CSE420 Compilers Design
javatpoint, tutorialspoint, geeksforgeeks

Compilers Design

Week 0

- Theories of Compilers Design used in NLP, Machine Translation, ~~Compilers~~ Information Retrieval, Semantic Web Technology, Query Language Processing.

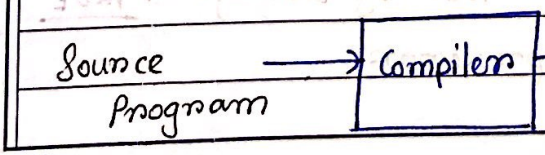
- procedure for identifying syntax errors.

"Compilers Basics"

Compilers is a translator that converts the high-level language into the machine language.
Compilers is used to show errors to the programmer.

A compilers is a computer program that translates computer code written in one programming language (the source language) into another language. (the target language.)

- Programming problems are easier to solve in high-level languages. (HLL)
- Solutions are usually more efficient (faster, smaller) when written in Machine Language.
- Compilers are the bridges → connects the software with hardware; Tools to translate programs written in HLL to efficient executable code.
  Another important role of the compilers is to report any errors in the source program that it detects during the translation process. (compilers errors)

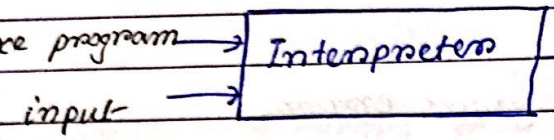


(Compilers is a software program)

(diff from a compiler)

Q. What is an interpreter?

Ans: An interpreter requires no pre-computation it directly takes the source program and runs the input on it producing the desired output. (line by line / instruction by instruction)



Q. Difference bet^n Interpreter and Compilers

↳ slower

* The interpreter takes less memory (no exe file needed)
* The interpreter is invoked each time the CPU has to run any piece of instruction (as no files .exe generated)

Compilers & interpreters do similar jobs, but there are differences:

To run a program you've written, it must first be translated into machine code so the computer can read it. This is what compilers and interpreters do.

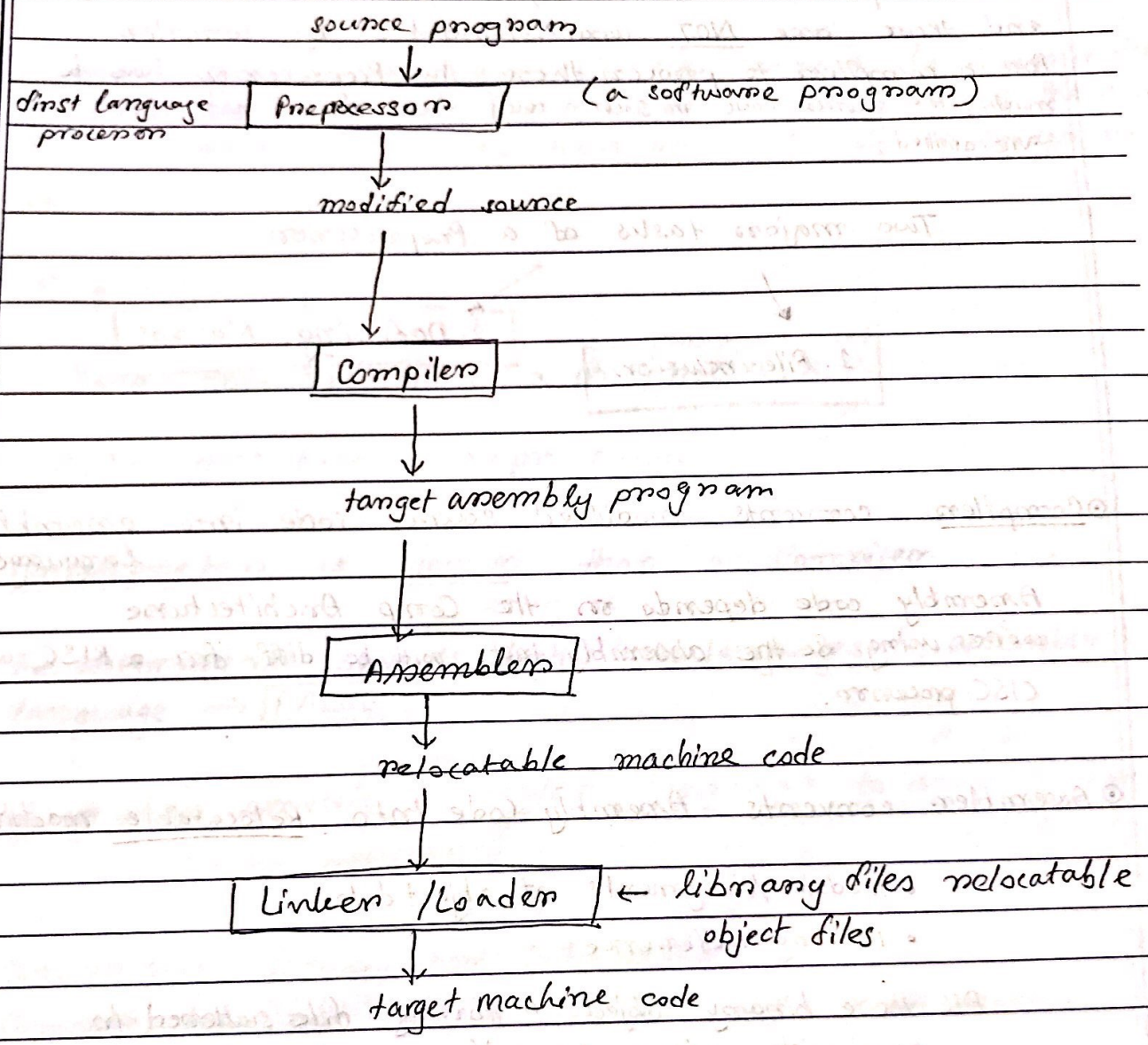
However, Compilers convert the code all at once, save it, then run it. ↳ (C, C++) Java

Whereas, interpreters translate the code one line at a time, as it is run. ↳ (Python, Ruby) JavaScript

* Interpreter is slower
* The interpreter converts HLL to Assembly Language ⇒ FALSE
* ~~The interpreter~~
* The compiler converts HLL to Assembly Language ⇒ TRUE
* Java does not have a preprocessor.

- A Language Processing System is a system that actually converts the source program written in one language into a target machine code consisting of 1s and 0s by incorporating different language processors such as preprocessors, compilers, assemblers, linkers and loaders.

Q Steps of a Language Processing System.



(asks the Preprocessor to include the contents of standard io. header file into our program)

⊙ instructions preceded by a hash symbol are Preprocessor directives / Preprocessor instructions and these are NOT understandable by Compiler. For a compiler to process these, the Preprocessor has to modify the source code in such a way that these instructions are omitted.

Two major tasks of a Preprocessor

1. File inclusion
2. Defining Macros

⊙ Compiler converts modified source code into assembly language.

Assembly code depends on the Comp. Architecture we're using. So the assembly inst. will be diff for a RISC and CISC processor.

⊙ Assembler converts Assembly code into Relocatable machine code.

- modules / fragments of object files
- library reference

All these binary object & library files scattered in hard disk, after Assembler converted A. code to M. code

⊙ Linker is a software program that links all these library & object files into one single separate executable file. keeping in mind the order of instructions.

(Linker links relocatable object files into executable file)

Sumary: { * Compilers
* Compilers Vs Interpreters
* Steps of LP System.

@Loaders loads executable machine code into main memory for execution.

Loaders will take the executable file produced by Linkers in the hard disk and it will load them into the memory by mapping them into memory addresses, after that those segments will be loaded for execution

Quiz - 01

1. Java does not have a preprocessor.

   C, C++ does have a preprocessor

2. Interpreter is slower than a Compiler

3. The interpreter converts higher level languages to assembly language → FALSE

4. The compiler converts higher-level languages to assembly language.
   ↳ TRUE

Compiler runs faster than Interpreter.
Compilers take less time to execute than interpreters because the run time is slower.
Interpreter doesn't have any compile time as the code never compiles, it directly runs.

"Phases of Compiler"

July 12th, 2020

Week 1, Lecture - 2

Phases of a Compiler (LSSICT)

- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- Intermediate code generation
- Code optimization
- Code generator

built-in keyword → (lexeme / meaningful sequence of characters)

Lexical Analyzer/scanner will find lexime (variable, no., operator) and generate tokens for the same.
This token stream fed as input into Syntax Analyser/Parser. which will generate a syntax tree basing on a grammatical structure. or CFG
Semantic Analyzer checks for inconsistency in meaning. (array size can never be declared as a double)

ICG → (3 address code generated called Intermediate Representation)

(Machine-independent code optimizers)

MICO's task → to reduce the no. of lines or to reduce the no. of variables in 3-address code to decrease Compilation Time and increase Efficiency.

Code Generator → generates Assembly Instructions which is hardware-specific.

(Machine-dependent code optimizers)

MDCO → to reduce no. of lines + variables in assembly code. (optional)

Symbol Table: a table that contains all the variables used in the Software program (index/pointers to) for Compilers.
It includes variables and also information regarding the variables such as the name, the type, the scope, the storage allocation for that variable.

All the phases of compilers will interact with Symbol table to install or modify updates regarding the variables.

Symbol Table contains: • Variables • Procedures (names, no. and types of ~~argument used~~)

↳ 1. Records the identifiers used in the source program
2. It's a data structure with collection of records.

Analysis - Synthesis model of compilation (all phases of compilers broadly classified into 2 major categories)

Analysis	Synthesis
front-end of compilers as it links the compilers to the software part of the machine	back-end of compilers as it links the compilers to the machine's hardware part
Breaks up the source program into constituents	Constructs the target program
[Scanner, Parser, Semantic Analyzer]	[Code optimizers, Code generator]
Analysis Phase	Synthesis Phase

Syntax tree from CFG.

(scan the entire input statement characters by characters) to find a pattern

(contains Regular Expression) id → letter(letter|digit)*

Lexical Analysis divides the program text into lexemes or tokens.

- Tokens are the 'words' of the programming language.
- Lexeme are meaningful sequences of characters from source character stream.

token ⇒ < token_name, attribute_value >

Lexical analyser also updates Symbol Table entries, removes the white spaces, removes the comments.

Syntax → understand sentence structure

Semantic analysis → performs type checking and performs type conversion. } looks for meaning

ICG
- 3 address code which are assembly-like instructions

why? ⇒ (max^m 3 addresses (variables/digits) per line)
- At most 1 operator on the RHS of equal sign
- All the 3 instructions are always executed line by line (in order)

~~Code Optimatio~~

Code Optimization → task is to reduce no. of lines or variables to improve the time and space efficiency of the generated code.

Peephole optimization (small fragment of code)

Global optimization (takes more (entire code) time)

Code Generation

→ takes as input the intermediate code representation
→ generates assembly code that is hardware specific

Compiler

* Each phase can encounter errors (detects & reports it along the way.
  - lexical error
  - syntax error
  - semantic error
* Should be able to proceed & process the rest of the program after an error detected.
* Should be able to link the errors with the source program.

Quiz-02

1. Code Generation is NOT part of Analysis Part.
2. Operators such as "−", "+", "=" get a token name. **False**
3. Structure of a sentence is checked during Syntax Analysis part

   The compiler checks for inconsistency in meaning of a sentence in the Semantic Analysis Part
4. Code Optimizer reduces the no. of lines and variables.

Week 02, Lecture 01

Lexical Analysis

July 12th, 2020

(tokenize the text of the source program)

Lexical Analysis is the first phase of Compiler Design.
Also called Scanner.
~~It removes the whitespace or commen~~

Lexical Analyzer breaks the syntaxes into a series of tokens, by removing any whitespace or comments in the source code.

if / (/ x / / / 2 / == / =

- if → keyword
- (→ operator
- x → identifier
- 2 → no.
- == → ~~relationoperator~~ relational operator
- = → assignment operator

else → keyword

even/odd → identifier

Lexical analyzer also connected to Symbol Table which is a data structure. It inserts the Identifier & procedure/func token and the class object and instance token to the symbol table with their attributes.

Lexical Analyzer (key fun's)

- Scan input
- remove White Space, New Lines, tabs
- identify tokens
- create Symbol Table
- insert tokens into ST
- generate errors if there's any
- send tokens to parser

Parser
- perform syntax analysis (validating syntaxes)
- actions dictated by token orders
- update symbol table entries
- create abstract representation of the source (using parse tree)
- generate errors

Q. What are major terms for Lexical Analysis?

- **Token** → a classification for a common set of strings
  example: <Identifiers>, <number>, etc.
- **Pattern** → the rules which characterize the set of strings for a token
- **Lexeme** → actual sequence of characters that matches pattern and is classified by a token.
  (identifiers: x, count, name, etc.)
  sum,

lexeme 1 2 3
token <number>

- No specific pattern for defining keywords. (if, const)
(Each of the word has diff patterns, so no ~~comm~~ common pattern in general)
(if, const, do, while, so on)

literal (token) – any seq of characters or sentence within the double quotation mark is called a literal.

