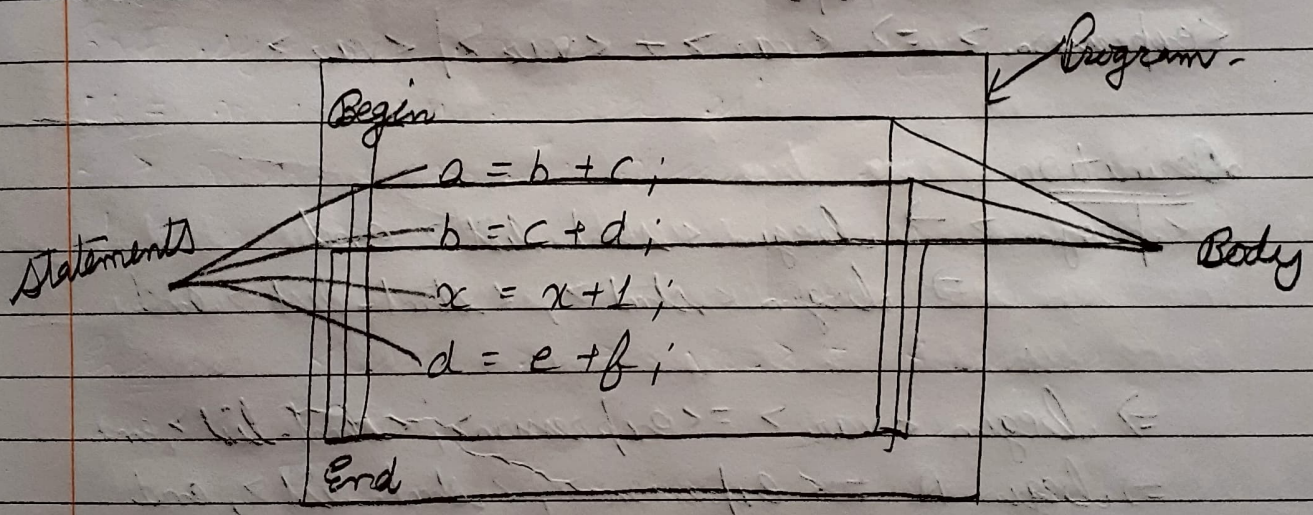
Recursion is used in BNF to write production rules to define 'one or more' of a symbol. e.g, a number can be made up of one or more digits. You may start out by writing a regular production rule:

<number> ::= <digit> | <digit> <digit> | <digit> <digit> <digit> | ...

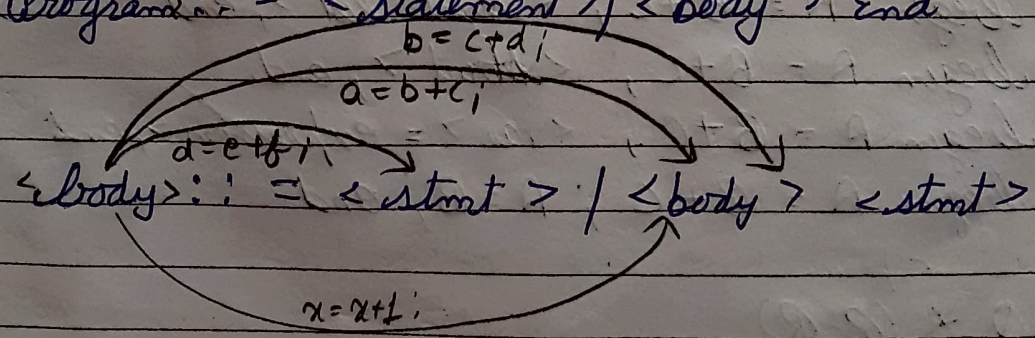
But where would you stop? How many digits can a number have? Recursion allows an elegant solution:

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

<digit> ::= 0|1|2|3|4|5|6|7|8|9.



<Program> ::= <statement> | <body> End



<body> ::= <stmt> | <body> <stmt>

Grammar and Derivation in BNF :

A grammar is generative device for defining languages. The sentences of the languages are generated through a sequence of applications of the rules beginning with a special non-terminal of the grammar called start symbol.

⇒ The sequence rule application is called derivation.

eg : A = B+C ; A=B

<prg> ⇒ begin <stmt-list> end

<stmt_list> ⇒ <stmt> | <stmt> <stmt-list>

<stmt> ⇒ <var> = <expression>

<var> ⇒ A|B|C

<expression> ⇒ <var> + <var> | <var>

Derivation:

<program> → begin <stmt-list> end

⇒ begin <stmt> ; <stmt-list> end

⇒ begin <var> = <expression> ; <stmt-list> ; end

⇒ begin A = <expression> ; <stmt-list> ; end

⇒ begin A = <var> + <var> ; <stmt-list> ; end

⇒ begin A = B + C ; <stmt-list> ; end

⇒ begin A = B + C ; <stmt> ; end

⇒ begin A = B + C ; <var> = expression> ; end

⇒ begin A = B + C ; A = B ; end

ii) $\underbrace{P}_{\text{identifier}} = \underbrace{Q * (P-R)}_{\text{expression}}$

Sol. <expression> ⇒ <id> = <expression>

⇒ P = <expression>

⇒ P = <id> * <expression> $P = \underbrace{Q}_{\text{identifier}} * \underbrace{(P-R)}_{\text{Expression}}$

⇒ P = Q * <expression>

$P = Q * (<expression>)$
$P = Q * (<id> - <expression>)$
$P = Q * (P - <expression>)$
$P = Q * (P - R)$

Hence Proved.

iii) $A = B * (C + D)$

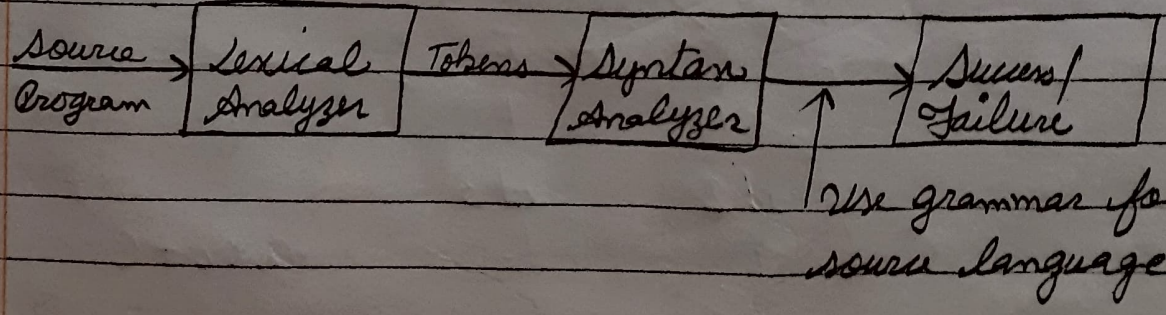
Sol $<expression> \Rightarrow <id> = <exp>$
$\Rightarrow A = <id> * <exp>$
$\Rightarrow A = B * <exp>$
$\Rightarrow A = B * (<exp>)$
$\Rightarrow A = B * (<id> + <exp>)$
$\Rightarrow A = B * (C + <exp>)$
$\Rightarrow A = B * (C + D)$

Hence Proved

OR

Parsing

Parsing also known as syntax analysis can be defined as a process of analysis a text which contains a sequence of tokens, to define its grammatical structure w.r.t given grammar. So, it is a process of deriving a string from a given grammar.



