28/08/2023

Compiler: Compiler is a translator that converts high level language into target language (machine level / low level).

» It converts the source code into target code, all at once and creates an Object ~~eta~~ file.

Assembler: Assembler is a translator that converts high level language into machine level language.

Interpreter: Interpreter is a translator that performs line by line execution.

» It is slower when compared to compiler.

→ Compilation can be done in one or many passes.

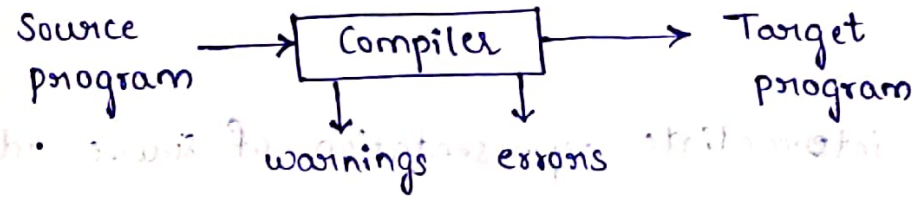
One pass :- HLL → MLL.

more passes :- HLL → LLL , LLL → MLL.

29/08/2023

Compiler: It is a software or a program that converts a program written in high-level language (source code) to a low-level language (object or target code) language.

» It also reports errors present in the source program.



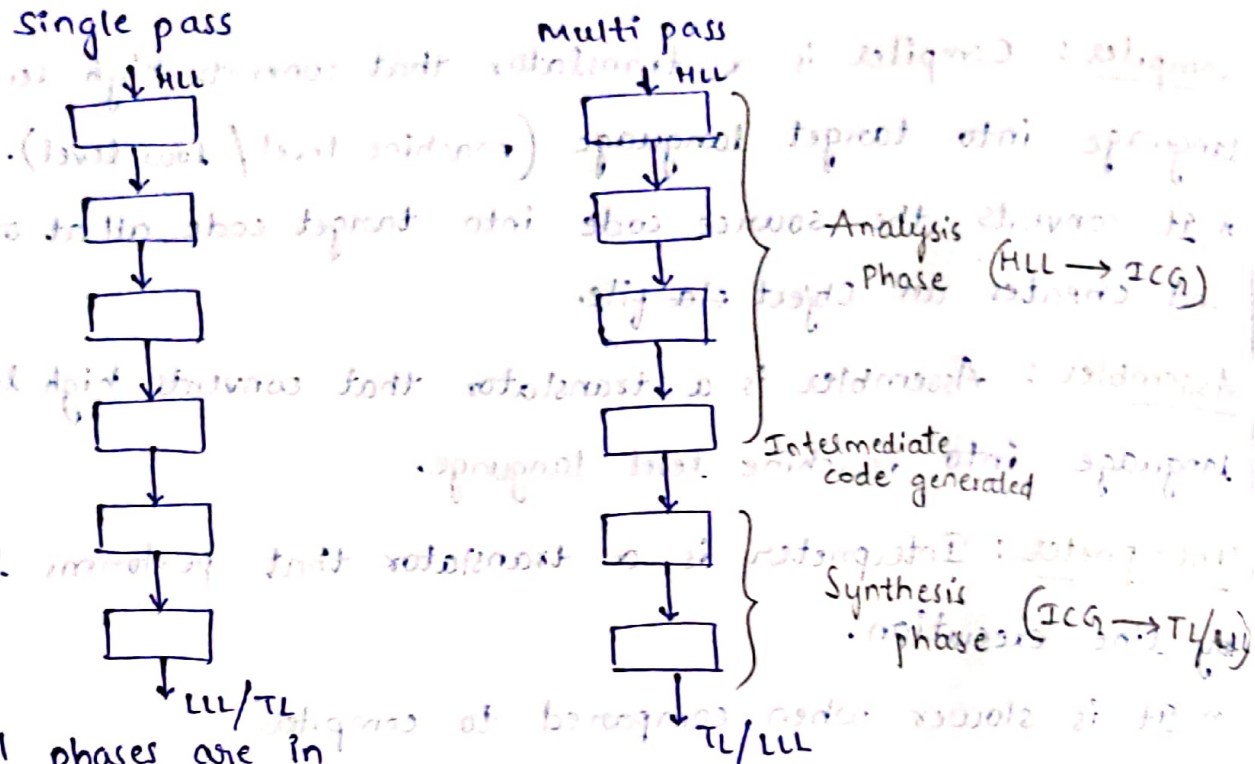
Types of compilers

1) **Single pass compiler**:

It is a type of compiler that processes the source code only once. (All 6 phases are performed at once).

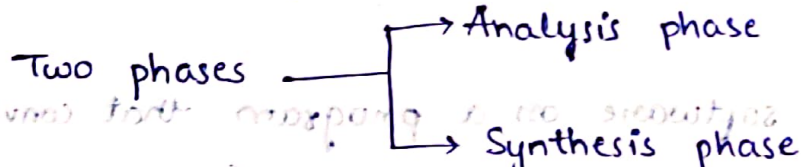
2) **Multi pass compiler**: 4-phases — Pass I
2-phases — Pass II.

It is a type of compiler that processes the source code multiple times i.e, to convert source code into target/object code.



All phases are in single module.

There are two parts of compilation process (phases).



Analysis Phase

It breaks up the source code or program into smaller parts and creates an intermediate representation of the source program.

Synthesis Phase

It takes the intermediate representation of source code as input and creates the target code that is desired.

Properties of the compiler

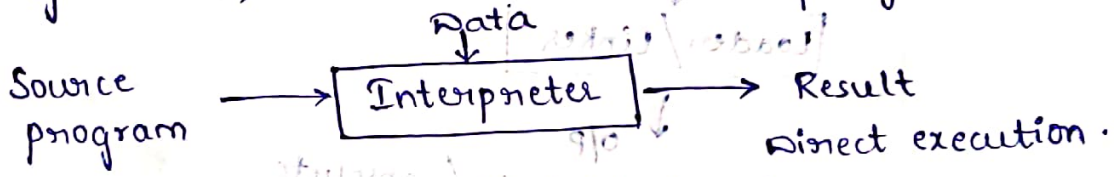
When a compiler is built, it should posses the following properties.

» The compiler itself must be bug-free.

» It must generated correct machine code.

» The generated machine code must run fast.
» Compilation time must be proportional to program size.
» The compiler must be portable.
» It must give good diagnostics and error messages.
» The generated code must work well with existing debuggers
» It must have consistent optimization.

Interpreter :- An interpreter is a kind of translator which produces the result directly when the source program and data is given to it as input. It does not produce the object code, rather each time the program needs execution

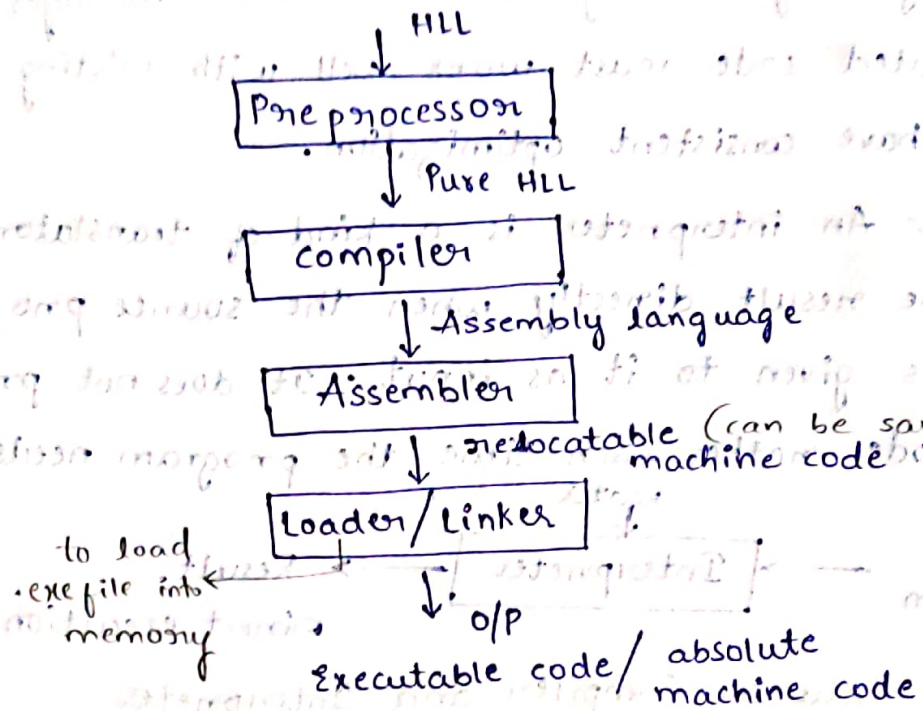


** Differences between Compiler and Interpreter

Compiler	Interpreter
» It translates entire program as input.	» It takes single instruction as input.
» Intermediate object code is generated.	» No intermediate object code is generated.
» Memory requirement is more (since, object code is generated).	» Memory requirement is less.
» Program need not be compiled every time.	» Every time, high level program is converted into low level langua
» Errors are displayed after the entire program is checked.	» Errors are displayed for every instruction interpreted.
» C, Java, C++ are compiled languages.	» Python is interpreted language.

→ The main goal of any translator is that, it must preserve the meaning of the source code.

Language Processing System



The high-level language programs are fed into a series of tools and operating system components to get the desired code that can be used by the machine. This is called language processing system.

» Preprocessor : In preprocessing, HLL is converted into pure HLL. Preprocessor removes the preprocessing directives (#include<stdio.h>) and adds the respective files ie, file inclusions. It also performs macro expansion, operation conversions (Ex: a-- ⇒ a = a-1)

» Compiler : During compilation, compiler converts pure HLL to assembly language.

» Assembly language : It is a low-level programming language. It is not in binary form.

» Assembler : For every platform, we have different assemblers. Assembler for one platform will not work for another platform.

Assembly code is converted into ~~relocatable~~ executable machine code

» Relocatable machine code : that can be loaded anywhere in the memory.

Assembly code $\xrightarrow{\text{converted}}$ executable machine code.

» Linker / Loader : It converts relocatable machine code into absolute machine code.

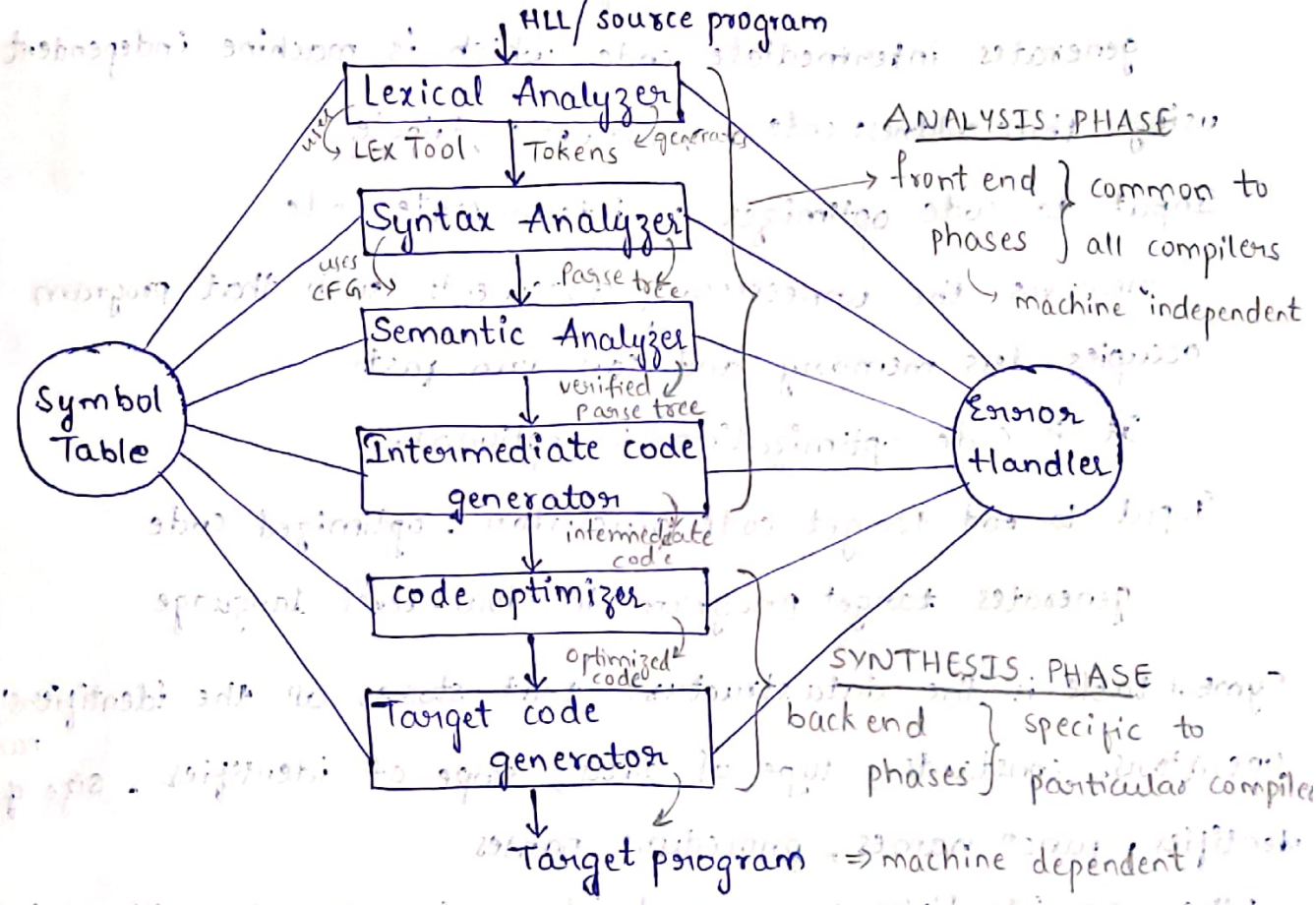
» Linker : It links different object files into a single executable file.

» Loader : It loads the executable file in the memory and executes it.

» Absolute machine code : Code that is assembled to work at one specific address.

30/08/2023

Phases of the Compiler



① Input to lexical analyzer : HLL source code

generates stream of tokens using LEX Tool that uses regular exp.

regular expression → that has rules for a valid token.

removes all spaces, commas, semi colons, tab spaces.

Errors like repeated declarations, using keywords as identifiers are detected.

② Input to syntax analyzer : stream of tokens

generates parse tree

uses CFG (Context Free Grammar) to check the syntax

identifies all the syntactical errors (if any).

③ Input to semantic analyzer : parse tree

checks for compatibility errors, parentheses matching, if-else statements, control flow statements

generates verified parse tree

④ Input to Intermediate code generator : verified parse tree

generates intermediate code, which is machine independent.

using three address code, quadra pole, tripole

$$\begin{cases} a = b + c + d \\ t_1 = b + c \\ t_2 = t_1 + d \end{cases}$$

⑤ Input to code optimizer : intermediate code

removes the unnecessary statements such that program occupies less memory and can run fast.

It is optional.

⑥ Input to target code generator : optimized code

generates target program in low-level language.

Symbol table is the data structure that stores all the identifiers, operators, constants, type of data, scope of identifier, size of identifier, name, func^n names, procedure names

When an identifier is encountered again, then it will not be entered into symbol table rather, it points to the already entered identifier.

It is also used to retrieve the identifiers.

It is associated with each phase.

Error handler is associated with ~~eo~~ every phase.

It handles all the errors and ensures smooth compilation of the entire program.

Show how an input $a = b + c * 60$ get processed in compiler. show the output of each stage of compiler. Also show the contents of symbol table.

$a = b + c * 60$

↓

lexical analyzer

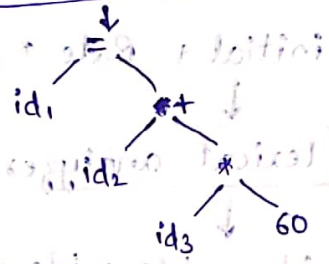
↓

$id_1 = id_2 + id_3 * 60$ (stream of tokens)

↓

syntax analyzer

↓



(parse tree)

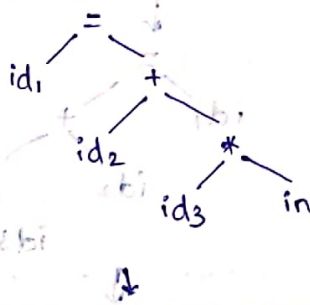
Symbol table

a	...
b	...
c	...
⋮	...

↓

semantic analyzer

↓



(verified parse tree)

if a, b, c are floating point

↓

Intermediate code generator

↓

$t_1 = \text{int to float}(60)$

$t_2 = id_3 * t_1$

$t_3 = id_2 + t_2$

$id_1 = t_3$

(3-address code)

↓

code optimization

$t_1 = id_3 * 60.0$

$id_1 = id_2 + t_1$

↓

target code generator

↓

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

(Target code)

Describe the output for the various phases of the compiler w.r.to following statement

position = initial + Rate * 60

position = initial + Rate * 60

↓

lexical analyzer

↓

$id_1 = id_2 + id_3 * 60$

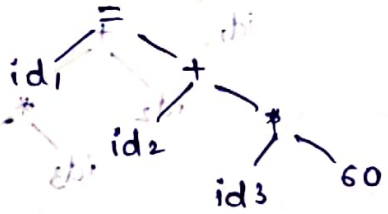
Stream of tokens :

↓

syntax analyzer

Symbol table

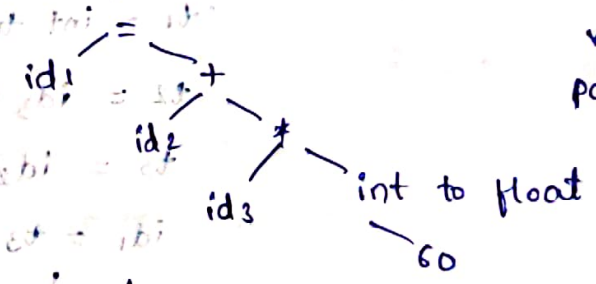
position	...
initial	...
Rate	...
⋮	...



parse tree

↓

semantic analyzer



verified parse tree

↓

intermediate code generator

↓

$t_1 = \text{int to float}$

$t_2 = id_3 * t_1$

$t_3 = id_2 * + t_2$

~~t_4~~ $id_1 = t_3$

3-address code

↓

code optimization

↓

$t_1 = id_3 * 60.0$

$id_1 = id_2 + t_1$

↓

target code generator

↓

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

↓

target code

19/2023

Show the output generated by each phase for the following example.

price = amount + rate * 50

price = amount + rate * 50

↓

lexical analyzer

↓

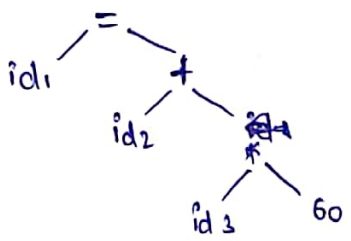
$id_1 = id_2 + id_3 * 50$

stream of tokens

↓

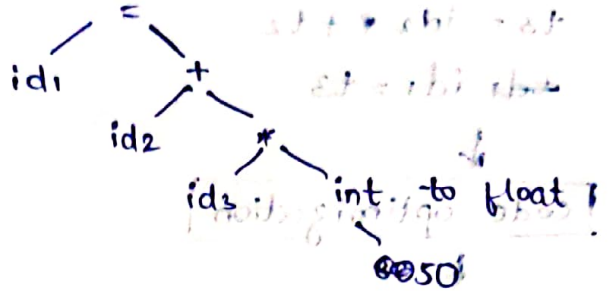
syntax analyzer

↓



parse tree

Semantic analyzer



intermediate code generator

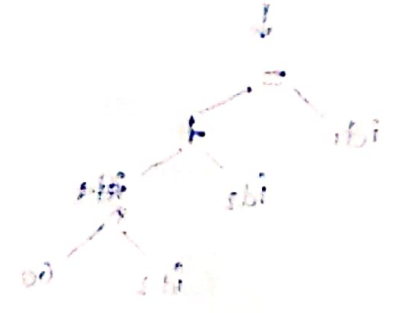
```
t1 = int to float
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

code optimization

```
t1 = id3 * 50.0
id1 = id2 + t1
```

Target code generator

```
MOVF  id3, R2
MULF  #50.0, R2
MOVF  id2, R1
ADDF  R2, R1
MOVF  R1, id1
```



Show the output generated by each phase for the following example.

$x = (a+b) * (c+d)$

$x = (a+b) * (c+d)$

↓

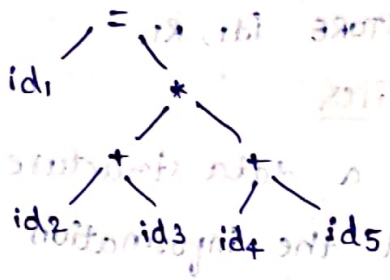
lexical analyzer

↓

$id_1 = (id_2 + id_3) * (id_4 + id_5)$ stream of tokens

↓

Syntax analyzer

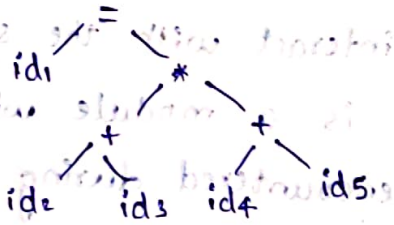


parse tree / syntax tree

↓

semantic analyzer

↓



verified parse tree

↓

Intermediate code generator

↓

$t_1 = id_2 + id_3$

$t_2 = id_4 + id_5$

$t_3 = t_1 * t_2$

$id_1 = t_3$

3-address code

↓

code optimization

↓

$t_1 = id_2 + id_3$

$t_2 = id_4 + id_5$

$id_1 = t_1 * t_2$

↓

code generator

↓

~~load id1, id2~~

~~ADD~~

LOAD R1, id2

ADD R1, id3

LOAD R2, id4

ADD R2, id5

MUL R1, R2

STORE id1, R1

Phases of the compiler

Symbol Table : It is a data structure being used by the compiler to store all the information related to identifiers.

Ex: its type, scope, size, location, name etc

» It helps the compiler to function smoothly by finding the identifiers quickly.

» All the phases interact with the symbol table manager.

Error Handler : It is a module which takes care of the events which are encountered during compilation, and it takes care to continue the compilation process even if error is encountered.

Lexical Analysis :

» Reads the program and converts it into tokens using a tool called Lex tool.

» Tokens are identified by regular expressions which are understood by the lexical analyzer.

» Removes whitespaces, comments, tab spaces etc.

» Lexical analyzer is also called as Scanner

Syntax Analyzer / Parser :

» constructs parse tree

» Takes the tokens one by one and uses CFG to construct parse tree

» If it is not possible to construct the parse tree, then input is syntactically incorrect and error messages are displayed.

» By using Context Free Grammars (CFG), the input has to be checked whether it is in the desired format or not.

» Syntax errors can be detected by it, if the input is not according to the grammar given.

» It takes input as tokens and checks their syntactic correctness.

Semantic Analysis:

» Once the syntax is checked in the syntax analysis phase, the next phase is semantic analysis, which determines the meaning of the source string. For ex: meaning of the source string is matching of parantheses in the expression, matching of if-else statements, performing arithmetic operations of the expressions that are type-compatible, checking the scope of the operation.

Intermediate code generation:

» The intermediate code is a kind of code, which is easy to generate and this code can be easily converted to target code.

» This code can be in variety of forms such as 3-address code, quadraples, tripules, post-fix notation.

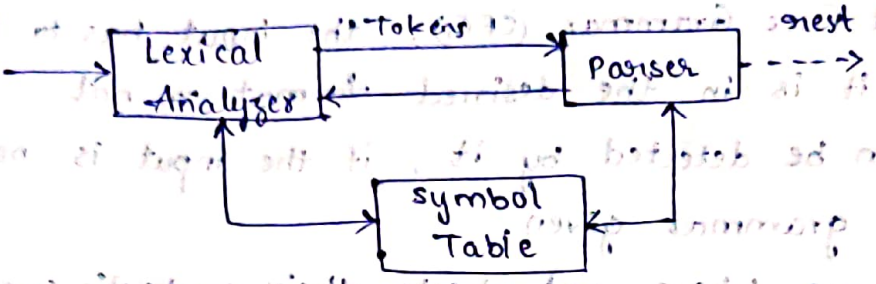
Code ~~optimation~~ Optimization:

» The code optimization phase attempts to improve the intermediate code. This is necessary to have faster executing code or less consumption of memory. Thus, by optimizing the code, the overall running time of the target program can be improved.

Code generation:

» In code generation phase, the target code gets generated. The intermediated code instructions are translated into sequence of machine instructions.

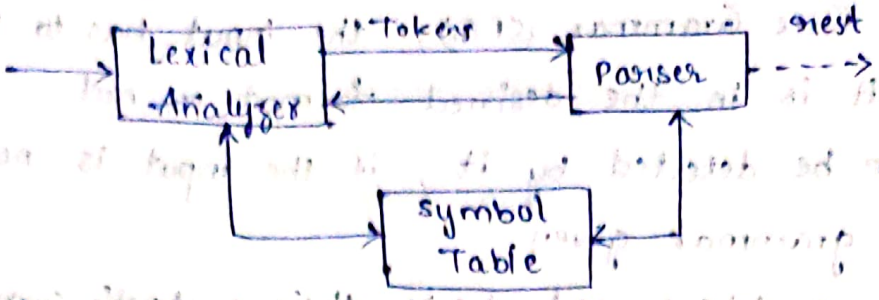
Lexical Analysis Phase



» Lexical Analysis is the first phase of the compiler, also called as Linear Analysis or ~~Scanner~~ Scanning.

» In this phase, the stream of characters making up source code is read from left to right and grouped into tokens that are sequence of characters having collective meaning.

Lexical Analysis Phase



» Lexical Analysis is the first phase of the compiler, also called as linear Analysis or Scanner Scanning.

» In this phase, the stream of characters making up source code is read from left to right and grouped into tokens that are sequence of characters having collective meaning.

» Tokens include identifiers, operators, keywords...

4/9/2023

Functions of Lexical Analyzer

» It produces stream of tokens.

» It eliminates blank spaces and comments.

» It generates the symbol table which stores the information about identifiers, constants encountered in the input.

» It keeps track of the line numbers.

» It reports the errors encountered while generating the tokens.

Tokens :

It describes the class or category of input string.

ex :- identifiers, keywords, constants.

Patterns :

Set of rules that describe the tokens

Lexemes :

Sequence of characters in the source program that are matched with the patterns or the tokens.

ex :- num is a valid token ie, identifier that follows the pattern

Ex: Identify tokens in the expression —

if (a < b)

if – keyword
(→ operator
a → identifier
< → relational operator
b → identifier
) → operator

**

Q) Define lexeme, token and pattern. Identify the lexemes that make up the tokens in the following program set.
Identify Indicate the corresponding token and pattern.