Language & Regular Expressions.

* A Regular Expression is a set of Rules / Techniques for constructing Sequences of Symbols (Strings) from an alphabet.

Let Σ be an alphabet, r a Regular Expression, then $L(r)$ is the Language that's characterized by the Rules of r.

Precedence in important in Parsing regular expressions

1. "*" has the highest precedence
2. Concatenation as middle precedence
3. "|" has lowest precedence
4. Use parentheses to override these rules.

* Equality vs. Equivalence

We say two r.e. as equivalent, not equal !!

• Algebraic law of R.E.

1. Concatenation is associative $R(ST) = (RS)T = RST$ ~~$R(ST) = (RS)T = RST$~~
2. "|" is associative $R|(S|T) = (R|S)|T = R|S|T$
3. "|" is commutative $R|S = S|R$
4. Concatenation is distributes over | $R(S|T) = RS|RT$, $(R|S)T = RT|ST$
5. ϵ is identity for concat $\epsilon R = R\epsilon = R$
6. * is idempotent $(R^*)^* = R^*$
7. Relation bet^n * and ϵ.

$$R^* = (R|\epsilon)^*$$

$id = \ell(\ell|d)^*$

important ✱ identifier = letter (letter | digit)*
ID

regular expression of identifier

letter = a|b|c|...|z
digit = 0|1|2|..|9

For defining id's i.e., the first character must be a letter

$x^+ = x(x^*)$

(+ → one or more)

num → digit+

(? → zero or one)

Q. Lexical analyzer scan away blanks, new lines, tabs
Can we define tokens for these?

ws → delim+

Whitespace consists one or more delimetres.

blank → blank
tab → tab
newline → newline
delim → blank | tab | newline

Ans: No token is returned to Parser

Lexical Analyzer does NOT send any whitespace token to the Parser, it just scans away blanks, new lines, tabs

Week 2
Lecture 2

Lexical Analysis (Part 2)

RE defines the pattern (of a string)
FA is used to recognize the pattern.

(NFA → we can move to two or more states with same input symbol) and it accepts epsilon

Automata is an idealized machine to recognize / identify patterns.

Finite State Automata
→ finite number of states

FSA is a program / recognizer that takes a string as an input and determines whether its a sentence of the language or not.

R.E. defines the pattern of a sentence or a string.

one start state
many final states } each state labelled with a state name
directed edges (labelled with symbols)

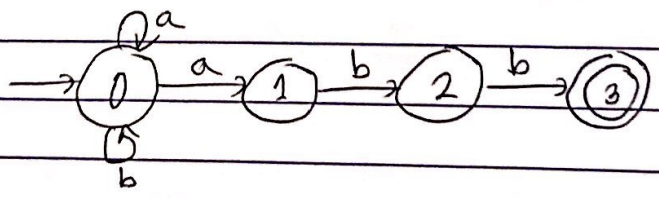
two types → DFA (deterministic)
→ NFA (non-deterministic)

The regular expression is compiled into a recognizer by constructing a generalized transition diagram called a Finite Automaton.

A DFA is an NFA with the following restrictions:
- ε moves are NOT allowed.
- for every state, there's one and only one path for an input symbol

(a|b)*abb

NFA



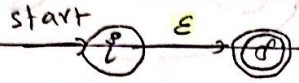
DFA

draw later

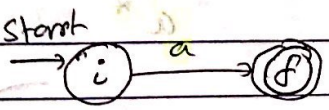
Both DFA and NFA are "recognizers" of regular sets.
But time-space trade space exists

DFAs are (faster) recognizers (can be much bigger too)
drawback
so more storage space

Converting RE to NFAs.
(Thompson's Construction)

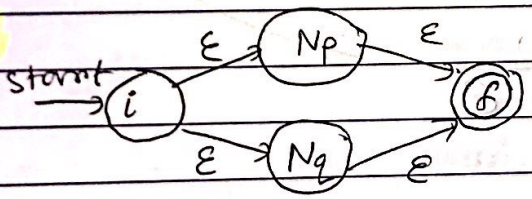


rule 1 • ε ~~stn~~ empty string is a r.e. denoting $\{\epsilon\}$
rule 2 • a is a r.e. denoting $\{a\}$ for any a in Σ

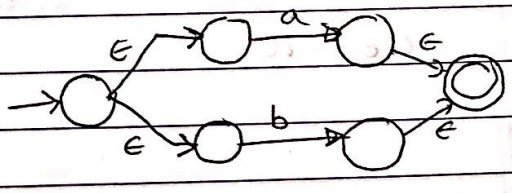


If P and Q are r.e. with NFAs N_p, N_q:

rule 3 #
P|Q (union)

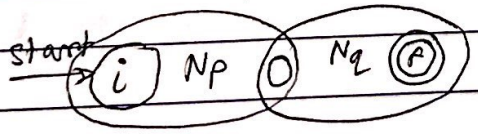


example
a|b



rule 4 #
PQ (concatenation)

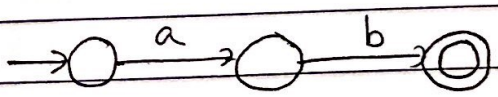
P.Q



example
a b

state
the last ~~symbol~~ of N_p is
first ~~symbol~~ of N_q
state

a.b

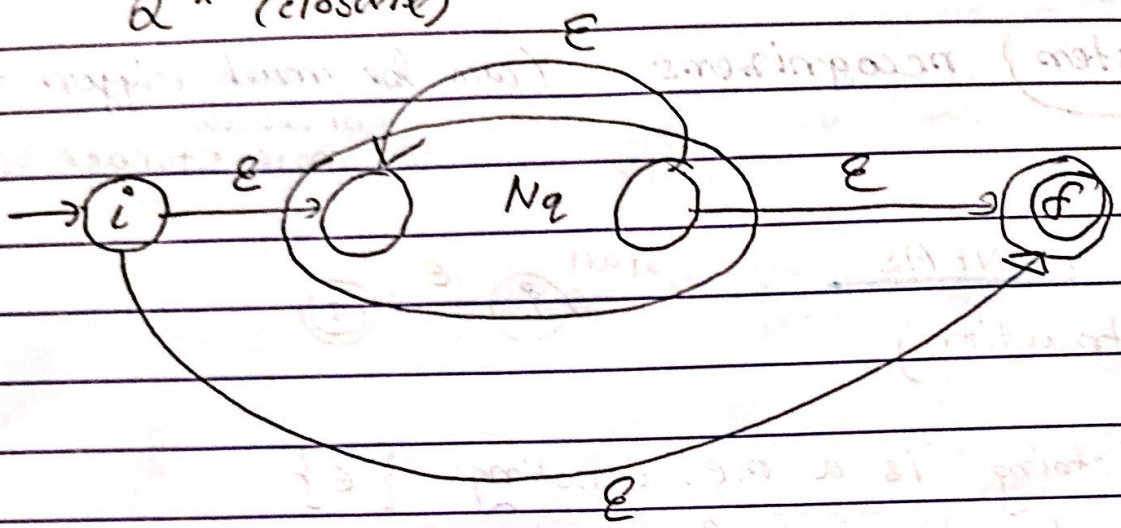


last state of a
first state of b

rule 5

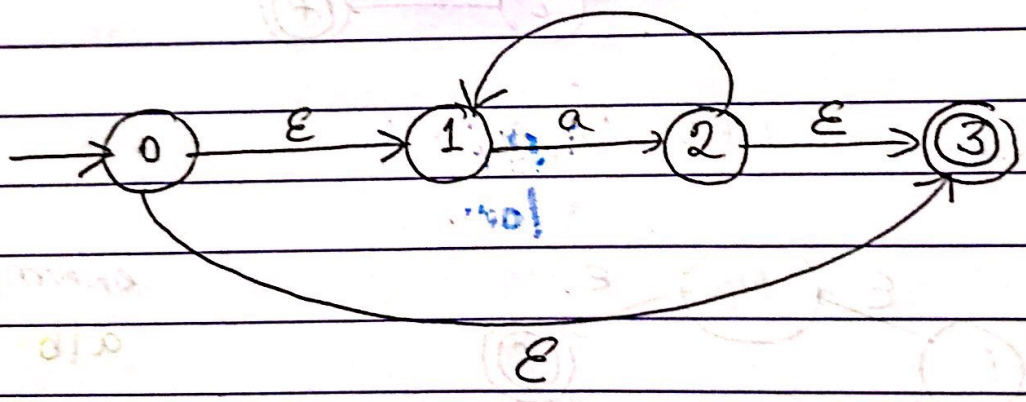
If Q is a r.e. with NFA Nq

Q* (closure)



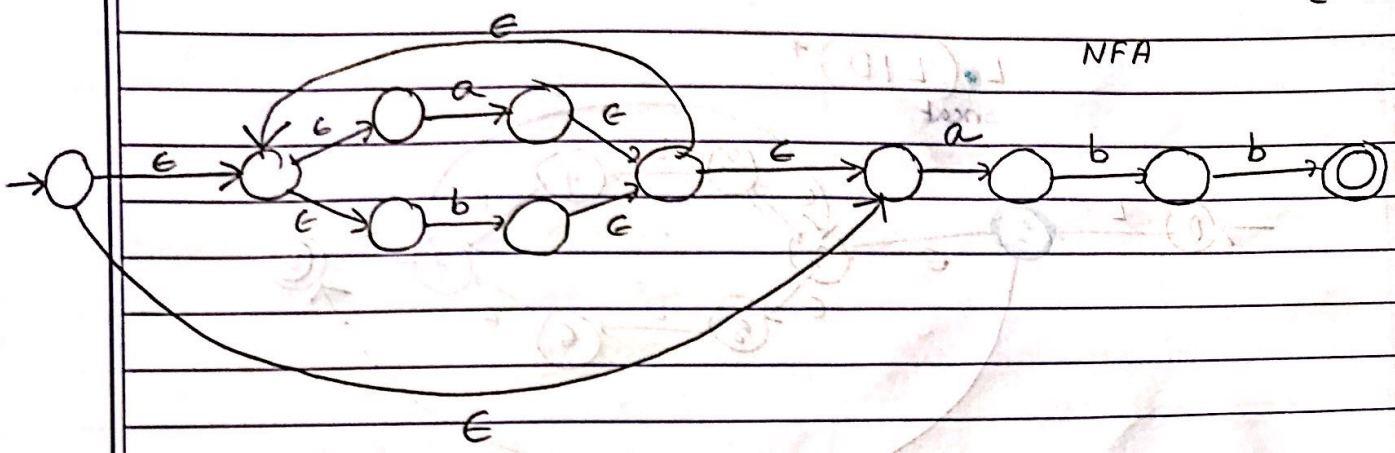
example

a*



aε = a

Example 1 $(a|b)^* abb$ — Using Thompson's C. rule



H.W.

Example 2 $(ab^*c) \mid (a(b|c^*))$

* → highest
. → middle
| → lowest

Precedence Of operators

Precedence of Operators

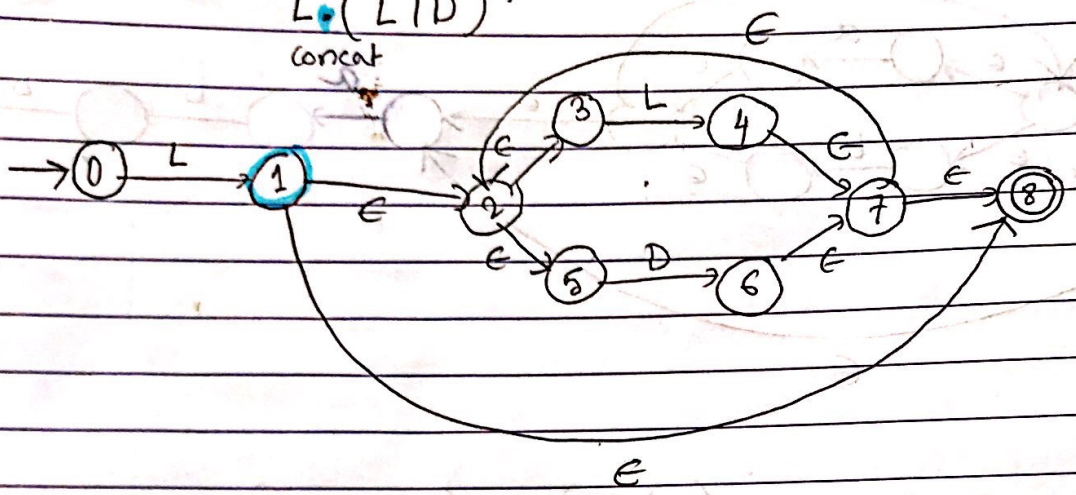
* → highest
. → medium
| → lowest

Example 3

id → letter (letter | digit)*

L.(L|D)*

concat



Quiz-03

1. Lexical Analysis does not generate errors → False
2. Token and lexeme mean the same thing → False
3. We cannot use decimal point in RE as we only use digits → False

Lecture 5
Lexical Analysis

July 22, 2020

Converting RE to DFA

RE → NFA → DFA
(subset construction in 331 Automata)

- direct method RE → DFA (don't have to create NFA)

First, augment the R.E.
adding # at the end.

r → (r) #

Then, we create syntax tree for this augmented r.e.