- Parser is a compiler that is used to break the data into smaller elements coming from lexical analysis phase.

A parser takes I/P in the form of sequence of tokens and produces O/P in the form of parse tree.

Functions:

1. A parsing can perform context free syntax analysis.
2. Constructs and intermediate representations in the form of parse tree.
3. Produces meaningful error messages (like ;, {, }, ~}
4. Attempt error correction (but only several errors detected, not all).

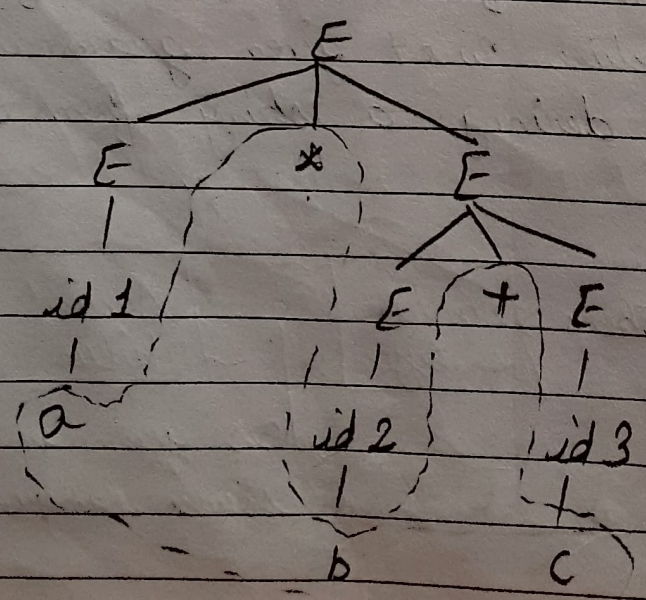
What is parse tree?

Ans: Parse tree is an ordered rooted tree that represents the syntactic structure of a string according to some context free grammar.

eg

Grammar: $E \rightarrow E*E \mid E+E \mid E$

Program: $a*b+c$



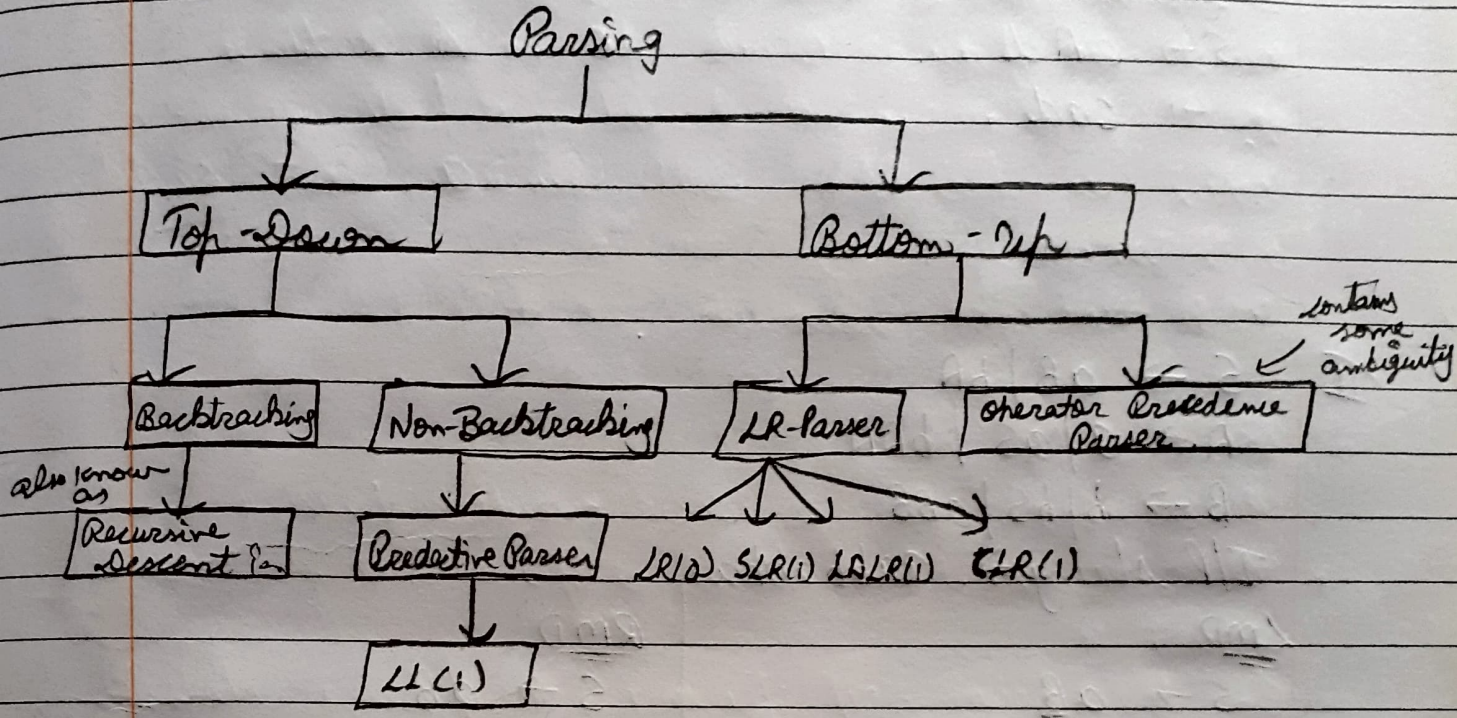
LMD

$a*b+c$

Types of parsing

Parsing is of two types:

1) Top-Down Parsing 2) Bottom-up Parsing



1) **Top-Down Parser**: The process of constructing the parse tree which starts from the root and goes down to the leaf is called top down parser.

When the parser starts constructing the parse tree from the start symbol and then tries to tranform the start symbol to the input, it is called top-down parsing.

- It constructs from grammar which is free from ambiguity and left recursion.
- It uses left most derivation (LMD) to construct a parse tree. It allows a grammar which is free from left factoring.

It generates parse from given i/p string with the help of grammar productions by expanding non-terminals i.e start from the start symbol and ends on the terminal. It uses LMD.