```
void swap (int i, int j){
    int t;
    t = i;
    i = j;
    j = t;
}
```

Lexeme		Token
void	—	keyword
swap	—	identify
(	—	operator
int	—	keyword
&	—	operator
i	—	identifier
,	—	separation operator
int	—	keyword
j	—	identifier
)	—	operator
{	—	operator
int	—	keyword
t	—	identifier
;	—	operator
*	—	operator

Lexeme		Token
t	—	identifier
=	—	operator
i	—	identifier
;	—	operator
i	—	identifier
=	—	operator
j	—	identifier
;	—	operator
j	—	identifier
=	—	operator
t	—	identifier
;	—	operator
}	—	operator

Patterns

(i) Identifiers

» Identifiers are a collection of letters

» Identifier is a collection of alphanumeric characters)

» The first character of an identifier must be a letter.

(ii) Operator

» Operator can be arithmetic, logical, relational operators.

» The paranthesses are considered as operators

» comma is treated as separation operator

» assignment is denoted by operator

(iii) keyword

» keywords are special words to which some meaning is associated with.

» int, void are keywords for denoting data types.

Q)

```
int max (int a, int b)
{
    if (a > b)
        return a;
    else
        return b;
}
```

~~Token~~	
int	keyword
max	identifier
(	operator
a	identifier
b	identifier
{	operator
if	keyword
>	relational operator
return	keyword
else	keyword
}	operator

lexeme	~~token~~
int	- keyword
max	- identifier
(	- operator
int	- keyword
a	- identifier
,	- operator
int	- keyword
b	- identifier
)	- operator
{	- operator
if	- keyword
(	- operator

lexeme	~~Token~~.	token
a	-	identifier
>	-	operator
b	-	identifier
)	-	operator
return	-	keyword
a	-	identifier
else	-	keyword
return	-	keyword
b	-	identifier
}	-	operator

operator

lexeme — token

Patterns

(i) Identifiers

» Identifiers are a collection of letters

» Identifier is a collection of alphanumeric characters

» The first character of an identifier must be a letter.

(ii) Operator

» Operator can be arithmetic, logical, relational

» Parantheses are considered as operators

» Comma is treated as separation operator.

» Assignment is denoted by operator.

» greater than symbol (>) is an operator

(iii) keyword

» keywords are special words to which some meaning is associated with.

» int, void are keywords for denoting data types.

» return is a keyword

» if, else are keywords for checking whether a condition is true.

Differences between Phases and Pass

Phase	Pass
1) The processing of compilation is carried out in various ways. These steps are referred as Phases. The phases are: Lexical phase, Syntax phase, Semantic phase, Intermediate code generation, code optimization, Target code generation	1) Various phases are logically grouped together to form a Pass. The process of compilation can be carried out in single pass or in multiple passes.

2) The task of compilation is carried out in analysis and synthesis phase.

3) The phases namely -
lexical analysis
Syntax analysis
Semantic analysis
Intermediate code generation
are machine independent.
code optimization
Target code generation
are machine dependent.

2) The task of compilation is carried out in single or multiple passes.

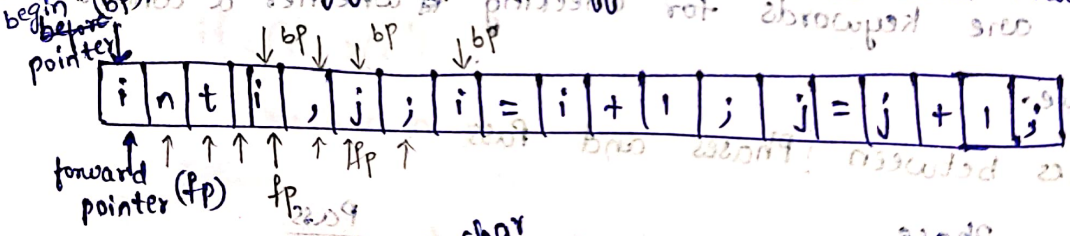
3) The execution of program in passes, The compilation model can be viewed as front end and back end

05/9/2023

** Input Buffering