Input: A regular expression r
Output: A DFA D that recognizes L(r)
(the language defined by r.e. r)

Syntax Tree

- leaves (alphabet or ε) # also
- internal nodes (operators) •, *, |
  concatenation, star-node, or-node
  cat-node

$(a|b)^*abb$

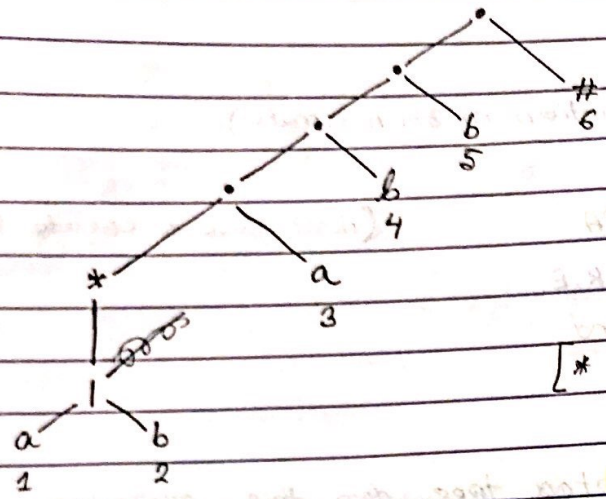
* Add # at end of r.e. to augment it.

$((a|b)^*abb)\#$
1 2 3 4 5 6

To construct ST
- Consider precedence of operators
- Scan r.e. from Left to Right (L to R)
- Parentheses define scope of operation

(non-ε leaf is associated with a unique no.)

abcd, ε, #
number these



[* precedence higher than concat]

We define the position of each non-ε leaf.

Dont define position for ε symbol.

To evaluate **followpos**, we need 3 more functions to be defined for the nodes (not just for leaves) of the syntax tree.

- firstpos(n)
- lastpos(n)
- nullable(n)

$(a|b)^*a$

firstpos(n) = 1, 2, 3

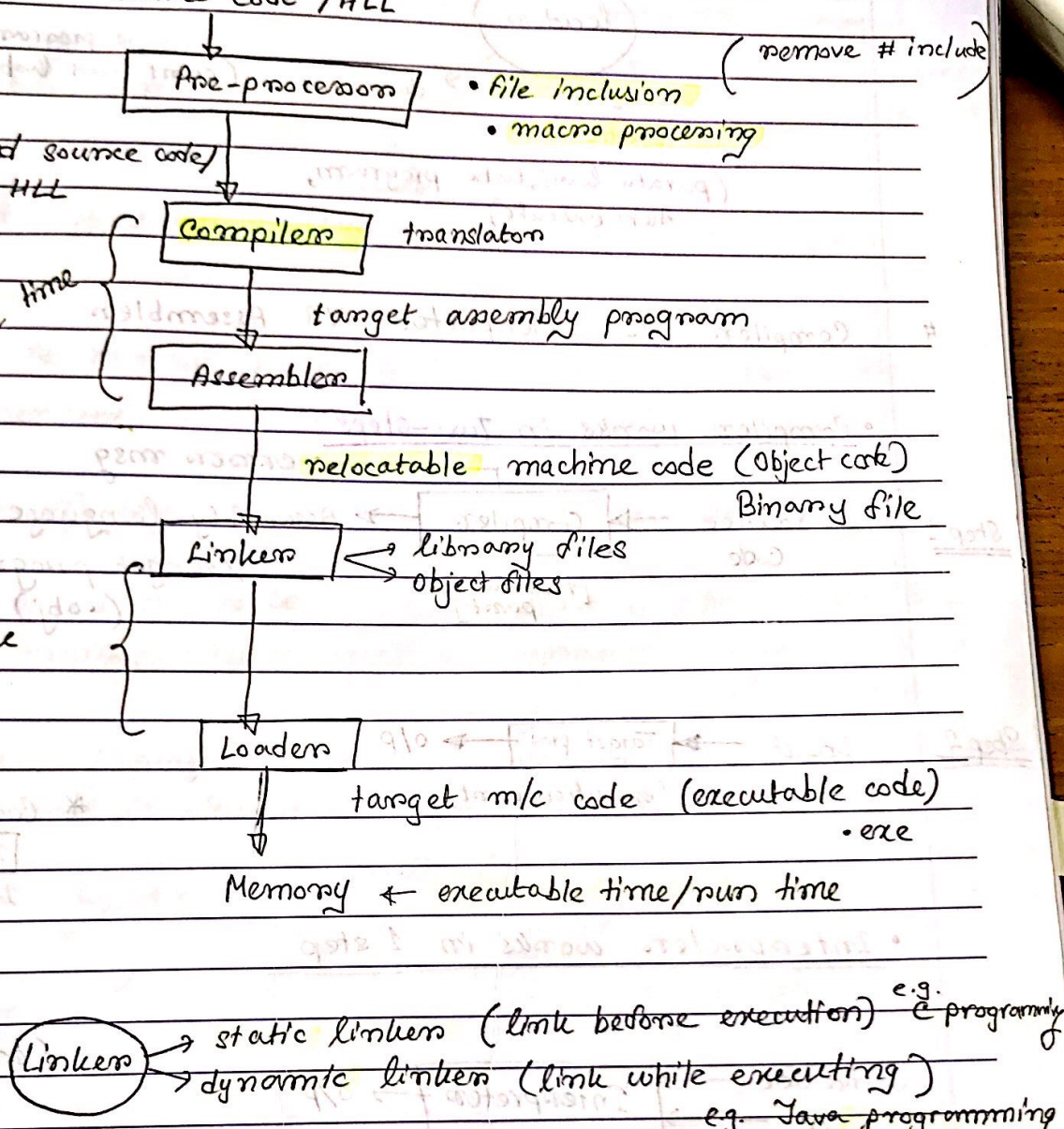
lastpos(n) = 3

nullable node $(a|b)^*$ → ε

nullable : nodes that are root of some sub-expression that generate empty string.

Revision of Lecture 1

Language Processing System

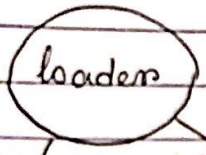


• translation
• execution

Linkers → static linkers (link before execution) e.g. C programming
→ dynamic linkers (link while executing) e.g. Java programming

C program faster than Java, because lower execution time.
Because static linking.

disadv of static linking, translating process slow.



dynamic (part of program or load libe)

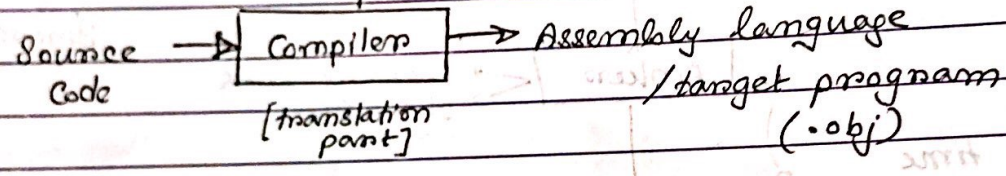
static

(purata load, libe program, then execute)

Compiler VS Interpreter VS Assembler

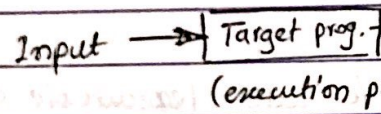
• Compiler works in Two-steps

Step 1



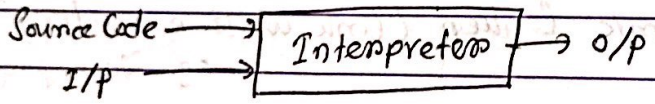
(purata scan)

Step 2



* Compiler is faster than Interpreter.

• Interpreter works in 1 step



(scan line by line)

(line-by-line translate + execute)

Q. Differences between Compiler & Interpreter.

	Compiler	Interpreter
1.	Compiler scans the whole program and translate into machine code	1. Interpreter scans one line at a time and translate into machine code.
2.	Intermediate object is generated	2. Intermediate object code is not generated.
3.	It takes more memory since object code is generated.	3. It takes less memory.
4.	Compiler is faster	4. Interpreter is slower.
5.	Program need to not to be compiled everytime.	5. Every time higher level program is converted into lower level program.
6	Errors are displayed after entire program is checked. For example: C, C++	6. Errors are displayed for every instruction interpreted. So efficient error diagnosis. For example: Basic, Python

assembly language to machine code. The output of assemblers is called object file.

Java is a compiled language.
Python is an interpreted language.

- High-level language – If a program contains #define or #include directives such as #include or #define it is called HLL. They are closer to humans but far from machines.

The (#) tags are called pre-processor directives. They direct the preprocessor about what to do.

- Preprocessor – The preprocessor removes all the #include directives by including the files called file inclusion and all the #define directives using macro expansion.

- Assembly Language – Its neither in binary form nor high level. It is an intermediate state that is a combination of machine instructions and some other useful data needed for execution.

Compilers and Interpreters do similar jobs, but they're different in the way they read input.

~~Compiler reads the inputs in one go,~~

The Compiler in one go reads the inputs, does the processing and executes the source code whereas the interpreter does the same line by line.

Compiler scans the entire program and translates it ~~as~~ as a whole into machine code whereas an interpreter translates the program one statement at a time.

Compilers FASTER than Interpreters.

- Linker loads a variety of object files into a single file to make it executable.

The loader loads it in memory and executes it.

Interpreter is another common kind of language processor.

• frontend is platform-independent

L 8 S 1 | C T

frontend/analysis, backend/synthesis

Instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user.

The machine-language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs.

Interpreter gives better error diagnostics than a compiler, because it executes the source program statement by statement.

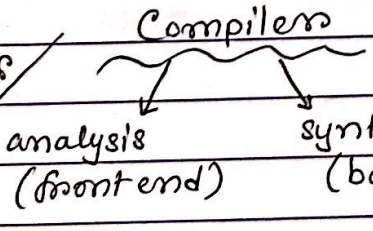
Assembly language is easier to produce as output and is easier to debug.

Compiler produces assembly-language program as its output. The assembly language is then processed by a program called an Assembler that produces relocatable machine code as its output.

Linker resolves external memory addresses, where the code in one file may refer to a location in another file.

Loader puts together all of the executable object files into memory for execution.

Fig: phases of a compiler



• Compilation process operates as a sequence of phases.

LSSI CT

Symbol Table stores info about the entire source program, is used by all phases of the compiler.

Symbol Table helps the compiler to function smoothly by finding the identifiers quickly.

LSSL CT

Phases of a Compiler

Analysis Phase

An intermediate representation is created from the given source code.

1. Lexical Analyzer
2. Syntax Analyzer
3. Semantic Analyzer
4. Intermediate Code Generator

Synthesis Phase

Equivalent target program is created from the intermediate representation.

~~5. Code generator~~
~~6. Code optimizer~~

5. Code Optimizer
6. Code Generator

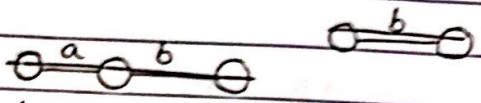
Q. Why does a lexical analyzer use buffer pairs to scan input?

Q. lexical error → initializing a variable name starting with a number

Q. DFA is a finite pattern recognizer

Q. Is string 'abbc' accepted by r.e. "ab|b*c" ⇒ (NO)

Q. Syntax Analyzer deals with the Grammatical structure of a programming language.



- Line 1 and 2 (both)

Q. Compiler phases are broadly classified into analysis & synthesis parts.

Q.
- One buffer cannot hold all input characters, it needs two.
- ~~One buffer has less capacity than the other.~~
- To increase space complexity

Q. C++ does not use an interpreter.

Q. Syntax error → int[] arr = new int[3.5]

Q. Syntax → missing ;

Q. True → $aa^*(b|c) = a^*a(c|b)$

Q. Which of the following is a r.e. of all the binary numbers divisible by 8

Q. $\Sigma\{0,1\}$ is described by r.e. $(0+1)^* 0(0+1)^* 0(0+1)^*$

Set of all strings containing atleast two 0's

Week 3

Lecture 1, Part 1 — August, 2020

331

RE → NFA → DFA

Converting RE to DFA directly

RE → DFA (direct method)

Step 1: RE augmentation

(adding additional symbol hash #)

$r \rightarrow (r)\#$ — this symbol is concatenated with whole r.e.

Step 2: Then, we create a syntax tree for this augmented regular expression.

—

Construction of DFA from RE.

1. **Input:** A regular expression r.
2. **Output:** A DFA D that recognizes $L(r)$. ($L(r)$ — the language defined by r.e. r)

Method:

① Construct Syntax Tree for augmented RE $(r)\#$

② Construct the nullables, firstpos, lastpos and followpos for ST.