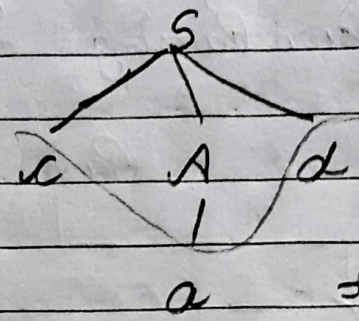
eg $S \to cAd$

$A \to ab|a$

Input string : cad

$S \to cAd$

$\to cad$



$\Rightarrow$ cad

II $S \to aB|bA$

$A \to a|aS|bAA$

$B \to b|bS|aBB$

I/P string $\Rightarrow$ aabbabba

LMD

$S \to aB$

$\to aaBB$

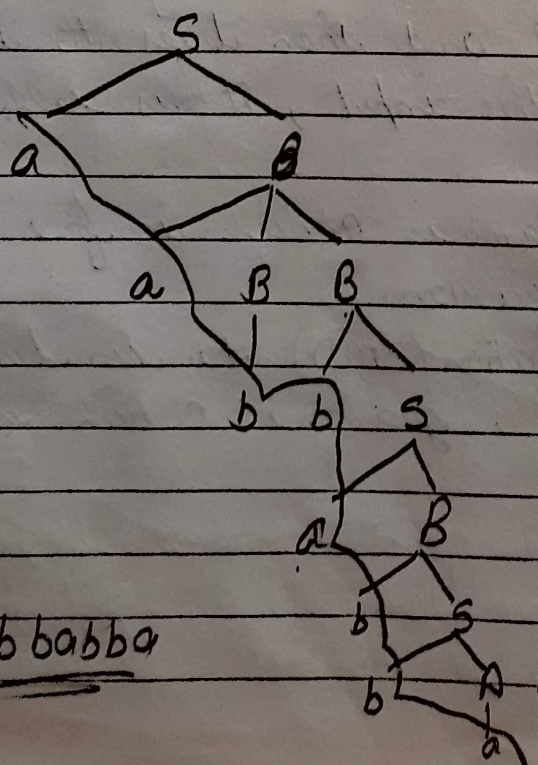
$\to aabbS$

$\to aabbaB$

$\to aabbabS$

$\to aabbabbA$

$\to aabbabba$



$\Rightarrow$ aabbabba

RMD

$S \to aB$

$\to aaBB$

$\to aaBbS$

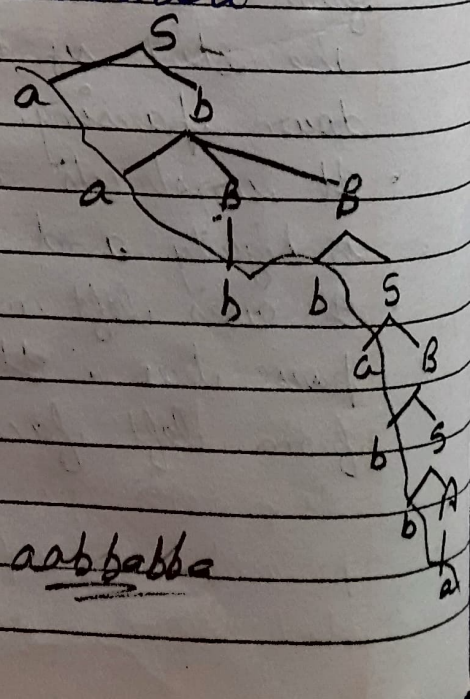
$\to aaBbaB$

$\to aaBbabS$

$\to aaBbabbA$

$\to aaBbabba$

$\to aabbabba$



$\Rightarrow$ aabbabba

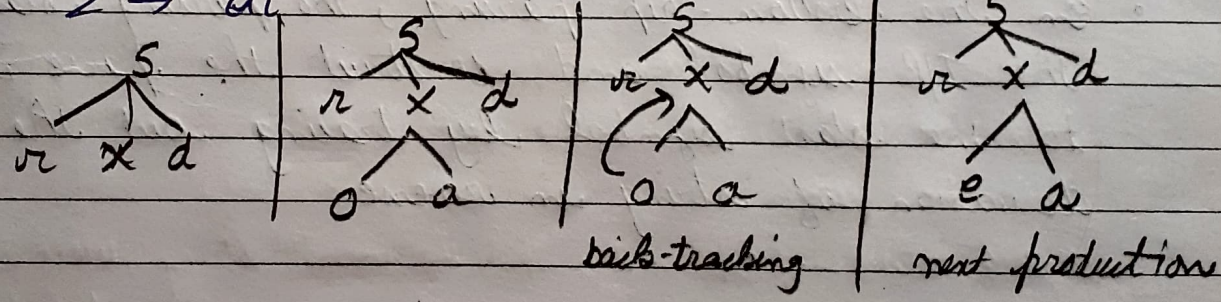
Recursive Descent Parsing: It is a common form of top down parsing. It is called recursive as it uses recursive ~~producedure~~ procedures to process the input. Recursive descent parsing suffers from backtracking.

Backtracking: It means, if one derivation of a production fails, the syntax analyzer restarts the process using different rules of same production. This technique may process the I/P string more than once to determine the right production.

eg $S \rightarrow rXd \mid rZd$

$X \rightarrow oa \mid ea$

$Z \rightarrow ai$



Predictive Parsing / Non-Backtracking: It requires no backtracking. It can determine which A-production must be chosen to derive the input string.

It chooses the correct A-production by looking ahead at the I/P string. It allows looking ahead a fixed number of I/P symbols from the I/P string.

LL(1) Parser: Also known as Non-recursive descent parser
In LL1, the first L stands for Left to Right and second L stands for Left Most Derivation. 1 stands for a number of Look Ahead tokens used by parser while parsing a sentence.

- LL(1) parsing is constructed from the grammar which is free from left recursion, common prefix and ambiguity.
- This is non-recursive.

2) Bottom-up Parser:

- It is also known as shift-reduce parsing.
- It is used to construct a parse tree for an I/P string.
- In the bottom-up parsing, the parsing starts with the I/P symbol and construct the parse tree up to the start symbol by tracing out the rightmost derivation of string in reverse.
- That generates the parse tree for the given I/P string with the help of grammar productions by comprising non-terminals i.e starts from non terminals and ends on the start symbol (uses reverse of RMD).
- LR Parser: It is the bottom-up parser that generates parse tree for the given string by using unambiguous grammar.

LR Types: LR(0), SLR(1), LALR(1), CLR(1).

- Operator Precedence parser: Generates parse tree from given grammar and string but the only condition is two consecutive non-terminal and epsilon never appear on the RHS of any production.

Left Recursion

A production of grammar is said to have left recursion if the left most variable of its RHS is same as variable of its LHS.

- A grammar containing a production having left recursion is called as Left Recursive Grammar.

Direct Left Recursion: There is the production of the form like

$$A \rightarrow A\alpha / \beta.$$

Indirect Left Recursion: First symbol on the RHS of a rule can derive the symbol on the left.

Why to remove the left recursion?
Top-down parser cannot accept the grammar having left recursion. So, we have to remove the left recursion but preserve the language generated by grammar.

$$A \rightarrow A\alpha / \beta$$

Ex: A()
{
A();
{
α;
}
}

Rule: $A \rightarrow A\alpha / \beta$ ← Left Recursion

$A \rightarrow \beta A'$
$A' \rightarrow \alpha A' / \varepsilon$] Remove Left Recursion

eg 1) $E \rightarrow E+T \mid T$

comparing, $E \rightarrow E+T \mid T$ with $A \rightarrow A\alpha \mid \beta$

$\therefore A = E, \alpha = +T, \beta = T$

$A \rightarrow A\alpha \mid \beta$ is changed to $A \rightarrow \beta A'$ and $A' \rightarrow \alpha A' \mid \varepsilon$

$\therefore$ $A \rightarrow \beta A'$ means $E \rightarrow TE'$

$A' \rightarrow \alpha A' \mid \varepsilon$ means $E' \rightarrow +TE' \mid \varepsilon$

LL(1) Parser

A top down parser that uses a one-token lookahead is called an LL(1) parser. The first L indicates that the input is read from left to right. The second L says that it produces a left-to-right derivation.

LL(1)

- L → Left to right scan the I/P
- L → Use LMD
- (1) → No. of lookahead

Construction of Predictive LL(1) Parser

1. Check for left recursion in the grammar, if there is left recursion in the grammar, remove that,
2. Find first() and follow() function.

   First(): If there is a variable, and from that variable, if we try to drive all the strings then the beginning Terminal Symbol is called the First.

   Follow(): What is the Terminal Symbol which follows a variable in the process of derivation.
3. Construct passing tree.
4. Stack implementation
5. Parse the I/P string

⇒ First() function: First(α) is a set of terminal symbol that begins in string derived from alpha.