```
int i, j;
i = i+1 ;
j = j+1 ;
```

lexical analyzer uses buffer to ~~store~~ avoid multiple system calls.



when it reads each char then, fp is incremented until space is identified.
when it encounters blank space, then two pointers is identified.
a token is identified.
Here, int is identified which is a keyword.
Then, begin pointer is moved to i
both fp and bp are at i
Then, fp is incremented.
next char is ,
so, bp is also shifted to ,

When ; is encountered it indicates end of the statement so, both bp and fp are moved to it.

» Problem of using buffer is size.

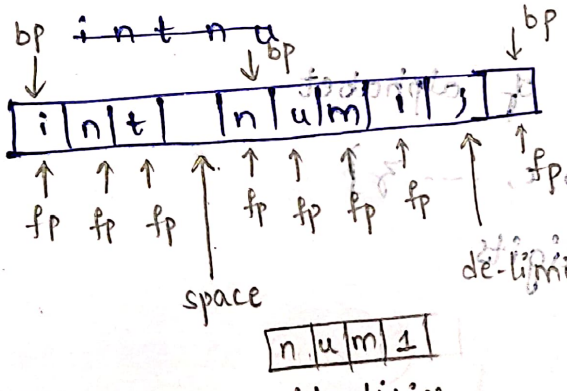
categories of buffer :-

(i) one buffer

(ii) two buffer

(i) In single buffer, data is overwritten when buffer size is insufficient. In this method, data may be lost.

(ii) To overcome the problems of single buffer, we use two buffer. When the end point of buffer (eof) is reached, the remaining data is stored in second buffer. And, when second ~~dat~~ buffer is reached to its end, it again uses first buffer. Thereby, the data of first buffer is lost.



Specifications of tokens (UNIT - I) 1st Half

» To specify tokens, regular expressions are used.

When a pattern is matched by some regular expression then, token can be recognized.

Language and String

» Language : Finite set of non-empty strings

» String : Collection of finite number of alphabet or letters.

ε indicates null string.

» The length of the string is denoted by $|s|$.

» The empty string is denoted by ε

» The empty set of strings is denoted by $\emptyset$

Operations on Language

1) Union of two languages ~~L1 and L2~~ L1 and L2 :-

$L1 \cup L2 = \{$ set of strings in L1 and set of strings in L2$\}$

2) Concatenation of two languages L1 and L2 :-

$L1 \cdot L2 = \{$set of strings in L1 followed by set of strings in L2$\}$

3) Kleen closure of language $L \Rightarrow L^*$

L^* denotes 0 or more concatenations of L

~~Include~~ This may contain ε

4) Positive closure of L $\Rightarrow L^+$

L^+ denotes 1 or more concatenations of L

Example

Let L represent the set of alphabet.

$L = \{ A, B, C, \ldots, Z, a, b, c, \ldots z \}$

D be the set of digits

$D = \{ 0, 1, 2, 3, \ldots, 9 \}$

Then, by performing various operations the new language is generated as follows :-

1) $L \cup D$ is a set of alphabet and digits

2) $L \cdot D$ (or) LD is a set of strings consisting of alphabet followed by digits.

L^5 is a set of strings having length of 5 each

3) L^* is a ~~strino~~ set of strings having all the strings including ε.

4) L^+ is a set of strings except ε.

Suppose, $L = \{a\}$ then,

$$L^* = \{\varepsilon, a, aa, aaa, aaaa, \ldots\ldots\}$$

$$L^+ = \{a, aa, aaa, aaaa, \ldots\ldots\}$$

suppose, $L = \{a, b\}$ then

$$L^* = \{\varepsilon, a, b, ab, ba, aa, bbb, \ldots\}$$

$$L^+ = \{a, b, ab, ba, aaa, bbb, \ldots\}$$

3) L^* is a string set of strings having all the strings including ε.

4) L^+ is a set of strings except ε.

Suppose, $L = \{a\}$ then,

$L^* = \{\varepsilon, a, aa, aaa, aaaa, \ldots\}$

$L^+ = \{a, aa, aaa, aaaa, \ldots\}$

suppose, $L = \{a, b\}$ then

$L^* = \{\varepsilon, a, b, ab, ba, aa, bbb, \ldots\}$

$L^+ = \{a, b, ab, ba, aaa, bbb, \ldots\}$

11/09/2023

Regular exp. has set of rules.

Regular set

The finite set which denotes a regular language and the set which can be described by regular expressions is called Regular set.

Ex: A set of identifiers.

Because, it can be represented using regular expressions.

Regular Expressions

Regular expressions are mathematical symbolisms which describe the set of strings of specific language.

~~Rules for writing~~

Following are the rules that describe the definition of regular expressions over the input set denoted by Σ

1. ε is a regular expression that denotes the set containing the empty string
2. If R_1 and R_2 are regular expressions, then

$R = R_1 + R_2$ $(R = R_1 | R_2)$ is also a regular expression which represents Union operation.

3. If R_1 and R_2 are regular expressions, then

$R = R_1 R_2$ is also a regular expression which represents concatenation operation.

4. If R_1 is a regular expression, then

$R = R_1^*$ is also a regular expressions which denotes Kleen closure.

* The language denoted by regular expressions is called Regular Set / Regular Language.

Q) Write a regular expression for the language accepting all the combinations of a's including null over $\Sigma = \{a\}$

$L = \{\varepsilon, a, aa, aaa, \ldots\}$

$L = a^*$

Q) Accept all the combinations of a except the null string

$L = \{a, aa, aaa, \ldots\}$

$L = a^+$

Q) Write a regular expression for the language containing the strings of length 2 over $\Sigma = \{0, 1\}$

$L = \{00, 01, 10, 11\}$

OR ← $(0+1)$ for length 1

$RE = (0+1)(0+1)$

Q) Write a regular expression for a language combining strings which ends with abb over $\Sigma = \{a, b\}$

$L = \{abb, aabb, babb, ababb, baabb, \ldots\}$

$RE = (a+b)^* abb$

$\because (a+b)^* = \{\varepsilon, ab, aa, bb, ba, \ldots\}$

Q) construct a regular expression for the language containing all the strings having any number of a's and b's except ε.

$RE = (a+b)^+$ ($\because L = \{a, b, aa, bb, ab, \ldots\}$ null string)

Q) construct regular expression for the language accepting all the strings ending with 00 over the set $\Sigma = \{0,1\}$

$RE = (0+1)^*00$

$L = \{000, 100, 0100, 1000, 01100, 10100, \ldots 00\}$

Q) write a regular exp for the language accepting the strings which are starting with 1 and ending with 0 over the set $\Sigma = \{0,1\}$

$RE = 1(0+1)^*0$

$L = \{10, 1010, 1110, 100100, 100100, \ldots\}$

Q) Write RE to denote the language L over Σ^* where $\Sigma = \{a, b, c\}$ in which every string will be such that any number of a's followed by any number of b's followed by any number of c's.

$RE = a^*b^*c^*$

$L = \{a, b, c, ab, bc, aabc, abcc, \ldots\}$

Q) write a RE to denote L over Σ^* where, $\Sigma = \{a, b\}$ such that the third character from right end of the string is always a.

$RE = (a+b)^*a(a+b)(a+b)$

Q) construct RE for the language which consists of exactly 2 b's over the set $\Sigma = \{a, b\}$

$RE = a^* b\, a^* b\, a^*$

Q) Describe the language for the following REs.

(1) $(0+1)^*\, 0\, (0+1)^*$

The language L containing all the strings made up of 0's and 1's which contains atleast one zero.

(2) $0^*1\,0^*1\,0^*10^*$

The language L containing all the strings made up of 0's and 1's which contains exactly 3 ones.

12/09/2023

Finite Automata

» Finite Automata is an abstract computing device.

» It is a mathematical model of a system with discrete inputs, outputs, states and set of transitions from state to state that occurs on input symbols from alphabet Σ

A string is said to be Accepted String, if all the characters are accepted by the finite automata, based on the specific patterns.

To accept a string, the final state has to be reached.

num1 ↑↑↑↑ ✓

nu#m1 ↑↑x ✗

Finite Automata Representation

1) Graphical representation (Transition diagram)

2) Tabular form (Transition Table)

3) Mathematical form (Transition function (or) Mapping function)

Formal definition of Finite Automata

A finite Automata is a 5-tuple notation. They are

$M = (Q, \Sigma, \delta, q_0, F)$ where,

Q is a finite set of states

Σ is a finite set of input symbols or alphabet

δ is Q

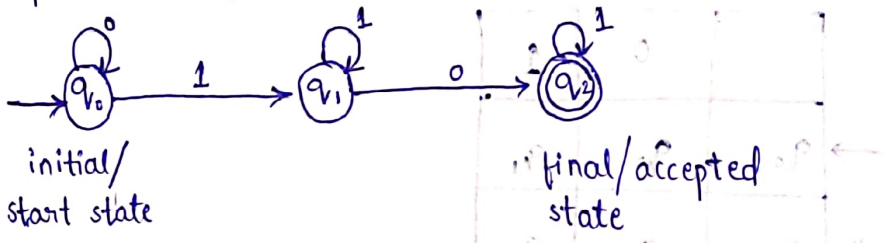
$\delta : Q \times \Sigma \to Q$ is a transition function.

$q_0 \in Q$ is the start state, also called as Initial state

F is the set of accepted states, also called as final state.

Transition Diagram

It is a directed graph associated with the vertices of a graph corresponds to the state of finite automata.



$Q = \{q_0, q_1, q_2\}$

$\Sigma = \{0, 1\}$

q_0 = initial state

$F = q_2$ (final state)

Each state accepts one symbol at a time.

Here, q_0 takes the symbol 0 and stays at q_0 itself

Transition Table

It is the tabular representation of the transition function that takes two arguments (a state and symbol) and returns a value (the next state).

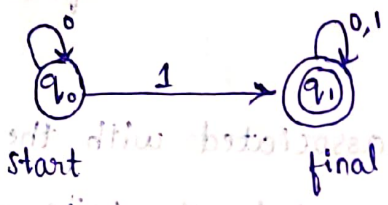
Rows corresponds to states

Columns correspond to input symbols

Entries correspond to next states

The start state is marked with an arrow. →

The accepted state is marked with *



	symbols / states	0	1
→	q_0	q_0	q_1
*	q_1	q_1	q_1

Transition function / Mapping function

» It is denoted by δ.

» Two parameters are passed to this transition function.

(i) current state

(ii) Input symbol.

» The transition function returns a state, which can be called as next state.

δ (current state, input symbol) = next state.

Ex:-

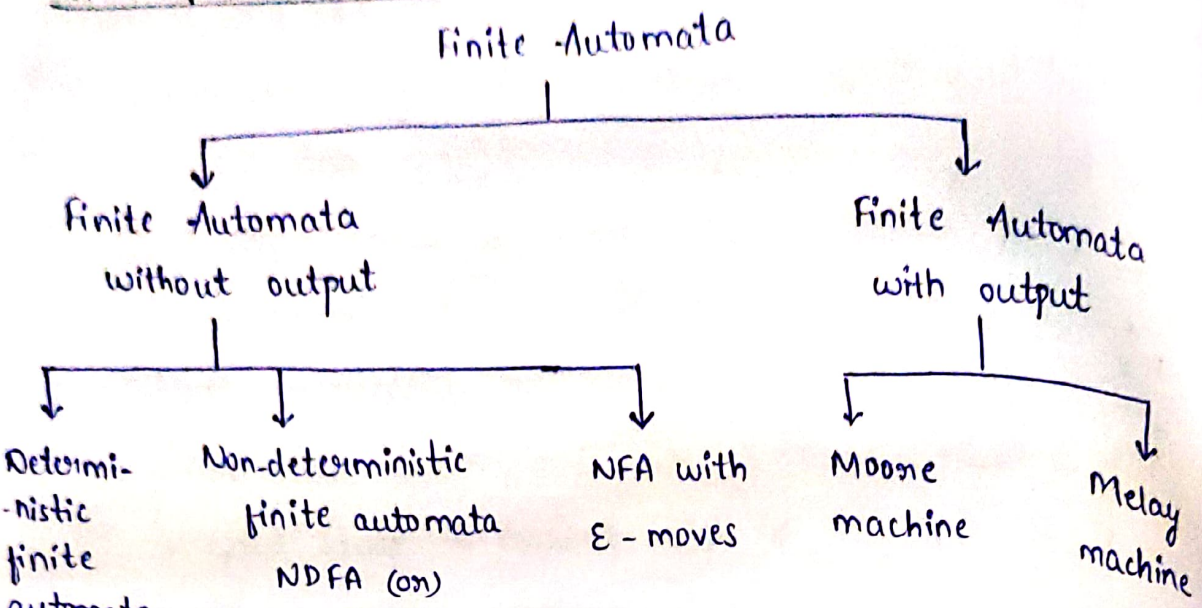
$\delta(q_0, 0) = q_0$

$\delta(q_0, 1) = q_1$

$\delta(q_1, 0) = q_1$

$\delta(q_1, 1) = q_1$

Types of Finite Automata

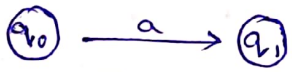


13/9/2023

Deterministic finite Automata (DFA)

1) DFA, if the machine can read an input string one symbol at a time.

2) In DFA, there is only one path for specific input from the current state to ~~the~~ next state.



3) DFA does not accept the null move ie; DFA cannot change state without any input symbol.

4) DFA can contain multiple final states. It is used in lexical analysis in compiler.

Formal Definition of DFA

DFA is a collection of 5 tuple notation, ~~as~~,

$$M = (Q, \Sigma, \delta, q_0, F)$$

Q : finite set of states

Σ : finite set of input symbols

δ : transition function

q_0 : start state

F : final ~~set~~ state

Acceptance of a Language

Language acceptance is defined by, if a string w is accepted by machine M ie; if it is reaching the final state F by taking the string w.

Not accepted if not reaching the final state.

Ex :-

⇒ L = { } ⇒ ∅

If language contains nothing ⇒ q_0 (no string)

It is represented by single state.

⇒ L = { ε } ⇒ q_0 (no move) — initial & final state.

ε → Len string.

we don't have I/p so, it will not move to next state.

⇒ If we are able to express reg. exp. for ε

It means that our DFA can accept it.

⇒ DFA to accept 'a' ⇒

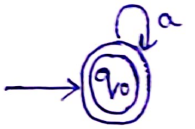
⇒ DFA to accept zero or more no. of a's

L = { ε, a, aa, aaa, ... }

regular exp. = a^*

No. of states depends on min length string

NOTE : No. of states = min str length + 1

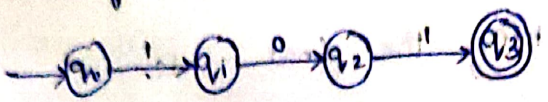


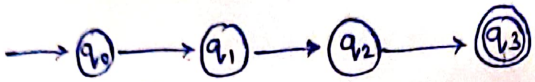
Q) Design DFA which accepts 101 string over

$\Sigma = \{0,1\}$

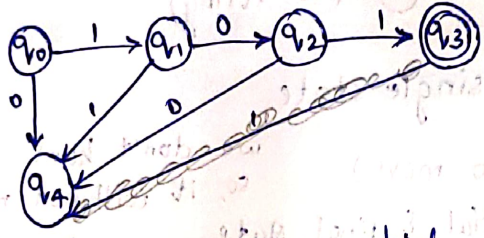
$L = \{101\}$, length of the string = 3

no. of states = 3+1 = 4





2nd I/P: 0101

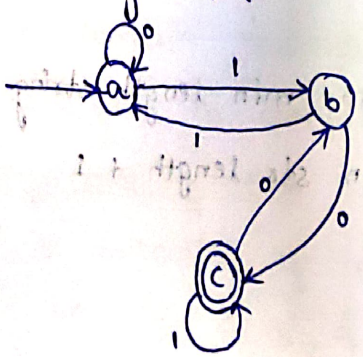


dead / trap state : which will never reach final state.

Q) Let DFA be $Q = \{a, b, c\}$, $\Sigma = \{0, 1\}$, $q_0 = \{a\}$, $F = \{c\}$ and construct a DFA based on transition table

	0	1
a	a	b
b	c	a
c	b	c

Transition diagram :-



Accepted string: $L = \{10, 010, 1000, \ldots\}$

Q) Design DFA $\Sigma = \{a, b\}$

(i) str starting with a

(ii) str ending with a

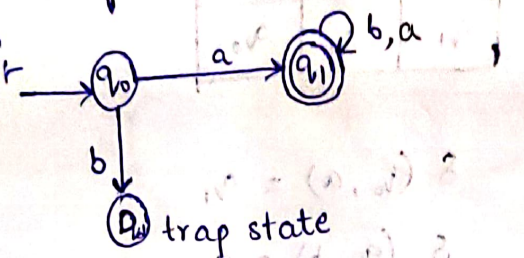
15/09/2023

(i) $L = \{ a, ab, abb, aba, ababaa, \dots \}$

min length str = a $\Rightarrow$ len = 1

no. of states = 1 + 1 = 2

Transition diagram :-



It is not required to take another state after q_1

(ii) ~~$L = \{a, aa\}$~~

(iii) ~~$L = \{a, aa, ab, abab\}$~~

Transition table :-

	a	b
→ q_0	q_1	q_d
* q_1	q_1	q_1

Transition function :-

$\delta(q_0, a) = q_1$

$\delta(q_0, b) = q_d$

$\delta(q_1, a) = q_1$

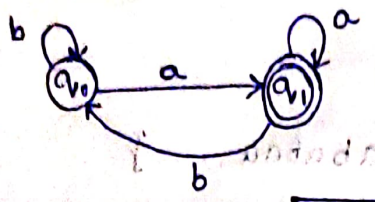
$\delta(q_1, b) = q_1$

(ii) $L = \{ a, aa, aba, ba, baa, ababa, \dots \}$ - baba,

min length str = a $\Rightarrow$ len = 1

no. of states = 1 + 1 = 2

Transition diagram :



Transition table :

	a	b
→ q_0	q_1	q_d
* q_1	q_1	q_0

Transition function :

$\delta(q_0, a) = q_1$

$\delta(q_0, b) = q_0$

$\delta(q_1, a) = q_1$

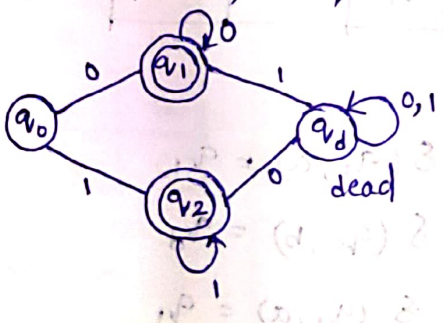
$\delta(q_1, b) = q_0$

Multiple final States

Q) Draw a DFA for the lang. accepting strings with 0's and 1's only over the input alphabet, $\Sigma = \{0, 1\}$

$L = \{ 0, 00, 1, 11, 000, 1111, 00000, \ldots \}$

Transition diagram



Transition Table

	0	1
→ q_0	q_1	q_2
* q_1	q_1	q_d
* q_2	q_d	q_2

Transition function :-

$\delta(q_0, 0) = q_1$

$\delta(q_0, 1) = q_2$

$\delta(q_1, 0) = q_1$

$\delta(q_1, 1) = q_d$

$\delta(q_2, 0) = q_d$

$\delta(q_2, 1) = q_2$

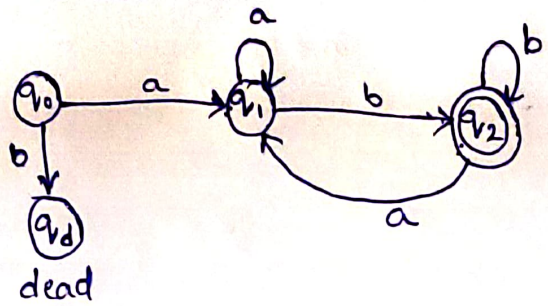
Q) Construct a DFA which accept a language of all the strings starting with 'a' and ending with 'b'.

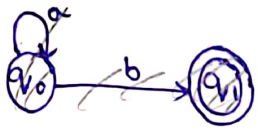
$L = \{ ab, aab, abb, abab, abaabb, \ldots \}$

min length str = 2

no. of states = 2+1 = 3

Transition diagram :-





Transition Table :-

	a	b
→ q_0	q_1	q_d
→ q_1	q_1	q_2
* q_2	q_1	q_2

Transition function :-

$\delta(q_0, a) = q_1$

$\delta(q_0, b) = q_d$

$\delta(q_1, a) = q_1$

$\delta(q_1, b) = q_2$

$\delta(q_2, a) = q_2$

$\delta(q_2, b) = q_2$

20/9/2023

Non- Deteriministic finite Automata (NFA)

» NFA stands for Non- Deteriministic finite Automata.

» It is easy to construct an NFA, when compared to DFA for a given regular expression.

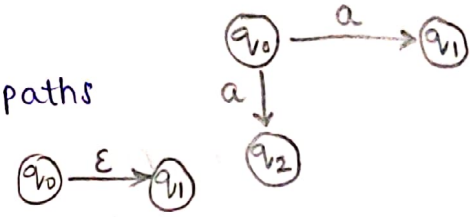
» The finite automata are called NFA, when there exists many paths for specific input from the current state to the next state.

» Each NFA can be translated into DFA but, every NFA is non- DFA.

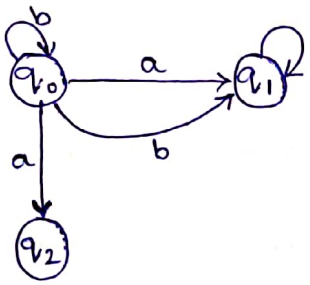
» NFA is defined in the same way as DFA but, with the following two exceptions :-

(i) It contains multiple next paths

(ii) It contains ε- transitions



Ex :-



» NFA also have 5 tuples which are same as DFA, but with different transition function

$$\delta : Q \times \Sigma \longrightarrow 2^Q$$

↳ finite set of states

In DFA,

$\delta : Q \times \Sigma \rightarrow Q$

» NFA is defined as a 5 tuple notation

$$M = (Q, \Sigma, \delta, q_0, F) \text{ , where}$$

Q : finite set of states

Σ : finite set of input symbols

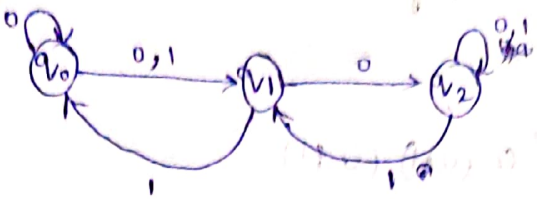
δ : transition function

q_0 : start state

F : end state

Q) Construct the transition table for the given DFA.

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



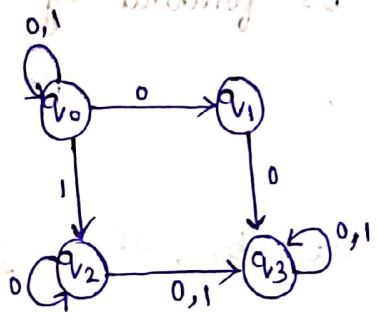
Transition Table :-

	0	1
→ q_0	$\{q_0, q_1\}$	q_1
q_1	q_2	q_0
* q_2	q_2	$\{q_1, q_2\}$

Q) Design NFA transition diagram for the transition table given.

	0	1
→ q_0	$\{q_0, q_1\}$	$\{q_0, q_2\}$
q_1	$\{q_3\}$	
q_2	$\{q_2, q_3\}$	$\{q_3\}$
* q_3	$\{q_3\}$	$\{q_3\}$

Transition diagram :-



Transition function :-

$\delta(q_0, 0) = \{q_0, q_1\}$

$\delta(q_0, 1) = \{q_0, q_2\}$

$\delta(q_1, 0) = \{q_3\}$

$\delta(q_1, 1) = \emptyset$

$\delta(q_2, 0) = \{q_2, q_3\}$

$\delta(q_2, 1) = \{q_3\}$

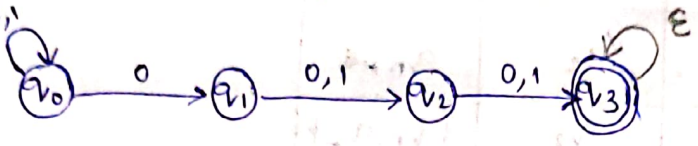
$\delta(q_3, 0) = \{q_3\}$

$\delta(q_3, 1) = \{q_3\}$

Q) Design NFA with $\Sigma = \{0,1\}$ accepting all strings in which the third symbol from the right end is always 0.

RE :- $\underbrace{(0+1)^*}_{\text{self loop}}\ 0\ (0+1)\ (0+1)$

~~bc~~ & min length str = 4



NFA-ε

It is also a 5-tuple notation

$M = (Q, \Sigma, \delta, q_0, F)$

Here, $\Sigma = \{0, 1, \varepsilon\}$

$\Sigma' = \Sigma \cup \{\varepsilon\}$

Σ' → (NFA-ε) ; Σ → (NFA)

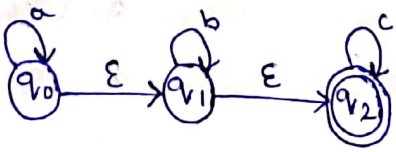
Q) Construct NFA with ε which accepts all strings over the alphabet $\Sigma = \{a, b, c\}$ where, string contains multiple a's followed by multiple b's followed by multiple c's.

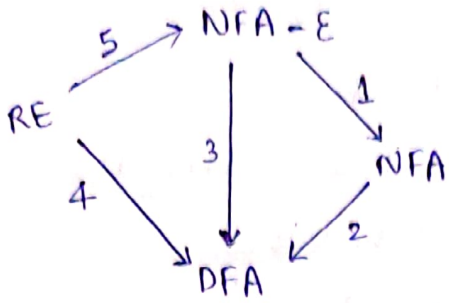
RE = $a^* b^* c^*$

~~L = {a, ab, ac~~ $L = \{a, b, c, ab, bc, ac, \ldots\}$

DFA :- NFA

NFA-ε :-





21/09/2023

① NFA-ε to NFA

2 steps

→(i) ε-closure (state) = self state + ε-move to other state

→(ii) $\delta'(q, a)$ = ε-closure $(\delta(\hat{\delta}(q, \varepsilon), a))$

where, $\hat{\delta}(q, \varepsilon) \Rightarrow$ ε-closure (q)

Ex:

NFA - ε

(i) Find ε-closure of states (q_0, q_1, q_2)

ε-closure (q_0) = (self state for q_0) + (ε-move to other state) → $q_0 \to q_1$ $q_1 \to q_2$

= $\{q_0, q_1, q_2\}$

ε-closure (q_1) = (self state for q_1) + (ε-move to other state)

= $\{q_1, q_2\}$

ε-closure (q_2) = (self state for q_2) + (ε-move to other state)

= $\{q_2\}$

(ii) $\delta'(q_0, 0)$ = ε-closure $(\delta(\hat{\delta}(q_0, \varepsilon), 0))$

= ε-closure $(\delta(q_0, q_1, q_2), 0)$

= ε-closure $(\delta(q_0, 0) \cup \delta(q_1, 0) \cup \delta(q_2, 0))$

= ε-closure $(q_0 \cup \emptyset \cup \emptyset)$

= ε-closure (q_0)

$\delta'(q_0, 0) = \{q_0, q_1, q_2\}$

$\delta'(q_0, 1) = \varepsilon\text{-closure}\ (\delta\ (\hat{\delta}\ (q_0, \varepsilon), 1))$

$= \varepsilon\text{-closure}\ (\delta\ (q_0, q_1, q_2), 1)$

$= \varepsilon\text{-closure}\ (\delta\ (q_0, 1) \cup \delta\ (q_1, 1) \cup \delta\ (q_2, 1))$

$= \varepsilon\text{-closure}\ (\phi \cup q_1 \cup q_2)$

$= \varepsilon\text{-closure}\ (q_1 \cup q_2)$

$= \varepsilon\text{-closure}\ (q_1) \cup \varepsilon\text{-closure}\ (q_2)$

$= \{q_1, q_2\} \cup \{q_2\}$

$\Rightarrow \delta'(q_0, 1) = \{q_1, q_2\}$

$\delta'(q_1, 0) = \varepsilon\text{-closure}\ (\delta\ (\hat{\delta}\ (q_1, \varepsilon), 0))$

$= \varepsilon\text{-closure}\ (\delta\ (q_1, q_2), 0)$

$= \varepsilon\text{-closure}\ (\delta(q_1, 0) \cup \delta(q_2, 0))$

$= \varepsilon\text{-closure}\ (\phi \cup \phi)$

$= \varepsilon\text{-closure}\ (\phi)$

$\Rightarrow \delta'(q_1, 0) = \phi$

$\delta'(q_1, 1) = \varepsilon\text{-closure}\ (\delta\ (\hat{\delta}\ (q_1, \varepsilon), 1))$

$= \varepsilon\text{-closure}\ (\delta\ (q_1, q_2), 1)$

$= \varepsilon\text{-closure}\ (\delta\ (q_1, 1) \cup \delta(q_2, 1))$

$= \varepsilon\text{-closure}\ (q_1, q_2)$

$\delta'(q_1, 1) = \{q_1, q_2\}$

$\delta'(q_2, 0) = \varepsilon\text{-closure}\ (\delta\ (\hat{\delta}\ (q_2, \varepsilon), 0))$

$= \varepsilon\text{-closure}\ (\delta\ (q_2), 0)$

$= \varepsilon\text{-closure}\ (\delta\ (q_2, 0))$

$= \varepsilon\text{-closure}\ (\phi)$

$\Rightarrow \delta'(q_2, 0) = \phi$

$\delta'(q_2, 1) = \varepsilon\text{-closure}\ (\delta\ (\hat{\delta}\ (q_2, \varepsilon), 1))$

$= \varepsilon\text{-closure}\ (\delta(q_2), 1)$

$= \varepsilon\text{-closure}\ (\delta\ (q_2, 1))$

$= \varepsilon\text{-closure}\ (q_2)$

$\delta'(q_2, 1) = \{q_2\}$

Transition Table

	0	1
q_0	$\{q_0, q_1, q_2\}$	$\{q_1, q_2\}$
q_1	$\emptyset$	$\{q_1, q_2\}$
q_2	$\emptyset$	$\{q_2\}$

Equivalent NFA diagram

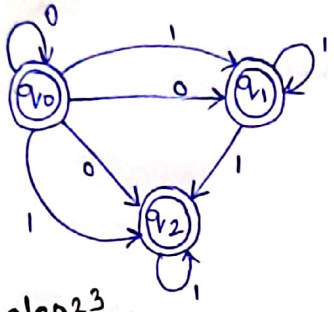
Initial state is always q_0

In NFA-ε, q_2 is final state, so check for

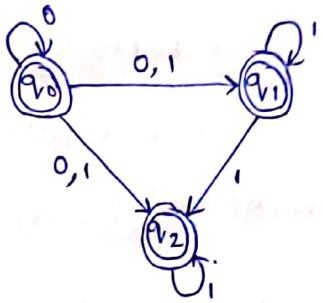
ε-closure (q_0), it contains q_2 so, q_0 is final state

ε-closure (q_1), it contains q_2 so, q_1 is also final state

ε-closure (q_2), it contains q_2 so, q_2 is also final state

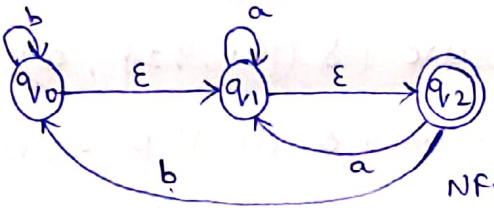


(or)



22/09/2023

Ex (2)



NFA - ε

(i) to find ε-closure (state) here; q_0, q_1, q_2 are states.

= self state + ε-moves to other state

ε-closure (q_0) = self state for q_0 + ε-moves to other state

$= \{q_0, q_1, q_2\}$

ε-closure (q_1) $= \{q_1, q_2\}$

ε-closure (q_2) $= \{q_2\}$

$\delta'(q_0, a)$ = ε-closure $(\delta(\delta(q_0, \varepsilon), a))$

= ε-closure $(\delta(\varepsilon\text{-closure}(q_0), a))$

= ε-closure $(\delta(\{q_0, q_1, q_2\}, a))$

= ε-closure $(\delta(q_0, a) \cup \delta(q_1, a) \cup \delta(q_2) \cup \delta(q_2, a))$

= ε-closure $(\emptyset \cup q_1 \cup q_1)$

= ε-closure (q_1)

$\Rightarrow \delta'(q_0, a) = \{q_1, q_2\}$

$\delta'(q_0, b)$ = ε-closure $(\delta(\varepsilon\text{-closure}(q_0), b))$

= ε-closure $(\delta(\{q_0, q_1, q_2\}, b))$

= ε-closure $(\delta(q_0, b) \cup \delta(q_1, b) \cup \delta(q_2, b))$

= ε-closure $(q_0 \cup \emptyset \cup q_0)$

= ε-closure (q_0)

$\Rightarrow \delta'(q_0, b) = \{q_0, q_1, q_2\}$

$\delta'(q_1, a)$ = ε-closure $(\delta(\varepsilon\text{-closure}(q_1), a))$

= ε-closure $(\delta(\{q_1, q_2\}, a))$

= ε-closure $(\delta(q_1, a) \cup \delta(q_2, a))$

= ε-closure $(q_1 \cup q_1)$

= ε-closure (q_1)

$\Rightarrow \delta'(q_1, a) = \{q_1, q_2\}$

$\delta'(q_1, b)$ = ε-closure $(\delta(\varepsilon\text{-closure}(q_1), b))$

= ε-closure $(\delta(\{q_1, q_2\}, b))$

= ε-closure $(\delta(q_1, b) \cup \delta(q_2, b))$

= ε-closure $(\emptyset \cup q_0)$

= ε-closure (q_0)

$\Rightarrow \delta'(q_1, b) = \{q_0, q_1, q_2\}$

$\delta'(q_2, a)$ = ε-closure $(\delta(\varepsilon\text{-closure}(q_2), a))$

= ε-closure $(\delta\{q_2\}, a)$

= ε-closure $(\delta(q_2, a))$ ~~$\cup \delta(q_2, a)$~~

= ~~ε-closure $(q_1) \cup q_1$~~

= ε-closure (q_1)

$\rightarrow \delta'(q_2, a) = \{q_1, q_2\}$

$\delta'(q_2, b)$ = ε-closure $(\delta(\varepsilon\text{-closure}(q_2), b))$

= ε-closure $(\delta(q_2, b))$

= ε-closure (q_0)

$\Rightarrow \delta'(q_2, b) = \{q_0, q_1, q_2\}$

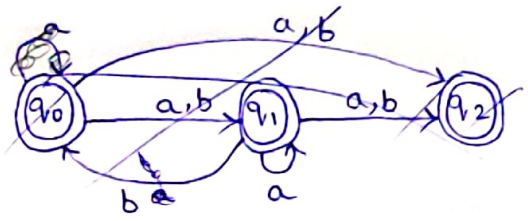
Transition Table

	a	b
q_0	$\{q_1, q_2\}$	$\{q_0, q_1, q_2\}$
q_1	$\{q_1, q_2\}$	$\{q_0, q_1, q_2\}$
q_2	$\{q_1, q_2\}$	$\{q_0, q_1, q_2\}$

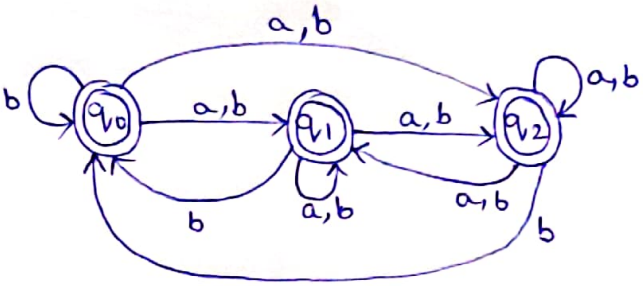
Equivalent NFA diagram

In NFA-ε, q_2 is the final state.

If q_2 is present in ~~each~~ any of the ε-closure (q_0), ε-closure (q_1), ε-closure (q_2) then, it becomes the final state.



Here, q_2 is present in ε-closure (q_0), ε-closure (q_1) and ε-closure (q_2).



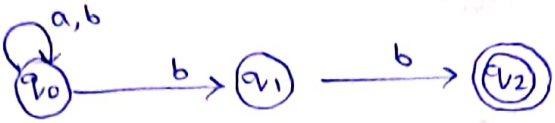
25/9/2023

② NFA to DFA

DFA transition function :- $\delta : Q \times \Sigma \rightarrow Q$

NFA transition function :- $\delta : Q \times \Sigma \rightarrow 2^Q$

Ex (1) :-



Step 1 :- construct NFA transition table.

	a	b
q_0	q_0	q_0, q_1
q_1	ϕ	q_2
q_2	ϕ	ϕ

Step 2 :- construct DFA transition table

	a	b
q_0	q_0	$\{q_0, q_1\}$
q_0, q_1	q_0	$\{q_0, q_1, q_2\}$
q_0, q_1, q_2	q_0	$\{q_0, q_1, q_2\}$

(i) 1st row in DFA TT will be 1st row in NFA TT

(ii) Next row in DFA TT will be last col of its previous row.

$\delta((q_0, q_1), a) \Rightarrow \delta(q_0, a) \cup \delta(q_1, a)$

$\Rightarrow q_0 \cup \phi$

$\Rightarrow q_0$

$\delta((q_0, q_1), b) \Rightarrow \delta(q_0, b) \cup \delta(q_1, b)$

$\Rightarrow (q_0, q_1) \cup (q_2)$

$\Rightarrow q_0, q_1, q_2$

$\delta((q_0, q_1, q_2), a) \Rightarrow \delta(q_0, a) \cup \delta(q_1, a) \cup \delta(q_2, a)$

$\Rightarrow q_0 \cup \phi \cup \phi$

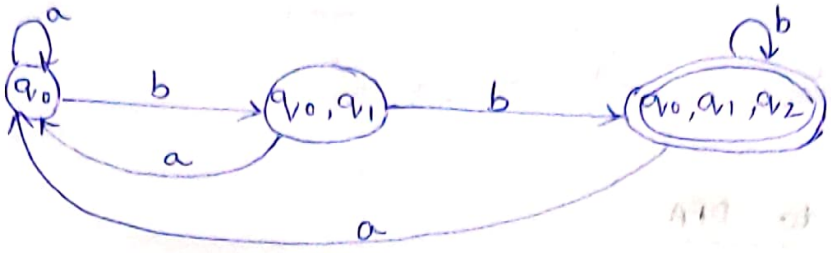
$\Rightarrow q_0$.

$\delta((q_0, q_1, q_2), b) \Rightarrow \delta(q_0, b) \cup \delta(q_1, b) \cup \delta(q_2, b)$

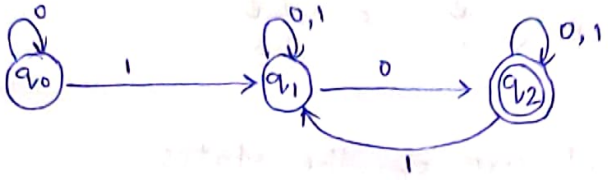
$\Rightarrow (q_0, q_1) \cup q_2 \cup \phi$

$\Rightarrow q_0, q_1, q_2$

step-3 :- construct transition diagram for DFA



Ex (2) :-



step 1 :- construct NFA transition table

	0	1
q_0	q_0	q_1
q_1	q_1, q_2	q_1
q_2	q_2	q_1, q_2

step 2 :- construct DFA transition table

	0	1
q_0	q_0	q_1
q_1	q_1, q_2	q_1
q_1, q_2	q_1, q_2	q_1, q_2