③ Construct D states : set of states of D

D trans : transition table for D

Q. Intermediate Code Generator produces machine-dependent intermediate code → FALSE

Q. Interpreter better for error diagnosis.

Q. Preprocessor is used to expand macros into Source Language statements.

Q $R^* = (R^*)^*$ valid

$R^* = (R|\varepsilon)^*$ valid

$R^+ = R + U\varepsilon$ valid

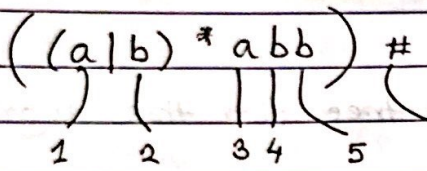
$\varepsilon.R.\varepsilon = \varepsilon$ invalid

Q. Set of all strings over $\Sigma = \{0,1\}$ that begins & ends with 0 is:

$0(0+1)^* 0$

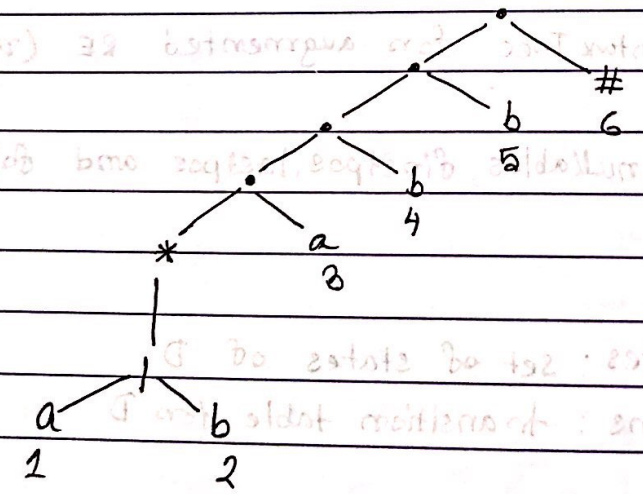
Syntax Tree has two types of nodes.

- Leaves contain: Alphabet symbols or ε, # symbol (non-ε leaf associated with unique position no.)
- Internal nodes contain: Operators. (cat-node, or-node, star-node)



- When constructing Syntax Tree, consider Precedence of the Operators.
- Scan the r.e. from Left to Right
- Parentheses (brackets) define the scope of the Operation.

[* precedence higher than concat]

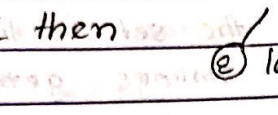


[define the position of each of the non-ε leaf. Don't define position for ε-symbol]

Fig: Syntax Tree for ((a|b)*abb)#

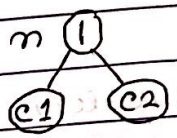
Nullable: nodes that are the root of some sub-expression that generate empty string.

- If n is a leaf labeled by ε then
  nullable(n) = true



- If n is a leaf labeled with position i
  nullable(n) = false

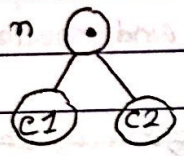
- If n is an or-node (|) with children c1 and c2.



(or)

nullable(n) = nullable(c1) or nullable(c2)

- If n is a cat-node (•) with children c1 and c2.

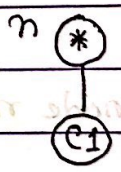


(both)

nullable(n) = nullable(c1) and nullable(c2)
(both)

- If n is a star-node (*) with children c1.

nullable(n) = true



nullable because it will produce the empty string.

Rules for firstpos (See slide) } Lecture 1, Part 2
Rules for lastpos (See slide) }

Summary table

To evaluate followpos, we need 3 more functions to be defined for the nodes (not just for leaves) of the Syntax Tree.

- **firstpos(n)** – the set of the positions of the first symbols of strings generated by the sub-expression rooted by n.
- **lastpos(n)** – the set of the positions of the last symbols of strings generated by the sub-expression rooted by n.
- **nullable(n)** – true if the empty string is a member of strings generated by the sub-expression rooted by n false otherwise.

To determine followpos, we need to determine firstpos, lastpos and nullable.

$(a|b)^* a$

firstpos : 1 or 2 or 3

lastpos : 3

~~w = {aa, ab, ba, aa}~~

$w = \{aa, ba, \epsilon a\}$

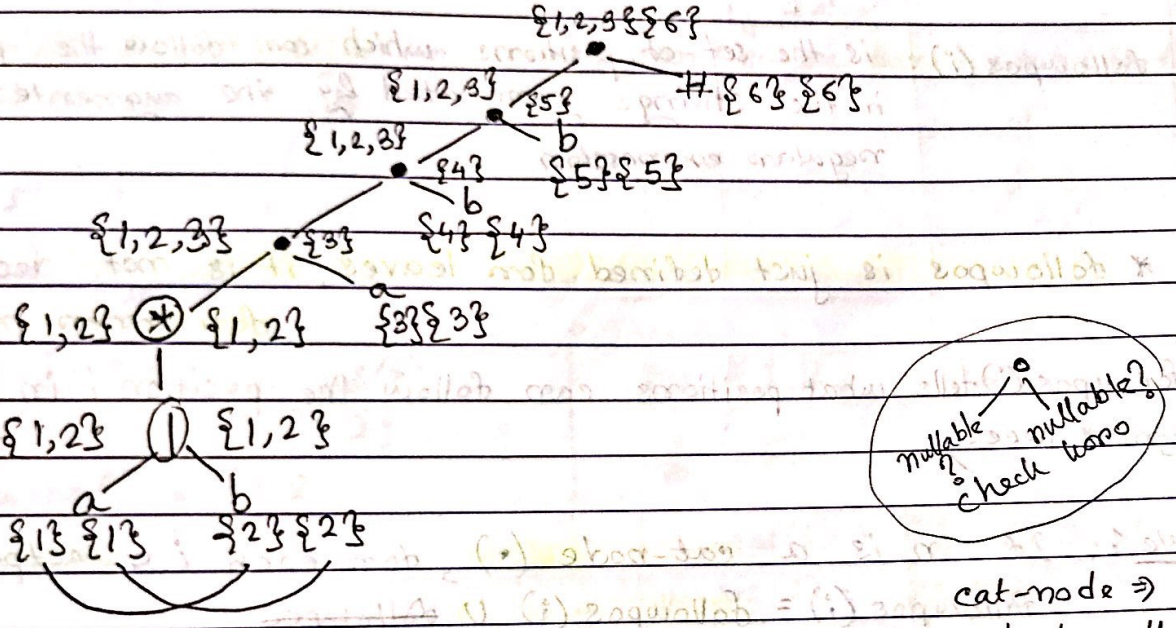
1 2 3

nullable node is the node n that produces empty string.

nullable node : $(a|b)^*$

Firstpos and lastpos example (Lecture 1, Part 3)

- First, find leaves: firstpos() and lastpos()
- Traverse the tree in a bottom-up approach.



cat-node ⇒
check nullability
and
c1, c2

Week 03

August 2020

* Lecture 2, Part 1 *

Lexical Analysis

RE → DFA (direct method)

followpos function returns the possible next positions of the designated alphabet of r.e.

followpos(i): is the set of positions which can follow the position i in the strings generated by the augmented regular expression.

* followpos is just defined for leaves, it is not defined for inner nodes.

(followpos(i) tells what positions can follow the position i in the syntax tree.)

Rule 1: If n is a cat-node (•), for each $i \in lastpos(c1)$

$followpos(i) = followpos(i) \cup firstpos(c2)$

Rule 2 If n is a (*) star-node, for each $i \in lastpos(n)$

$followpos(i) = followpos(i) \cup firstpos(n)$

Followpos only for (•) catnode (*) starnode

• After computing firstpos and lastpos for each node, follow-pos of each position can be computed by making depth-first traversal of the Syntax Tree.

- After constructing ST, determine firstpos and lastpos of leaf nodes and interior nodes.
- We'll determine followpos only for (*) star-node and (·) cat-node.
- Traverse the ST using depth-first traversal technique.
- or-node doesn't define any next position^ so we don't consider this for followpos.

Star node

$i \in \{1,2\}$, firstpos(*) = {1, 2}

followpos(1) = {1, 2}

followpos(2) = {1, 2}

Cat node

$i \in \{1,2\}$, firstpos(c2) = {3}

followpos(1) = {3}

followpos(2) = {3}

so {1, 2, 3}

$i \in \{3\}$, firstpos(b) = {4}

followpos(3) = {4}

$i \in \{4\}$, firstpos(b) = {5}

followpos(4) = {5}

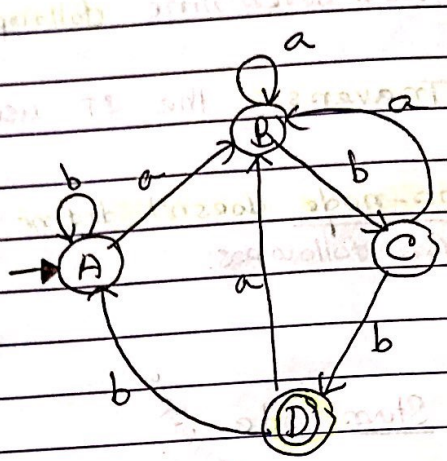
followpos(5) = {6}

use these two to construct DFA
- ST with firstpos and lastpos
- followpos Table

In DFA construction procedure,
we start with firstpos of root.

root:

Dstate	a	b
{1,2,3} = A	{1,2,3,4} = B	{1,2,3} = A
{1,2,3,4} B	{1,3} C	{2,4} D
{2}		{4}



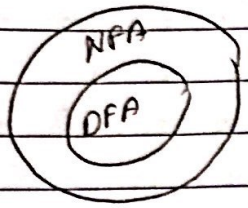
root firstpos

Dstate	a	b
{1,2,3} ≡ A	{1,2,3,4} ≡ B	{1,2,3} ≡ A
B ≡ {1,2,3,4}	{1,2,3,4} ≡ B	{1,2,3,5} ≡ C
C ≡ {1,2,3,5}	{1,2,3,4} ≡ B	{1,2,3,6} ≡ D
D ≡ {1,2,3,6}	{1,2,3,4} ≡ B	{1,2,3} ≡ A

- 6 last state

DFA
recognize pattern generated by RegEx.

Most of the Lexical Analyzers uses DFA simulation for pattern/token recognition.



input → Stream of tokens → Parser → output → Parse Tree / Syntax Tree

operator (internal node)
operand (leaf node)

Lexical error → misspelling of keywords (printf, return)

Syntax error → ",) missing

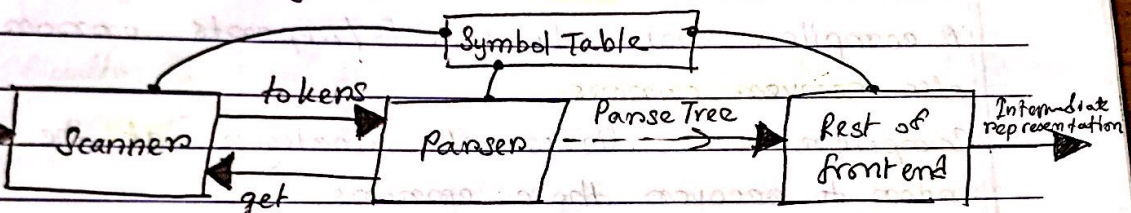
Parsing

Week 4, Lecture - 01
(SRF)

Parsing / Syntax Analysis

When a lexical analyzer processes these lexemes into tokens, these lexemes are then taken as inputs by Parser.
The Syntax Analyser / Parser will apply a grammatical structure on those tokens in order to generate a Syntax Tree. This ST follows a basic CFG.
The ST is passed to the Semantic Analyser which will check for any meaning ful inconsistency within the tree.

Syntax tree is used to guide translation.



Scanner & Parser work simultaneously side by side in order to generate a Parse Tree from a given stream of tokens.

- collecting token info
- type checking
- Intermediate code generation

When a parser takes a stream of tokens as inputs and generates a Parse Tree, it causes the func to get next tokens to the Scanner in order to generate more tokens so as to get more Syntax Trees.

Both Scanner & Parser take help from Symbol Table, by updating and modifying the values and info of variables.

Lexical error - misspelt keyword / typos
Syntactic error - extra hyphen or extra closing bracket.
Semantic error - type errors, undefined IDs.
Logical errors cannot be handled by Compilers, rather these are handled by the Programmers themselves.

Compilers cannot detect Logical Errors.

Most of the errors-detection happens during Syntactic Analysis level. Adequate error reporting is a major task to do. Inadequate recovery may introduce more errors.

A compiler not only detects/reports errors, it's task is to also recover errors.
Compilers uses different strategies at the Syntactic level in order to recover those errors.
One such approach is called Panic Mode.

Panic Mode Recovery Approach discards token until it finds a "synchronizing" token.
When Syntax Analyzer finds one or a set of tokens that doesn't fall into a grammatical structure (CFG), the Syntax Analyzer will keep on deleting those tokens one ~~or~~ by one until it finds a synchronizing token.

Synchronizing token could mean the end of something.
It could mean the end of a class, end of a func, end of a statement and so on

} end ;

Adv: - simple to implement
- does not go into infinite loop
- commonly used method

Disadv: - may skip over large sections of source with some errors.