Rules for calculating First ( ):

R1: For a production rule $X \to \varepsilon$, then
First $(X) = \{\varepsilon\}$

R2: For any terminal symbol a, the
First ( ) $= \{a\}$

R3: For a production rule $X \to Y1, Y2, Y3$, then
$\Rightarrow$ if $\varepsilon \notin$ First (x) then First $(X) =$ First $(Y1)$
$\Rightarrow$ if $\varepsilon \in$ First $(Y1)$ then First $(X) = \{$First $(Y1) - \varepsilon\} \cup \{Y2,$

eg
$S \to A$
$A \to aB \mid Ad$
$B \to b$
$C \to g$

Sol 1) Remove Left Recursion
$A \to Ad \mid aB \Rightarrow A \to A\alpha \mid \beta$
$A \to aBA' \Rightarrow A \to \beta A'$
$A' \to dA' \mid \varepsilon \Rightarrow A' \to \alpha A' \mid \varepsilon$

New productions are:	First ( )
$S \to A$	First $(S) = \{$First $(A) = \{a\}$
$A \to aBA'$	First $(A) = \{a\}$
$A' \to dA' \mid \varepsilon$	First $(A') = \{d, \varepsilon\}$
$B \to b$	First $(B) = \{b\}$
$C \to g$	First $(C) = \{g\}$

Follow( ) function: Follow (α) is a set of terminal symbol that appear immediately to the right of α.

eg $S \to aAc$

Follow(A) = {c} ← Right most terminal

If there is no symbol after A the eg.

$S \to aA$, then

Follow(A) = {S}

Rule for calculating follow() function?

R1: For the start symbol S place \$ in follow(S)

i.e. Follow(S) = {\$}

R2: For any production rule as $A \to \alpha B$ then

Follow(B) = Follow(A)

R3: For any production rule A determines α ($A \to \alpha$) then $B \to \beta$

⇒ if $\varepsilon \notin$ First(β) then, Follow(B) = First(β)

⇒ If $\varepsilon \in$ First(β) then, Follow(B) = {First(B) − ε} ∪ {Follow(A)}

eg

$S \to A$	⇒ Follow(S) = {\$}
$A \to aBA'$	⇒ Follow(A) = Follow(S) = {\$}
$A' \to dA' \mid \varepsilon$	⇒ Follow(A') = Follow(A) = {\$}
$B \to b$	⇒ Follow(B) = Follow(A') = {d}
$C \to g$	⇒ Follow(C) = Not applicable (NA)

First ()	Follow ()
First(S) = First(A) = {a}	Follow(S) = {\$}
First(A) = {a}	Follow(A) = {\$}
First(A') = {d, ε}	Follow(A') = {a}
First(B) = {b}	Follow(B) = {First(A') − ε} ∪ {Follow(A)}
First(C) = {g}	= {d − ε} ∪ {\$}
	= {d}
	Follow(C) = NA.

Table:

Production	First()	Follow()
$S \to A$	$\{a\}$	$\{\$\}$
$A \to aBA'$	$\{a\}$	$\{\$\}$
$A' \to dA' / \varepsilon$	$\{d, \varepsilon\}$	$\{\$\}$
$B \to b$	$\{b\}$	$\{d\}$
$C \to g$	$\{g\}$	$\{NA\}$

Parse Table:

	a	d	b	g	$ (NT)
S	$S \to A$	—	—	—	—
A	$A \to aBA'$	—	—	—	—
A'	—	$A \to dA'$	—	—	$A \to \varepsilon$
B	—	—	$B \to b$	—	—
C	—	—		$C \to g$	—

Stack Implementation: I/P string: abd$

Stack	I/P	Production
S $	abd$	$S \to A$
A $	abd$	$A \to aBA'$
aBA'$	abd$	Pop a
BA'$	bd$	$B \to b$
bA'$	bd$	Pop b
A'$	d$	$A' \to dA'$
dA'$	d$	Pop d
A'$	$	$A' \to \varepsilon$
$	$	Accept

Parse Tree (A)

Syntax Directed Translation (SDT)
(Semantic Analysis Phase)

In SDT, along with grammar it can identify some informal notations and these notations are known as ~~syn~~ semantic rules.

SDT
Grammar + Semantic Rules
- Strings
- Numbers
- Address value

Parse Tree → I/P

↓

Annotated Tree → O/P (Add some text or comments within a tree)

⇒ Attaching attributes to the variables of the CFG and defining semantic action of each production of grammar is called SDT.

Semantic Actions: It is an action that is executed whenever the Parser will recognize the I/P string generated by CFG.

eg $A \rightarrow BC$
{semantic action}

⇒ Semantic action is written in curly braces attached with a production.

☆ Parse tree generated by Syntax Analyser → [Semantic Analysis] → Complete Parse Tree (Semantic checked)

Semantic Analysis

- In SDT, every non-terminal can get 0 or more attributes depending on type of attributes. The attributes can be represented by (VAL).
- The SDT is a context-free grammar.
- It is used to evaluate the order of semantic rules.
- In Translation scheme, the semantic rules are embedded within the right side of the production.
- The position at which an action is to be executed is shown by enclosed between braces { }. It is written within the right side of the production.

Example:

Productions	Semantic Rule (Action
$S \rightarrow E\$$	{Print E.val}
$E \rightarrow E+E$	{E.val = E.val + E.val}
$E \rightarrow E*E$	{E.val = E.val * E.val}
$E \rightarrow (E)$	{E.val = E.val}
$E \rightarrow I$	{E.val = I.val}
$I \rightarrow I\,digit$	{I.val = 10* I.val + lex val}
$I \rightarrow digit$	{I.val = lex val}

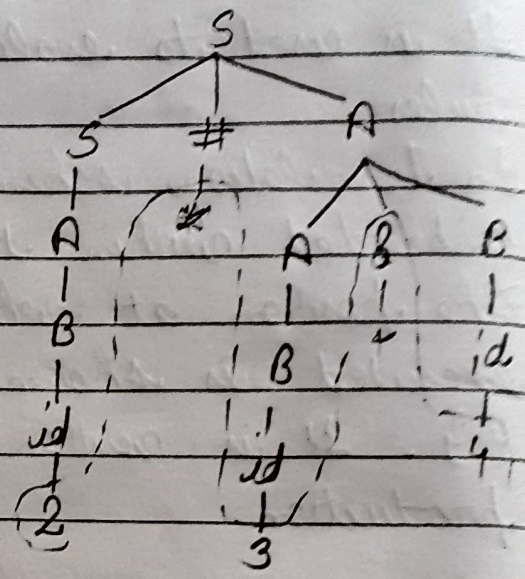
Implementation of SDT:

- SDT is implemented by constructing a parse tree and performing the actions in a left to right, Depth first order.
- SDT is implementing by parse the I/P and a parse tree as a result.