$\delta(q_1, 0) = q_1, q_2$

$\delta(q_1, 1) = q_1$

$\delta((q_1, q_2), 0) = \delta(q_1, 0) \cup \delta(q_2, 0)$

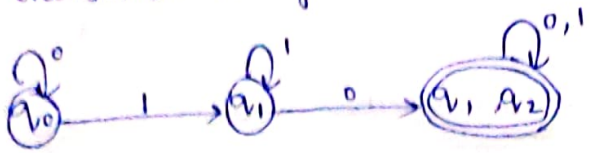
$= (q_1, q_2) \cup q_2$

$= q_1, q_2$

$\delta((q_1, q_2), 1) = \delta(q_1, 1) \cup \delta(q_2, 1)$

$= \delta \cdot q_1, q_2$

step3 :- DFA transition diagram.



(3) NFA-ε to DFA

Ex (1):

step1 :- Find ε-closure of the states

ε-closure (state) = self state + ε-move to other state

ε-closure (q_0) = $\{q_0, q_1, q_2\}$

ε-closure (q_1) = $\{q_1, q_2\}$

ε-closure (q_2) = $\{q_2\}$

step2 :- constnuct DFA transition table

(i) 1st state in DFA is ε-closure of q_0 (initial state)

		0	1
*	q_0, q_1, q_2	$\{q_0, q_1, q_2\}$	$\{q_1, q_2\}$
*	q_1, q_2	$\emptyset$	$\{q_1, q_2\}$

$\delta((q_0, q_1, q_2), 0)$ = ε-closure $\delta((q_0, q_1, q_2), 0)$

$\Rightarrow$ ε-closure $(\delta(q_0, 0) \cup \delta(q_1, 0) \cup \delta(q_2, 0))$

$\Rightarrow$ ε-closure (q_0) = $\{q_0, q_1, q_2\}$

$\delta((q_0, q_1, q_2), 1)$ = ε-closure $(\delta(q_0, q_1, q_2), 1)$

= ε-closure $(\delta(q_0, 1) \cup \delta(q_1, 1) \cup \delta(q_2, 1))$

= ε-closure $(\emptyset \cup q_1 \cup q_2)$

= ε-closure $(q_1 \cup q_2)$

= ε-closure (q_1) $\cup$ ε-closure (q_2)

= $\{q_1, q_2\} \cup \{q_2\}$

$\Rightarrow$ $\delta((q_0, q_1, q_2), 1)$ = $\{q_1, q_2\}$

$\delta((q_1, q_2), 0) = \varepsilon\text{-closure}(\delta(q_1, q_2), 0)$

$= \varepsilon\text{-closure}(\delta(q_1, 0) \cup \delta(q_2, 0))$

$= \varepsilon\text{-closure}(\emptyset \cup \emptyset)$

$= \varepsilon\text{-closure}(\emptyset)$

$= \emptyset$

$\delta((q_1, q_2), 1) = \varepsilon\text{-closure}(\delta(q_1, q_2), 1)$

$= \varepsilon\text{-closure}(\delta(q_1, 1) \cup \delta(q_2, 1))$

$= \varepsilon\text{-closure}(q_1 \cup q_2)$

$= \varepsilon\text{-closure}(q_1) \cup \varepsilon\text{-closure}(q_2)$

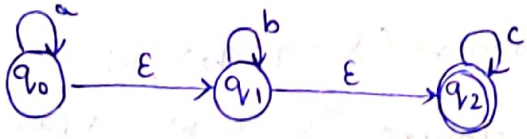
$= \{q_1, q_2\} \cup \{q_2\}$

$= \{q_1, q_2\}$

step-3 :- Draw transition diagram for DFA

(or)

Ex (2) :-



step 1 : ε-closure of states

ε-closure $(q_0) = \{q_0, q_1, q_2\}$

ε-closure $(q_1) = \{q_1, q_2\}$

ε-closure $(q_2) = \{q_2\}$

step 2 : DFA transition table

	a	b	c
q_0, q_1, q_2			
q_1, q_2			
q_2			

$\delta((q_0, q_1, q_2), a) = \varepsilon\text{-closure}(\delta((q_0, q_1, q_2), a))$

$= \varepsilon\text{-closure}(\delta(q_0, a) \cup \delta(q_1, a) \cup \delta(q_2, a))$

$= \varepsilon\text{-closure}(q_0)$

$= \{q_0, q_1, q_2\}$

$\delta((q_0, q_1, q_2), b) = \varepsilon\text{-closure}(\delta((q_0, q_1, q_2), b))$

$= \varepsilon\text{-closure}(\delta(q_0, b) \cup \delta(q_1, b) \cup \delta(q_2, b))$

$= \varepsilon\text{-closure}(\phi \cup q_1 \cup \phi)$

$= \varepsilon\text{-closure}(q_1)$

$= \{q_1, q_2\}$

$\delta((q_0, q_1, q_2), c) = \varepsilon\text{-closure}(\delta((q_0, q_1, q_2), c))$

$= \varepsilon\text{-closure}(\delta(q_0, c) \cup (q_1, c) \cup (q_2, c))$

$= \varepsilon\text{-closure}(q_2)$

$= \{q_2\}$

$\delta((q_1, q_2), a) = \varepsilon\text{-closure}(\delta(q_1, q_2), a)$

$= \varepsilon\text{-closure}(\delta(q_1, a) \cup \delta(q_2, a))$

$= \varepsilon\text{-closure}(\phi)$

$= \phi$

$\delta((q_1, q_2), b) = \varepsilon\text{-closure}(\delta(q_1, q_2), b)$

$= \varepsilon\text{-closure}(\delta(q_1, b) \cup \delta(q_2, b))$

$= \varepsilon\text{-closure}(q_1)$

$= \{q_1, q_2\}$

$\delta((q_1, q_2), c) = \varepsilon\text{-closure}(\delta(q_1, q_2), c)$

$= \varepsilon\text{-closure}(\delta(q_1, c) \cup \delta(q_2, c))$

$= \varepsilon\text{-closure}(q_2)$

$= \{q_2\}$

$\delta(q_2, a) = \varepsilon\text{-closure}(\delta(q_2, a))$

$= \varepsilon\text{-closure}(\phi)$

$= \phi$

$\delta(q_2, b) = \varepsilon\text{-closure}(\delta(q_2, b))$

$= \varepsilon\text{-closure}(\emptyset)$

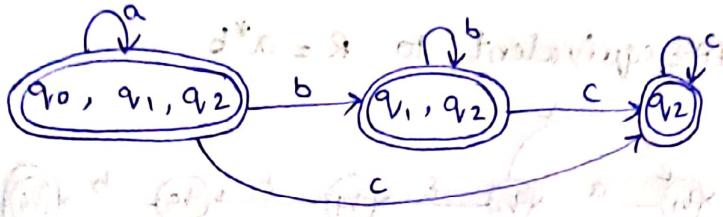
$= \emptyset$

$\delta(q_2, c) = \varepsilon\text{-closure}(\delta(q_2, c))$

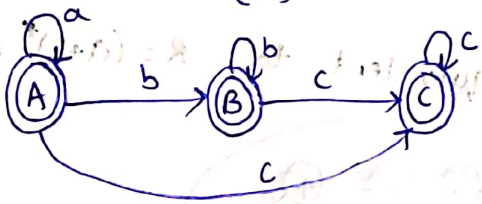
$= \varepsilon\text{-closure}(q_2)$

$= q_2$

step 3 : DFA transition diagram



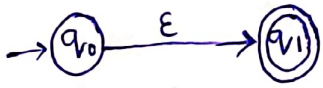
(OR)



27/09/2023

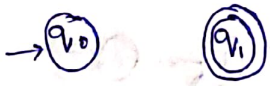
(5) Regular Expression to NFA-ε

(1) RE = ε (null string)



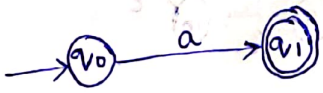
states can be changed without any input.

(2) RE = ∅ (empty set of strings)

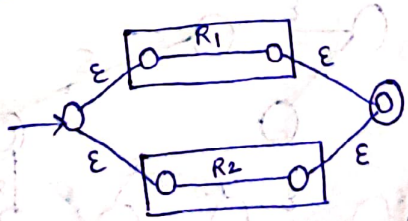


no transition

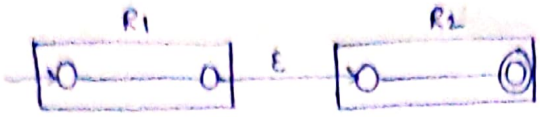
(3) RE = a



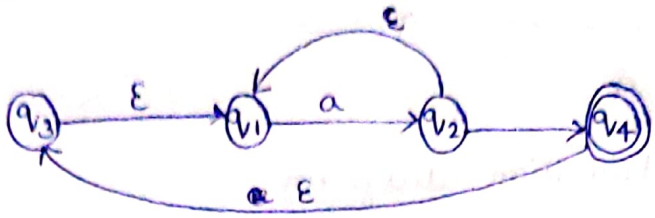
(4) union : RE = $R_1 + R_2$ (or) $R_1 | R_2$



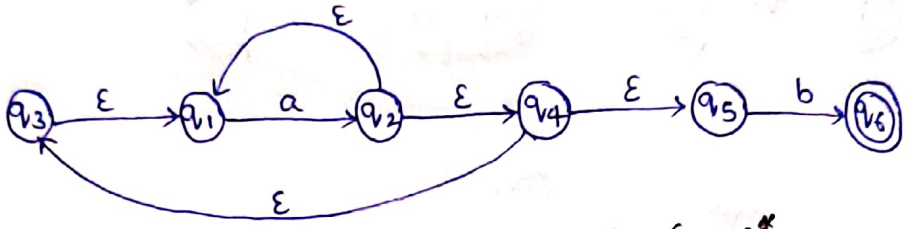
(5) concatenation : RE = $R_1 \cdot R_2$ (or) $R_1 R_2$



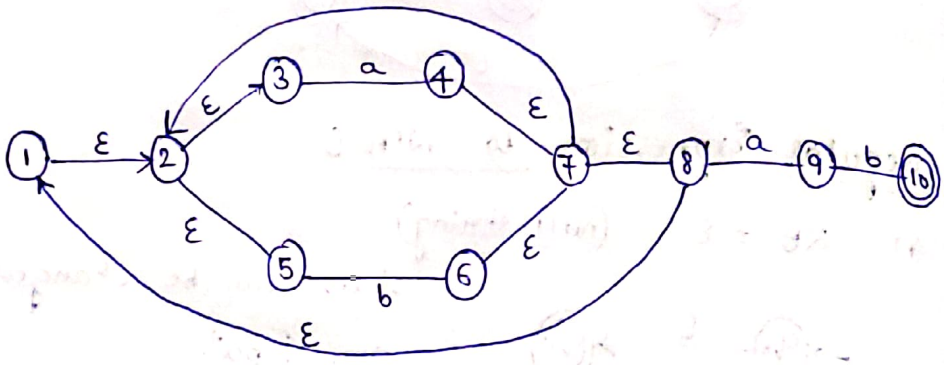
(6) closure : RE = a^*



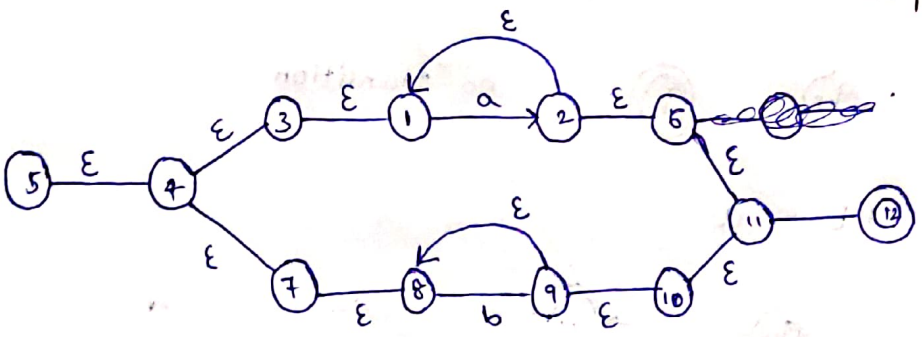
Q) construct NFA-ε equivalent to $R = a^*b$



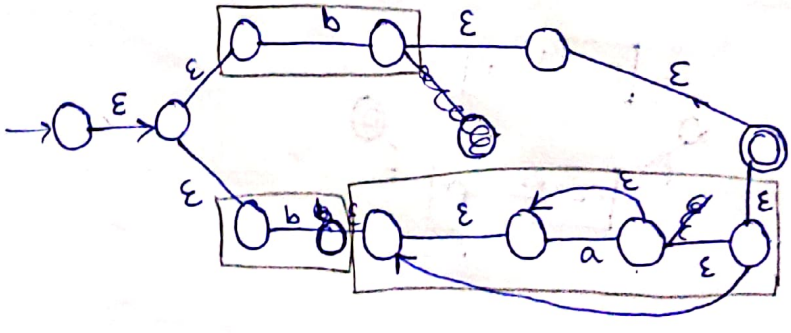
Q) construct NFA-ε equivalent to $R = (a+b)^* ab$



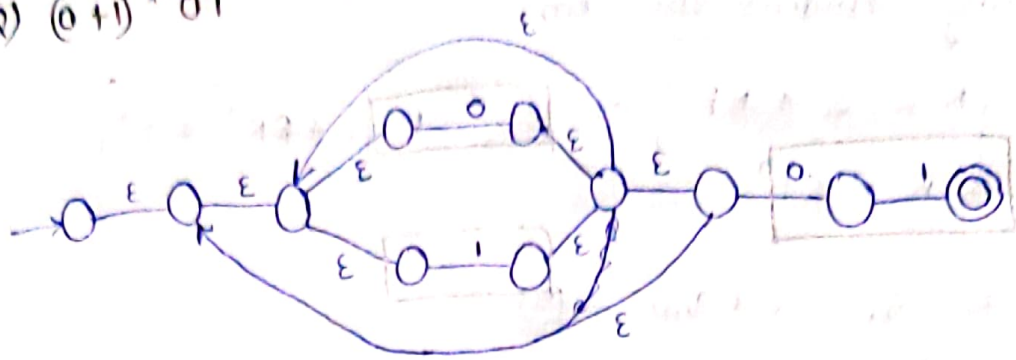
Q) Construct NFA-ε equivalent to $R = a^* + b^*$ (or)
$R = a^* | b^*$



Q) RE = $b + ba^*$



Q) $(0+1)^*01$



Scanned
27/9/23

④ DFA to RE

step (i) For each state q_0, q_1, q_2, identify all the states that comes into the state and, write in equation format

step (ii) For initial state, add ε

step (iii) : Calculate all the equations.

step (iv) : Result is the value of the final state..

Q)

(i) write the eqns for each state individually by considering only incoming edges.

$$q_1 = q_1 0 + \varepsilon \quad \text{—①}$$

ε → q_1 is initial state

$$q_2 = q_1 1 + q_2 1 \quad \text{—②}$$

$$q_3 = q_2 0 + q_3 0 + q_3 1 \quad \text{—③}$$

Now, simplify the eqns including final states.

If there are multiple final states then, apply union operation.

→ By using Arden's Theorem

$$\begin{bmatrix} R = Q + RP \\ R = QP^* \end{bmatrix} \text{---} ④ \quad ; \quad [\varepsilon + RR^* = R^*]$$

eq ① $\Rightarrow q_1 = \varepsilon + q_1 0$ ---⑤

on comparing eq's ④, ⑤ ; we get

$R = q_1$, $Q = \varepsilon$, $RP = q_1 0$

$\therefore q_1 = 0^*$

eq ② $\Rightarrow q_2 = q_1 1 + q_2 1$

$\Rightarrow q_2 = 0^* 1 + q_2 1$ ---⑥

on comparing eq's ④, ⑥ ; we get

$$q_2 = 0^* 1 1^*$$

∴ Resultant Regular Expression is,

$$RE = q_1 + q_2$$

(∵ q_1 and q_2 are final states, we appl perform union operation on q_1 and q_2)

$\Rightarrow RE = 0^* + 0^* 1 1^*$

$= 0^* (\varepsilon + 11^*)$

$= 0^* 1^*$ $\quad (\because \varepsilon + 11^* = 1^*)$

(b)

(a) $q_1 = q_1 0 + q_3 0 + \varepsilon$ ---①

$q_2 = q_2 1 + q_1 1 + q_3 1$ ---②

$q_3 = q_2 0$ ---③

The above eq's cannot be simplified by using Adenn's Theorem.

substitute eq (3) in eq (2)

$\therefore$ eq (2) $\Rightarrow q_2 = q_2 1 + q_1 1 + q_2 01$

$\Rightarrow q_2 = q_1 1 + q_2 (1 + 01)$

$$\begin{bmatrix} R = Q + RP \\ R = QP^* \end{bmatrix}$$

$\Rightarrow q_2 = q_1 1 (1+01)^*$ —(4)

Now, eq (3) $\Rightarrow q_3 = q_1 1 (1+01)^* 0$ —(5)

Then, eq (1) $\Rightarrow q_1 = q_1 0 + q_1 1 (1+01)^* 00 + \varepsilon$

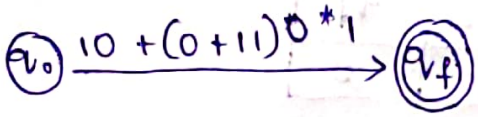
$\Rightarrow q_1 = q_1 (0 + 1 (1+01)^* 00) + \varepsilon$

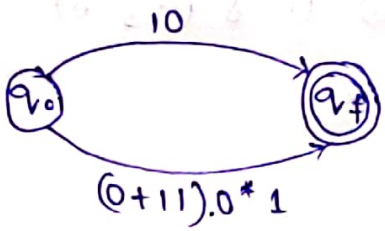
$\Rightarrow q_1 = \varepsilon + q_1 (0 + 1 (1+01)^* 00)$

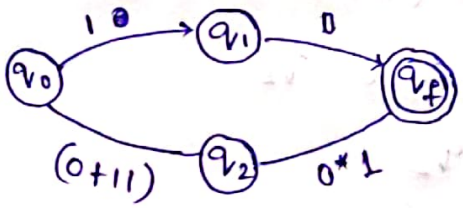
$$\begin{bmatrix} R = Q + RP \\ R = QP^* \end{bmatrix}$$

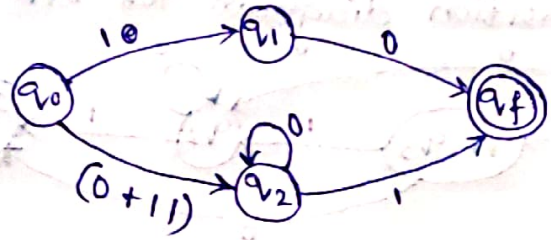
$\Rightarrow q_1 = (0 + 1 (1+01)^* 00)^*$

Q) Design DFA from the given RE : $10 + (0+11) 0^* 1$

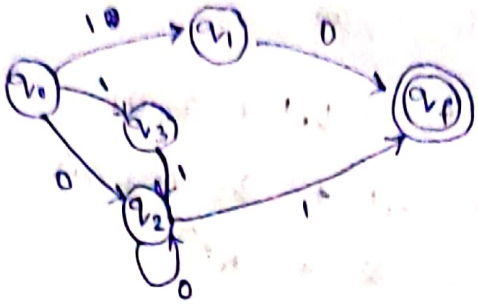
step 1 :- 

step 2 :- 

step 3 :- 

step 4 :- 

step 5 :-



NFA

step 6 :- Transition Table for NFA

	0	1
q_0	q_2, q_3	q_1, q_3
q_1	q_f	$\emptyset$
q_2	q_2	q_f
q_3	$\emptyset$	q_2

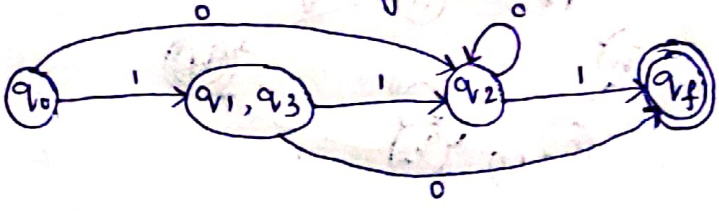
step 7 :- Transition Table for DFA

	0	1
q_0	q_2	q_1, q_3
q_1, q_3	q_f	q_2
q_2	q_2	q_f
q_f	$\emptyset$	$\emptyset$

$$\delta((q_1, q_3), 0) = \delta(q_1, 0) \cup \delta(q_3, 0)$$
$$= q_f \cup \emptyset$$
$$= q_f$$

$$\delta((q_1, q_3), 1) = \delta(q_1, 1) \cup \delta(q_3, 1)$$
$$= \emptyset \cup q_2$$
$$= q_2$$

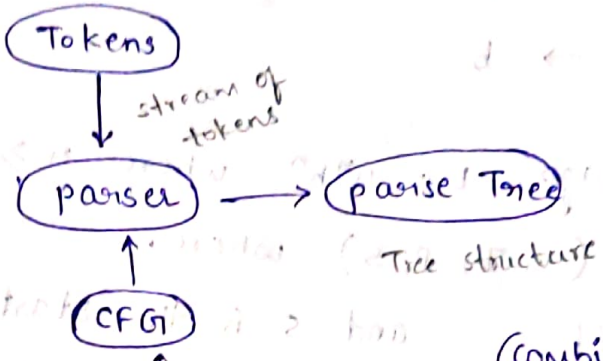
step 8 :- Transition diagram for DFA



03/10/2023 UNIT - 2

Syntax Analysis

A parser or syntax analysis is a process which takes the input string w and produces either a parse tree (syntactic structure) or generates the syntactic errors.



production rules - (combinations of terminal & non-terminal symbols)

S = T + const

↓ start

non-terminal symbols → other than capitals

capitals

Every non-terminal symbol is ended with ~~non~~-terminal

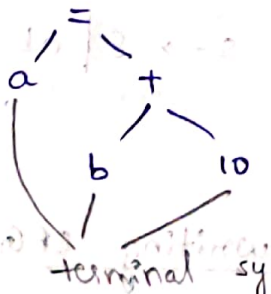
Ex: a = b + 10

S → T + const

T → b

const → 10

S → a



Parse Tree is generated by using terminal & non-terminal symbols.

Context Free Grammar (CFG)

The Context Free Grammar is a collection of the following :-

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

where, V : set of non-terminal symbols

T : set of terminal symbols (other than capitals)

S : start symbol

P : set of production rules

lexical analyzer & syntax analyzer work together

Ex: (S, V, T, A, B)

The production rules are given in the following form

Non Terminal $\longrightarrow (V \cup T)^*$

Ex: $E \rightarrow E+E \mid E*E \cong E \rightarrow E+E \mid E*E$

$E \rightarrow T \mid T*F$

$T \rightarrow a$

$F \rightarrow b$

Q) The language $L = a^n b^n$ where $n \geq 1$

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

$V = \{S\}$ and S is the start symbol.

$T = \{a, b\}$ Give the production rules. ie, $P = ?$

Ans: $n \geq 1$ implies that $L = \{ab, aabb, aaabbb, \ldots\}$

↓ min length string

$P = \{$

$S \rightarrow aSb$ $\because a \varepsilon b = ab$

$S \rightarrow \varepsilon \mid ab$ for only ab

$S \rightarrow \varepsilon \mid aSb \mid$ for any no: of a's and b's

$\}$

Rules for writing CFG

1) A single non-terminal should be at LHS
2) The rule should be always in the form of LHS $\longrightarrow$ RHS where, RHS may be the combination of terminal and non-terminal symbols and LHS is non-terminal symbol
3) The null derivation can be specified as

non-terminal $\longrightarrow \varepsilon$

4) One of the non-terminals should be start symbol and conventionally, we should write the write the rules for these non-terminal symbols.

$a \underset{\varepsilon}{S} b$

ab

$a\, a \underset{\varepsilon}{S} b\, b$

$aabb$

$a S b$

$a\, a S b\, b$

$a\, a\, a \underset{\varepsilon}{S} b\, b\, b$

$aaabbb$

Q) Write a CFG for while statement in C language

Ans:- P = { S → while (condition) stmt
(S → non-terminal)

condition → id relOp id

relOp → > | >= | < | <= | != | ==

S → while (condition) stmt L

stmt L → stmt L | stmt

while (cond)

while (cond)
{
}

}

V = { S, condition, relOp, stmt L }

T = { while, (,), <, >, <=, >=, !=, ==

Q) what is the string generated by CFG A → (A) A

Every non-terminal symbol should be ended with the terminal symbol.

Here, A is not ending with terminal symbol.

As there is no terminal symbol generation by the given CFG, we cannot derive any string. from

A → (A) A

A → ( ) A

A → ( ) | ε

Possible strings :-

take, A → (A) A

A → (ε) ε

A → ( )

A → (( )) ( )

A → (( )) ε

A → (( ))

Derivation and Parse Trees

Derivation from S means, generation of string w from S.

For constructing derivation, two rules are important.

(1) Choice of non-terminal from several others.

(2) Choice of rule from production rules for corresponding non-terminal.

Derivation can be of two types

1) Left most derivation (LMD) — replace non-ter with ter symbol from left end

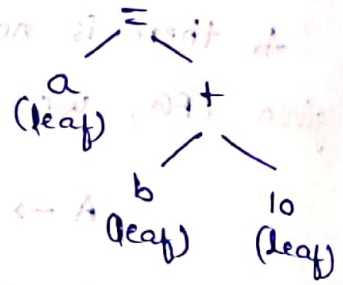
2) Right most derivation (RMD) — replace ter with non Ter from right most side

Parsing:

The syntax Analyzer is the second phase in compilation. The syntax analyzer (parser) basically checks for the syntax of language.

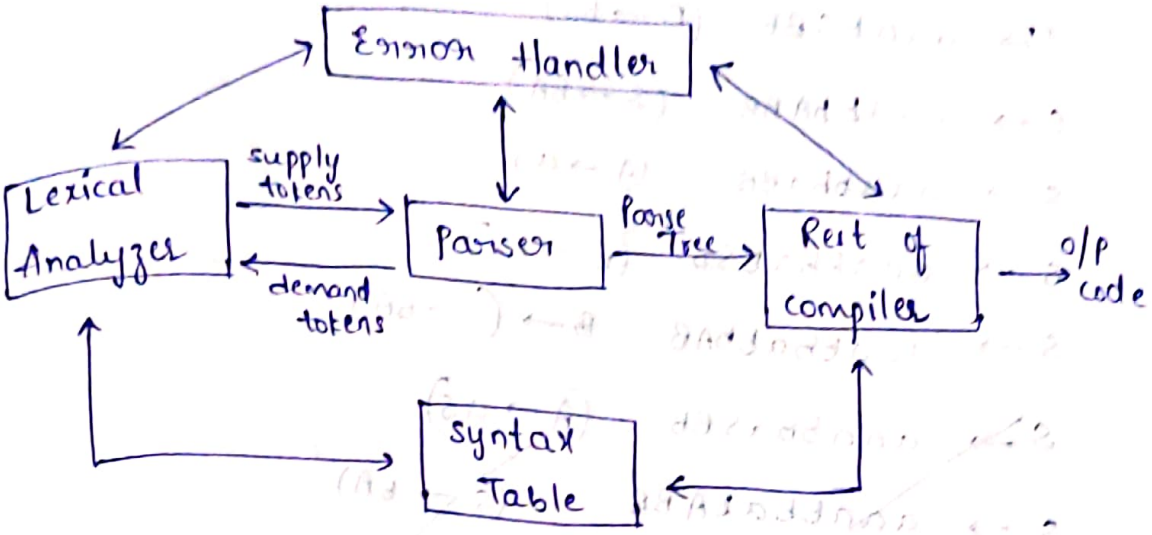
Ex: a = b + 10

This statement is first given to lexical analyzer. It will divide into group of tokens. Syntax analyzer takes the tokens as input and generates a tree like structure called parse tree.



Role of Parser

In the process of compilation, the parser and LA work together. That means, when parser requires stream of tokens, it invokes lexical analyzer. In return, the lexical analyzer supplies tokens to syntax analyzer.



The parser collects sufficient no. of tokens and builds a parse tree. Thus by building the parse tree, parser finds the syntactical error if any. It is also necessary that, the parser should recover from commonly occuring errors, so that remaining task of process the i/p can be continued.

Why Lexical analyzer and Syntax analyzer are separated out?

→ It speeds up the process of compilation

→ The errors in the source input can be identified precisely (correctly).

04/10/2023

Q) Let G be context free Grammar for which the production rules are given as below.

$S \rightarrow aB \mid bA$

$A \rightarrow a \mid aS \mid bAA$

$B \rightarrow b \mid bS \mid aBB$.

Derive the string aaabbabbba using above grammar.

Ans:

$w = aaabbabbba$

$S \rightarrow aB$

$S \rightarrow aaBB \quad (B \rightarrow aBB)$

$S \rightarrow aaaBBB \quad (B \rightarrow aBB)$

~~$S \rightarrow aaabBB \quad (B \rightarrow b)$~~

S → aaabSBB (B → bS)

S → aaabbABB (S → bA)

S → aaabbaBB (A → a)

S → aaabbabSB (B → bS)

S → aaabbabbAB (B → bA)

S → aaabbabbbA (A → a)

S → aaabbabbbb (A → bAA)

LMD

S → aB

S → aaBB (B → aBB)

S → aaaBBB (B → aBB)

S → aaabBB (B → b)

S → aaabbB (B → b)

S → aaabbaBB (B → aBB)

S → aaabbabB (B → b)

S → aaabbabbS (B → bS)

S → aaabbabbbA (S → bA)

S → aaabbabbbba (A → a)

RMD

S → aB

S → aaBB (B → aBB)

~~S → aaaBBB (B → aBB)~~

S → aaBbS (B → bS)

S → aaBbbA (S → bA)

S → aaBbbba (A → a)

S → aaaBBbbba (B → aBB)

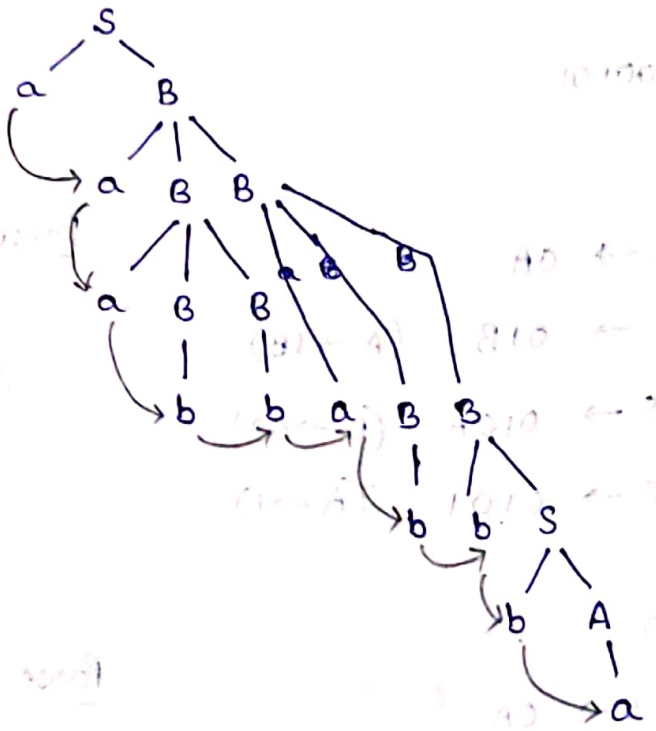
S → aaaBbbbba (B → b)

$S \rightarrow aaabSbbba \quad (B \rightarrow bS)$

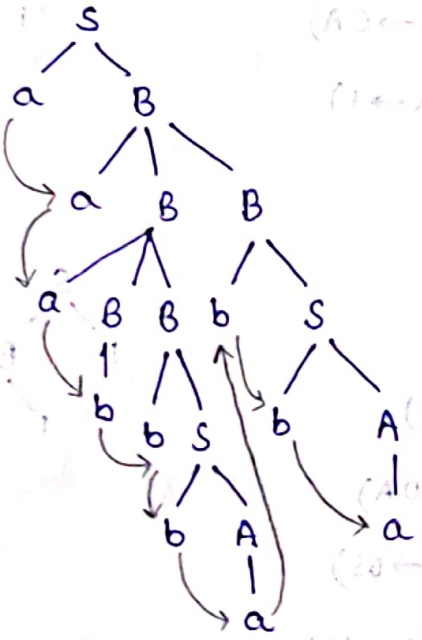
$S \rightarrow aaabbAbbba \quad (S \rightarrow bA)$

$S \rightarrow aaabbabbba \quad (A \rightarrow a)$

Parse Tree for LMD



Parse Tree for RMB



Q) $S \rightarrow 0A \mid 1B \mid 0 \mid 1$

$A \rightarrow 0S \mid 1B \mid 1$

$B \rightarrow 0A \mid 1S$

(i) 0101

(ii) 1100101

Ans:

(i) LMD

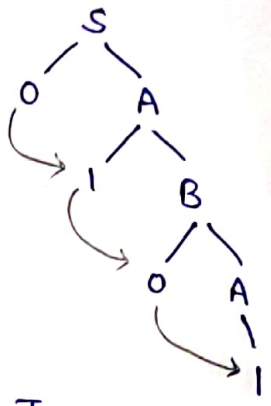
$S \rightarrow 0A$

$S \rightarrow 01B \quad (A \rightarrow 1B)$

$S \rightarrow 010A \quad (B \rightarrow 0A)$

$S \rightarrow 0101 \quad (A \rightarrow 1)$

Parse Tree



RMD

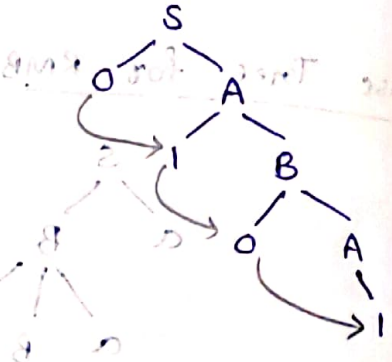
$S \rightarrow 0A$

$S \rightarrow 01B \quad (A \rightarrow 1B)$

$S \rightarrow 010A \quad (B \rightarrow 0A)$

$S \rightarrow 0101 \quad (A \rightarrow 1)$

Parse Tree



(ii) LMD

$S \rightarrow 1B$

$S \rightarrow 11S \quad (B \rightarrow 1S)$

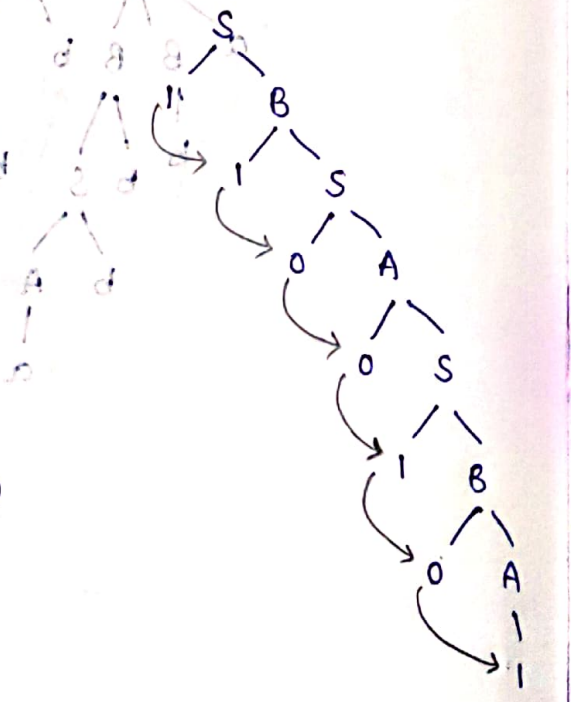
$S \rightarrow 110A \quad (S \rightarrow 0A)$

$S \rightarrow 1100S \quad (A \rightarrow 0S)$

$S \rightarrow 11001B \quad (S \rightarrow 1B)$

$S \rightarrow 1100101A \quad (B \rightarrow 0A)$

$S \rightarrow 1100101 \quad (A \rightarrow 1)$



RMD

$S \rightarrow 1B$

$S \rightarrow 11S \quad (B \rightarrow 1S)$

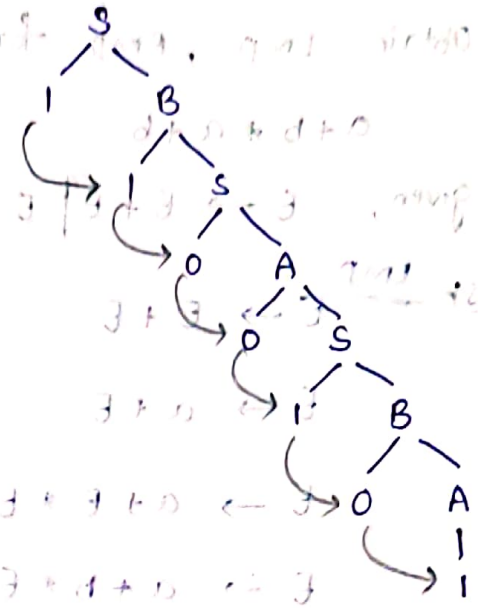
$S \rightarrow 110A \quad (S \rightarrow 0A)$

$S \rightarrow 1100S \quad (A \rightarrow 0S)$

$S \rightarrow 11001B \quad (S \rightarrow 1B)$

$S \rightarrow 110010A \quad (B \rightarrow 0A)$

$S \rightarrow 1100101 \quad (A \rightarrow 1)$



HW

Q) Consider the grammar

$S \rightarrow (L) \mid a$

$L \rightarrow L, S \mid S$

(i) what are the terminals and non-terminals and start symbol

(ii) Find the parse trees for the following sentences.

(1) (a,a)

(2) (a, (a, a))

(3) (a, ((a,a), (a,a)))

(iii) Construct LMD and RMD

(iv) what languages thus the grammar generate.

Ans (iv) This grammar generates the strings with well-formedness of parantheses.

(i) Terminal symbols :-

$T = \{ (,), , , a \}$

Non-terminal Symbols :-

$V = \{ L, S \}$

start symbol = S

P = $S \rightarrow (L) \mid a$

Q) Obtain LMD, RMD for the string

a + b * a + b

given, $E \rightarrow E+E \mid E-E \mid E*E \mid E \mid a \mid b$

Ans: LMD

$E \rightarrow E + E$

$E \rightarrow a + E \quad (E \rightarrow a)$

$E \rightarrow a + E * E \quad (E \rightarrow E * E)$

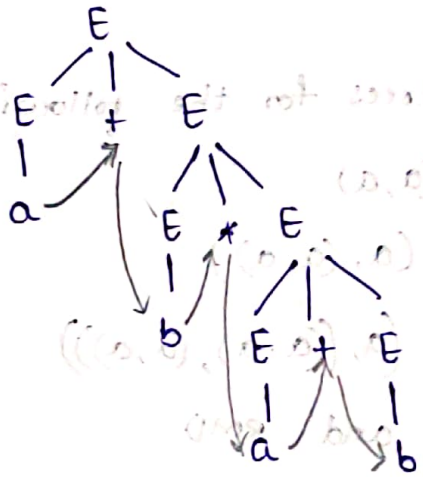
$E \rightarrow a + b * E \quad (E \rightarrow b)$

$E \rightarrow a + b * E + E \quad (E \rightarrow E + E)$

$E \rightarrow a + b * a + E \quad (E \rightarrow a)$

$E \rightarrow a + b * a + b \quad (E \rightarrow b)$

Parse Tree



RMD

$E \rightarrow E + E$

$E \rightarrow E + b \quad (E \rightarrow b)$

$E \rightarrow E * E + b \quad (E \rightarrow E * E)$

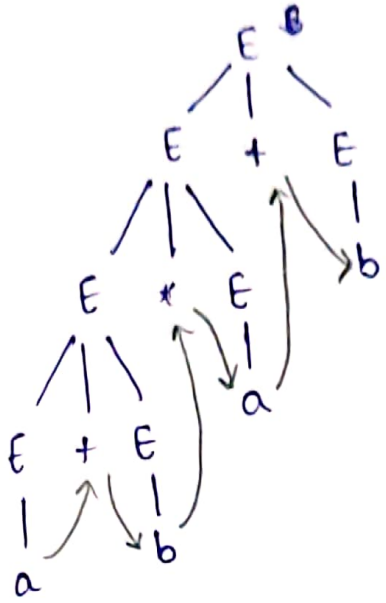
$E \rightarrow E * a + b \quad (E \rightarrow a)$

$E \rightarrow E + E * a + b \quad (E \rightarrow E + E)$

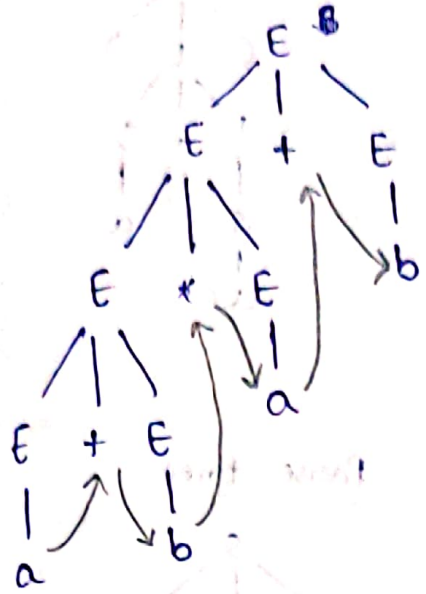
$E \rightarrow E + b * a + b \quad (E \rightarrow b)$

$E \rightarrow a + b * a + b \quad (E \rightarrow a)$

Parse Tree



Parse Tree



HW

Sol :-

(ii), (iii)

i) (a, a)

$S \to (L) \mid a$

$L \to L,S \mid S$

LMD :-

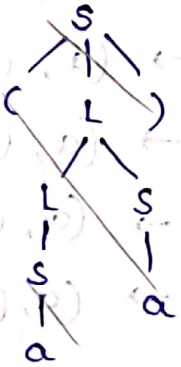
$S \to (L)$

$S \to (L,S)$ $(L \to L,S)$

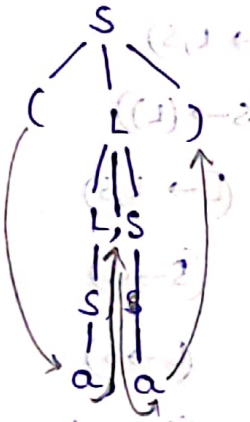
$S \to (S,S)$ $(L \to S)$

$S \to (a,S)$ $(S \to a)$

$S \to (a,a)$



Parse tree :-



RMD:-

$S \to (L)$

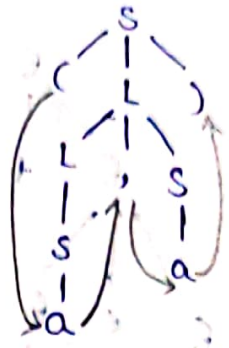
$S \to (L, S)$ $(L \to L, S)$

$S \to (L, a)$ $(S \to a)$

$S \to (S, a)$ $(L \to S)$

$S \to (a, a)$ $(S \to a)$

Parse tree



2) $(a, (a, a))$

LMD:-

$S \to (L)$

$S \to (L, S)$ $(L \to L, S)$

$S \to (S, S)$ $(L \to S)$

$S \to (a, S)$ $(S \to a)$

$S \to (a, (L))$ $(S \to (L))$

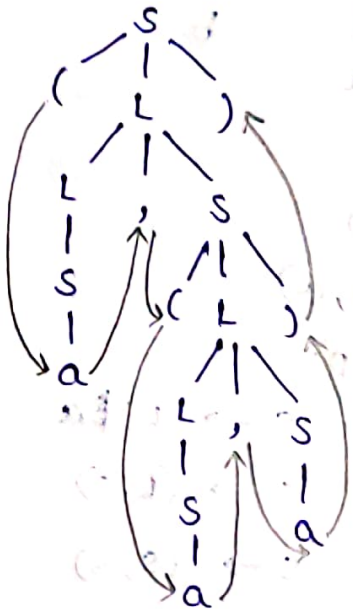
$S \to (a, (L, S))$ $(L \to L, S)$

$S \to (a, (S, S))$ $(L \to S)$

$S \to (a, (a, S))$ $(S \to a)$

$S \to (a, (a, a))$ $(S \to a)$

Parse tree



RMD:-

$S \to (L)$

$S \to (L, S)$ $(L \to L, S)$

$S \to (L, (L))$ $(S \to (L))$

$S \to (L, (L, S))$ $(L \to L, S)$

$S \to (L, (L, a))$ $(S \to a)$

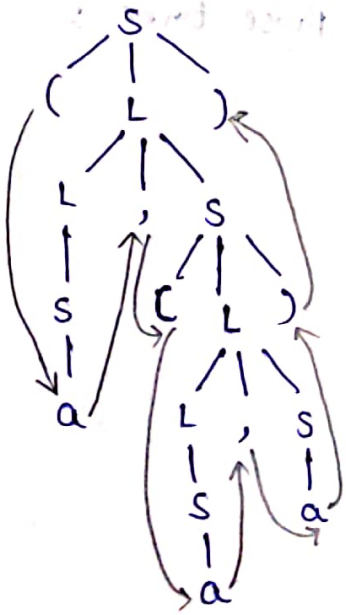
$S \to (L, (S, a))$ $(L \to S)$

$S \to (L, (a, a))$ $(S \to a)$

$S \to (S, (a, a))$ $(L \to S)$

$S \to (a, (a, a))$ $(S \to a)$

Parse tree



3) (a, ((a,a), (a,a)))

LMD :-

$S \to (L)$

$S \to (L,S)$ $(L \to L,S)$

$S \to (S,S)$ $(L \to S)$

$S \to (a,S)$ $(S \to a)$

$S \to (a,(L))$ $(S \to (L))$

$S \to (a,(L,S))$ $(L \to L,S)$

$S \to (a,(S,S))$ $(L \to S)$

$S \to (a,((L),S))$ $(S \to (L))$

$S \to (a,((L,S),S))$ $(L \to L,S)$

$S \to (a,((S,S),S))$ $(L \to S)$

$S \to (a,((a,S),S))$ $(S \to a)$

$S \to (a,((a,a),S))$ $(S \to a)$

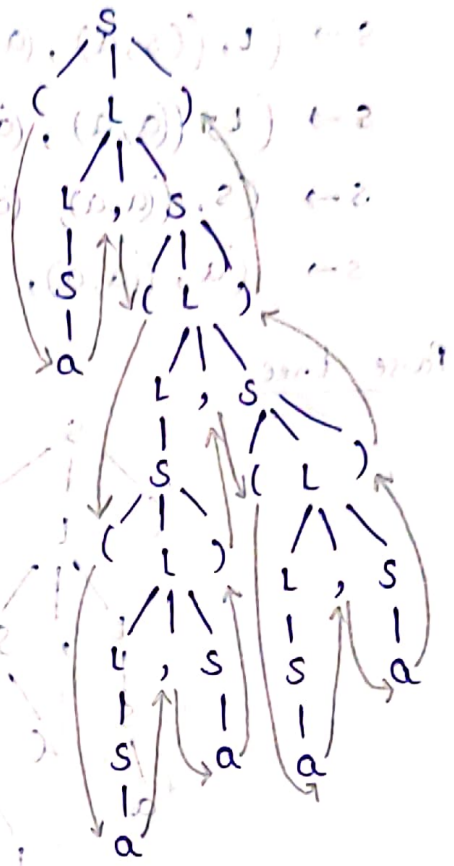
$S \to (a,((a,a),(L)))$ $(S \to (L))$

$S \to (a,((a,a),(L,S)))$ $(L \to L,S)$

$S \to (a,((a,a),(S,S)))$ $(L \to S)$

$S \to (a,((a,a),(a,S)))$ $(S \to a)$

$S \to (a,((a,a),(a,a)))$ $(S \to a)$



RMD :-

$S \to (L)$

$S \to (L,S)$ $(L \to L,S)$

$S \to (L,(L))$ $(S \to (L))$

$S \to (L,(L,S))$ $(L \to L,S)$

$S \to (L,(L,(L,S)))$ $(S \to (L))$

$S \to (L,(L,(L,a)))$ $(S \to a)$

$S \to (L,(L,(S,a)))$ $(L \to S)$

$S \to (L,(L,(a,a)))$ $(S \to a)$

$S \rightarrow (L, ((L,S), (a,a)))$ $(L \rightarrow L,S)$

$S \rightarrow (L, ((L,a), (a,a)))$ $(S \rightarrow a)$

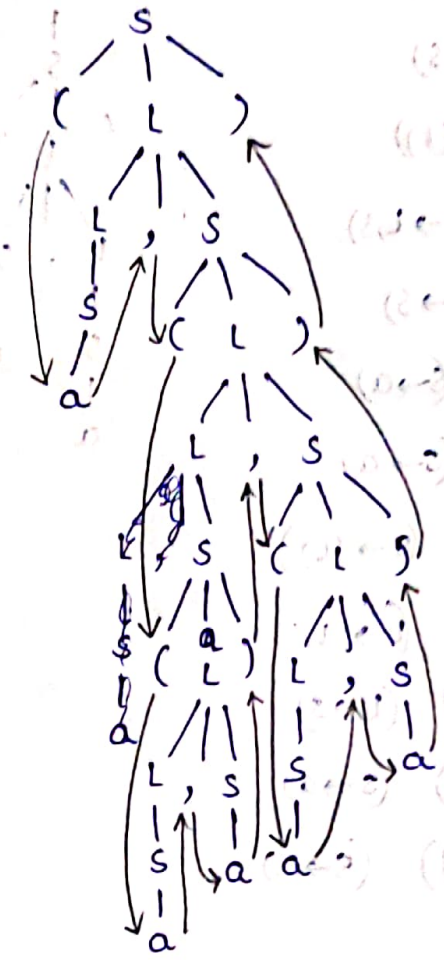
$S \rightarrow (L, ((S,a), (a,a)))$ $(L \rightarrow S)$

$S \rightarrow (L, ((a,a), (a,a)))$ $(S \rightarrow a)$

$S \rightarrow (S, ((a,a), (a,a)))$ $(L \rightarrow S)$

$S \rightarrow (a, ((a,a), (a,a)))$ $(S \rightarrow a)$

Parse tree



05/10/2023

Ambiguous Grammar

A grammar G is said to be ambiguous if it generates more than one parse tree for sentences of language L(G)

We need to remove the ambiguity so that there would be a single parse tree.

To eliminate the ambiguity, we use some techniques

Q) $E \rightarrow E+E \mid E*E \mid (E) \mid id$

(i) $\omega = id + id + id$

LMD

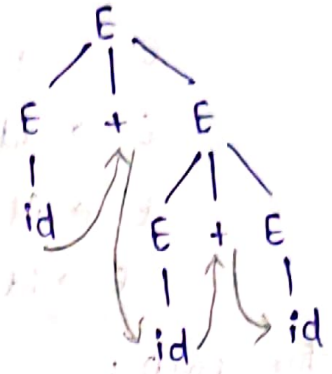
$E \rightarrow E + E$

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

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

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

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



RMD

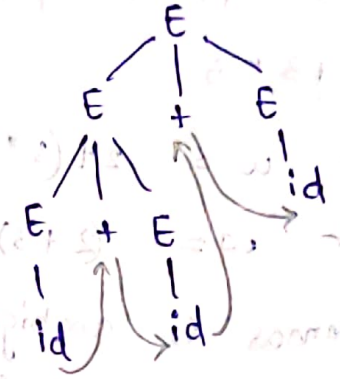
$E \rightarrow E + E$

$E \rightarrow E + id$

$E \rightarrow E + E + id$

$E \rightarrow E + id + id$

$E \rightarrow id + id + id$



(ii) $\omega = id + id * id$

LMD

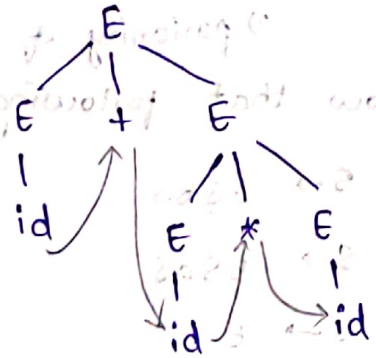
$E \rightarrow E + E$

$E \rightarrow id + E$

$E \rightarrow id + E * E$

$E \rightarrow id + id * E$

$E \rightarrow id + id * id$



Parse Tree 1

RMD

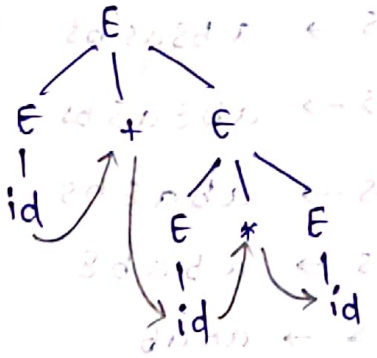
$E \rightarrow E + E$

$E \rightarrow E + E * E$

$E \rightarrow E + E * id$

$E \rightarrow E + id * id$

$E \rightarrow id + id * id$



LMD:-

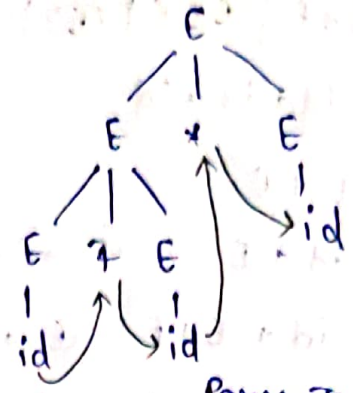
$E \rightarrow E * E$

$E \rightarrow E + E * E$

$E \rightarrow id + E * E$

$E \rightarrow id + id * E$

$E \rightarrow id + id * id$



Parse Tree 2

for LMD;

there are two possible parse trees.

If $\omega = 2 + 3 * 5$

Parse Tree 1 :- $\omega = 2 + (3 * 5)$ ✓

Parse Tree 2 :- $\omega = (2 + 3) * 5$ ✗

∴ Given grammar is ambiguous.

So, To eliminate this ambiguity :-

consider- 1) associativity

2) priority of operators

Q) Show that following grammar is ambiguous.

$S \rightarrow aSbS$

$S \rightarrow bSaS$

$S \rightarrow \varepsilon$

$\omega = abab$

$S \rightarrow aSbS$

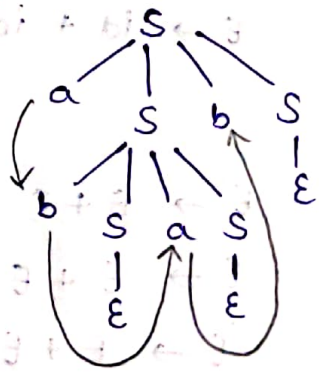
$S \rightarrow abSaSbS$ $(S \rightarrow bSaS)$

$S \rightarrow abSaSbS$ $(S \rightarrow \varepsilon)$

$S \rightarrow abaSbS$ $(S \rightarrow \varepsilon)$

$S \rightarrow abab\varepsilon$ $(S \rightarrow \varepsilon)$

$S \rightarrow abab$



Parse Tree 1

$S \rightarrow aSbS$

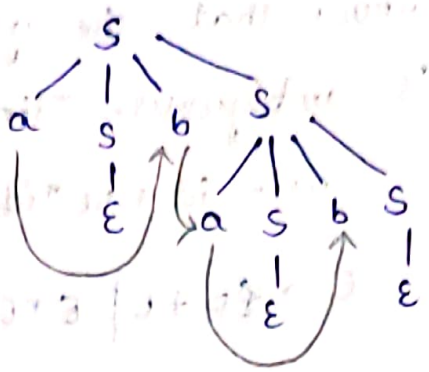
$S \rightarrow a\varepsilon bS$

$S \rightarrow abaSbS$

$S \rightarrow aba\varepsilon bS$

$S \rightarrow ababS$

$S \rightarrow abab$



Parse Tree 2

Thus, the given grammar is ambiguous.

Hence proved

Removal of Ambiguity

The ambiguity in the given grammar occurs due to use of single non-terminal more in multiple definitions or production rules.

> If single non-terminal is being used repeatedly, then the ambiguity arises

By introducing multiple non-terminals, the ambiguity can be removed.

For removing ambiguity of grammar, defining arithmetic expressions, we apply two rules.

1) If the grammar has left associative operators $(+, -, *, /)$ then, apply Left Recursion.

Ex:- $E \rightarrow E + T$ (Left recursion grammar)

2) If the grammar has right associative operator $(\uparrow)$ (exponential operator) then, apply Right Recursion.

Ex:- $E \rightarrow T + E$ (Right recursive production rule)

Q) Show that the grammar $E \to E+E \mid E*E \mid (E) \mid id$ is ambiguous. Eliminate the ambiguity.

$w = id + id * id$ ⇒ left recursive (∵ +, *)

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