(During panic-mode recovery, sets of tokens produced by follow can be used as synchronizing tokens.)

2# **Phrase Level Recovery**

works by inserting, deleting, or modifying tokens within a code segment.

Compiler corrects the program by deleting or inserting tokens so it can proceed to parse from where it was.

Adv: does not get into infinite loop
... constantly inserting tokens and never scanning the actual source.

- Generally used for error-repairing compilers.

Disadv: Difficulty: Point of error detection might be much later the point of occurence.

3# **Error Productions**

- Augment the CFG with "Error Productions"; now the CFG accepts anything.

Used with: - LR (Bottom-up parsing)
- Parser Generators.

Error productions are prod. rules that are written within a CFG

4# **Global Correction**

Theoretical Approach (too time consuming)

It takes grammatically incorrect input string and a given grammar G, and make min^m/least ~~modificita~~ modifications to x in order to convert it into correct string y such that the string y is acceptable by grammar G.

CFG

• A CFG is a formal model consisting of 4 units

Context Free Grammar

- **Terminals**
  - keywords
  - token classes
  - Punctuation
- **Non-terminals**
  - Any symbol appearing on LHS of any rule
- **Start symbol**
  - non-terminal on the LHS of the first rule
- **Rules / Productions**
  - BNF (Backus-Normal form)
  - Stmt ::= if Expn then Stmt els Stmt

4-tuples (V) (T) (P) (S)

- (V) → Variable / non-terminal (capital letter)
- (T) → Terminal (small letter)
- (P) → Productions
- (S) → Start variable

CFG for ~~S ⇒ 01 / 0S1~~

S ⇒ 01	0S1
LHS	RHS

} Production / Rule

Variable, V = {S}

Terminal, T = {0, 1}

S = Start variable

Q. Identify V, T, P, S

① $E \Rightarrow E+T \mid T$

$T \Rightarrow T*F \mid F$

$F \Rightarrow (E) \mid id$

V = { E, T, F }

T = { +, *, (,), id }

S = E

A CFG is a set of rules / productions used to generate patterns of strings.

(2)

$S \Rightarrow (L) \mid a$

$L \Rightarrow L,S \mid S$

$V = \{S, L\}$

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

$S = S$

Equivalence of CFGs: If two CFG's generate the same language, we say they are "equivalent"

$G_1 \approx G_2$ whenever

$$L(G_1) = L(G_2)$$

In a derivation,

for Leftmost Derivation (L to R) → always expand the leftmost non-terminal

for Rightmost Derivation (R to L) → always expand the rightmost non-terminal / variable

Parse Tree is another representation of Left Leftmost and Rightmost Derivations. Both derivations will have the same Parse Tree until and unless the grammar is ambiguous.

(The parse tree only remembers: which rule to use in replacing it) (See slide)

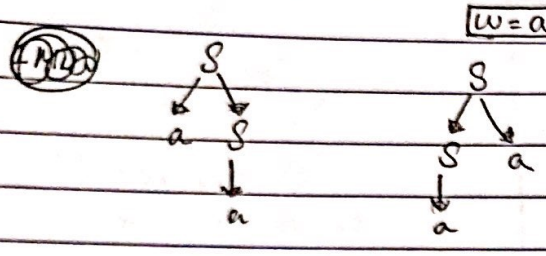
Please Note : ↓ x | ✓

w = string
G = grammar

Q. Find whether the given grammar is Ambiguous or not

① $S \rightarrow aS / Sa / a$

$w = aa$



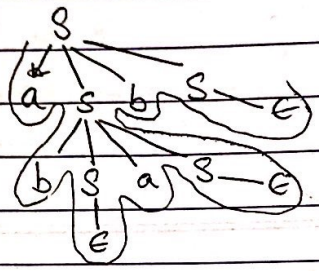
Two Parse Trees, so Ambiguous.

(for same G and w)

② $S \rightarrow aSbS \mid bSaS \mid \epsilon$

$w = aabb$

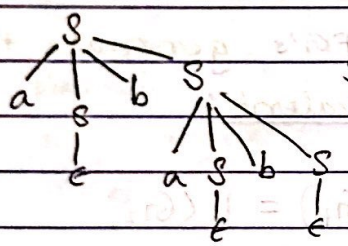
$w = abab$



$ab \in a \in b \in$

$= abab$

First S expand



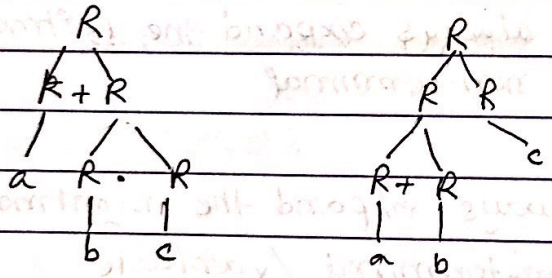
Second S expand

abab

So, Ambiguous.

③ $R \rightarrow R+R \mid RR \mid R^* \mid a \mid b \mid c$

$w = a+bc$



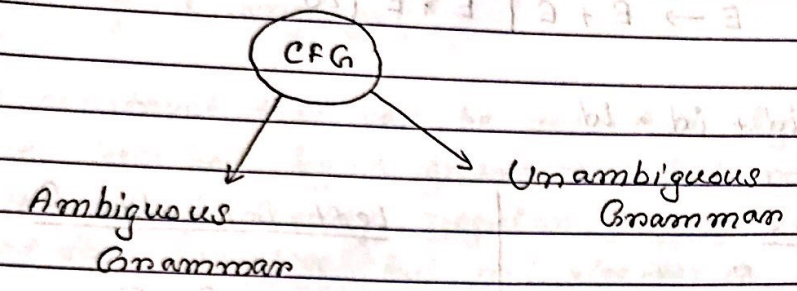
So, Ambiguous Grammar

(Parser always matches terminals from left to right.)

Ambiguous Grammar
(TNR)

Week 4, Lecture 2
7th August, 2020

CFG is divided into two parts based on the number of derived trees.



A grammar can have more than one parse tree generating a given string of terminals.
Such a grammar is said to be ambiguous.

Ambiguous Grammars produce either one or more or all of the below:

- more than one Parse Tree
- more than one Derivation Tree
- more than one Syntax Tree
- more than one Leftmost Derivation
- more than one Rightmost Derivation.

Most of the parsers, except one parser, will not allow ambiguous grammars.

(derivation means we're to generate a string consisting of only ~~non terminals~~ of terminals from the body of the Production Rules)

A sequence of terminals and nonterminals in a derivation is called "Sentential forms" #

Example

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

Sentence: $id + id * id$

Left Derivation 1	Left Derivation 2
$E \rightarrow E + E$	$E \rightarrow E * E$
$\rightarrow id + E$	$\rightarrow E + E * E$
$\rightarrow id + E * E$	$\rightarrow id + E * E$
$\rightarrow id + id * E$	$\rightarrow id + id * E$
$\rightarrow id + id * id$	$\rightarrow id + id * id$

Two different L.D. while generating the same sentence of terminals. So by definition, this grammar is Ambiguous.

Q. How to remove ambiguity in a grammar so as to make it Unambiguous Grammar.

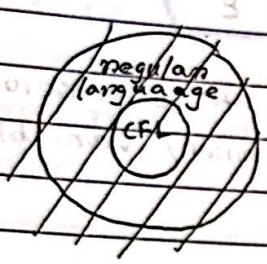
[Notice: all these lines that you generate in / get in the way of generating the string of terminals is known as Sentential Forms of the derivation process.]

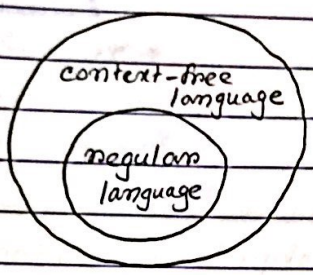
$(id * E)$

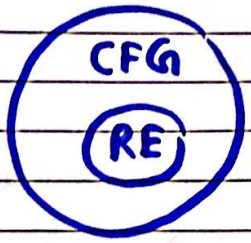
Left Recursion and Left Factoring

Q. What is the difference between CFG and Regular Grammar? When and why do we use CFG instead of R.G. when we're implementing a language?

Every construct that can be described by a Regular Expression can be described by a grammar, but not vice-versa.
Alternatively, every regular language is a context-free language, but not vice-versa.







Finite automata cannot count the no. of a's preceded before b. F.A. lacks the ability of counting or memorizing, therefore, we cannot implement such languages using F.A. In these cases, we need the concepts of CFGs to implement them.

$$L = \{a^n b^n \mid n \geq 1\}$$

Left Recursive → Why is it bad? → Infinite loop

A grammar is left recursive if it can be represented in the form: $A \Rightarrow^{+} A\alpha$

Why is this a problem for top-down parsing?

✓ Left Recursive is bad for Top-Down Parsing. (let the parsers go into an infinite loop, hence Left Recursive is bad.)

Left Recursive
Any variable A in any of its sentential form is calling itself to the left of the sentential form produces such a context.

Important

Converting left recursive grammar into right recursive one.

Left Recursion Removal

Simplest Case: "Immediate Left Recursion"

Given: $A \rightarrow A\alpha \mid \beta$

Transform into:

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

where A' is a new nonterminal / variable

Left and Right Recursive Grammar

Left Recursive

$A \rightarrow A\alpha \mid \beta$

- Causes an infinite loop.

* So we have to eliminate a left-recursive grammar. We've to change Left Recursive grammar into its Right Recursive form.

Right Recursive

$A \rightarrow \beta R$

$R \rightarrow \alpha R \mid \epsilon$

- This never causes an infinite loop because Parser always checks the inputs from Left to Right.

Left Recursion Elimination (example)

Grammar

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

(Step 1)	$E \rightarrow TE'$ $E' \rightarrow +TE' \mid \epsilon$	
(Step 2)	$T \rightarrow FT'$ $T' \rightarrow *FT' \mid \epsilon$	
	$F \rightarrow (E) \mid id$	as it is

Left Recursion in more than one step.

- At first, eliminate non-immediate left recursion for prod. rules
- Then, we've to eliminate the immediate left recursion for the same prod. rule.
  Do this for each and every prod. rule.

Example 1

$S \rightarrow Af \mid b$ — left recursive → Yes, indirect
$A \rightarrow Ac \mid Sd \mid e$ — left recursive → Yes, direct

↓

$S \rightarrow Af \mid b$
$A \rightarrow Ac \mid e \mid Afd \mid bd$

$A \rightarrow e \mid bd \mid Afd \mid Ac$

↓

$S \rightarrow Af \mid b$

$A \rightarrow eA' \mid bdA'$
$A' \rightarrow fdA' \mid cA' \mid \varepsilon$

#

Example 2

$S \rightarrow Af \mid b$
$A \rightarrow Ac \mid Sd \mid Be$ — indirect left recursive
$B \rightarrow Ag \mid Sh \mid k$ — direct left recursive; indirect left recursive

↓

$S \rightarrow Af \mid b$
$A \rightarrow Ac \mid Be \mid Afd \mid bd$
$B \rightarrow Ag \mid k \mid Afh \mid bh$

↓

$S \rightarrow Af \mid b$
$A \rightarrow BeA' \mid bdA'$

$A' \rightarrow cA' \mid BfdA' \mid \varepsilon$
$B \rightarrow Ag \mid k \mid Afh \mid bh$

↓

$B \rightarrow BeA'g \mid BeA'fh \mid k \mid bh \mid bdA'g \mid$ ~~bdA'd~~ $bdA'fh$

↓

$B \rightarrow k \mid bh \mid BeA'g \mid BeA'fh \mid bdA'g \mid bdA'fh$

↓

$B \rightarrow kB' \mid bhB' \mid bdA'gB' \mid bdA'fhB'$

$B' \rightarrow \varepsilon \mid eA'gB' \mid BeA'fhB'$

Finally,

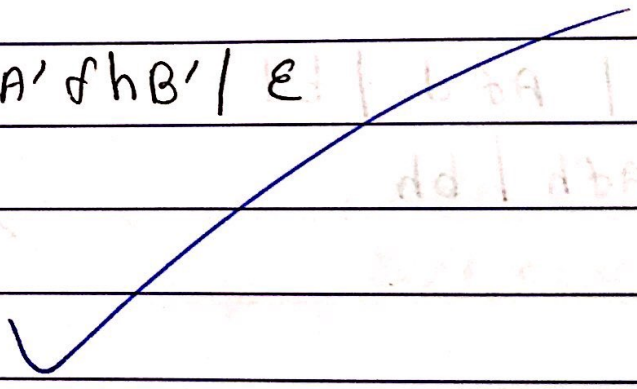
$S \rightarrow Af \mid b$

$A \rightarrow Be\,A' \mid bd\,A'$

$A' \rightarrow cA' \mid fd\,A' \mid \varepsilon$

$B \rightarrow k\,B' \mid bh\,B' \mid bd\,A'g\,B' \mid bd\,A'fh\,B'$

$B' \rightarrow eA'g\,B' \mid e\,A'fhB' \mid \varepsilon$



Important

Left Factoring → to eliminate nondeterminism

Week 05, Lecture 02

- problem of limited lookaheads
- problem of generating same CFGs on doing the same operations twice or thrice (this takes more memory/time for Parser)

How do we avoid these problems?
We factor the common terms out.