Production	Semantic Rules
$S \to S\#A \mid A$	$\{S.val := S.val * A.val\}$
$A \to A\&B \mid B$	$\{A.val := A.val + B.val\}$
$B \to id$	$\{B.val := id.lexval\}$

Let I/P = 2#3&4

$\Rightarrow 2 * 3 + 4$

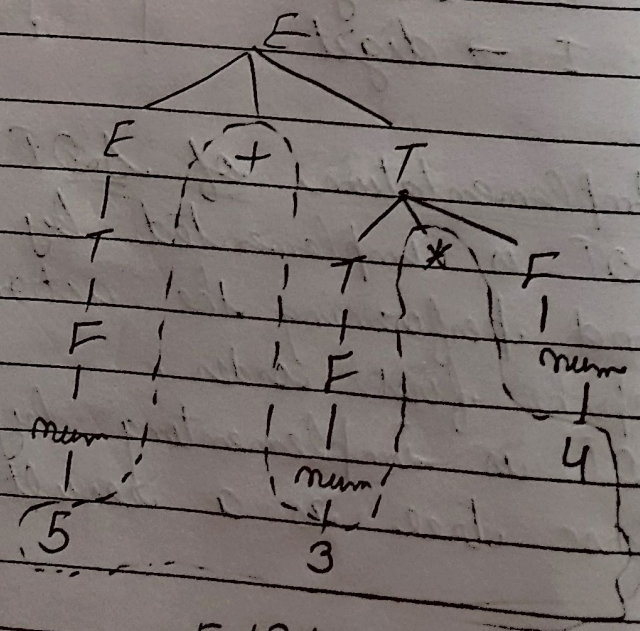


$2 * 3 + 4 = \underline{10}$

e.g

Production	Semantic action
$E \to E+T$	$\{E.val := E.val + T.val\}$
$E \to T$	$\{E.val := T.val\}$
$T \to T*F$	$\{T.val := T.val * F.val\}$
$T \to F$	$\{T.val := F.val\}$
$F \to num$	$\{F.val = num.lexval\}$

I/P : $5 + 3 * 4$



$5 + 3 * 4$

Applications of SDT:

1) Executing arithmetic expressions.
2) Conversion from infix to postfix.
3) Conversion from binary to decimal.
4) Converting number of reductions
5) Converting syntax tree.
6) Generating intermediate code.
7) Type checking.
8) Storing type information into symbol table.

Date 19th April 2022

Evaluation of SDT

SDT involves passing information bottom up and top-down approach the parse tree in form of attributes attached to the nodes.

Evaluation of SDT

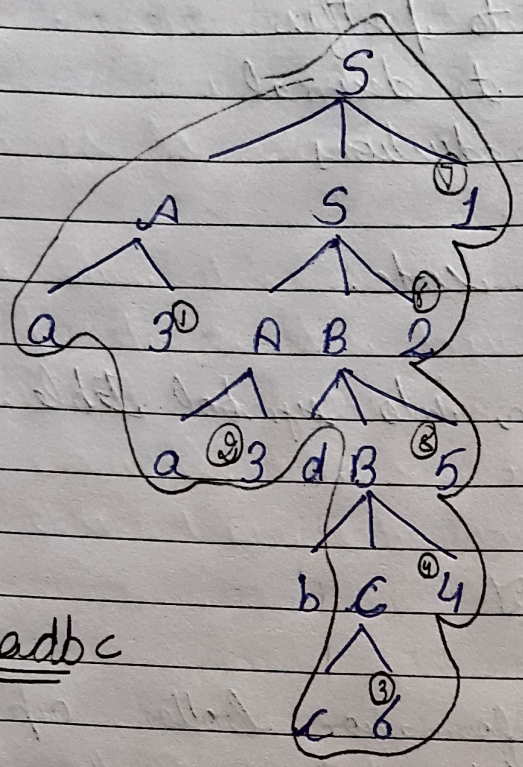
- Top-Down Approach
- Bottom-up Approach

Example:

Production	Rule
$S \rightarrow AS$	PrintF{1}
$S \rightarrow AB$	PrintF{2}
$A \rightarrow A$	PrintF{3}
$B \rightarrow bC$	PrintF{4}
$B \rightarrow dB$	PrintF{5}
$C \rightarrow c$	PrintF{6}

Top-down approach:

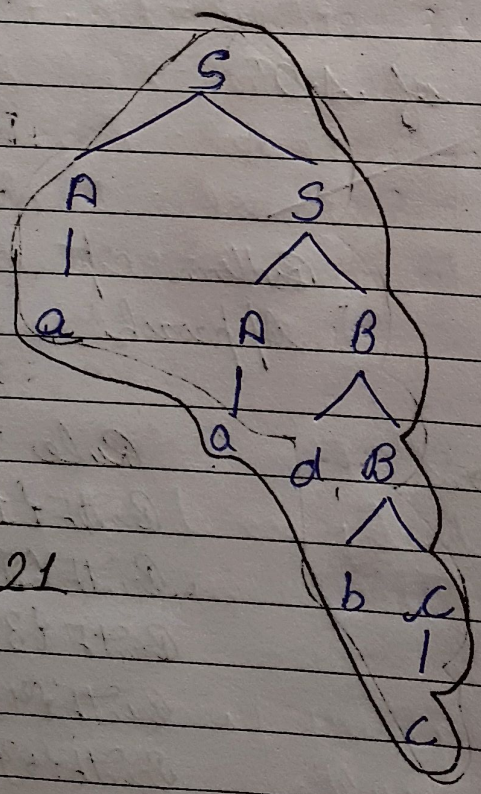
Input: aadbc



3364521

aadbc

Bottom-up approach:



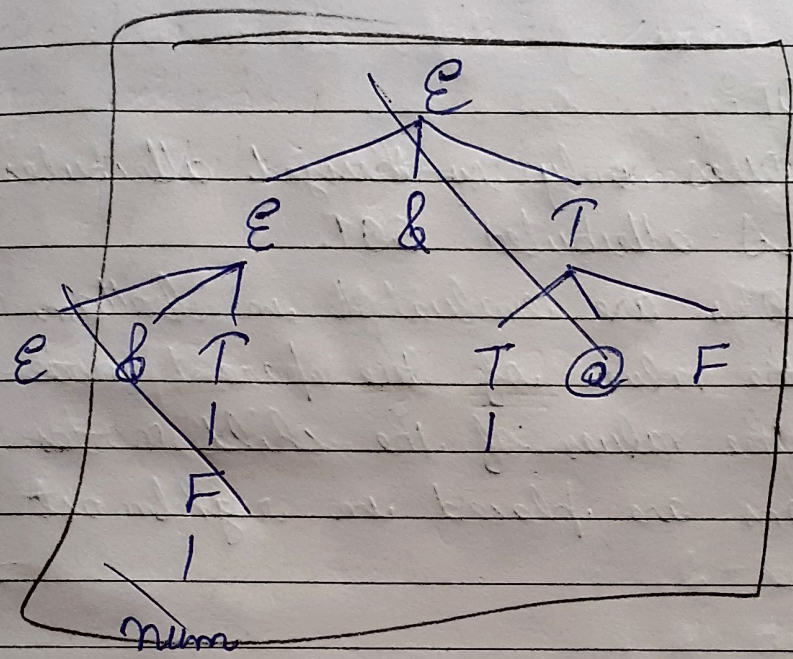
aadbc

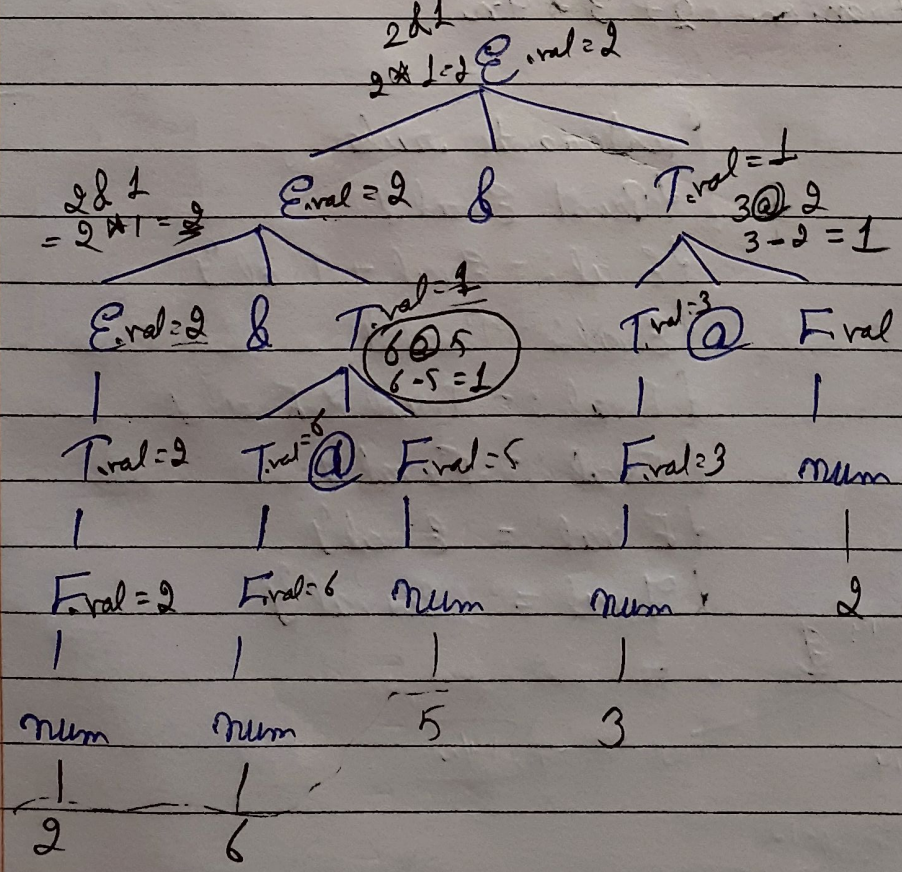
3364521

Eg

Production	Semantic rules
$E \to E \& T \mid T$	$\{E.val := E.val \& T.val\}$
$T \to T @ F \mid F$	$\{T.val := T.val \& F.val\}$
$F \to num$	$\{F.val = num.lexval\}$

I/P: ~~2 & 6 @ 5 @ 3 & 2~~ 2 & 6 @ 5 & 3 @ 2





T.val @ F.val

$6 - 5 = 1$

E.val & T.val

$2 * 1 = 2 \Rightarrow$

$\boxed{E.val = 2}$

T.val @ F.val

$= 3 - 2 = 1$

$\boxed{T.val = 1}$

E.val & T.val

$2 * 1 = 2$

Print $\{E.val = 2\}$

Types of SDT

There are two types of SDT:

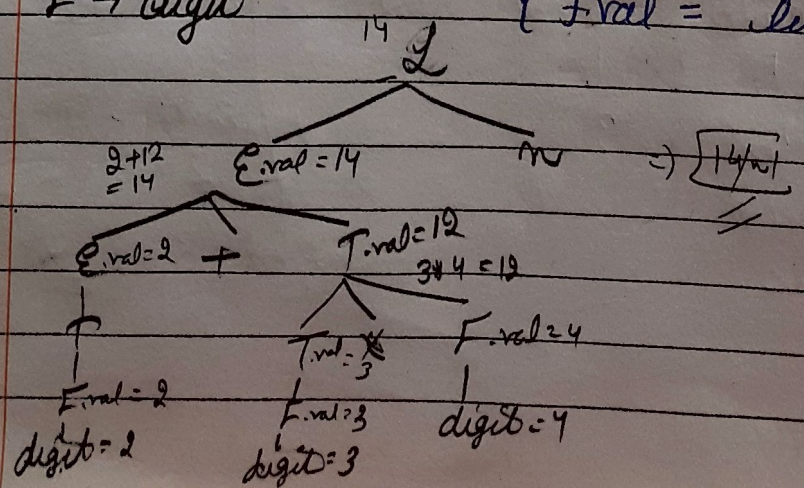
1. S-attribute SDT : Based on synthesized attributes
2. L-attribute SDT : Based on both synthesized and inherited attributes

1) S-attribute SDT :

- If an SDT uses only synthesized attributes, it is called as S-attributed SDT.
- S-attribute SDTs are evaluated in bottom-up parsing, as the values of the parent node depend upon the values of the child nodes.
- Semantic actions are placed in rightmost place of RHS.

Example: Expression: 2+3*4n

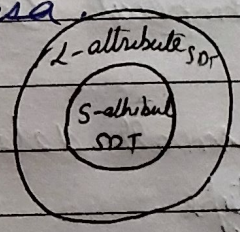
Production	Semantic rule
$L \rightarrow En$	{Print E.val}
$E \rightarrow E+T$	{E.val := E.val + T.val}
$E \rightarrow T$	{E.val := T.val}
$T \rightarrow T*F$	{T.val := T.val * F.val}
$T \rightarrow F$	{T.val := F.val}
$F \rightarrow (E)$	{F.val := E.val}
$F \rightarrow digit$	{F.val = lexval}



2) L-attribute SDT?

- If an SDT used both synthesized attributes and inherited attributes with a restriction that inherited attribute can inherit values from left siblings only, it is called as L-attribute SDTs.
- Attributes in L-attribute SDTs are evaluated by depth first and left-to-right parsing manner.
- Semantic actions are placed anywhere in RHS.

Note: If a definition is S-attributed, then it is also L-attributed, but not vice-versa.

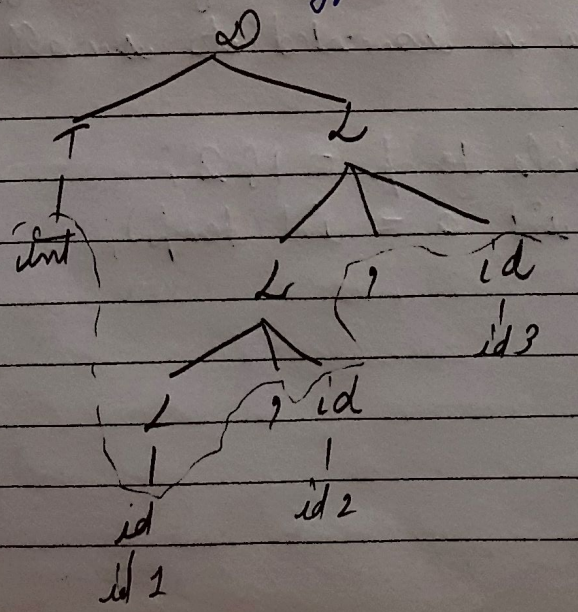


Example: Expression: int id1, id2, id3.

L - inherited
T - synthesized

Production	Semantic rules
$D \to TL$	{L.in := T.Type}
$T \to int$	{T.Type := integer}
$T \to real$	{T.Type := real}
$L \to L, id$	{L.in := L.in}
$L \to id$	{addtype (id.entry, L.in)}
	{addtype (id.entry, L.in)}

L - identifier



int id1, id2, id3.

Type Conversion

Type conversion is a process to convert data type of one variable into another data type. It is done in case when ~~of~~ more than one data type variables are present in one statement.

There are two types of type conversion:

1) Implicit Data Type Conversion
2) Explicit Data Type Conversion

1) **Implicit Data Type Conversion:** It is done automatically by the compiler. In this small data type is converted into large data type.