$E \to E+E$ (∵ $E \to E+E$; $E \to E+T \mid T$; $E \to T$)

$\Rightarrow E \to E+T \mid T$

~~$E \to E*E$~~

~~$T \to T*F$~~

$\Rightarrow T \to T*F \mid F$ (∵ $E \to T$ & $E \to E*E$, $T \to T*T$, $T \to T*F \mid F$ → $T \to F$)

$\Rightarrow F \to (E) \mid id$

LMD

$E \to E+T$

$E \to T+T$ ($E \to T$)

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

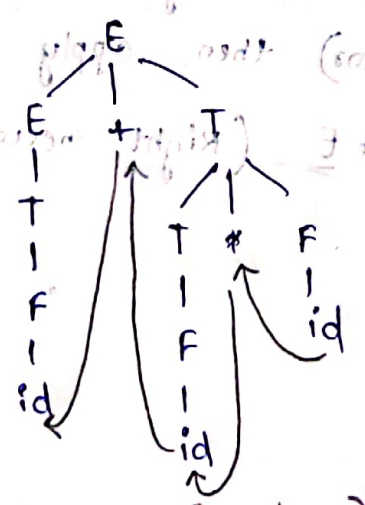
$E \to id+T$ ($F \to id$)

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

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

$E \to id+id*F$

$E \to id+id*id$



2 + (3 * 5)

2 + 15 ① ⇒ 17

② ⇒

RMD

$E \rightarrow E + T$

$E \rightarrow E + F * F$

$E \rightarrow E + T * id$

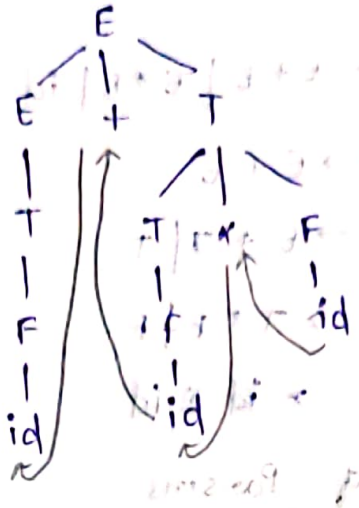
$E \rightarrow E + F * id$

$E \rightarrow E + id * id$

$E \rightarrow T + id * id$

$E \rightarrow F + id * id$

$E \rightarrow id + id * id$



Q) bExp → bExp OR bExp
| bExp AND bExp
| Not bExp
| True
| False

bExp → bExp OR bExp

~~bExp → T OR bExp~~

bExp → bExp OR T | T

T → T OR F | F

F → F AND G | G

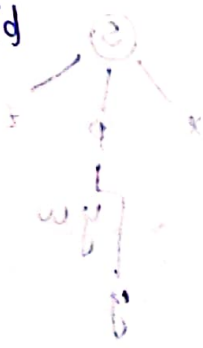
G → Not H | H

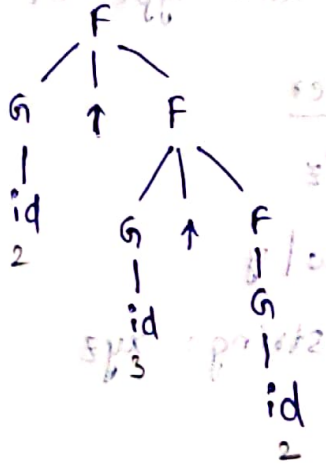
H → True | False

Q) $2 \uparrow 3 \uparrow 2$

$F \rightarrow G \uparrow F \mid G$

$G \rightarrow id$





$(2 \uparrow (3 \uparrow 2))$

$= 2^{3^2} = 2^9$

06/10/2023

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

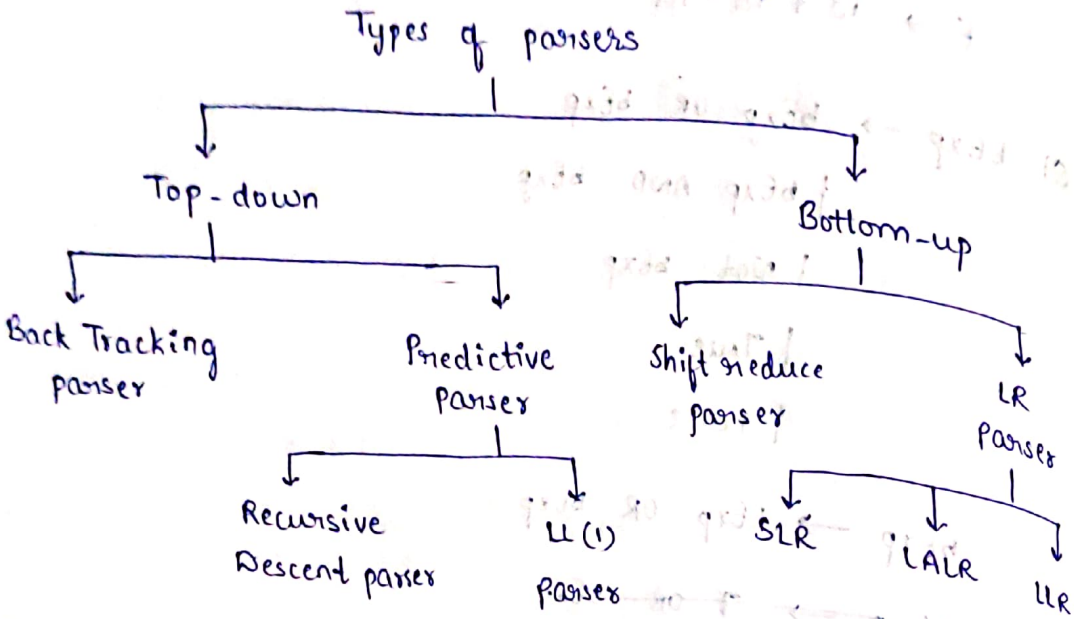
$E \rightarrow E+E$

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

Types of Parsers



Top-down Parser

When parse tree can be constructed from root & expanded to leaves then, such type of parser is called Top-down Parser.

Bottom-up Parser

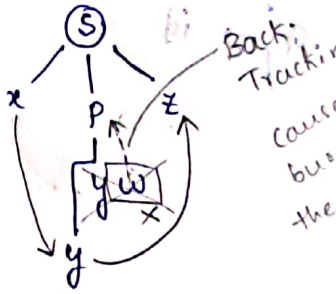
When the parse tree can be constructed from leaves to root then, such type of parser is called Bottom-up Parser.

Top-down Parser

$S \rightarrow xPz$

$P \rightarrow yw \mid y$

* string: xyz



Problems of Top-down Parser

» Back tracking
» Left Recursion
» left Factoring
» Ambiguity

Left Recursion

case (i) :- $A \rightarrow A\alpha \mid \beta$

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid \varepsilon$$

Ex: $E \rightarrow E+T \mid a$
$A \rightarrow A\alpha \quad \beta$
$$E \rightarrow aE'$$
$$E' \rightarrow TE' \mid \varepsilon$$

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

First, identify if it is left recursion.

Take, $E \rightarrow E+T \mid T$
$(A \rightarrow A\alpha \mid \beta$ then, $A \rightarrow \beta A'$, $A' \rightarrow \alpha A' \mid \varepsilon)$
$$E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \varepsilon$$

Thus, left recursion is eliminated.

Now, $T \rightarrow T*F \mid F$
$\left(A \rightarrow A\alpha \mid \beta \right.$ then, $A \rightarrow \beta A'$, $A' \rightarrow \alpha A' \mid \varepsilon\left.\right)$
$$T \rightarrow FT'$$
$$T' \rightarrow *FT' \mid \varepsilon$$

And, $F \rightarrow (E) \mid id$

∴ After removing left recursion,

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

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

Q) $S \rightarrow S0S1S \mid 01$

$\left(A \rightarrow A\alpha \mid \beta \quad \text{then,} \quad A \rightarrow \beta A', \; A' \rightarrow \alpha A' \mid \varepsilon\right)$

$S \rightarrow 01S'$

$S' \rightarrow 0S1SS' \mid \varepsilon$

Q) $S \rightarrow (L) \mid x$

$L \rightarrow L,S \mid S$

↳ terminal

Take, $L \rightarrow L,S \mid S$

$\left(A \rightarrow A\alpha \mid \beta \quad \text{then,} \quad A \rightarrow \beta A', \; A' \rightarrow \alpha A' \mid \varepsilon\right)$

$L \rightarrow SL'$

$L' \rightarrow ,SL' \mid \varepsilon$

After removing left recursion,

$S \rightarrow (L) \mid x$

$L \rightarrow SL'$

$L' \rightarrow ,SL' \mid \varepsilon$

case (ii): $A \to A\alpha_1 \mid A\alpha_2 \mid \beta_1 \mid \beta_2$

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

$A' \to \alpha_1 A' \mid \alpha_2 A' \ldots \mid \varepsilon$

Q) $expr \to expr + expr \mid expr * expr \mid id$

($A \to A\alpha_1 \mid A\alpha_2 \mid \beta$)

$expr \to id\,expr' \mid expr'$

~~$expr' \to +expr' \mid *expr' \mid \varepsilon$~~

$expr' \to +expr\ expr' \mid *expr\ expr' \mid \varepsilon$

10/10/2023

Consider the following grammar

$A \to ABb \mid Aa \mid a$

$B \to Be \mid d$

remove the left recursion.

(i) $A \to ABb \mid a$ ($A \to A\alpha \mid \beta$) then $A \to \beta A'$, $A' \to \alpha A' \mid \varepsilon$

$A \to aA'$

$A' \to BbA' \mid \varepsilon$

(ii) $A \to Aa \mid a$ ($A \to A\alpha \mid \beta$) then, $A \to \beta A'$, $A' \to \alpha A' \mid \varepsilon$

$A \to aA'$

$A' \to aA' \mid \varepsilon$

(iii) $B \to Bc \mid d$ ($A \to A\alpha \mid \beta$) then, $A \to \beta A'$, $A' \to \alpha A' \mid \varepsilon$

$B \to dB'$

$B' \to eB' \mid \varepsilon$

∴ The production rules after removing left recursion are:

$A \to aA'$

$A' \to aA' \mid BbA' \mid \varepsilon$

$B \to dB'$

$B' \to eB' \mid \varepsilon$

Left Factoring

(It) is used when it is not clear that which of the two alternatives is used to expand the non-termin

If $A \to \alpha\beta_1 | \alpha\beta_2$ is a production, then it is not possible to take a decision whether to choose Rule 1 or Rule 2. In such situations, the above grammar can be left factored as :-

$A \to \alpha A'$ when $A \to \alpha\beta_1 | \alpha\beta_2$ is given.

$A' \to \beta_1 | \beta_2 \cdots$

Ex :-

$S \to iEtS | iEtSeS | a$

$E \to b$

take, $S \to \underbrace{iEtS}_{\alpha\beta_1} | \underbrace{iEtSeS}_{\alpha\beta_2} | a$ then, $A \to \alpha A'$

$A' \to \beta_1 | \beta_2$

$S \to iEtSS' | a$

$S' \to \varepsilon | eS$

$E \to b$

Q) Do left factoring in the following grammar

$A \to aAB | aA | a$

$B \to bB | b$

Ans :- (1) $A \to \underbrace{aAB}_{\alpha\beta_1} | \underbrace{aA}_{\alpha\beta_2} | \underbrace{a}_{\alpha\beta_3}$ then, $A \to \alpha A'$

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

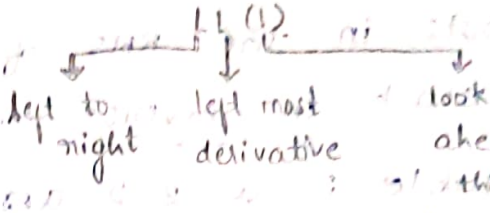
$A \to aA'$

$A' \to AB | A | \varepsilon$

(ii) $B \rightarrow bB \mid b$
$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$
$B \rightarrow bB'$
$B' \rightarrow B \mid \varepsilon$

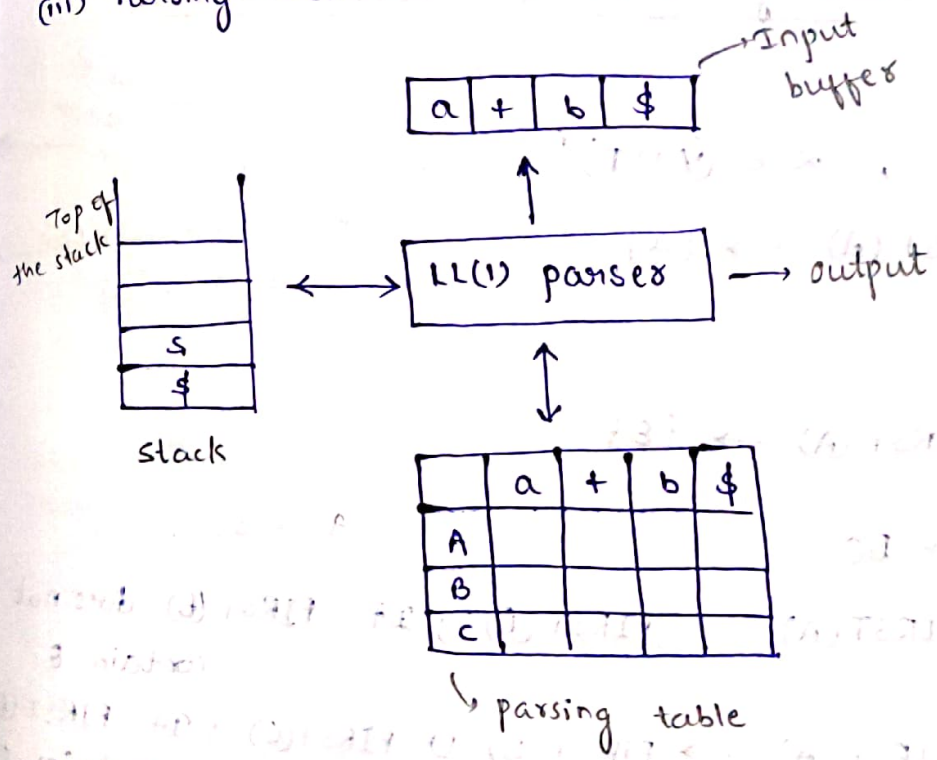
then, $A \rightarrow \alpha A'$
$A' \rightarrow \beta_1 \mid \beta_2$

Predictive LL(1) parser



» For LL(1) parser,
the first L means that, the input is scanned from left to right.
the second L means that, it uses left most derivation for input string.
the number 1 in the input string means that, it uses only 1 input symbol (look ahead symbol) to predict the parsing process.

» The data structures used by LL(1) parser are
(i) Input buffer
(ii) Stack
(iii) Parsing table



» LL(1) parser uses input buffer to store the input tokens
$ indicates that, it is the end of the string.
» stack is used to hold the left sentential form.
The symbols in the RHS of the rule are pushed into the stack in reverse order.
» Parsing table is a 2-D array. It has rows for the non-terminal symbols and columns for terminal symbols.

Construction of Predictive LL(1) parser

For the construction of this parser, we need to follow the steps given below:

1) Computation of FIRST & FOLLOW functions
2) Construct the predictive parsing table using the FIRST & FOLLOW functions.
3) Parse the input string with the help of predictive parsing table

Rules for constructing FIRST & FOLLOW functions

FIRST :- [it is found for symbols that are on the LHS of Production rule]

1) If $A \rightarrow a\alpha$, $\alpha \in (V \cup T)^*$, then, FIRST(A) $\rightarrow$ {a}

Ex: $A \rightarrow aBaC$ (a: terminal; BaC: combination of terminals & non-terminals)

FIRST(A) $\rightarrow$ FIRST(aBaC)
$\Rightarrow$ FIRST(A) $\rightarrow$ {a}

2) If $A \rightarrow \varepsilon$
then, FIRST(A) $\rightarrow$ {ε}

3) If $A \rightarrow BC$
then, FIRST(A) $\rightarrow$ FIRST(B); If FIRST(B) does not contain ε

$A \rightarrow \overbrace{BC}^{\text{all are non-terminal symbols}}$

4) If FIRST(A) $\rightarrow$ FIRST(B) $\cup$ FIRST(C); If FIRST(B) contains ε

FOLLOW :- [It is found for symbols that are on the RHS of production rule]

1) If 'S' is a starting symbol

then, FOLLOW (S) = { $ }

2) If $A \rightarrow \alpha B \beta$

then, FOLLOW (B) = FIRST (β) ; if FIRST (β) doesn't contain ε

3) If $A \rightarrow \alpha B$

↳ Here, B should be the last symbol.

then, FOLLOW (B) → FOLLOW (A)

4) If $A \rightarrow \alpha B \beta$ where $\beta \rightarrow \varepsilon$ then, B becomes the last symbol.

then, FOLLOW (B) → FOLLOW (A)

FOLLOW :- [It is found for symbols that are on the RHS of Production rule]

1) If 'S' is a starting symbol
then, FOLLOW (S) = { $ }

FOLLOW never contains ε

2) If $A \rightarrow \alpha B \beta$
then, FOLLOW (B) = FIRST (β) ; if FIRST (β) doesn't contain ε

3) If $A \rightarrow \alpha B$ → Here, B should be the last symbols
then, FOLLOW (B) → FOLLOW (A)

4) If $A \rightarrow \alpha B \beta$ where $\beta \rightarrow \varepsilon$ then, B becomes the last symbol.
then, FOLLOW (B) → FOLLOW (A)

11/10/2023
Consider the following grammar. Remove left recursion and left factoring and construct LL(1) parsing table.

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

After removing left recursion :-
given; $E \rightarrow E+T \mid T$

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

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

$F \rightarrow (E) \mid id$

1) FIRST & FOLLOW functions

FIRST (E) → FIRST (TE')
FIRST (T) → FIRST (FT')
FIRST (F) → FIRST ((E)) ∪ FIRST (id) = { (, id }
a

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

Now,

FIRST (E') = FIRST (+TE') ∪ FIRST (ε) (∵ E' → +TE' | ε)

= { +, ε }

Now,

FIRST (T) = ~~FIRST~~ { (, id }

FIRST (T') → FIRST (*FT') ∪ FIRST (ε)

= { *, ε }

FIRST (F) = { (, id }

FOLLOW (E) → { $,) } (∵ E is the starting symbol)

FOLLOW (E) = FIRST ( ))

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

FOLLOW (T) = FIRST (E') = { +, ε } ✗ E → TE'

But, FOLLOW should not contain ε.

FOLLOW (T) = ~~FOLLOW~~ FOLLOW(E) = { +, $,) }

FOLLOW (T') = FOLLOW (T)

= { +, $,) }

FOLLOW (F) = FIRST (T') = { *, ε } ✗

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

2) Constructing the parsing 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			$F \to (E)$		$F \to id$	

To construct the parsing table, following rules are used.

1) $M[A, a] = A \to \alpha$, if a is in FIRST (α)

2) $M[A, b] = A \to \alpha$, if ε is in FIRST (α)
b is in FOLLOW (A)

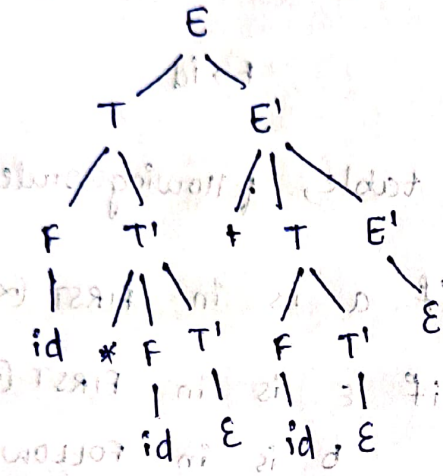
3) Parsing the input string.

ω = id * id + id

stack	input	output
$E	id * id + id $	
$ E' T	id * id + id $	$E \to TE'$
$ E' T' F	id * id + id $	$T \to FT'$
$ E' T' id	id * id + id $	$F \to id$
$ E' T'	* id + id $	
$ E' T' F *	* id + id $	$T' \to *FT'$
$ E' T' F	id + id $	
$ E' T' id	id + id $	$F \to id$
$ E' T'	+ id $	$T' \to \varepsilon$
$ E'	+ id $	
$ E' T +	+ id $	$E' \to +TE'$

check : $E \to id$ in Parsing table

$ E' T	id $	
$ E' T' F	id $	$T \rightarrow FT'$
$ E' T' id	id $	$F \rightarrow id$
$ E' T'	$	
$ E'	$	$T' \rightarrow \varepsilon$
$	$	$E' \rightarrow \varepsilon$



Q) Construct the predictive parsing table for given grammar

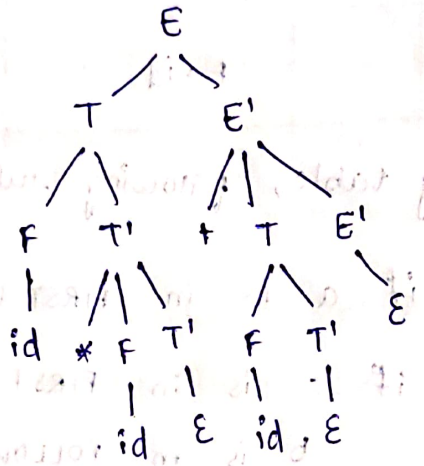
$E \rightarrow E+T \mid T$

$T \rightarrow TF \mid F$

$F \rightarrow F* \mid a \mid b$

Sol:

$ E' T	id $	
$ E' T' F	id $	T → FT'
$ E' T' id	id $	F → id
$ E' T'	$	
$ E'	$	T' → ε
$	$	E' → ε



Q) Construct the predictive parsing table for given grammar

$E \rightarrow E+T \mid T$

$T \rightarrow TF \mid F$

$F \rightarrow F* \mid a \mid b$

Sol:-

$E \rightarrow \underset{A}{E} \underset{\alpha}{+T} \mid \underset{\beta}{T}$

$E \rightarrow TE'$

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

$T \rightarrow \underset{A}{TF} \underset{\alpha}{} \mid \underset{\beta}{F}$

$T \rightarrow FT'$

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

~~$F \rightarrow F* \mid a \mid b$~~

$F \rightarrow$

$F \rightarrow \underset{A}{F*} \underset{\alpha}{} \mid \underset{\beta}{a}$

$F \rightarrow aF'$

$F' \rightarrow *F' \mid \varepsilon$

$F \rightarrow \underset{A}{F*} \underset{\alpha}{} \mid \underset{\beta}{b}$

$F \rightarrow bF'$

$F' \rightarrow *F' \mid \varepsilon$

After removing left recursion :-

$E' \to +TE' \mid \varepsilon$
$T' \to FT' \mid \varepsilon$
$F \to aF' \mid bF'$
$F' \to *F' \mid \varepsilon$
$E \to TE'$

$E \to TE'$
$E' \to +TE' \mid \varepsilon$
$T \to FT'$
$T' \to FT' \mid \varepsilon$
$F \to aF' \mid bF'$
$F' \to *F' \mid \varepsilon$

1) FIRST & FOLLOW functions

FIRST (E) = FIRST (TE')

FIRST (T) = FIRST (FT')

FIRST (F) = FIRST (aF') ∪ FIRST (bF')
($a \in$ a) ($a \in$ b)
= {a, b}

⇒ FIRST (E) = {a, b}

⇒ FIRST (T) = {a, b}

⇒ FIRST (F) = {a, b}

Now,

FIRST (E') = FIRST (+TE') ∪ FIRST (ε)

⇒ FIRST (E') = {+, ε}

FIRST (T') = FIRST (FT') ∪ FIRST (ε)

⇒ FIRST (T') = {a, b, ε} (∵ FIRST (F) = {a, b})

FIRST (F') = FIRST (*F') ∪ FIRST (ε)

⇒ FIRST (F') = {*, ε}

⇒ FOLLOW (E) = {$}

FOLLOW (T) = FIRST (E')
= {+, ε}
⇒ FOLLOW (E)

$E \to \underset{\alpha}{T}\underset{\beta}{E'}$

$E \to \underset{\alpha}{T}\underset{\beta}{\varepsilon}$

⇒ FOLLOW (T) = {+, $}

FOLLOW (F) = FIRST (T')

⇒ FOLLOW (F) = {a, b, ε} = {a, b, +, $}

FOLLOW (E') = FOLLOW (E)

⇒ FOLLOW (E') = {$}

FOLLOW (T') = FOLLOW (T)

⇒ FOLLOW (T') = {+, $}

FOLLOW (F') = FOLLOW (F)

⇒ FOLLOW (F') = {a, b, ε, +, $}

2) Constructing parsing table.

	+	*	a	b	$
E			E→TE'	E→TE'	
E'	E'→+TE'				E'→ε
T			T→FT'	T→FT'	
T'	T'→ε		T'→FT'	T'→FT'	T'→ε
F			F→aF'	F→bF'	
F'	F'→ε	F'→*F'	F'→ε	F'→ε	F'→ε

12/10/2023

Consider the grammar

S → iEtSS' | a

S' → eS | ε

E → b

Check whether it is LL1 grammar.

Step 1 :- FIRST & FOLLOW functions

FIRST (S) = {i, $, a}

FIRST (S') = {e, ε}

FIRST (E) = {b}

FOLLOW (S) = { $, e, ε }

if S → iEtSS' FIRST (S') = {e, ε}

FOLLOW (S) = { $, e }

replace S' by ε

S → iEtSS'
α β
Follow (S)

S' → eS
α β
Follow (S')

FOLLOW (S') = FOLLOW (S) = { $, e }

FOLLOW (E) = FIRST (tSS') = {t}

step 2 :-

	i	t	a	e	b	$
S	S → iEtSS'		S → a			
S'				S' → eS S' → ε		S' → ε
E					E → b	

Here, one of the cells of parsing table contains more than one production rule.

Hence, given grammar is not an LL(1) grammar

Q) Find FIRST & FOLLOW functions

S → ABCDE

A → a | ε

B → b | ε

C → c C

D → d | ε

E → e | ε

② Q) S → ACB | CbB | Ba

A → da | BC

B → g | ε

C → h | ε

③ Q) Construct the predictive parsing table for the grammar

S → (L) | a

L → L, S | S

① FIRST (S) = FIRST (ABCDE) = FIRST (A) ∪ FIRST (B) ∪ FIRST (C) ∪ FIRST (D) ∪ FIRST (E)

FIRST (A) = {a, ε}

FIRST (B) = {b, ε}

FIRST (C) = {c}

FIRST (D) = {d, ε}

FIRST (E) = {e, ε}

~~= FIRST (S)~~

~~= {a, b, c, d, e, ε}~~

FIRST (S) = FIRST (ABCDE)

= FIRST (A) ∪ FIRST (BCDE)

= {a, ε} ∪ {b, ε} ∪ {c}

= {a, b, c, ε}

FOLLOW (S) = {$}

FOLLOW (A) = FIRST (BCDE) = FIRST (B) = {b, ε}

= {b, FIRST (CDE) = FIRST (C)

= {c} ;

= {b, c}

FOLLOW (B) = FIRST (CDE)

= FIRST (C) = {c}

FOLLOW (C) = FIRST (DE)

= FIRST (D) ∪ FIRST (E)

= {d, FIRST (BCE)} ∪ {e, FIRST (D)}

= {d, FIRST (B) ∪ FIRST (CE)} ∪ {e, d, FIRST (E)}

= {d, {b, FIRST (DE)} ∪ {c}} ∪ {e, d, e, $}

= {d, {b, FIRST (D) ∪ FIRST (E)} ∪ {c}} ∪ {e, d, $}

= {d, b, d,

FOLLOW (S) = {$}

FOLLOW (A) = FIRST (BCDE) = FIRST (B) ∪ FIRST (C)
= {b, c}

FOLLOW (B) = FIRST (CDE)
= {c}

FOLLOW (C) = FIRST (DE)
= {d, e, $}

FOLLOW (D) = FIRST (E)
= {e, d, $}

FOLLOW (E) = d, {$}

(2) FIRST (S) = FIRST (ACB) ∪ FIRST (CbB) ∪ FIRST (Ba)
= {d, g, h, ε} ∪ {h, ε} ∪ {g, ε}
= {d, g, h, ε}

FIRST (A) = FIRST (da) ∪ FIRST (BC)
= {d} ∪ {g, h, ε}
= {d, g, h, ε}

FIRST (B) = {g, ε}

FIRST (C) = {h, ε}

FOLLOW (S) = {$}

FOLLOW (A) = FIRST (CB) = {h, ε} ∪ FIRST (B) ∪ FOLLOW (S)
= {h, g, $}

FOLLOW (B) = FOLLOW (S) = {$}
FIRST (a) = {h, g, $, a}

~~FOLLOW (C) = FIRST (B) = {g, ε}~~

FOLLOW (C) = FIRST (B) ∪ FIRST (bB) ∪ FOLLOW (A)
= {g, b, h, $}

(3) $S \rightarrow (L) \mid a$

$L \rightarrow L, S \mid S$

Removing left recursion :

$S \rightarrow (L) \mid a$

$L \rightarrow SL'$

$L' \rightarrow ,SL' \mid \varepsilon$

FIRST & FOLLOW functions

FIRST (S) = FIRST((L)) ∪ FIRST (a) = { (, a }

FIRST (L) = FIRST (SL') = { (, a }

FIRST (L') = FIRST (,SL') ∪ FIRST(ε) = {, , ε }

FOLLOW (S) = { $, , ,) }

FOLLOW (L) = FIRST ( )) = {) }

FOLLOW (L') = FOLLOW(L) = {) }

Passing table.

	(	)	,	a	$
S	$S \rightarrow (L)$			$S \rightarrow a$	
L	$L \rightarrow SL'$			$L \rightarrow SL'$	
L'		$L' \rightarrow \varepsilon$	$L' \rightarrow ,SL'$		

LR(0) Parser

$S \to AA$ — 1

$A \to aA$ — 2

$A \to b$ — 3

Include new produce rule

⇒ Augmented grammar is the result

step 1 :- Include new production rule

Augmented grammar :- $S' \to S$, $S \to AA$, $A \to aA$, $A \to b$

step 2 :- Closure operation Introduce . at the RHS at left most.

I_0: $S' \to .S$, $S \to .AA$, $A \to .aA$, $A \to .b$ } closure of the production rules

step 3 :- Apply goto operation (move . towards right)

(no I/p symbol is scanned) $S \to .AA$

(1 I/p symbol is scanned) $S \to A.A$

(all I/p symbols are scanned) $S \to AA.$

$S' \to .S \xrightarrow{S}$ I_1: $S' \to S.$

$S \to .AA \xrightarrow{A}$ I_2: $S \to A.A$, $A \to .aA$, $A \to .b$

(Check symbols that are after . & include its produce rules)

$A \to .aA \xrightarrow{a}$ I_3: $A \to a.A$, $A \to .aA$, $A \to .b$

$A \to .b \xrightarrow{b}$ I_4: $A \to b.$

$S' \to S.$ ——— (no I/P symbol)

$S \to A.A$ ——A—— $S \to AA.$ I5

$A \to .aA$ ——a—— $A \to a.A$ already exists.

$A \to .b$ ——b—— $A \to b.$ already exists

$A \to a.A$ ——A—— $A \to aA.$ I6

$A \to a.A$ → $A \to a.A$

$A \to .aA$ →

$A \to .b$ → $A \to b.$

$A \to b.$ ——— (no I/P symbol)

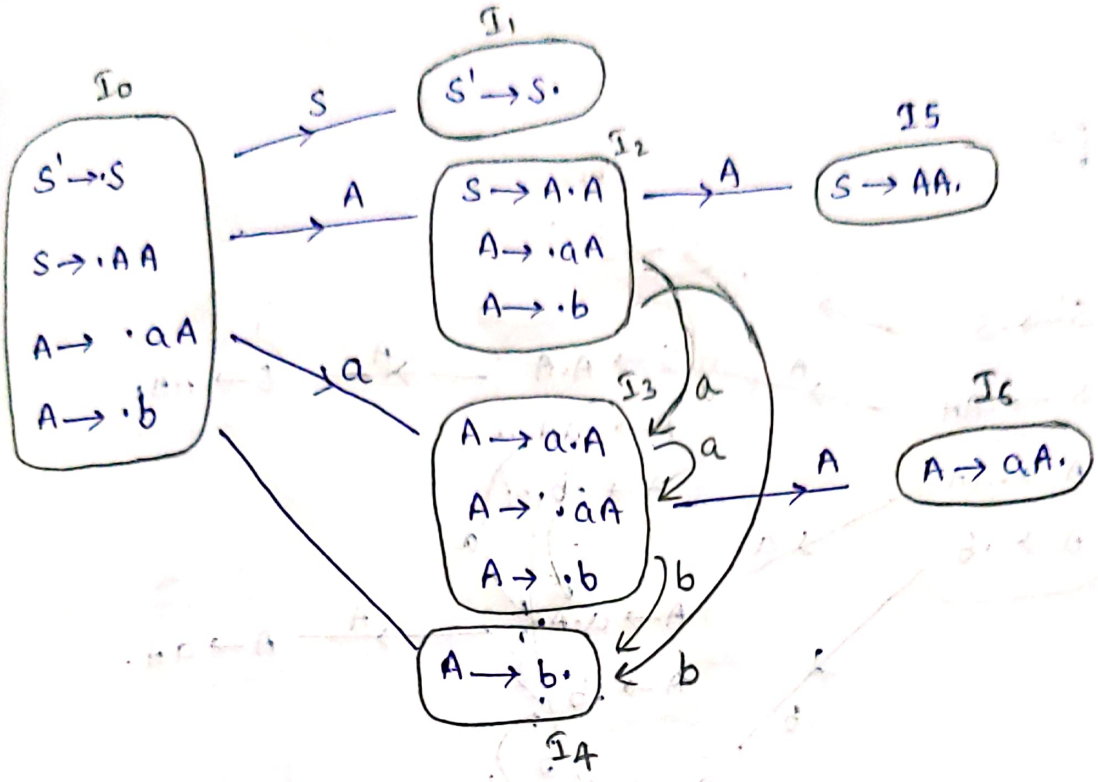
step 4 : construct parsing table.

FOLLOW (S) = {$}

FOLLOW (A) = FIRST (A) ∪ FOLLOW(S) = {a, b, $}

$A \to aA \mid b$

	Action			goto	
	a	b	$	A	S
0	S3	S4		2	1
1			accept		
2	S3	S4		5	
3	S3	S4		6	
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		



SLR (1) parser

$S \rightarrow AA$

$A \rightarrow aA$

$A \rightarrow b$

step 1

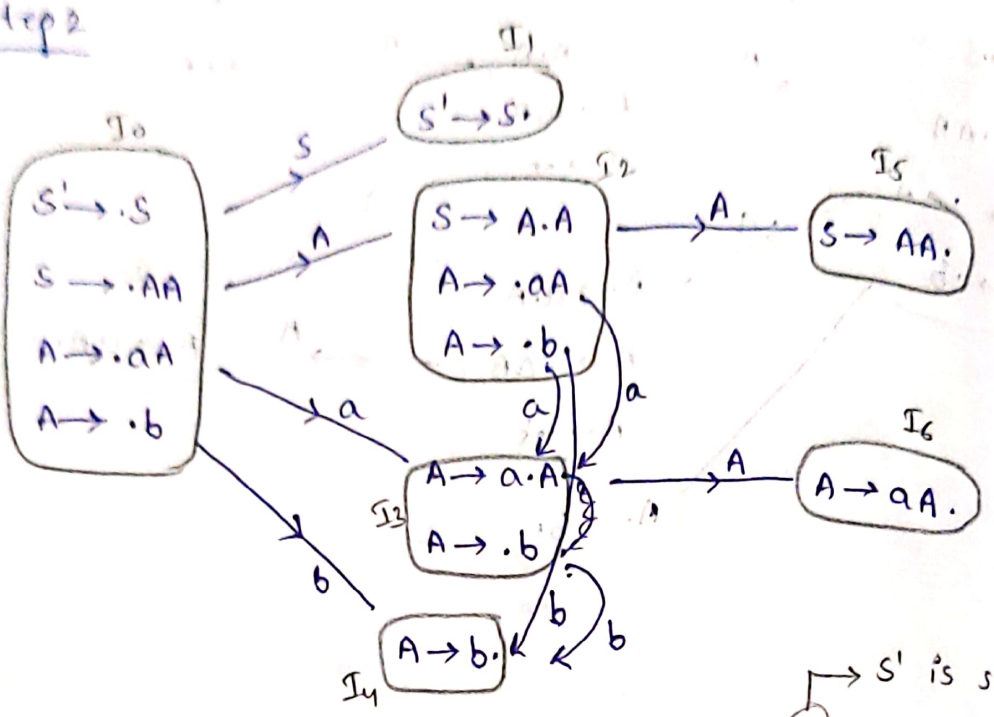
$S' \rightarrow S$

$S \rightarrow AA$

$A \rightarrow aA$

$A \rightarrow b$

Step 2



I1 : (S') → S. → S' is starting symbol, so write accept

Step 3

	Action			goto	
	a	b	$	A	S
0	S3	S4		2	1
1			accept		
2	S3	S4		5	
3	S3	S4		6	
4	r3	r3	r3		
5			r1		
6	r2	r2	r2		

I6 ⇒ (A) → aA. — (2)

↳ A is not starting symbol

So, take FOLLOW(A) = {a, b, $}

In these places, write r2

18/9/2023

UNIT - 3

Syntax Directed Definition (SDD)

SDD = CFG + Semantic Rules

(associated with)

$E \rightarrow E + T \Rightarrow E.val = E.val + T.val$ (val: attribute)

$E \rightarrow T \Rightarrow E.val = T.val$ (semantic rule)

An SDD is a Context Free Grammar together with semantic rule.

Attributes are associated with grammar symbols and semantic rules are associated with production rules.

Attributes may be numbers, strings, references, datatypes etc.

Ex:

Production rule	semantic rules
$E \rightarrow E + T$	$E.val = E.val + T.val$

Types of Attributes (2)

1) Synthesized attribute : If a parent node takes value from its children, then it is called Synthesized Attribute.

Ex: $A \rightarrow BCD \Rightarrow A.S = B.S$, $A.S = C.S$, $A.S = D.S$

(A: parent node; B C D: child nodes)

2) Inherited Attribute : If a node takes a value from its parent or sibling nodes then, it is called Inherited Attribute

Ex: $A \rightarrow BCD \Rightarrow C.i = A.i$, $C.i = B.i$, $C.i = D.i$

(A: parent; B C D: siblings; C: child)

** Types of SDDs (2)

1) S-attributed SDD

2) L-attributed SDD

1) S- Attributed SDD

» An SDD that uses only synthesized attributes is called as an S-attributed SDD.

Ex: A → BCD

A.S = B.S

A.S = C.S

A.S = D.S

» Semantic actions are always placed at the right end of the productions.

Ex: A → BCD { .... }

» Attributes are evaluated with Bottom-up Parsing

2) L- Attributed SDD

» It uses both synthesized and inherited attributes. Inherit attributes can inherit values from either parent or left sibling only

Ex: A → BCD

C.i = A.i

C.i = B.i

D.i = C.i

Ci = D.i (X)

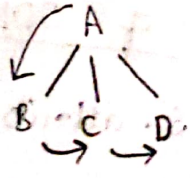
» Semantic actions are placed anywhere on RHS.

Ex: A → { .... } BCD

A → BCD { .... }

A → B { .... } CD

» Attributes are evaluated by traversing parse string depth-first, left to right



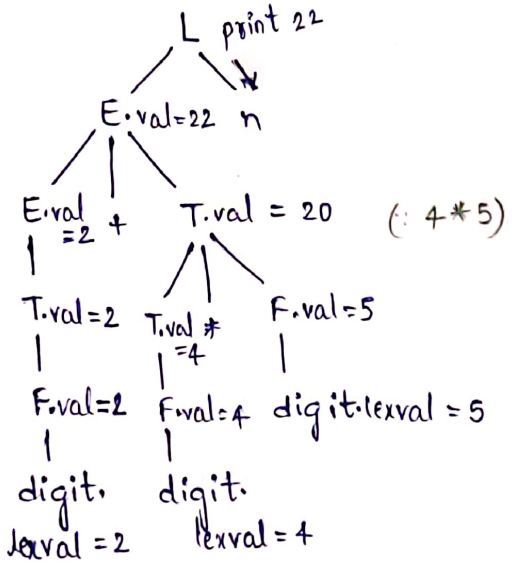
Examples

SDT
Syntax
Directed
Translation

(i) S- attributed SDD

Production rules for Desktop calc.	Semantic rules
$L \rightarrow En$ (↵ new line)	print (E.val)
$E \rightarrow E_1 + T$	E.val = E_1.val + T.val
$E \rightarrow T$	E.val = T.val
$T \rightarrow T_1 * F$	T.val := T_1.val * F.val
$F \rightarrow (E)$	F.val = (E.val)
$F \rightarrow$ digit	F.val = digit.lexval

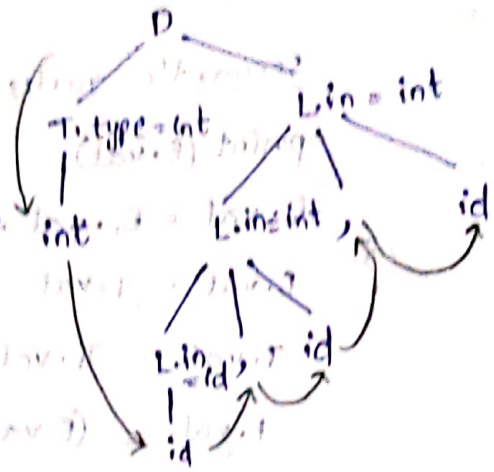
Evaluate the expression 2+4*5n



(ii) L- attributed SDD

Production	Semantic rules
$D \rightarrow TL$	L.in = T.Type
$T \rightarrow$ int	T.Type = integer
$T \rightarrow$ real	T.Type = real
$L \rightarrow L_1$, id	L_1.in = L.in add type (id.entry, L.in)
$L \rightarrow$ id	add type (id.entry, L.in)

Evaluate int id1, id2, id3



20/10/2023

Q) LL(1) Parser

$\omega = id + id * id$

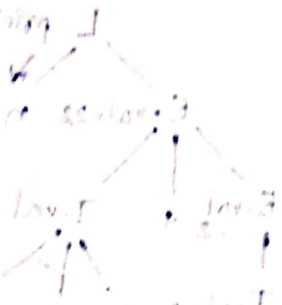
(1) $E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$



Sol:

step 1 : FIRST & FOLLOW functions

FIRST (E) = FIRST (TE')

FIRST (T) = FIRST (FT')

FIRST (F) = FIRST ((E)) ∪ FIRST (id)

= {(, id}

Now,

FIRST (E') = FIRST (+TE') ∪ FIRST (ε)

= {+, ε}

FIRST (T) = FIRST (FT')

= {(, id}

FIRST (T') = FIRST (*FT') ∪ FIRST (ε)

= {*, ε}

~~FIRST (F)~~ = FIR

FOLLOW (E) = {$,)}

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

FOLLOW (T) = FIRST (E')

= {+, ε} (x)

= {+, $,)}

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

FOLLOW (F) = FIRST (T') = {*, ε} (x)

= {*, $, +,)}

F → (E)
α B β

E → TE'
α B β
α B β

E → TE'

E → TE'
α B β

T → FT'
α B
α B β

T → FE
α B β

step 2 :- parsing table.

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

step 3 :- Parsing the input string.

w = id + id * id

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

$ E' T	id * id $	
$ E' T' F	id * id $	T → FT'
$ E' T' id	id * id $	~~F → id~~
$ E' T'	* id $	F → id
$ E' T' F *	* id $	T' → *FT'
$ E' T' F	id $	
$ E' T' id	id $	F → id
$ E' T'	$	
$ E'	$	T' → ε
$	$	E' → ε

Q) Construct SLR Parser

$S \rightarrow AA$

$A \rightarrow aA$

$A \rightarrow b$

03/11/2023

Q) SLR Parser.

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

Sol:

Step (1) Augumented Grammar :-

$E' \rightarrow E$

$E \rightarrow E+T$ —①

$E \rightarrow T$ —②

$T \rightarrow T*F$ —③

$T \rightarrow F$ —④

$F \rightarrow (E)$ —⑤

$F \rightarrow id$ —⑥

Step (2) closure :-

$E' \rightarrow \cdot E$