Factoring of common terms → Left Factoring

* So to eliminate the problem of generating the same grammar string multiple times, take a common grammar string and factor it out.

Given Grammar: $S \rightarrow iEtS \mid iEtSeS \mid a$
$E \rightarrow b$

Left factored equivalent of the Grammar:

$S \rightarrow iEtSS' \mid a$
$S' \rightarrow eS \mid \epsilon$
$E \rightarrow b$

Left Factoring saves our time and improves our efficiency.

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

Left factoring removes non determinism in grammar.

(non-deterministic grammars → does same symbol many options)
(common prefixes causes bad backtracking)

Q. Do left factoring on the following grammar.

$S \rightarrow aEbS \mid aEbSe \mid x$

$E \rightarrow y$

Ans:

$S \rightarrow aEbSX' \mid x$

$X' \rightarrow e \mid \varepsilon$

$E \rightarrow y$

Important

Top Down Parsing

Week 05, Lecture 03

- Recursive Descent Parsing is a form of Top Down Parsing.
- Computing First function → used in Predictive Parsing

Top Down Parsing

- Find a left-most derivation
- Build a parse tree
- Start building from the root and work down
- As we search for a derivation
  (must make choices: • which rule to use • where to use it)

Two types of **Top-Down Parsing**

Recursive Descent Parsing

- backtracking is needed
  (if a choice of a production rule does not work, we backtrack to try other alternatives.)
- It's a general parsing technique, but not widely used.
- Not efficient

Predictive Parsing

- no backtracking
- efficient
- needs a special form of grammars (LL(1) grammars)

* RDP takes a lot of time as a lot of backtracking involved. And the prod. rules we're choosing on the basis of priority. So, very inefficient approach.

Recursive Descent Parsing (Backtracking)

$S \rightarrow Aa \mid Ce$

$A \rightarrow aaB \mid aaba$

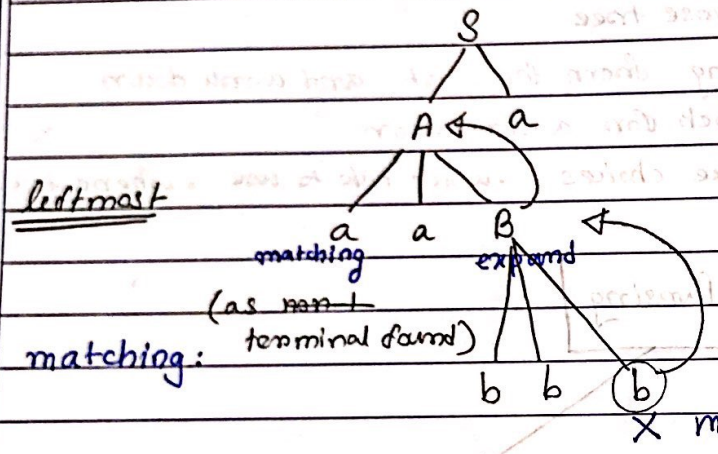
$B \rightarrow bbb$

$C \rightarrow aaD$

$D \rightarrow bbd$

Input: aa bbde

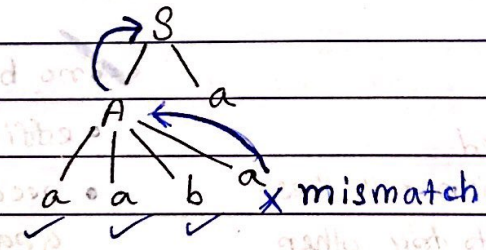
(priority - written or quoted before)

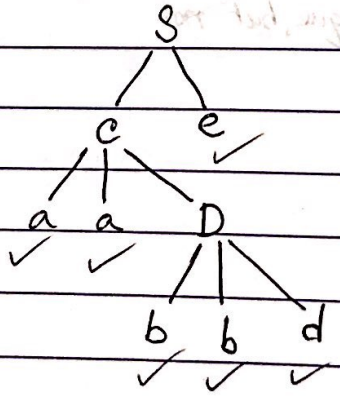


leftmost

matching:

backtrack to see other option





Parse Tree

* First Func is needed to be used ~~in/done in~~ in order to do Predictive Parsing

First Function

Find the first terminal

1. $S \rightarrow aB$
   $B \rightarrow d$

   combo: @d

   FIRST(S) = {a}
   FIRST(B) = {d} ✓

   FIRST(α) = {a}

2. $S \rightarrow AB$
   $A \rightarrow c$
   $B \rightarrow d$

   combo: ⓒd

   FIRST(S) = {c}
   FIRST(A) = {c}
   FIRST(B) = {d} ✓

   FIRST(α) = {c}

3. $S \rightarrow AB$
   $A \rightarrow c \mid \varepsilon$
   $B \rightarrow d$

   combo: cd , εd
   = @d , ⓓ

   FIRST(S) = {c, d}
   FIRST(A) = {c, ε}
   FIRST(B) = {d} ✓

   FIRST(α) = {c, d}

4. $S \rightarrow AB$
   $A \rightarrow c \mid \varepsilon$
   $B \rightarrow d \mid \varepsilon$

   combo: cd , cε, εd, εε
   = @d , ⓒ, ⓓ, ⓔ

   FIRST(α) = {c, d, ε}

   FIRST(A) = {c, ε}
   FIRST(B) = {d, ε}
   FIRST(S) = {c, d, ε} ✓

6. $S \rightarrow aSe \mid STS$

$T \rightarrow RSe \mid Q$

$R \rightarrow rSr \mid \varepsilon$

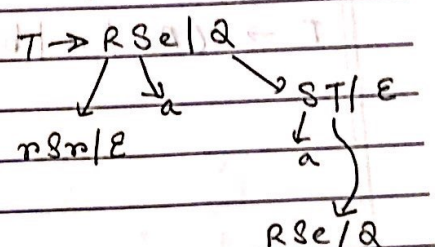
$Q \rightarrow ST \mid \varepsilon$

Ans: FIRST(S) = {a} ✓

FIRST(T) = {r, a, ε} ✓

FIRST(R) = {r, ε} ✓

FIRST(Q) = {a, ε} ✓



combo: rSrSe, εSa, aT, ε

= r, εSa, a, ε

= r, Sa, a, ε

= r, a, ε

$R \rightarrow rSr \mid \varepsilon$

combo: r, ε

$Q \rightarrow ST \mid \varepsilon$

↓

a

combo: aT, ε

= a, ε

5. $P \rightarrow i \mid c \mid nTS$

$Q \rightarrow P \mid aS \mid bSeST$

$R \rightarrow b \mid \varepsilon$

$S \rightarrow c \mid Rn \mid \varepsilon$

$T \rightarrow RSq$

Ans: $FIRST(P) = \{i, c, n\}$ ✓

$FIRST(Q) = \{i, c, n, a, b\}$ ✓

$FIRST(R) = \{b, \varepsilon\}$ ✓

~~$FIRST(S) = \{c, b, \varepsilon\}$~~

~~$FIRST(T) = b, \varepsilon$~~

$S \rightarrow c \mid Rn \mid \varepsilon$

$\swarrow$

~~$b \mid \varepsilon$~~

combo: $c, bn, \varepsilon n, \varepsilon$

$= c, bn, n, \varepsilon$

$= c, b, n, \varepsilon$

* $FIRST(S) = \{c, b, n, \varepsilon\}$ ✓

$T \rightarrow RSq$

$\swarrow \quad \searrow$

$b \mid \varepsilon \qquad c \mid Rn \mid \varepsilon$

$\swarrow \searrow$

$b \quad \varepsilon$

~~$FIRST(R)$~~

* $FIRST(T) = \{b, c, n, q\}$ ✓

combo:

$bcq, bbnq, bEnq,$

$bEq, Ecq, Ebnq,$

$EEnq, EEq$

$\Rightarrow b, b, b, b, c, b,$

n, q

7.

$S \rightarrow aSe \mid B$
$B \rightarrow bBeCf \mid C$
$C \rightarrow cCg \mid d \mid \varepsilon$

FIRST (S) = {a, b, c, d, ε}
FIRST (B) = {b, c, d, ε}
FIRST (C) = {c, d, ε} ✓

8.

$S \rightarrow (A) \mid \varepsilon$
$A \rightarrow TE$
$E \rightarrow \&TE \mid \varepsilon$
$T \rightarrow (A) \mid a \mid b \mid c$

FIRST (S) = {(, ε} ✓
FIRST (A) = {(, a, b, c} ✓
FIRST (E) = {&, ε} ✓
FIRST (T) = {(, a, b, c} ✓

TNR Practice for LR(1)

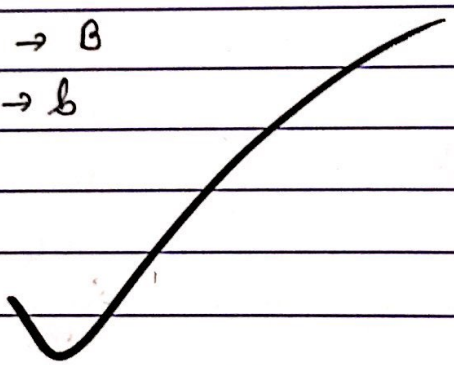
Q. Given the following grammar,

$S \to A$
$S \to \alpha b$
$A \to a A b$
$A \to B$
$B \to b$

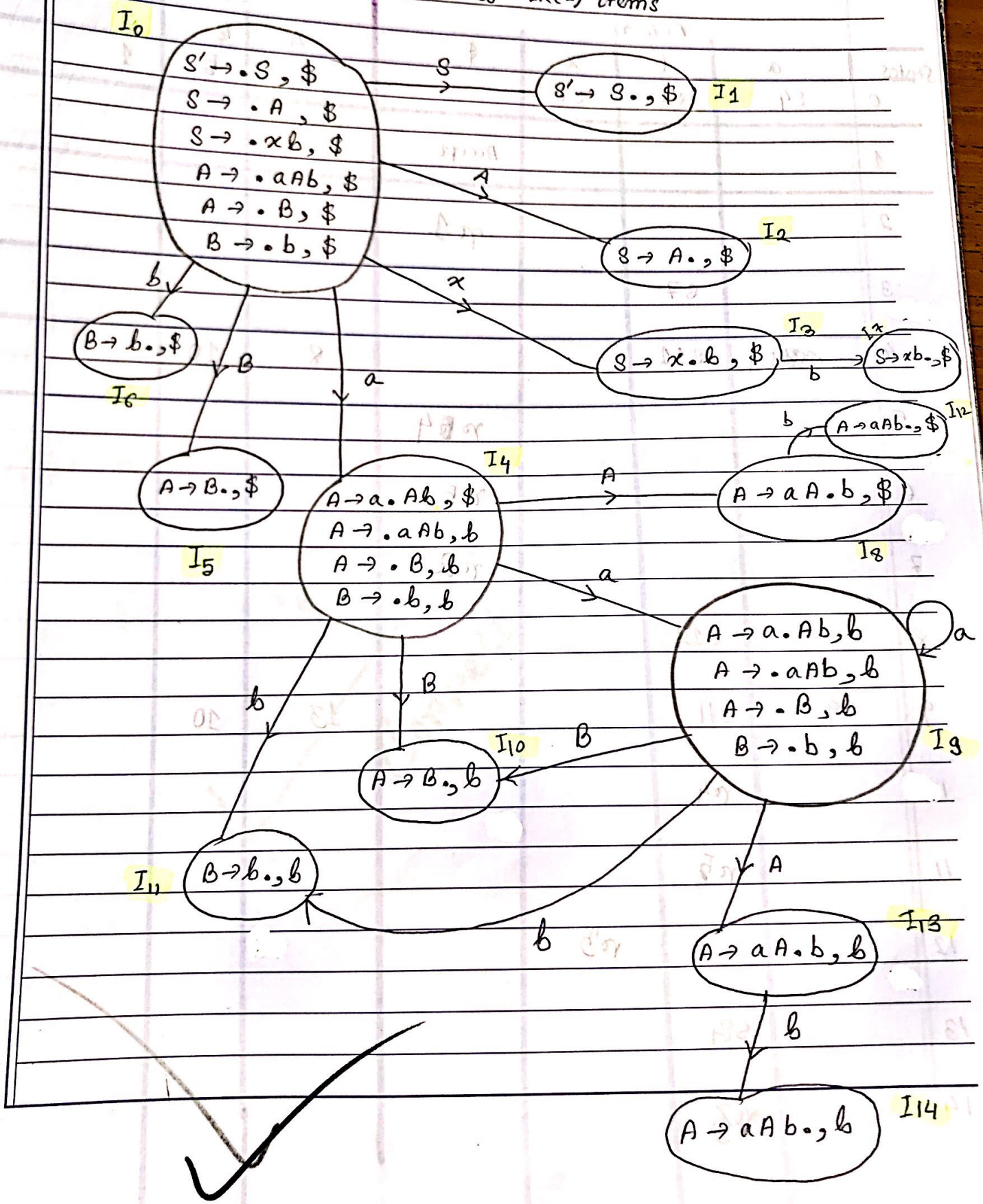
Compute the LR(1) items and construct the Parsing Table.