Example: int a = 2;
char b = "A";
int c = a+b;

O/P: 67

[ASCII Code of A = 65
∴ c = 2+65 = 67.]

2) **Explicit type conversion:** This is done by user manually. In this, the large data type is converted into small data type.

Example: double temp = 18.2;
int a = int(temp);

O/P: 18.

Polymorphic Function

Polymorphic means having many forms. It allows us to perform a single action in different ways. It allows to define one interface with multiple implementation. A function that takes different arguments and executes the code is called polymorphic function.

Types of polymorphic function:

1. Compile Time Polymorphism
2. Run time Polymorphism

1) Compile Time polymorphism: Whenever an object is bound with their functionality at compile time is called compile time polymorphism.

Parts of compile time polymorphism:

a) Function overloading
b) Operator overloading.

a) Function overloading: To overload the function there are two methods:

1). Function contains a different number of argument
2). Function having different type of argument.

Function: Functions are the block of code.

eg
```
sum (int a, int b)
{
    return a+b;
}
```

```
class A
{
    int sum (int num1, int num2)
```

```
{
    return num1 + num2;
}
int sum( int num1, int num2, int num3)
{
    return num1 + num2 + num3;
}
}
int main(void)
{
    A obj;
    cout << obj.sum(20,15);
    cout << obj.sum(10,10,10);
    return 0;
}
```

b) Operator overloading: It allows the same operator name to be used for multiple operations. For example, '+'. In an application where operands are of integer type. In a class like string, the addition is used to concatenate two strings.

eg
```
int x = 5;
int y = 2;
int z = x+y;
```

eg
```
string A = "Compiler"
string B = "construction"
string C = A+B
```

2) Runtime polymorphism: This type of polymorphism can be achieved using method overriding and virtual functions.

Run Time Environment

1. A compiler must accurately implemented the abstractions in the source language definition.
2. Runtime environment is a state of the target machine, which may include software libraries, environment variables, etc., to provide services to the processes running in the system.
3. These abstractions typically include names, scope, binding, data types, operators, procedures, parameters and flow of control construct.
4. The compiler must cooperated with OS and other system software to support these abstraction on the target machine.
5. To do so, the compiler creates and manages a runtime environment in which it assumes its target programs are being executed.

Runtime Environment Issues!

- The layout and allocation of storage location for the objects named in the source program.
- The mechanism used by the target program to access variables.
- The linkage between procedures.
- The mechanism for passing parameters.
- The interface to the OS, I/P, O/P devices and other programs.

Source language issues:

1) **Procedure**: A program is made up of procedure, procedure declarations that associates and identifies with a statement.

Statement: When procedure call. Procedure name appear within an executable statement.

Identifier - Procedure name
Statement - Procedure Body
Procedure call - whenever procedure name appears within an executable statement

eg:

```
main()
{
int n;
read array();
Quick sort (1,n);
}

void Quicksort (int n, int m)
{
int c = position (n,m);
Quicksort (n, i-1);
Quicksort (i+1, m);
}
```

Procedure call

```
void readarray()
{
}

int partition (int y, int z)
{
}
```

2) **Activation Tree**: The activation tree is the way to control and leaves activations.

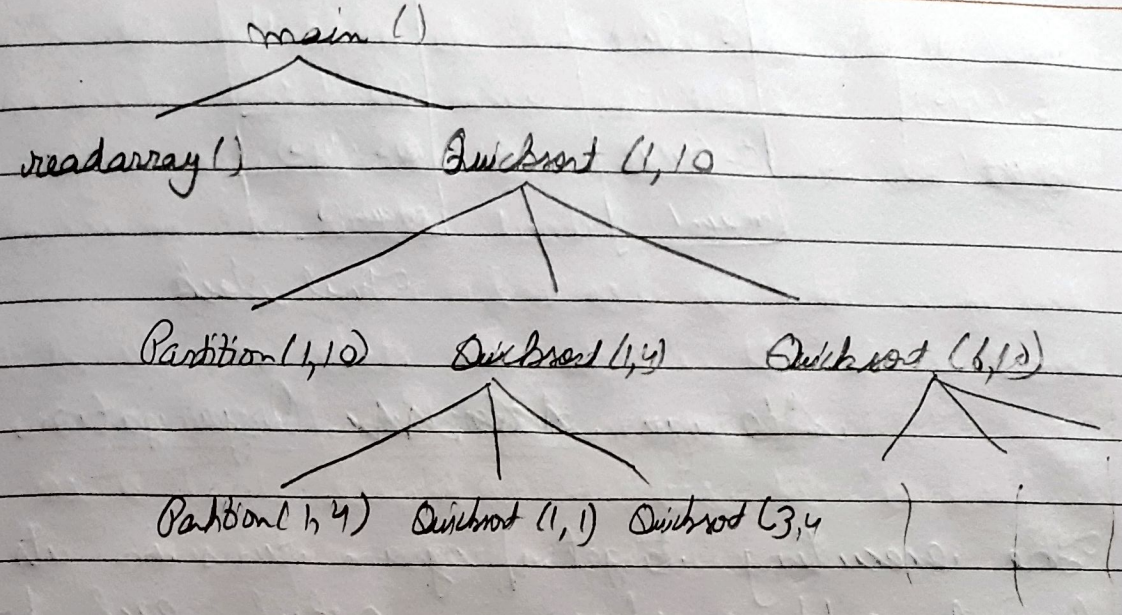
Each node represents an activation of a procedure. The root represents the activation of the main program. If a and b are two procedures the main their activation will be non-overlapping when is called after other nested procedure.

Ways control enters and leaves activation

3) The node for 'a' is the parent of node for 'b' if control flows from activation a to b.

4) The node for a is to the left of node b if the life time of a occurs before lifetime of b.

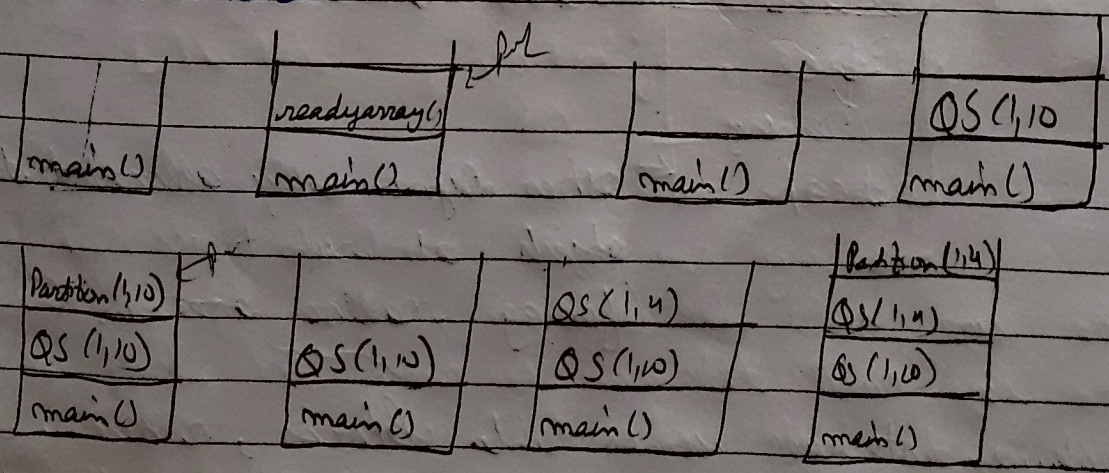
Each node represents an activation of a procedure

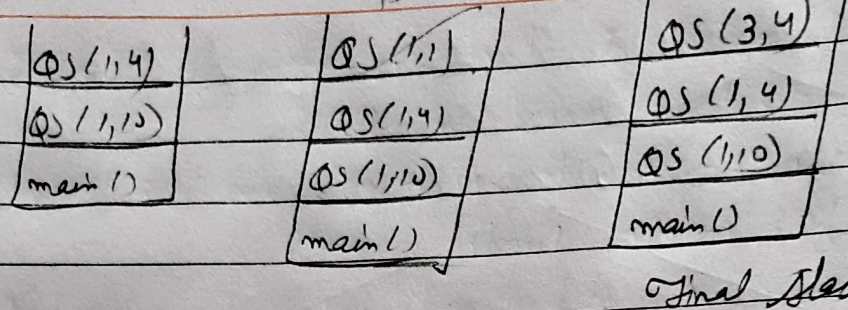


Control Stack

Definition: Keep track of live procedure activation in stack.