$E \rightarrow \cdot E+T$

$E \rightarrow \cdot T$

$T \rightarrow \cdot T*F$

$T \rightarrow \cdot F$

$F \rightarrow \cdot (E)$

$F \rightarrow \cdot id$

goto (3) goto ✓

I_0 ✓

$E' \to .E$

$E \to .E+T$

$E \to .T$

$T \to .T*F$

$T \to .F$

$F \to .(E)$

$F \to .id$

I_1: goto (I_0, E) ✓

$E' \to E.$

$E \to E.+T$

I_2: goto (I_0, T) ✓

$E \to T.$

$T \to T.*F$

I_3: goto (I_0, F) ✓

$T \to F.$

I_4: goto $(I_0, ()$ ✓

$F \to (.E)$

$E \to .E+T$

$E \to .T$ (I_2)

$T \to .T*F$ (I_2)

$T \to .F$ (I_3)

$F \to .(E)$ (I_4)

$F \to .id$ (I_5)

I_5: goto (I_0, id) ✓

$F \to id.$

I_6: goto $(I_1, +)$ ✓

$E \to E+.T$

$T \to .T*F$

$T \to .F$ I_3 ✓

$F \to .(E)$ I_4 ✓

$F \to .id$ I_5 ✓

I_7: goto $(I_2, *)$ ✓

$T \to T*.F$

$F \to .(E)$ I_4 ✓

$F \to .id$ I_5 ✓

I_8: goto (I_4, E) ✓

$F \to (E.)$

$E \to E.+T$ (I_6) ✓

I_9: goto (I_6, T)

$E \to E+T.$

$T \to T.*F$ (I_7)

I_{10}: goto (I_7, F)

$T \to T*F.$

I_{11}: goto $(I_8,)$

$F \to (E).$

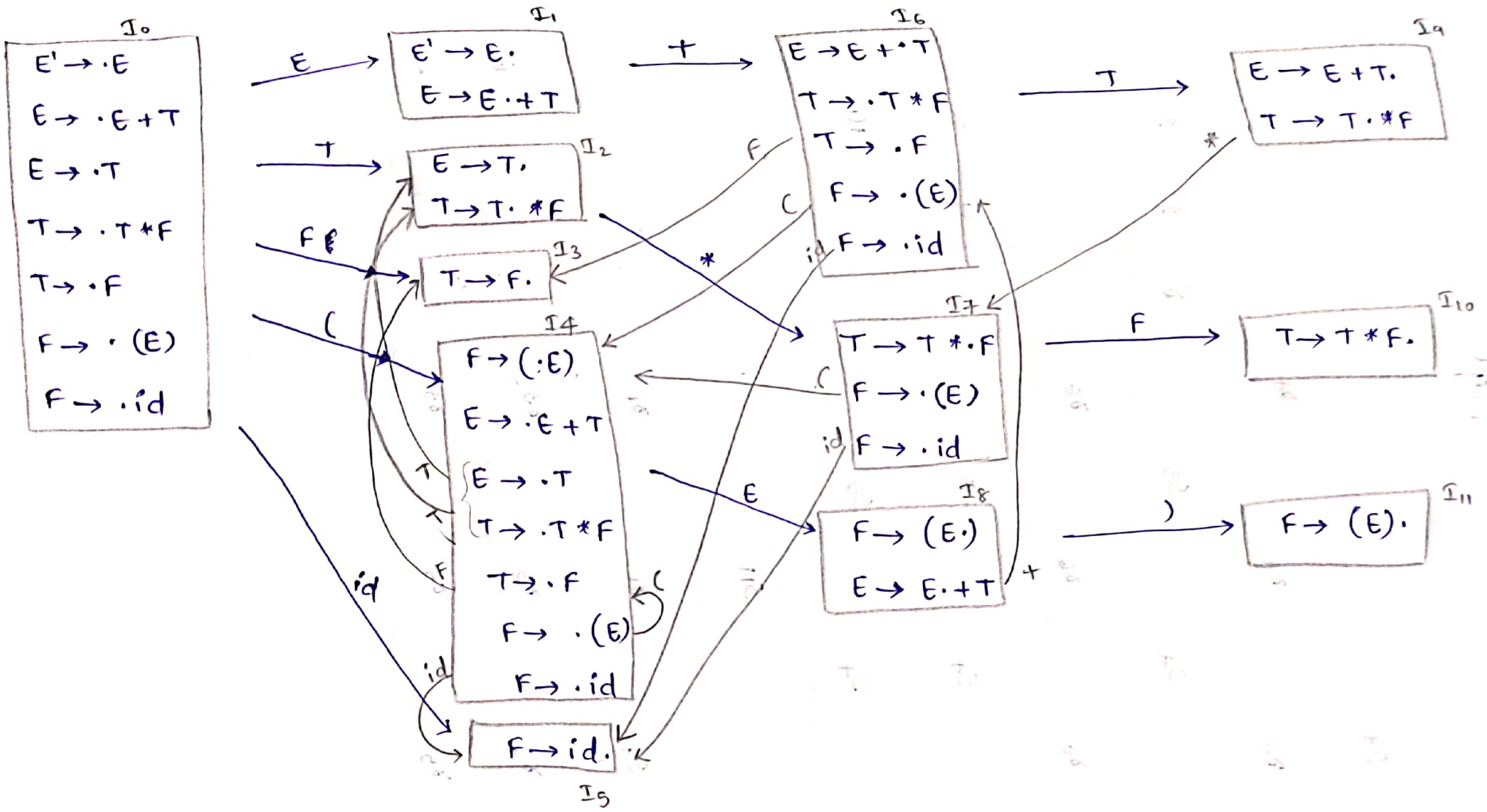
step (4) ✓ FIRST & FOLLOW

step (4) ✓ FOLLOW functions.

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

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

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


I0
E' → ·E
E → ·E+T
E → ·T
T → ·T*F
T → ·F
F → ·(E)
F → ·id
I1
E' → E·
E → E·+T
I2
E → T·
T → T·*F
I3
T → F·
I4
F → (·E)
E → ·E+T
E → ·T
T → ·T*F
T → ·F
F → ·(E)
F → ·id
I5
F → id·
I6
E → E+·T
T → ·T*F
T → ·F
F → ·(E)
F → ·id
I7
T → T*·F
F → ·(E)
F → ·id
I8
F → (E·)
E → E·+T
I9
E → E+T·
T → T·*F
I10
T → T*F·
I11
F → (E)·
E
T
F
(
id
+
*
)

step (5) :- parsing table.

	+	*	(	)	id	$	E	T	F
0			S_4		S_5		1	2	3
1	S_6					accept			
2		S_7							
3	γ_4	γ_4		γ_4		γ_4			
4			S_4		S_5		8	2	3
5	γ_6	γ_6		γ_6		γ_6			
6			S_4		S_5			9	3
7			S_4		S_5				10
8	S_6			S_{11}					
9	γ_1	S_7		γ_1		γ_1			
10	γ_3	γ_3		γ_3		γ_3			
11	γ_5	γ_5		γ_5		γ_5			

07/11/2023

Input string : id * id + id

Stack	input	Output / Action
0	id * id + id $	shift
0 id 5	* id + id $	reduce by F → id
0 F 3	* id + id $	reduce by T → F
0 T 2	* id + id $	shift
0 T 2 * 7	id + id $	shift
0 T 2 * 7 id 5	+ id $	reduce by F → id
0 T 2 * 7 F 10	+ id $	reduce by T → T * F
0 T 2	+ id $	reduce by E → T
0 E 1	+ id $	shift
0 E 1 + 6	id $	shift
0 E 1 + 6 id 5	$	reduce by F → id
0 E 1 + 6 F 3	$	reduce by T → F
0 E 1 + 6 T 9	$	reduce by E → E + T
0 E 1	$	Accept

08/11/2023

CLR Parser

SLR → no look ahead symbols.
» LR(0) items
CLR → we write look ahead symbols.
» LR(1) items.
⇓
LR(0) + look ahead symbols.

$S \to CC$ —①

$C \to cC$ —②

$C \to d$ —③

LR(1) = LR(0) items + look ahead symbols.

step1 : Augumented grammar

$S' \to S$

$S \to CC$

$C \to .cC$

$C \to d$

, is separator not a terminal

step2 : write look ahead symbols.

I_0 : $S' \to .S, \$$ → S is starting symbol → $S' \to .S\varepsilon, \$$ FIRST(ε, $) after S there's no symbol

$S \to .CC, \$$ (FIRST($))

$C \to .cC, c/d$ FIRST(C) (FIRST(C) = c/d) after C there's a symbol C

$C \to .d, c/d$ FIRST(C)

I_1 : goto (I_0, S)

$S' \to S., \$$

I_2 : goto (I_0, C)

$S \to C.C, \$$

$C \to .cC, \$$ FIRST(C $)

$C \to .d, \$$

I_3 : goto (I_0, c)

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

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

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

I_4 : goto (I_0, d)

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

I_5 : goto (I_2, C)

$S \to CC\cdot, \$$

I_6 : goto (I_2, c)

$C \to c\cdot C, \$$

$C \to \cdot cC, \$$ (I_6)

$C \to \cdot d, \$$ (I_7)

I_7 : goto (I_2, d)

$C \to d\cdot, \$$

I_8 : goto (I_3, C)

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

I_9 : goto (I_6, C)

$C \to cC\cdot, \$$

I_9 : goto (I_3, c)

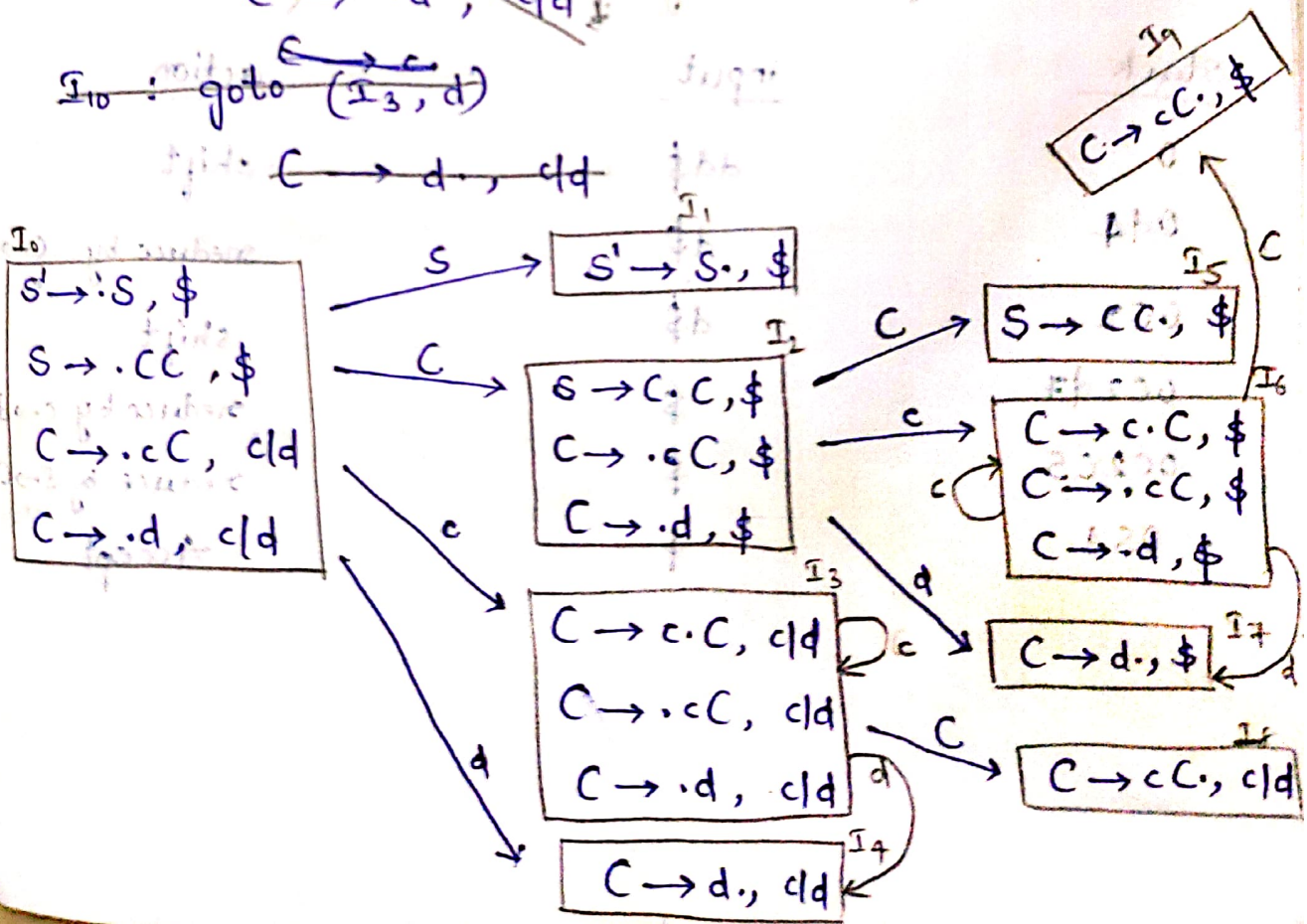
$C \to c\cdot C, c/d$

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

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

~~I_{10} : goto (I_3, d)~~

~~$C \to d\cdot, c/d$~~



	Action			goto	
	c	d	$	S	C
0	S_3	S_4		1	2
1			Accept		
2	S_6	S_7			5
3	S_3	S_4			8
4	γ_3	γ_3			
5			γ_1		
6	S_6	S_7			9
7			γ_3		
8	γ_2	γ_2			
9			γ_2		

Parse the input string $\omega = dd\$$. (same as SLR)

stack	input	action
0	dd$	shift
0d4	d$	reduce by C→d $\gamma_2 = 2$
0C2	d$	shift
0C2d7	$	reduce by C→d
0C2C5	$	reduce by S→CC $\gamma_2 = 4$
0S1	$	Accept

10/11/2023

LALR Parser

$S \to CC$

$C \to cC$

$C \to d$

I_{36} : $C \to c \cdot C$, $c|d|\$$

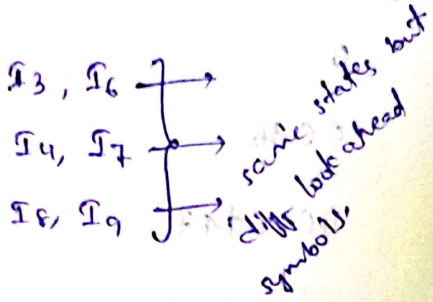
$C \to \cdot cC$, $c|d|\$$

$C \to \cdot d$, $c|d|\$$

I_{47} : $C \to d \cdot$, $c|d|\$$

I_{89} : $C \to cC \cdot$, $c|d|\$$

Same states with diff look ahead symbols are combined.



Parsing table

Instead of writing 3 & 6 separately ; write 36 as single state

In shift ↳ S_{36}

But not in reduce.

Combine the states

	c	d	$	S	C
0	S_{36}	S_{47}		1	2
1			Accept		
2	S_{36}	S_{47}			5
36	S_{36}	S_{47}			8,9
47	r_3	r_3	r_3		
5	~~S_{36}~~	~~S_{47}~~	r_1		~~8~~
89	r_2	r_2	r_2		

Minimum length string : cd$

dd$

Parse the input string cd$.

Stack	input	action
0	cd$	shift
0C36	d$	shift
0C36d47	$	reduce by C→d
0C36C89	$	reduce by C→cC
0C2	$	

Construct LALR parser and parse the input string w=id=id

S → L=R

S → R

L → *R

L → id

R → L

Augumented Grammar :-

S' → S

S → L=R — 1

S → R — 2

L → *R — 3

L → id — 4

R → L — 5

I_0 : $S' \to \cdot S, \$$

$S \to \cdot L = R, \$$

$S \to \cdot R, \$$

$L \to \cdot *R, = | \$$

$L \to \cdot id, = | \$$

$R \to \cdot L, \$$

I_1 : goto (I_0, S)

$S' \to S \cdot, \$$

I_2 : goto (I_0, L)

$S \to L \cdot = R, \$$

$R \to L \cdot, \$$

I_3 : goto (I_0, R)

$S \to R \cdot, \$$

I_4 : goto $(I_0, *)$

$L \to * \cdot R, = | \$$

$R \to \cdot L, = | \$$

$L \to \cdot *R, = | \$ \ (I_4)$

$L \to \cdot id, = | \$ \ (I_5)$

I_5 : goto (I_0, id)

$L \to id \cdot, = | \$$

I_6 : goto $(I_2, =)$

$S \to L = \cdot R, \$$

$R \to \cdot L, \$$

$L \to \cdot *R, \$$

$L \to \cdot id, \$$

I_7 : goto (I_4, R)

$L \to *R \cdot, = | \$$

I_8 : goto (I_4, L)

$R \to L \cdot, = | \$$

I_9 : goto (I_6, R)

$S \to L = R \cdot, \$$

I_{10} : goto (I_6, L)

$R \to L \cdot, \$$

I_{11} : goto $(I_6, *)$

$L \to * \cdot R, \$$

$R \to \cdot L, \$ \ (I_{10})$

$L \to \cdot *R, \$ \ (I_{11})$

$L \to \cdot id, \$ \ (I_{12})$

I_{12} : goto (I_6, id)

$L \to id \cdot, \$$

I_{13} : goto (I_{11}, R)

$L \to *R \cdot, \$$

I_{411} : $L \to *\cdot R$,

$R \to \cdot L$,

$L \to \cdot *R$,

$L \to \cdot id$,

I_{512} : $L \to id\cdot$

I_{713} : $L \to *R\cdot$

I_{810} : $R \to L\cdot$

Parsing table

	id	=	*	$	S	L	R
0	S_{12}		S_{11}		1	2	3
1				Accept			
2		S_6		γ_5			
3				γ_2			
4 11	S_{12}		S_{11}			8 10	7 13
5 12		γ_4		γ_4			
6	S_{12}		S_{11}			8 10	9
7 13		γ_3		γ_3			
8 10		γ_5		γ_5			
9				γ_1			

10 γ_5

11 S_{12} S_{11} 10 13

12 γ_4

13 γ_3

Input string : id = id

stack	Input	Action
0	id = id $	shift,
0 id 5 12	= id $	reduce by L → id
0 L 2	= id $	shift
0 L 2 = 6	id $	shift
0 L 2 = 6 id 5 12	$	reduce by L → id
0 L 2 = 6 L 8 10	$	reduce by R → L
0 L 2 = 6 R 9	$	reduce by S → L = R
0 S 1	$	Accept

5/11/2023

Shift - Reduce Parsers

» Shift - Reduce parsing is a process of reducing a string to the start of a grammar.

» It uses a stack to hold the grammar and an input to hold the string.

A string $\xrightarrow{\text{reduced to}}$ the starting symbol

» Shift - Reduce parsing performs two actions.

1) Shift
2) Reduce.

» At shift action, the current symbol string is pushed to stack.

» At each reduction, the symbol will be replaced by the non-terminals.

» The symbol is the right side of the production and non-terminal is the left side of the production.

Ex:-

$S \rightarrow S + S$

$S \rightarrow S - S$

$S \rightarrow (S)$

$S \rightarrow a$

Input string : $a_1 - (a_2 + a_3)$

Stack content	Input string	Action
$	$a_1 - (a_2 + a_3)$ $	shift a_1
$ a_1	$- (a_2 + a_3)$ $	reduce by S→a
$ S	$- (a_2 + a_3)$ $	shift -
$ S -	$(a_2 + a_3)$ $	shift (
$ S - (	$a_2 + a_3)$ $	shift a_2
$ S - (a_2	$+ a_3)$ $	reduce by S→a
$ S - (S	$+ a_3)$ $	shift +
$ S - (S +	$a_3)$ $	shift a_3
$ S - (S + a_3	$)$ $	reduce by S→a
$ S - (S + S	$)$ $	shift)
$ S - (S + S)	$ $	reduce by S→S+S
$ S - (S)	$ $	reduce by S→(S)
$ S - S	$ $	reduce by S→S-S
$ S	$ $	Accepted

9) Consider the grammar

E → 2E2
E → 3E3
E → 4

I/P : 32423

Stack content	Input string	Action
$	32423 $	shift 3
$3	2423 $	shift 2
$32	423 $	shift 4
$324	23 $	reduce by E→4
$32E	23 $	shift 2
$32E2	3 $	reduce by E→2E2

$3E	3$	shift 3
$3E3	$	reduce by E→3E3
$E	$	Accepted

15/11/2023

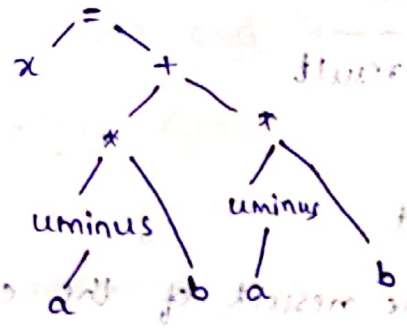
UNIT - IV

Formats of Intermediate code

1) Abstract syntax tree
2) Polish Notation
3) 3-address code

1) Abstract Syntax Tree

$x = (-a * b) + (-a * b)$



2) Polish Notation

or postfix notation

It is also called as Prefix notation, in which operators occur first and then operands are arranged.

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

$* + ab - cd$

3) Three Address Code

In three-address code form, at most 3 addresses are used to represent any statement.

General form of the 3-address code is :-

$a = b \ op \ c$

where; a, b, c are operands

op is operator

» Only single operation at RHS of the expression is allowed at a time.

Write 3-address code for the statement :- $a = b + c + d$

$a = b + c + d$

$t_1 = b + c$

$t_2 = t_1 + d$

$a = t_2$

Implementation of 3-address code

1) Quadraples
2) Triple representation
3) Indirect triple representation.

17/11/2023

① Quadraple Representation

op, arg1, arg2, result

op : ~~operand~~ operator

arg1, arg2 : two argument

result : used to store the result of the expression.

Q) $x = -a * b * -a * b$

$t_1 =$ uminus a

$t_2 = t_1 * b$

$t_3 =$ uminus a

$t_4 = t_3 * b$

$t_5 = t_2 * t_4$

$x = t_5$

Disadv : temp multiple variables are used. Hence, mem req is more. (Mem is wasted)

Quadraple representation :-

	op	arg1	arg2	result
(0)	uminus	a		t1
(1)	*	t1	b	t2
(2)	uminus	a		t3
(3)	*	t3	b	t4
(4)	*	t2	t4	t5
(5)	=	t5		x

② Triple Representation

op, arg1, arg2

address is used instead of temp variables.

	op	arg1	arg2
(0)	uminus	a	
(1)	*	(0)	b
(2)	uminus	a	
(3)	*	(2)	b
(4)	*	(1)	(3)
(5)	=	(4)	

Disadv:- becomes difficult when an error is occured. When value in a mem. loc is changed, it gets overwritten and the actual value may be lost.

③ Indirect Triple Representation

Instead of using direct mem locations, we use indirect addresses that has the same value in it.

Thus, even if the values are changed, result is not affected.

	op	arg1	arg2
(10)	uminus	a	
(11)	*	(10)	b
(12)	uminus	a	
(13)	*	(12)	b
(14)	*	(11)	(13)
(15)	=	(14)	

(0)	(10)
(1)	(11)
(2)	(12)
(3)	(13)
(4)	(14)
(5)	(15)

Q) $x = a * -(b+c)$

Quadraple :-

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

$t_1 = b+c$

$t_2 = \text{uminus } t_1$

$t_3 = a * t_2$

~~t_4 = x~~ $x = t_3$

	op	arg1	arg2	result
(0)	+	b	c	t1
(1)	uminus	t1		t2
(2)	*	a	t2	t3
(3)	=	t3		x

Triple :-

	op	arg1	arg2
(0)	+	b	c
(1)	uminus	(0)	
(2)	*	a	(1)
(3)	=	(2)	

Indirect Triple :-

	op	arg1	arg2
(10)	+	b	c
(11)	uminus	(10)	
(12)	*	a	(11)
(13)	=	(12)	

(0)	(10)
(1)	(11)
(2)	(12)
(3)	(13)

Q) $x = -(a*b) + (c+d) - (a+b+c+d)$

Quadraple :-

$t_1 = a * b$

$t_2 = \text{uminus } t_1$

$t_3 = c + d$

$t_4 = t_2 + t_3$

$t_5 = a + b$

~~$t_6 = c + d$~~

$t_7 = t_5 + t_6$

~~$t_8 = \text{uminus } t_7$~~

~~$t_9 = t_4 + t_8$~~

~~$t_8 = t_4 - t_7$~~

(0) $t_1 = a * b$

(1) $t_2 = \text{uminus } t_1$

(2) $t_3 = c + d$

(3) $t_4 = t_2 + t_3$

(4) $t_5 = a + b$

~~$t_6 = c + d$~~

(5) $t_6 = t_3 + t_5$

(6) $t_7 = t_4 - t_6$

(7) $x = t_7$

	op	arg1	arg2	result
(0)	*	a	b	t1
(1)	uminus	t1		t2
(2)	+	c	d	t3
(3)	+	t2	t3	t4
(4)	+	a	b	t5
(5)	+	t3	t5	t6
(6)	-	t4	t6	t7
(7)	=	t7		x

Triple :-

	op	arg1	arg2
(0)	*	a	b
(1)	uminus	(0)	
(2)	+	c	d
(3)	+	(1)	(2)
(4)	+	a	b
(5)	+	(2)	(4)
(6)	-	(3)	(5)
(7)	=	(6)	

Indirect Triple :-

	op	arg1	arg2
(10)	*	a	b
(11)	uminus	(10)	
(12)	+	c	d
(13)	+	(11)	(12)
(14)	+	a	b
(15)	+	(12)	(14)
(16)	-	(13)	(15)
(17)	=	(16)	

(0)	(10)
(1)	(11)
(2)	(12)
(3)	(13)
(4)	(14)
(5)	(15)
(6)	(16)
(7)	(17)

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

~~Quadruple~~ :-

$t_1 = \text{uminus } c$

$t_2 = b * t_1$

$t_3 = t_2 + t_2$

$a = t_3$

Quadruple :-

	op	arg1	arg2	result
(0)	uminus	c		t1
(1)	*	b	t1	t2
(2)	+	t2	t2	t3
(3)	=	t3		a

Triple :-

	op	arg1	arg2
(0)	uminus	c	
(1)	*	b	(0)
(2)	+	(1)	(1)
(3)	=	(2)	

Indirect Triple :-

	op	arg1	arg2
(10)	uminus	c	
(11)	*	b	(10)
(12)	+	(11)	(11)
(13)	=	(12)	

(0)	(10)
(1)	(11)
(2)	(12)
(3)	(13)

Q) $x = (a+b) * (c+d) - (a+b+c)$

$t_1 = a + b$

$t_2 = c + d$

$t_3 = t_1 + t_2$

$t_4 = t_1 + c$

$t_5 = t_3 - t_4$

$x = t_5$

Quadraple :-

	op	arg1	arg2	result
(0)	+	a	b	t_1
(1)	+	c	d	t_2
(2)	+	t_1	t_2	t_3
(3)	+	t_1	c	t_4
(4)	−	t_3	t_4	t_5
(5)	=	t_5		x

Triple :-

	op	arg1	arg2
(0)	+	a	b
(1)	+	c	d
(2)	+	(0)	(1)
(3)	+	(0)	c
(4)	−	(2)	(3)
(5)	=	(4)	

Indirect Triple :-

	op	arg1	arg2
(10)	+	a	b
(11)	+	c	d
(12)	+	(10)	(11)
(13)	+	(10)	c
(14)	−	(12)	(13)
(15)	=	(14)	

(0)	(10)
(1)	(11)
(2)	(12)
(3)	(13)
(4)	(14)
(5)	(15)

Q) $(x+y) * (y+z) * (x+y+z) = r$

$t_1 = x + y$

$t_2 = y + z$

$t_3 = t_1 * t_2$

$t_4 = t_1 + z$

$t_5 = t_3 + t_4$

$r = t_5$

Quadraple :-

	op	arg1	arg2	result
(0)	+	x	y	t1
(1)	+	y	z	t2
(2)	*	t1	t2	t3
(3)	+	t1	z	t4
(4)	+	t3	t4	t5
(5)	=	t5		r

Triple :-

	op	arg1	arg2
(0)	+	x	y
(1)	+	y	z
(2)	*	(0)	(1)
(3)	+	(0)	z
(4)	+	(2)	(3)
(5)	=	(4)	

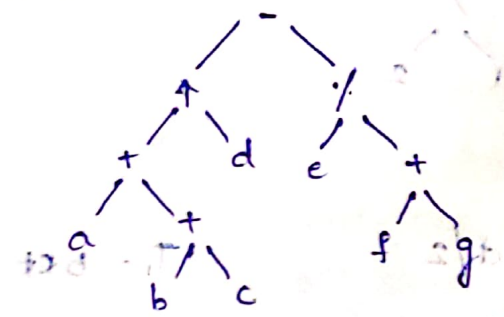
Indirect Triple :-

	op	arg1	arg2
(10)	+	x	y
(11)	+	y	z
(12)	*	(10)	(11)
(13)	+	(10)	z
(14)	+	(12)	(13)
(15)	=	(14)	

(0)	(10)
(1)	(11)
(2)	(12)
(3)	(13)
(4)	(14)
(5)	(15)

Construct the syntax tree and post fix notation for the following expression :- $(a+(b+c))\uparrow d - e / (f+g)$

Syntax tree :-



Postfix notation :- In this expression, the arrangement is

op1 , op2 , op

$a + (b+c) \uparrow d - e / (f+g)$ where $T_1 = bc+$

$a + T_1 \uparrow d - e / (f+g)$ where $T_2 = aT_1+$

$T_2 \uparrow d - e / (f+g)$ where $T_3 = T_2 d \uparrow$

$T_3 - e / (f+g)$ where $T_4 = fg+$

$T_3 - e / T_4$ where $T_5 = eT_4/$

$T_3 - T_5$ where $T_6 = T_3 T_5 -$

T_6

Now, backward substituting the values of temp variables

T_6

$T_3 T_5 -$

$T_3 eT_4 / -$

$T_3 efg+/-$

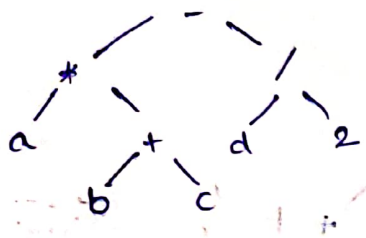
$T_2 d\uparrow efg+/-$

$aT_1 + d\uparrow efg+/-$

$abc+ + d\uparrow efg+/-$

Q) $a * (b+c) - d / 2$

syntax tree :-



Postfix notation :-

$a * (b+c) - d/2$

$a * T_1 - d/2$

$T_2 - T_3$

T_4

$T_1 = b + c$

$T_2 = a * T_1$

$T_3 = d2/$

$T_4 = T_2 T_3 -$

$\Rightarrow$ T_4

$T_2 T_3 -$

$T_2 d2/ -$

$aT_1 * d2/ -$

$abc + * d2/ -$

UNIT - 2 (last topic)

** Chomsky hierarchy of languages and recognizers (5m).

The grammar is represented with four tuples notation.

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

These grammars are divided into 4 types based on Chomsky hierarchy.

Type - 0 : unrestricted grammar

Type - 1 : Context sensitive grammar

Type - 2 : Context free grammar

Type - 3 : Regular grammar

Type -0 (Unrestricted Grammar) :-

→ The languages accepted by these grammars are called Recursively enumerable languages.

→ The languages are accepted by Turing machine.

→ Each production is in the form of $\alpha \to \beta$ where; (comb° of terminal & non-terminal)

$\alpha, \beta \in (V \cup T)^*$

→ α must contain atleast one non-terminal symbol.

→ Ex: $aA \to aBCd$ (✓) $ab \to dC$ (✗)

$D \to F$ (✓)

Type -1 (Context Sensitive Grammar) :-

→ The languages generated by this grammar are called Context Sensitive languages.

→ The automata that accepts context sensitive languages is linear bounded automata.

→ The production rule is in the form of $\alpha \to \beta$ where;

$\alpha, \beta \in (V \cup T)^*$ and $|\alpha| \leq |\beta|$

→ Ex: $aA \to aAB$ (✓)

$aA \to B$ (✗)

Type-2 (Context Free Grammar) :-

→ The language generated by this grammar is called Context free language.

→ These grammars are accepted by Push Down Automata.

→ The production rule is in the form of $\alpha \to \beta$ where;

$\alpha \in V$ only and $|\alpha| = 1$

→ Ex: $E \to E + T$

Type-3 (Regular Grammar) :-

→ This grammar generates regular languages.

→ Regular languages are accepted by finite automata.

→ The production rule is in the form of $\alpha \to \beta$ where;

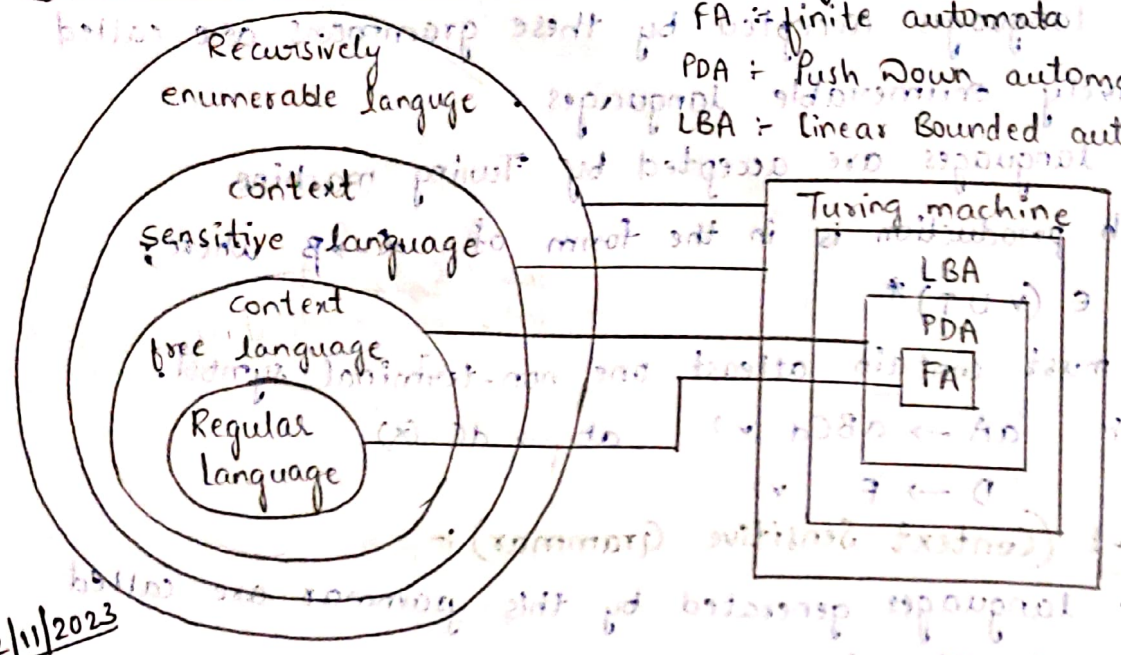
$\alpha \in V$ and $|\alpha| = 1$

→ Ex: $S \to a \mid aA$ Right linear (a followed by A)

or $S \to a \mid Aa$ Left linear (A followed by a)

Graphical Representation of Chomsky hierarchy

FA = finite automata
PDA = Push Down automata
LBA = Linear Bounded automata

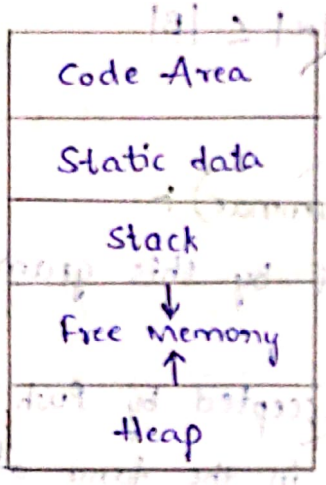


22/11/2023

Storage Organization

Runtime environment / Runtime Storage management

** Sub division of Runtime memory :



obj file is stored

to allocate memory for static progs

for dynamic progs including objects or pointers

Code Area : It stores the target executable code (object code)

Static Data : It is mainly useful to store static variables as well as global variables. Global variables are used throughout the program. Static variables means, the value of variable is persistent between different function calls.

Stack & Heap : In order to utilize the space in effective manner Heap and stack are used. They grow opposite to each other.

Stack : It works on the principle Last In First Out. Whenever a function is called activation ~~regard~~ record is created. To store the activation ~~regard~~ record information ; stack is used. The record is pushed on to the top of the stack.

Heap :- To allocate the memory dynamically.

. These two data structures are used depending on the situations.

. If there are too many functions or procedures, then stack is used more often.

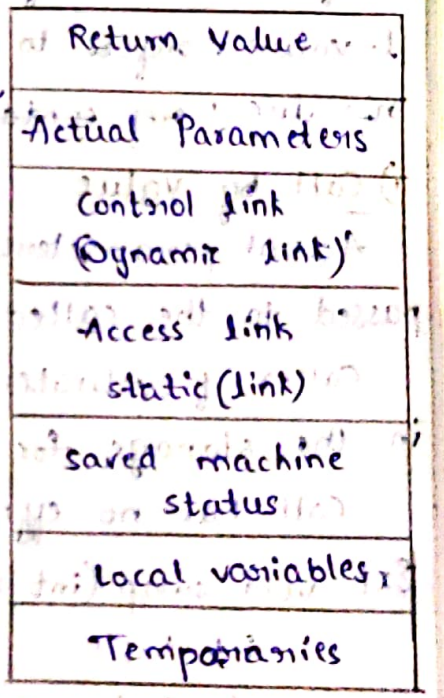
. The free memory is mostly occupied by the stack.

. If there are too many pointers or dynamic variable, then we need to allocate memory. This is done by using Heap.

Activation Record **

» The activation record is a block of memory used for managing information needed by a single execution of a func'n/proc.

» The content of activation record is shown below :-



Various fields of activation record are :-

Temporary values: The temporary variables are needed during the evaluation of expressions, such variables are stored in the temporary field of activation record.

Local variables : The local data is a data that is local to the execution of a procedure. This field stores the local data.

Saved (stored) Machine registers: This field holds the info. regarding the status of the machine just before the procedure is called. This field contains machine registers & program control.

Control link : This field is optional. It points to the activation record of the calling procedure. Also called as Dynamic link

Access link : It is optional. It refers to the non-local data in other activation record. Also called as static link.

Actual Parameters : It holds the info. about the actual parameters (ex) These params are passed to the called procedures.

Return value : This field is used to store the result of a function call.

The size of each field of activation record is determined at the time when a procedure is called.

** Storage Allocation Strategies

static
stack
heap
add, delete

24/11/2023

Parameter Passing

1. ~~Call~~ by value
2. Call by reference
3. Copy restore
4. Call by name.

Two types of parameters:

(1) Actual Parameters are the variables specified in function call.

(2) Formal Parameters are the variables declared in the definition of the called function.

l-value : refers to the storage.

r-value : ~~ref contai~~ content in the storage.

i = 10
l-value r-value

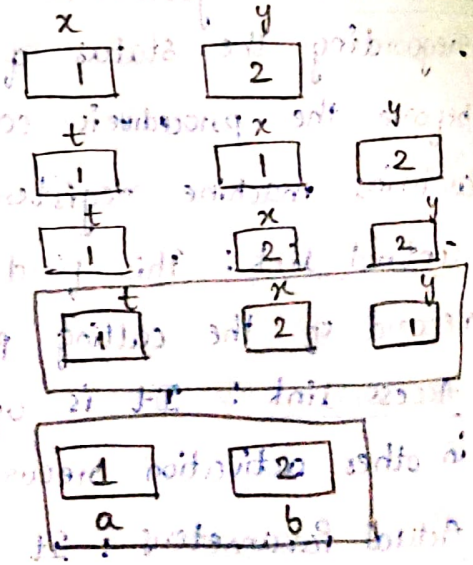
1) Call by Value

Actual parameters are evaluated and their r-values are passed to the called procedures.

Caller evaluates the actual parameters and places r-value in the storage for formals.

Call has no effect on the activation record of the caller.

Ex:
```
void swap (int x, int y){
    int t;
    t = x;
    x = y;
    y = t;
}
void main() {
    int a = 1, b = 2;
    swap (a, b);
    printf("a = %d, b = %d", a, b);
}
```

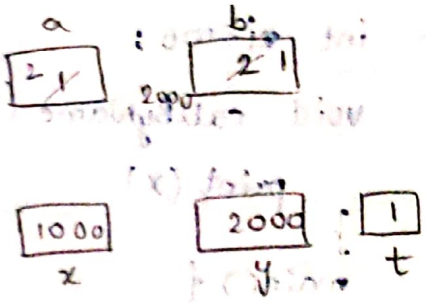


Output :- a = 1, b = 2

2) Call by reference / Call by address

The caller passes a pointer to each location of actual parameters. If actual parameter is a name then, l-value is passed. If actual parameter is an expression, then it is evaluated in a new location and the address of that location is passed.

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

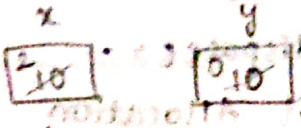


Output:- a=2, b=1

3) Copy restore

It is a hybrid method b/w call by value & call by reference. It is also called by copy in-copy out (or) value-result. Caller evaluates actuals and passes r-values to formals. l-values of actual parameters are determined before func call. When control returns, r-values of formals are copied back in to l-values of the actuals.

Ex:-
```
int y;
copyrestore (int x) {
    x=2;
    y=20;
}
main() {
    y=10;
    copyrestore (y);
    print (y);
}
```



r-value of formal x

l-value of actual y

y=2

O/P:- y=2

After control is returned to the main procedure, the r-value of x must be copied to the l-value of y.

4) Call by name

Actual parameters are substituted for the formals.

It is also called as Macro Expansion (or) Inline Expansion.

Ex:-
```
int i=100;
void callByName (int x){
    print (x)
}
main() {
    callByName (i=i*+1)
}
```

O/P :- 101

Language facilities for Dynamic Storage Allocation

The data under program control can be allocated dynamically. This allocation is done from heap memory.

There are two types of approaches used to allocate.

(1) Implicit Allocation

(2) Explicit Allocation

Explicit Allocation:

It is a kind of memory allocation that can be done with the help of some procedures. For example, In Pascal, memory allocation is done using "new" function, and dellocation is done using dispose().

Implicit Allocation

It is a kind of allocation that can be done automatically for storing the results of expression evaluation.

Ex:- LISP

SNOBOL

Garbage Collection

It is a substantial amount of memory which gets allocated dynamically but, it is unreachable memory.

If this memory remains unreachable then, it becomes waste of memory. Hence, a new technique called ~~Garge~~ co Garbage collection is introduced.

Java and lisp are the languages that (use) performs this technique.

*Dangling reference

It is a kind of complication that occurs due to explicit deallocation of memory. The dangling reference is a kind of reference that gets de-allocated which is referred to.

Ex:

```
int* dangle() {
    int i = 20;
    return &i;
}
main() {
    int *p;
    p = dangle();
}
```

i is local to dangle().

Block Structure & Non-Block Structure Storage allocation

The storage allocation can be done for two types of data variables.

(1) Local data

(2) Non-local data

Local data can be handled using activation records.

Non-local data can be handled using scope information.

Access to non-local names/data:

Scope rules of a language can be determined by references to non-local names.

There are two types of scope rules.

(1) Lexical or static scope rules.

(2) Dynamic scope rules.

Lexical or Static Scope rules & Dynamic Scope rules

» Determines the declarations applicable to a name at compile time.
  ↳ Lexical scope / static scope

• Determines the declarations applicable to a name at run time
  ↳ Dynamic scope

» Static scope is implemented by two methods.
  (1) Access Link (2) Display

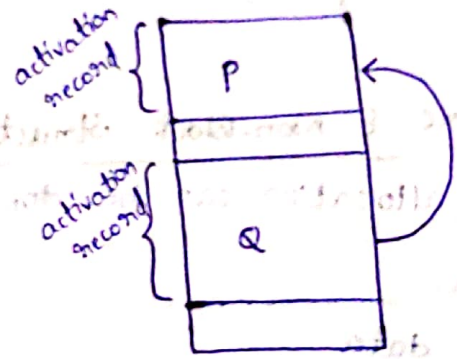
• Dynamic Scope is implemented by two methods.
  (1) Deep access (2) Shallow access.

» (1) Access Link :-

The implementation of lexical scope can be obtained by using pointer to each activation record. These pointers are called Access Link.

If a procedure Q is nested within procedure P, then access link of Q points to access link of most recent activation record of procedure P.

Ex:-
```
P {
    Q {
      ...
    }
}
```



(2) Display :-

It is expensive to traverse down access link everytime when a particular non-local variable is accessed.

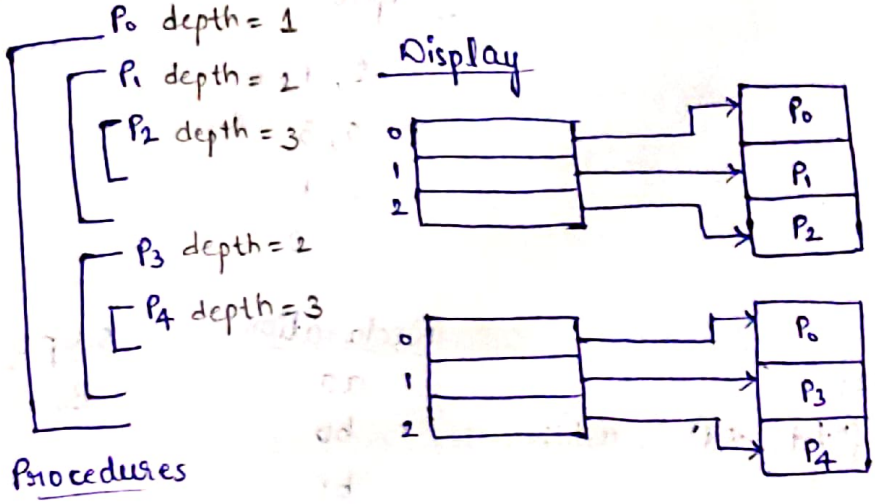
To speed up the access to non-locals, it can be achieved by maintaining an array of pointers called Display.

In display :-

» an array of pointers to activation record is maintained.
» array is indexed by nesting levels.
» the pointers point to only accessible activation record.

» the display changes when a new activation occurs and it must be reset when the control returns from the new activation.

Ex:-



If procedure is main(), then, depth = 1

Comparison between Access Link & Display

Access Link	Display
1) Access link will take more time to access especially when non-local variables are at many levels.	1) Display is used to generate the code efficiently.
2) It requires less space.	2) It requires more space at run time.

(1) **Deep Access**

To keep the stack of active variables, use control links instead of access link.

When you want to find a variable, then search the stack from top to bottom looking for most recent activation record that contains the space for desired variables.

In this method, a symbol tables needs to be used at run time.

(2) **Shallow Access**

Keep a central storage with one slot for every variable name. If the names are not created at run time, then the storage layout can be fixed at compile time. Otherwise, when new activation of procedure occurs, then that procedure changes the storage entries for its locals at entry and exit.

Difference b/w Deep Access and Shallow Access.

Deep Access	Shallow Access
1) It takes longer time to access the non-locals	1) It allows fast access.
2) It needs a symbol table at run time	2) It has overhead of handling procedures entry & exit.

Ex program :-

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

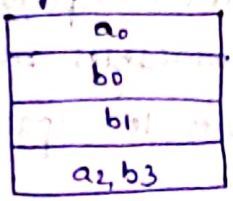
B0, B1, B2, B3

Output :-

2, 1
0, 3
0, 1
0, 0

Declaration	scope
a0	B0 → B2
b0	
b1	
a2	
b3	

Storage Allocation



6/12/2023

UNIT - 5

Principle sources of Optimization / Code Optimization Techniques

target code uses properties of target machine.

Machine Independent & Dependent Optimization Codes :-

→ Machine independent optimizations are program transformations that improve target code without taking into consideration any properties of target machine.

→ Machine dependent optimizations are based on register allocat' and utilization of special machine instruction sequence.

Advantages of Code Optimization

» It takes less time to execute the optimized code.

» Less consumption of memory.

The meaning shall be preserved.

Code Optimization Techniques

1) Common sub-expression elimination
2) Constant folding
3) Copy propagation
4) Dead code elimination
5) Code motion
6) Induction variable elimination & Reduction in strength.

① Common sub-expression elimination

It is done if —
- it was previously computed
- values of variables have not changed.

Ex:- a = b+c
b = a-d ⇒ value of b is changed
c = b+c ⇒ since b is changed, (b+c) is not common sub-expression
d = a-d ⇒ both a & d values are not changed.
& (a-d) is already computed as b = a-d

So, the optimized code is :-

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

② Constant folding

If the value of an expression is constant, use the constant instead (permoperformed at compile time)
Evaluate the expression & submit the result of expression.

Ex:- area = $(\frac{22}{7}) * r * r$

Here, $\frac{22}{7}$ is evaluated & and the result 3.14 is used.

⇒ area = 3.14 * r * r

③ Constant Copy propagation

Here, the constant value replaces a variable

Ex:- pi = 3.14, r = 5
area = pi * r * r
⇒ area = 3.14 * 5 * 5 (no need of declaring when values are known in advance)

Copy propagation
a = b+c
x = a
further instead of using x we can use a

③ Copy propagation :- duplication is not required.

④ Dead code elimination

It includes elimination of the statements which are never executed or even if executed, when the o/p is never used.

Ex:-
```
i=0;
if (i==1){
    a = x+i;
}
```
Here, when i is initialized as 0 the expression (i==1) never becomes true & its statements are never executed.

So, i=0;

Ex:-
```
int add (int x, int y){
    int z;
    z = x+y;
    return z;
    printf("%d", z);
}
```
when return stmt is encountered, the further stmts are not executed. Here, print stmt is never executed.

> A variable is live, if its value can be used subsequently in the program; otherwise it is called dead.

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

dead ← x = a
y = a*b
z = a+c

⑤ Code motion:

It is a technique which moves the code outside the loop; if it doesn't make any difference, in the values ~~before or after~~ when executed inside or outside the loop.

Ex:-
```
for (i=0; i<100; i++){
    x = y+z;
    a[i] = 6*i;
}
```
for loop has no effect on x=y+z. Its value is not effected even if it outside the loop.

If it is loop invariant computatn, only then we can move it outside the loop

so,
```
x = y+z;
for (i=0; i<100; i++){
    a[i] = 6*i;
}
```

> Moves the code outside of a loop
> x=y+z is called as "Loop Invariant computation"
> i=i+1 is "Loop variant computatio

⑥ Induction variable elimination & reduction in strength:

To reduce the strength; we replace the complex operators with those having less strength. Expensive operators are replaced with cheaper operators.

Ex:- b = a * 2

Instead, b = a+a.

Replace exponential by multiplicat"	Induction variable is the loop control variable.

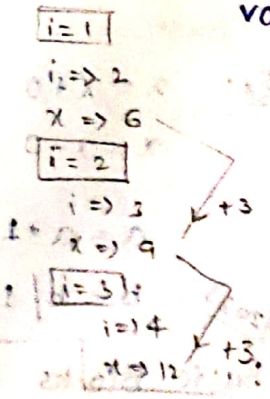
If the value of any variable in any loop gets changed every time, then the variable is called as Induction variable.

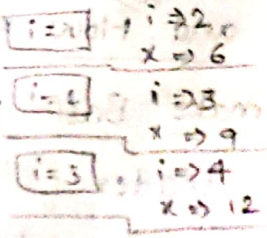
With each iteration, its value either gets incremented or decremented by some constant value.

Ex: B1
```
i := i+1
x := 3*i
y := a[x]
if y < 15, goto B2;
```

upon every increment of i by 1, x gets incremented by 3

So, instead of using i in x := 3*i we can use, x := x+3





So, B1
```
i := i+1
x := x+3
y := a[x]
if y < 15, goto B2
```

Ex: B1
```
i=1;
while (i<10)
{
  t = i*4;
  i = i+1;
}
```
⇒
```
t = 0;
while (t < 36)
{
  t = t+4;
}
```

Transformation on Basic block

It is classified into 2 types.

1) Structure preserving transformation
2) Algebraic preserving transformation.

- (i) Common sub expression elimination
- (ii) Dead code elimination
- (iii) Renaming of temp vars
- (iv) Interchange of 2 independent adjacent stmts

1)

(i) Ex:
```
x = y+z          x = y+z
y = x-w    ⇒     y = x-w
z = y+z          z = y+z
w = x-w          w = y
```

(ii)
```
B1 { a = a+2        { b = b+c
   { b = b+c   ⇒
B2 { b = b*c        { b = b*c
   { c = b+2        { c = b+2
```

(iii)
```
t1 = x*y          t1 = x*y
t2 = z-t1   ⇒     t2 = z-t1
t1 = t1*w         t3 = t1*w
```

Preserving x*y in t1 & x*y*z in t3

(iv) $t_1 = x * y$
$t_2 = z - t_1$
$t_3 = t_1 * w$
$w = t_2 + t_3$

$\Rightarrow$

$t_1 = x * y$
$t_3 = t_1 * w$
$t_2 = z - t_1$
$w = t_2 + t_3$

sub expressions can also be interchanged

(2)

Ex: $x = x + 0$
$x = x - 0$
$a = a * 1$
$b = b / 1$

$c = d ** 2$
$\Rightarrow$ pow(d,2)

These type of algebraic transform-ations can be done for improving the basic blocks.

8/12/2023

Basic Blocks and control Flow Graphs (Diagrams)

{single entry, single exit, no jump stmts}

Algorithm to make Basic Blocks

1) Identify the leader (1st line of code)
2) First line of the code is leader.
3) Address of conditional and unconditional goto are leaders.
4) Immediate next line of goto are leader.
5) Make basic block from leader to line before next leader.

Ex:-

Basic Blocks

B1:
1. i = 1

B2:
2. j = 1

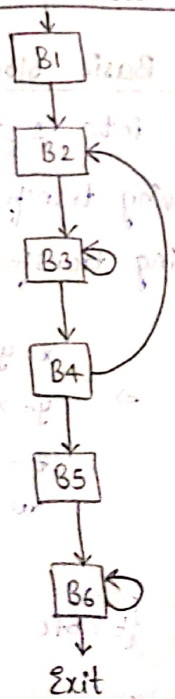
B3:
3. $t_1 = 10 * j$
4. $t_2 = t_1 + j$
5. $t_3 = 8 \times t_2$
6. $t_4 = t_3 - 8$
7. a[t_4] = 0 0
8. j = j+1
9. if j < 10 goto (3)

B4:
10. i = i+1
11. if i < 10 goto (2)

B5:
12. i = 1

B6:
13. $t_5 = i - 1$
14. $t_6 = 8 * t_5$
15. a[t_6] = 10
16. i = i+1
17. if i <= 10 goto (13)

Control Flow Graph



Control Flow ...

represent basic block and
» We can draw directed ed
• there is conditional or
• Block 2 immediately fol

Ex2:

B1:
1. p = 0
2. i = 1

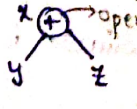
B2:
3. $t_1 = i * 4$
4. t_2 = a[t_1]
5. $t_3 = i * 4$
6. t_4 = a[t_3]
7. $t_5 = t_2 + t_4$
8. $t_6 = p + t_5$
9. p = t_6
10. $t_7 = i + 1$
11. i = t_7
12. if i <= 20 goto (3)

B3:
13. $t_8 = j + 1$
14. j = t_8

DAG representation of t...

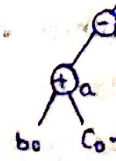
DAG => Directed Acyclic...
» DAG determines common...
» In DAG, leads are lab...
» Interior nodes are oper...
» Internal nodes also rep...

Ex1: $x = y + z$



Ex3: $a = b + c$

Here, (a-d) is the common sub-express...
$b = a - d$
$c = b + c$
$d = a - d$



Control Flow Graph : It is a directed graph in which nodes represent basic block and edges represent flow of control.

» We can draw directed edge from Block 1 to Block 2 if :
- there is conditional or unconditional jump from block 1 to block 2
- Block 2 immediately follow Block 1

Ex2 :

B1
1. p = 0
2. i = 1

B2
3. $t_1 = i * 4$
4. $t_2 = a[t_1]$
5. $t_3 = i * 4$
6. $t_4 = a[t_3]$
7. $t_5 = t_2 + t_4$
8. $t_6 = p + t_5$
9. p = t_6
10. $t_7 = i + 1$
11. i = t_7
12. if i <= 20 goto (3)

B3
13. $t_8 = j + 1$
14. j = t_8

code :

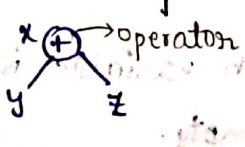
```
begin
    p = 0
    i = 1
do  begin
    p = p + a[i] * b[i]
    i = i+1
end
while (i <= 20)
    j = j+1
end
```

DAG representation of the Basic Blocks

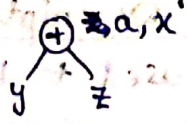
DAG ⇒ Directed Acyclic Graph

- DAG determines common sub-expressions.
- In DAG, leads are labelled by variable names or constants.
- Interior nodes are operators.
- Internal nodes also represent the result of the expression.

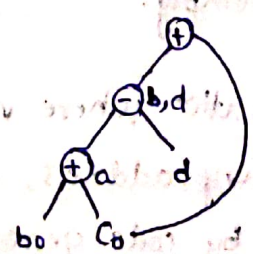
Ex1 : x = y + z



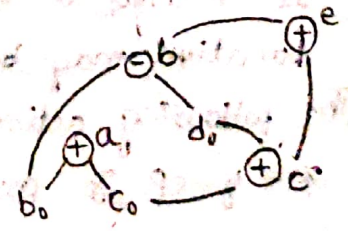
Ex2 : a = y+z
x = y+z



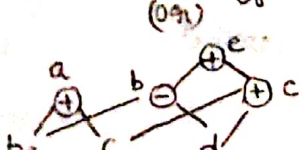
Ex3 : a = b+c
b = a-d
c = b+c
d = a-d



Ex4 : a = b+c
b = b-d
c = c+d
e = b+c



(or)

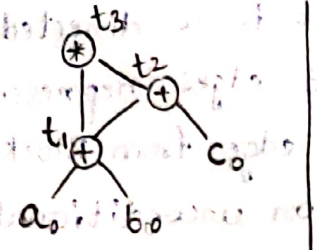


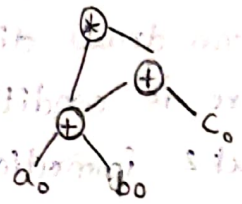
Ex 5 : $(a+b)*(a+b+c)$

$t_1 = a+b$

$t_2 = t_1 + c$

$\Rightarrow t_3 = t_1 * t_2$





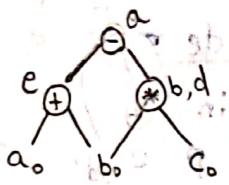
12/12/2022 ~~$t_1 * t_2$~~

Ex 6 : $d = b*c$

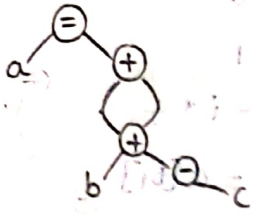
$e = a+b$

$b = b*c$

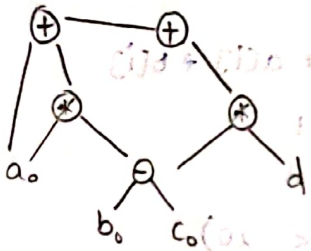
$a = e-d$



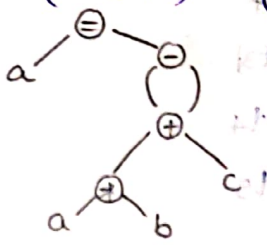
Ex 8 : $a = b*-c + b*-c$



** Ex 7 : $a + a*(b-c) + (b-c)*d$



Ex 9 : $a = (a*b+c) - (a*b+c)$



Applications of DAG

» Determines the common sub-expressions.

» Determining which names are used inside the block & computed outside the block.

» Determining which stmts of the block could have computed their value outside the block.

» Simplifying one list of quadruples by eliminating the common sub-expressions.

Code Generation Algorithm **

A simple code generator algorithm :-

» It generates the target code for a sequence of instructions.

» It uses a function getReg() to assign registers to variables

» It uses two data structures (i) register descriptor ; (ii) address descriptor

Register Descripton

It is used to keep track of which variable is stored in register.

Initially, all the registers are empty.

Address Descripton

It is used to keep track of location where variable is stored.

Location may be registers, memory address, stack, etc.

The following actions are performed by code generator for an instruct"

$x = y$ op z assumes that 'L' is location where the ~~addr~~ o/p of y op z is stored

1) Call the function getreg to get the location of L.
2) Determine the present location of y by consulting its address descriptor.
If y is not present in location L, then generate the instruction MOV y', L to copy the value of y to L
3) The present location of z is determined using step 2, and the instruction is generated as ~~MOV z' to L~~ ~~Op z~~ Op z', L
4) Now, L contains the value of y Op z that is assigned to x
If L is a register then, update its descriptor that it contains the value of x. Update the address descriptor of x to indicate that it is stored in address L.
5) If y and z have no future use, then update the descriptors to remove y and z.

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

$t_1 = a-b$

$t_2 = a-c$

$t_3 = t_1 + t_2$

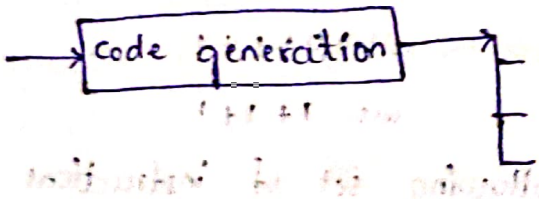
~~t4~~ $d = t_3 + t_2$

~~d = t4~~

Statement	code generatn	register descriptor	address descriptor
$t_1 = a-b$	MOV a, R0 SUB b, R0	R0 contains t_1	t_1 in R0
$t_2 = a-c$	MOV a, R1 SUB c, R1	R1 contains t_2 R0 contains t_1	t_2 in R1 t_1 in R0
$t_3 = t_1 + t_2$	ADD R1, R0	R0 contains t_3 R1 contains t_2	t_3 in R0 t_2 in R1
~~d~~ $= t_3 + t_2$	ADD R1, R0	R0 contains ~~d~~	d in R0

3/12/2022

Object Code forms



Absolute code: It is a machine code that contains references to actual addresses within program's address space. The generated code can be placed directly in the memory and the execution status immediately.

Relocatable code: Producing a re-locatable machine language program

as output allows sub-programs to be compiled separately. A set of relocatable object modules can be linked together and loaded for execution with the help of linker/loader.

The advantages of generating relocatable machine language code is that the flexibility to compile sub-programs separately and to call other previously compiled program from an object module.

Assembler code: Producing an assembly language program as o/p. makes the process of code generation easier we can generate symbolic instructions and use the MACRO facilities of the assembler to help in generation of code. But, generating assembler code as an o/p makes code generation slower, because assembling, linking & loading is required.

Cost of the Instruction

Addressing mode	from (src)	address (dest)	Added cost
Absolute	M (memory)	M	1
Register	R	R	0
Indexed	c (R)	c + content (R)	1
Indirect register	* (R)	content (R)	0
Indirect indexed	* c (R)	content + content (R)	1
Literal	# c	c	1

The cost of the instruction can be computed as,

1 + cost associated with source & destination addressing modes given by the added cost.

Ex:		cost	interpretation
	MOV R_0, R_1	1	cost: 0 + 0 + 1
	MOV R_0, M	2	cost: 0 + 1 + 1
	SUB 5(R_0), *10(R_1)	3	cost: 1 + 1 + 1

Q) Compute the cost of following set of instructions.

	cost	interpretation
MOV a, R_0	2	1 + 0 + 1
ADD b, R_0	2	1 + 0 + 1
MOV R_0, c	2	0 + 1 + 1
	Total cost: 6	

Q) Compute the cost of following set of instructions.

MOV *R1, *R0 :- 0+0+1 = 1

ADD *R2, *R1 :- 0+0+1 = 1

Total cost: 2

MOV B, R0 1+0+1 = 2

ADD C, R0 1+0+1 = 2

MOV R0, A 0+1+1 = 2

6

MOV B, A 1+1+0 = 3

ADD C, A 1+1+0 = 3

6

4/12/2023

** Peephole Optimization (done on target code)

This technique works locally on source code to transform it into an optimized code. small portion ↓ window or

Peephole optimization is a short inst sequence of target instructions that can be replaced by short or ~~long~~ faster sequence of instructions. peephole

It examines at most a few instructions transforming into other less expensive ones, such as turning multiplication of x by 2 into an addition of x with itself. ($x*2 \Rightarrow x+x$)

Characteristics

→ Redundant instruction elimination
→ Unreachable code
→ Flow of control optimization
→ Algebraic simplification

1. Redundant Instruction elimination :- At source code level, the following can be done by the user.

Ex1:

```
y = x+5          y = x+5
i = y      ⇒     k = y+10
z = i
k = z+10
```

(i, z redundant)

Ex2:

```
int add_ten(int x){
  int y, z;
  y = 10;
  z = x+y
  return z;
}
```

```
int add_ten(int x)
{
  return x+10;
}
```

2. Unreachable Code : It is a part of program code that is never accessed because of program constructs. Programmers may have accidentally return a piece of code, that can never be reached

Ex:-
```
int add_ten(int x){          int add_ten(int x){
    return x+10;      ⇒          return x+10;
    printf("x = %d", x);         }
}
```

In this function, printf statement is never executed as program control returns before its execution. Hence printf can be removed.

3. Flow of control Optimization : These are instances in a code where the program control jumps back & forth without performing any significant task. So, these jumps can be removed.

Ex:
```
goto L1                goto L3              goto L3
L1: goto L2      ⇒     L3: goto L3    ≡     L1: goto L3
L2: goto L3            L3: Mov a, R0        L2: goto L3
L3: Mov a, R0          ??                   L3: Mov a, R0
```

4. Algebraic Simplification : These are some situations where algebraic expressions can be made simple.

Ex:
```
y = x + 0        y = x
y = x - 0   ⇒    y = x
y = x * 1        y = x
y = x / 1        y = x
```

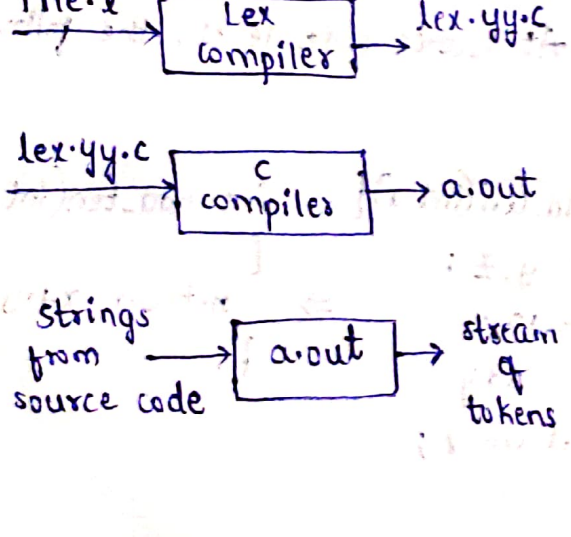
5. Machine Idioms

Ex:
```
a = a+1   ⇒   INC a
a = a-1       DEC a
```

LEX Tool ⇒ to generate stream of tokens

YACC Tool ⇒ works in second phase of compilation. i.e, Syntax Analysis

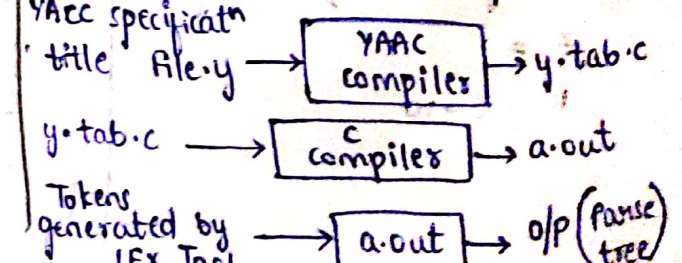
LEX Tool



YACC Tool (Yet Another Compiler-Compiler)

It is a tool for generating look-ahead left to right parser. (LALR(1))

It takes I/P from lexical analyzer & generates parse tree.



19/12/2023

Recursive Descent Parser

A parser that uses collection of recursive procedures for parsing the given input string is called Recursive Descent Parser.

Here, the CFG is used to build the recursive procedures.

The RHS of the production rule is directly converted to a program.

for each ~~non-terimo~~ non-terminal, a separate procedure is written and body of the procedure is RHS of the corresponding non-terminal.

Basic steps for the construction of RD parser

The RHS of the rule is directly converted into program code symbol by symbol.

If I/P symbol is non-terminal, then a call to the procedure corresponding to the non-terminal is made.

If I/P symbol is terminal, then it is matched with the lookahead from I/P. The lookahead ptr has to be advanced on ~~from~~ matching of the I/P symbol.

If the production rule has many alternatives, then all ~~its~~ these alternatives have to be combined into a single body of the procedure.

The parser should be activated by the procedure corresponding to the start symbol.

Ex: $E \rightarrow num\ T$

$T \rightarrow * num\ T / \varepsilon$

```
procedure E()
{
   if lookahead (num) {
      match(num);
      T();
   }
   else
      error();
   if lookahead = '$' then
      declare success;
} // End of procedure E
```

```
procedure T()
{
  if lookahead = '*' then {
     match('*');
  }
  if lookahed = 'num then
     match (num);
  else
     error();
  }
  else NULL; // indicates ε
}
```

The other alternative is combined into procedure T().

```
procedure match (token t)
{
if lookahead = t then
   lookahead = new_token; // lookahead ptr is advanced
else
   error();
```

```
procedure error()
{
   print("ERROR!");
```

Ex:- | 3 | * | 4 | $ |
↑

step-by-step process

Advantages of RD parsers

» They are simple to build
» They can be built with the help of parse trees.

Limitations

» They are not very efficient as compared to other parser techniques
» There are chances that program for RD parser may enter in infinite loop for some inputs.
» It is difficult to parse the string, if lookahead symbol is arbitarily long.

Q) E → T + E | T
T → V * T | V
V → id

Write procedures for non-terminals of the grammar to make a recursive descent parser.

```
procedure E()
{
    T();
    if lookahead = '+' then
        match '+';
```

```
procedure T()
{
```

Q) bexpr → bexpr or bterm | bterm

bterm → bterm and bfactor | bfactor

bfactor → bfactor | (bexpr) | true | false

Write RD parser where, or, and, (,), true, false are terminals.

bexpr = E ; bterm = T ; bfactor = F ; or = + ; and = *

E → E + T | T

T → T * F | F

F → F | (E) | true | false

Operator Precedence.

Symbol Table

comparision b/w static, stack and Heap Allocation.

	Static allocation	Stack allocation	Heap allocation.
1.	Stack allocation is done for all data objects at compile time	In stack allocation stack is used to manage runtime storage	In heap allocation, heap is used to manage dynamic memory allocation.
2.	Data structures can not be created dynamically bcoz in static allocation compiler can determine the amount of storage required by each data objects.	Data structures and data objects can be created dynamically.	Data structures and data objects can be created dynamically.
3.	Memory allocation: The names of data objects are bound to storage at compile time.	Memory Allocation: Using LIFO activation records and data objects are pushed onto the stack. The memory addressing can be done using index and registers.	Memory Allocation: A contiguous block of memory from heap is allocated for activation record or data object. A linked list is maintained for free blocks.
4.	Merits and limitations:- This allocation strategy is simple to implement but supports static allocation only. Similarly recursive procedures are not supported by stated allocation strategy.	Merits and limitations:- It supports dynamic memory allocation but it is slower than static allocation strategy. Supports recursive procedures but references to non local variables after activation record can not be retained.	Merits and limitations:- Efficient memory management is done using linked list. The deallocated space can be reused. But since memory block is allocated using best fit, holes may get introduced in the memory.

The heap allocation allocates the continuous block of memory when required for storage of activation records or other data objects. This allocated memory can be deallocated when activation ends. This deallocated (free) space can be further reused by heap manager.

3. The efficient heap management can be done by
   1. creating a linked list for the free blocks and when any memory is deallocated that block of memory is appended in the linked list
   2. Allocate the most suitable block of memory from the linked list. i.e. use best fit technique for allocation of block.

Storage Allocation Strategies:-

There are 3 different storage allocation strategies based on this division of run time storage. The strategies are:

1. Static allocation:- The static allocation is for all the data objects of compile time.
2. Stack allocation:- In this stack allocation a stack is used to manage the runtime storage.
3. Heap allocation:- In heap allocation the heap is used to manage the dynamic memory allocation.

1. Static allocation:-

1. The size of data objects is known at compile time. The names of these objects are bound to storage at compile time only and such an allocation of data objects is done by static allocation.
2. The binding of name with the amount of storage allocated do not change at run time. Hence the name of this allocation is static allocation.
3. In static allocation the compiler can determine the amount of storage required by each data objects And therefore it becomes easy for a compiler to find the addresses of these data in the activation record.
4. At compile time compiler can fill the addresses at which the target code can find the data it operates on.
5. FORTRAN uses the static allocation strategy.

Limitations of Static Allocation:-

1. The static allocation can be done only if the size of the data objects is known at compile time.

2. The data structures can not be created dynamically. It means that, the static allocation can not manage the allocation of memory at run time.

3. Recursive procedures are not supported by this type of allocation.

2. Stack Allocation:-

1. The stack allocation strategy is a strategy in which the storage is organized as stack. This stack is also called control stack.
2. As the activation begins, the activation records are pushed onto the stack and on completion of this activation the corresponding activation records can be poped.
3. The locals are stored in the each activation record. Hence locals are bound to corresponding activation record on each fresh activation.
4. The data structures can be created dynamically for stack allocation.

Limitations of stack allocation:-

- The memory addressing can be done using pointers and index registers. Hence this type of allocation is slower than static allocation.

3. Heap Allocation:-

1. If the values of non local variables must be retained even after the activation record, then such a retaining is not possible by stack allocation. This limitation of stack allocation is because bcoz of its Last in FIRST Out nature. For retaining of such local variables heap allocation strategy is used.