Ans: Step 1: Augment the Grammar

⓪ $S' \to S$
① $S \to A$
② $S \to \alpha b$
③ $A \to a A b$
④ $A \to B$
⑤ $B \to b$



Step 2: Canonical collection of LR(1) items



States	Action				GoTo		
	a	b	x	$	A	B	S
0	S4	S6	S3		2	5	1
1				Accept			
2				r1			
3		S7					
4	S9	S11			8	10	
5				r4			
6				r5			
7				r2			
8		S12					
9	S9	S11			13	10	
10		r4					
11		r5					
12			r3				
13		S14					
14		r3					

final states,

$I_2 \Rightarrow r1 \rightarrow \$$

$I_5 \Rightarrow r4 \rightarrow \$$

$I_6 \Rightarrow r5 \rightarrow \$$

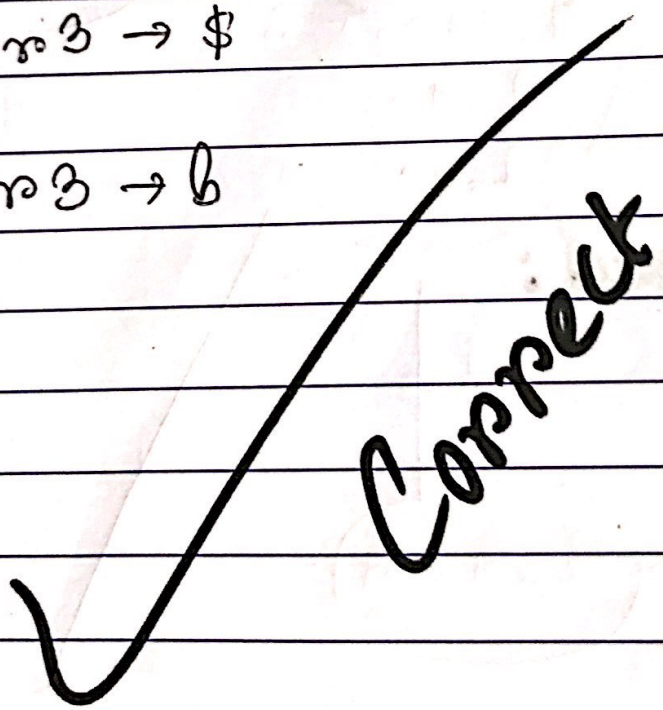
$I_7 \Rightarrow r2 \rightarrow \$$

$I_{10} \Rightarrow r4 \rightarrow b$

$I_{11} \Rightarrow r5 \rightarrow b$

$I_{12} \Rightarrow r3 \rightarrow \$$

$I_{14} \Rightarrow r3 \rightarrow b$



Ans 1:

(i)

	FIRST	FOLLOW
S	{a, b}	{$}
A	{a, ε}	{a, b}
B	{b, ε}	{a, b}

(ii)

	a	b	$
S	S → AaAb	S → BbBa	
A	A → a A → ε	A → ε	
B	B → ε	B → b B → ε	

(ii) No, it is not LL(1)

Since there are multiple entries in a single cell

Also, since grammar is Ambiguous

it is not LL(1)

(iv) Follow set of a variable can never contain an epsilon because,

Since, ~~error~~ epsilon transition is for ε-NFA and not DFA.

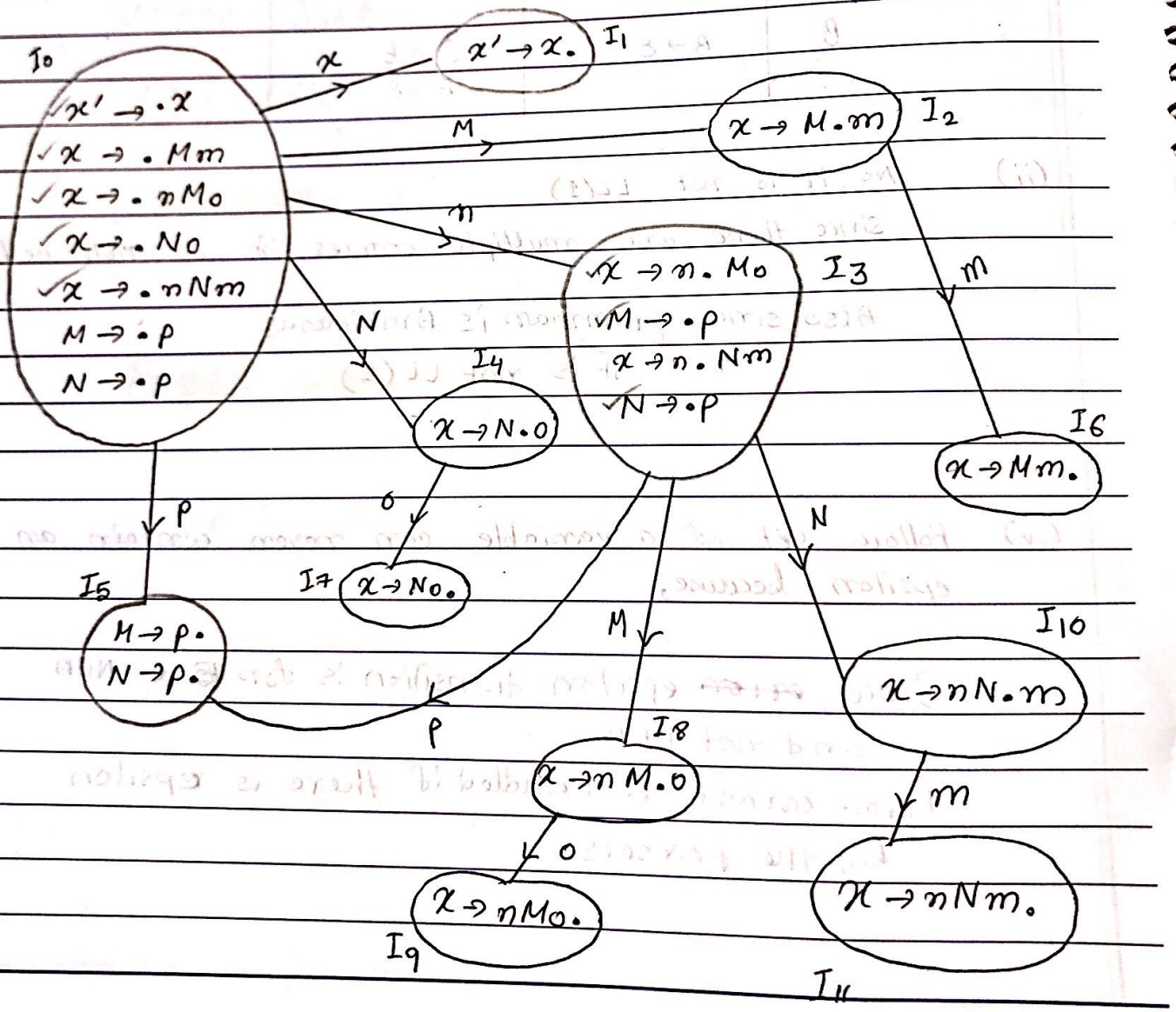
Error cannot be handled if there is epsilon by the parsers.

Ans: 8

Step 1: Augmented Grammar

$x' \to x$
$x \to Mm$
$x \to nMo$
$x \to No$
$x \to nNm$
$M \to p$
$N \to p$

Step 2: Canonical collection of LR(0) items



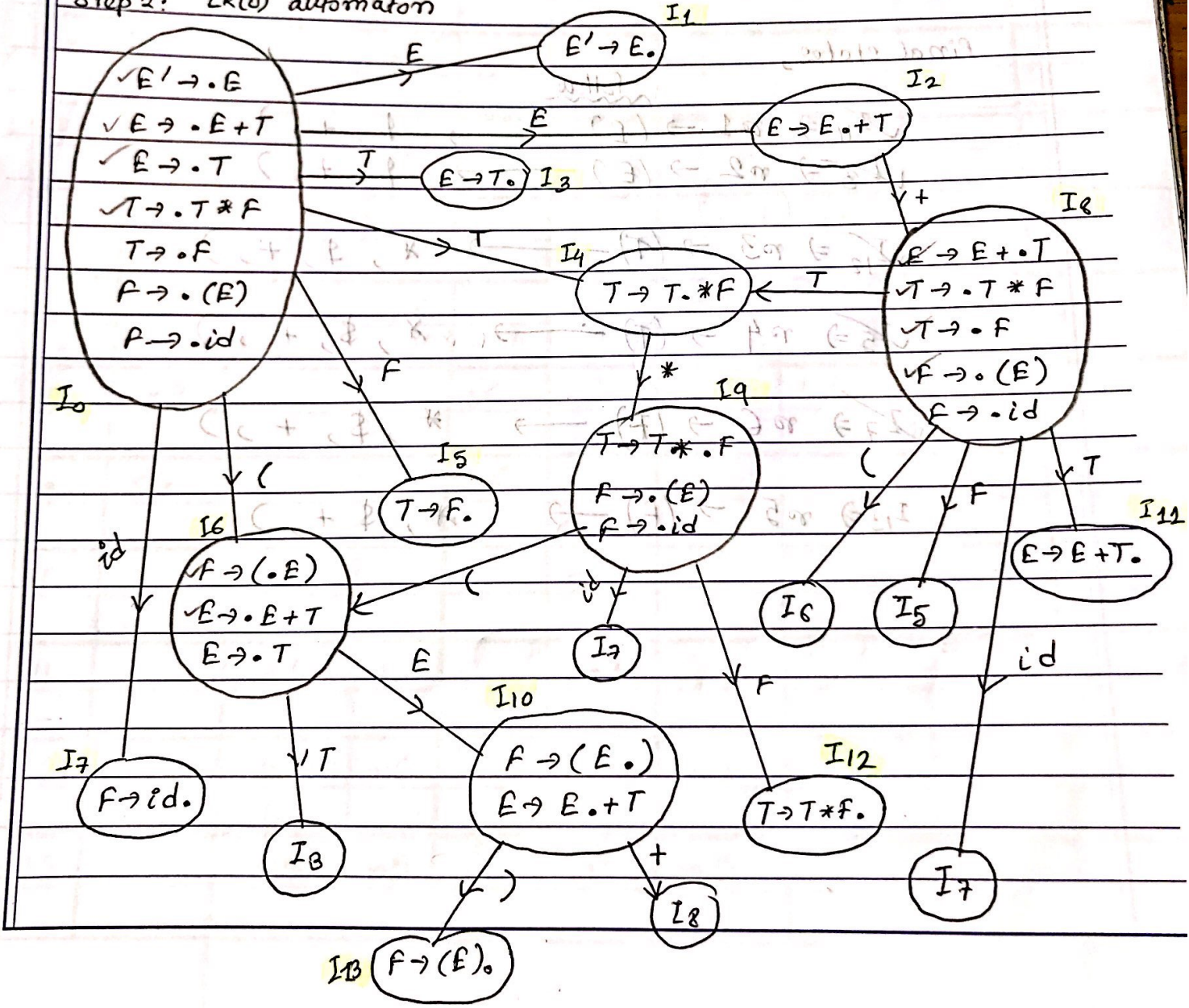
Ans 2:

Step 1: Augment Grammar

Pg-

$E' \to E$ ⓪
$E \to E+T$ ①
$E \to T$ ②
$T \to T*F$ ③
$T \to F$ ④
$F \to (E)$ ⑤
$F \to id$ ⑥

Step 2: LR(0) automaton



Variables

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

Final states,

Follow

$I_{11} \Rightarrow$ r1 → (E) ⟶ $, +,)

$I_3 \Rightarrow$ r2 → (E) ⟶ $, +,)

$I_{12} \Rightarrow$ r3 → (T) ⟶ *, $, +,)

$I_5 \Rightarrow$ r4 → (T) ⟶ *, $, +,)

$I_7 \Rightarrow$ r6 → (F) ⟶ *, $, +,)

$I_{13} \Rightarrow$ r5 → (F) ⟶ *, $ +,)

Subhi Bhuiyan
Pg-

	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s7			s6			1,2	3,4	5
1						accept			
2		s8							
3		r2			r2	r2			
4			s9						
5		r4	r4		r4	r4			
6							10	3	
7		r6	r6		r6	r6			
8	s7			s6				4,11	5
9	s7			s6					12
10		s8			s13				
11		r1			r1	r1			
12		r3	r3		r3	r3			
13		r5	r5		r5	r5			