- Flow of control in program corresponds to depth first traversal of activation tree.
- Use a stack called control stack to keep track of live procedure activation.
- Push the node when activation begins and pop the node when activation ends.
- When the node n is at the top of the stack the stack contains the node along the path from n to the root.



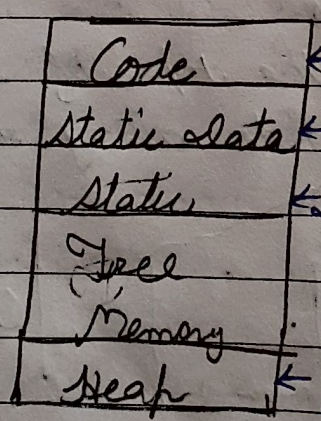


Final Stack

Storage Allocation Organization

- The executing target program runs in its own logical address space in which each program value has a location.
- The management and organization of this logical address space is shared between the compiler, OS and target machine.
- The operating system maps the logical address into physical address, which are usually spread throughout memory.

Subdivision of Runtime Memory:



← memory location are determined at the compile time.

← Location of static data can also be determined at compile time.

← Data object allocated at run time (Activation Records)

← other Dynamically allocated data objects at run-time. (for eg. malloc area in C).

- Runtime storage comes into blocks, where a byte is used to show the smallest unit of addressable memory. using four bytes a machine word can form

- Object of multibyte is stored in consecutive bytes and gives the first byte address.
- Runtime storage can be subdivided to hold the different components of an executing program:
  1) Generated executable code
  2) Static data objects
  3) Dynamic data object - heap
  4) Automatic data objects - stack.

Activation Record

- Control stack is a runtime stack which is used to keep track of the live procedure activations, i.e It is used to find out the procedures whose execution have not been completed.
- When the activation begins then the procedure name will push onto the stack and when returns (activation ends) then it will popped.
- Activation Record is used to manage the information needed by a single execution of a procedure.
- An activation record is pushed into the stack when a procedure is called, and is popped when the control returns to the caller function.
- The contents of Activation Records:

Return value: It is used by calling procedure to return a value to calling procedure.

Actual Parameters: It is used by calling procedure to supply parameters to called procedure.

Control Link: It points to activation record of the caller.

Return Value
Actual Parameters
Control Link
Access Link
saved machine status
Local data
Temporaries

Access Links: It is used to refer to non-local data held in other activation records.

Saved Machine Status: It holds the information about status of machine before the procedure is called.

Local Data: It holds the data i.e. local to the execution of the procedure.

Temporaries: It stores the value that arises in the evaluation of an expression.

Dynamic Memory Allocation

It is the process of assigning the memory space during the execution time or the runtime.

⇒ There are two types of dynamic memory allocation:
1) Static
2) Dynamic

1) Static: This is allocated by the compiler. Exact size and type of memory must be known at compile time. In this, we know how much memory we need is known before the program.
eg int a; , float b; , int a[]

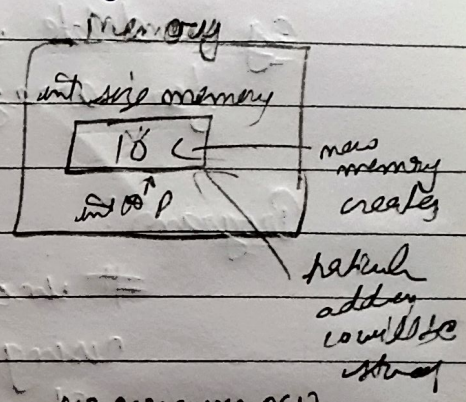
2) Dynamic: Many times we don't know how much memory we will need to store information. The size will be determine at only runtime. The size will determine ~~we~~ with the help of new and delete operator.

New operator: If Memory to be allocated to the variable new operator is used. It creates a

memory and returns memory address to the pointer.

Syntax: new datatype;

eg int *p; // int type pointer created (to hold memory)
p = new int; // p points to int type of memory
*p = 10; // particular location memory address 10 will be stored
cout << *p;

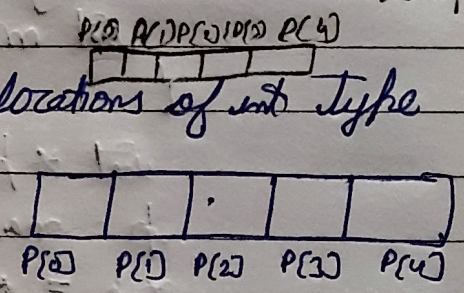


⇒ How to create an array using new operator?

We can also create an array.

int *p

p = new int [5]; // p points to five locations of int type



eg
```
# include <iostream>
using namespace std;
int main()
{
  int *p
  p = new int [5];
  p[0] = 10;
  p[1] = 20;
  p[2] = 30;
  p[3] = 40;
  p[4] = 50;
for (int i = 0 ; i<5 ; i++)
  {
    cout << p[i] << endl;
  }
  return 0;
}
```

P[0]	P[1]	P[2]	P[3]	P[4]
10	20	30	40	50

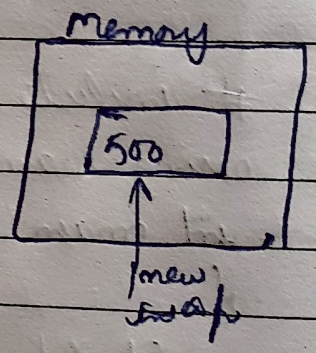
Delete operator: (Deallocate memory). It is used to release memory space when the object is no longer needed.

Syntax: delete pointer_name;

eg delete p;
for arrays: delete []p;

Program:

```
#include <iostream>
using namespace std;
int main()
{
int *p;
p = new int
*p = 500;
cout << "value in p: " << *p << endl;
```

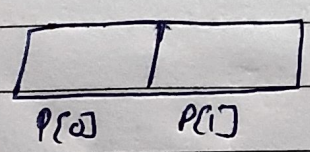



```
//Now I want to delete that value.
delete p;
cout << "value after delete:" << *p << endl;
return 0;
}
```

O/P:
Value in p : 500
Value after delete : 42
↓
any garbage value
(system provided)

Example for arrays:

```
# include <iostream>
using namespace std;
int main()
{
int *p;
p = new int [2];
p[0] = 10;
p[1] = 20;
delete [] p;
cout << "P[0] " << p[0] << endl;
cout << "P[1] " << p[1] << endl;
return 0;
}
```



O/P

P[0] = 22] Garbage
P[1] = 20] Values