Subhi

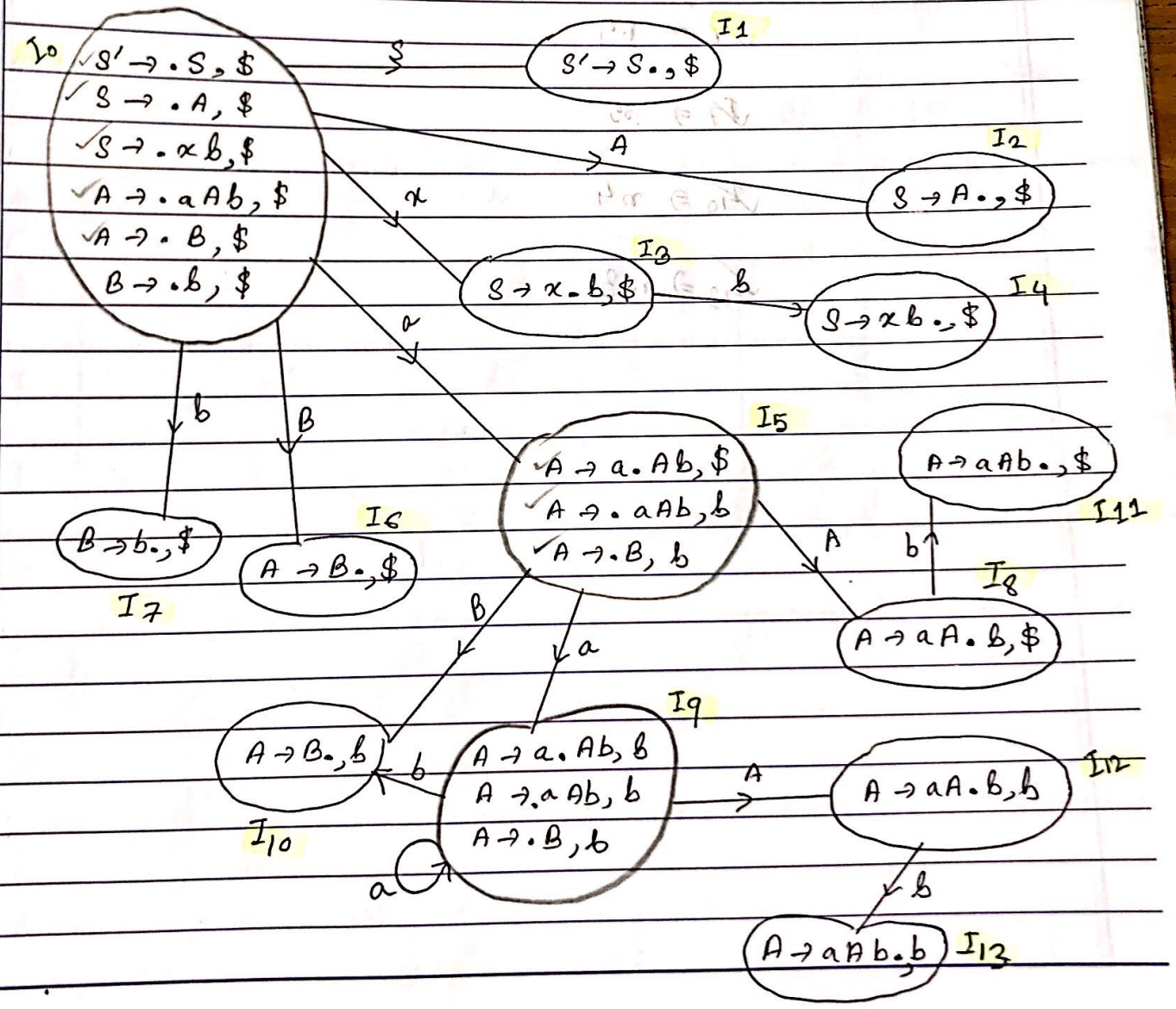
Pg-

Ans 4:

a) Step 1: Augment Grammar

0. $S' \to S$
1. $S \to A$
2. $S \to xb$
3. $A \to aAb$
4. $A \to B$
5. $B \to b$

Step 2: LR(1) items (canonical collection)



Final states,

$I_2 \Rightarrow r1$

$I_4 \Rightarrow r2$

$I_{11} \Rightarrow r3$

$I_6 \Rightarrow r4$

$I_7 \Rightarrow r5$

$I_{10} \Rightarrow r4$

$I_{13} \Rightarrow r3$

CLR(1) Parse Table

Subhi Bhriyan

Pg-

	Action				Go To		
	a	b	x	$	S	A	B
0	s5	s7	s3		1	2	6
1				accept			
2				r1			
3		s4					
4				r2			
5	s9					8	10
6				r4			
7				r5			
8		s11					
9	s9	s10				12	
10		r4					
11				r3			
12		s13					
13		r3					

Subhl Bhuiyan

Pg-

b)

input: a a b b b $

Stack

0	9	5	a	9	b	10	A	12	b	13	A	8	B	W	A	2	S	1

(0, a) ⇒ ~~s5~~

(5, a) ⇒ s9 ✓

(9, b) ⇒ s10 ✓

(10, b) ⇒ r4 (A) → |B|

L = 1

pop = 2L = 2

push

(A, 9) ⇒ 12

(12, b) ⇒ s13 ✓

(13, b) ⇒ r3 (A) → |aAb|

L = 3

pop = 6

push

(8, b) ⇒ s11 ✓

(5, A) → 8

(11, $) ⇒ r3 L = 3 pop = 6 push A

(A, 0) ⇒ 2

(2, $) → r1 S → A|

L = 1

pop = 2

(S, 0) ⇒ 1

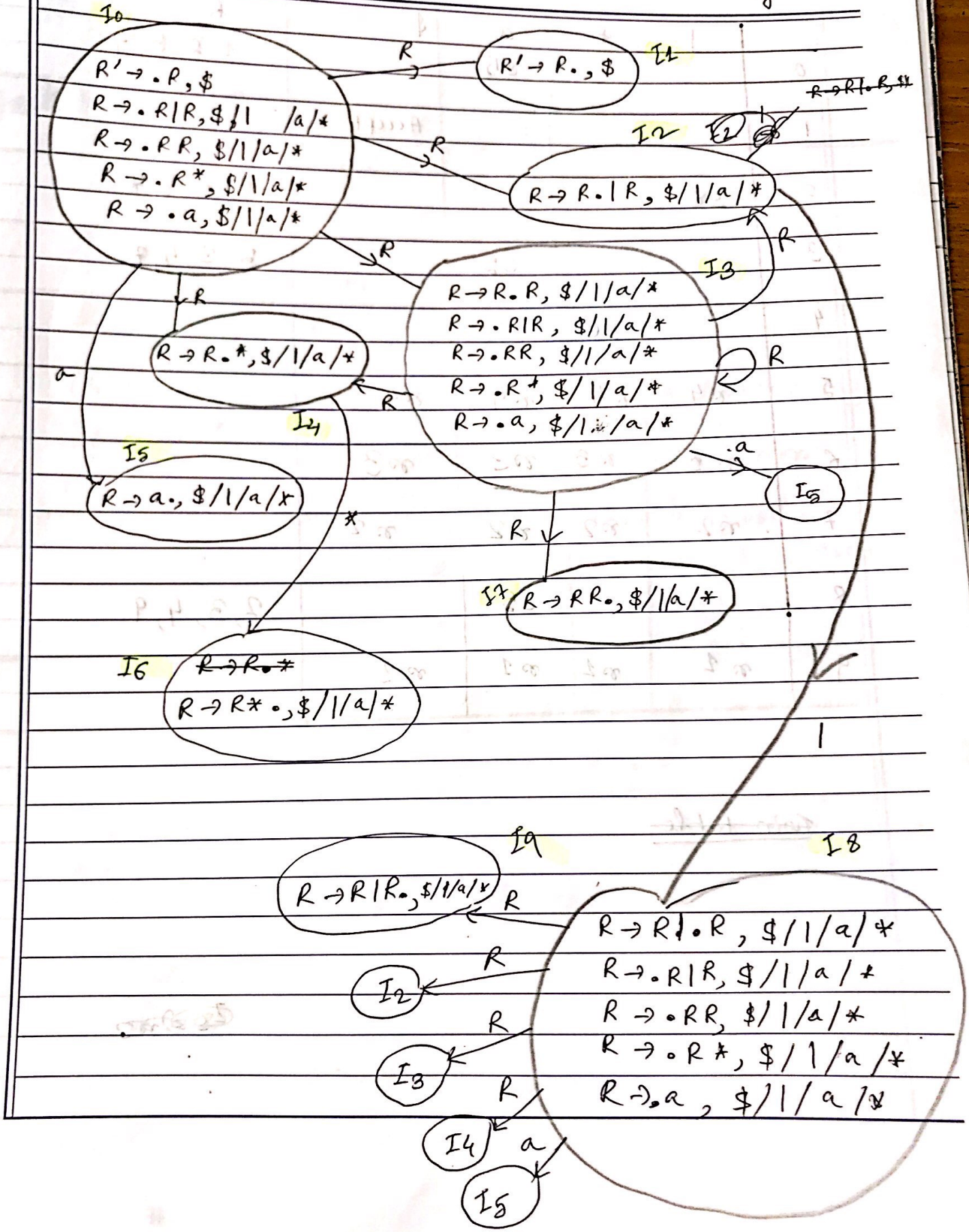
S

(1, $) ⇒ accept

Therefore, the string can be parsed.

Ans:-

Subhi Bhuiyan
Pg-



CLR(1) Parser for Q5

		*	a	$	R
0			s5		1, 2, 3, 4
1				Accept	
2	s8				~~2, 8, 4, 7~~
3			s5		2, 3, 4, 7
4		s6			
5	r4	r4	r4	r4	
6	r3	r3	r3	r3	
7	r2	r2	r2	r2	
8			s5		2, 3, 4, 9
9	r1	r1	r1	r1	

~~Twin table~~

~~L → R~~

RE: a*a | a$

Stack | 0 | a | 5 | R | 1 | 2 | 3 | 4 | * | 5 | R |

(0, a) ⇒ s5

(5, *) ⇒ r4 R → |a|
L=1 pop=2

(R, 0) ⇒ 1, 2, 3, 4

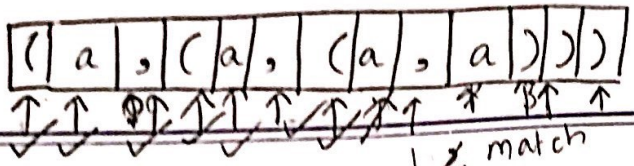
(4, *) ⇒ s5

(6, a) ⇒ r3 R → |R*| L=2 pop=4

(R, 2) ⇒ blank entry

So, Syntax error

Hence, the given RE cannot be parsed



Output	
	match
	match
$P \to (E)$	
$E \to PE'$	
$P \to a$	match
$E' \to ,PE'$	
$P \to (E)$	
$E \to PE'$	match
$P \to a$	
$E' \to ,PE'$	
$P \to (E)$	match
$E \to PE'$	
$P \to a$	
$E' \to ,PE'$	match
$P \to a$	
$E' \to \varepsilon$	
$E' \to \varepsilon$	match
$E' \to \varepsilon$	
	match
	match
	ε
	match

CSE420 Compilers Design
javatpoint, tutorialspoint, geeksforgeeks

Compilers Design

Week 0

- Theories of Compilers Design used in NLP, Machine Translation, ~~Compilers~~ Information Retrieval, Semantic Web Technology, Query Language Processing.

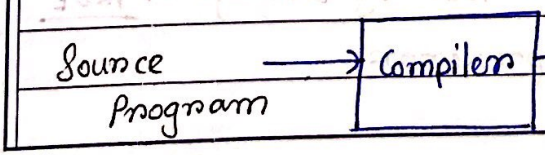
- procedure for identifying syntax errors.

"Compiler Basics"

Compiler is a translator that converts the high-level language into the machine language.
Compiler is used to show errors to the programmer.

A compiler is a computer program that translates computer code written in one programming language (the source language) into another language. (the target language.)

- Programming problems are easier to solve in high-level languages. (HLL)
- Solutions are usually more efficient (faster, smaller) when written in Machine Language.
- Compilers are the bridges → connects the software with hardware; Tools to translate programs written in HLL to efficient executable code.
  Another important role of the compiler is to report any errors in the source program that it detects during the translation process. (compiler errors)

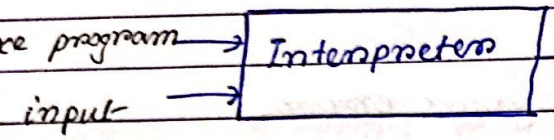


(Compiler is a software program)

(diff from a compiler)

Q. What is an interpreter?

Ans: An interpreter requires no pre-computation it directly takes the source program and runs the input on it producing the desired output. (line by line / instruction by instruction)



Q. Difference bet^n Interpreter and Compilers.
↳ slower

* The interpreter takes less memory (no exe file needed)
* The interpreter is invoked each time the CPU has to run any piece of instruction (as no files .exe generated)

Compilers & interpreters do similar jobs, but there are differences:

To run a program you've written, it must first be translated into machine code so the computer can read it. This is what compilers and interpreters do.

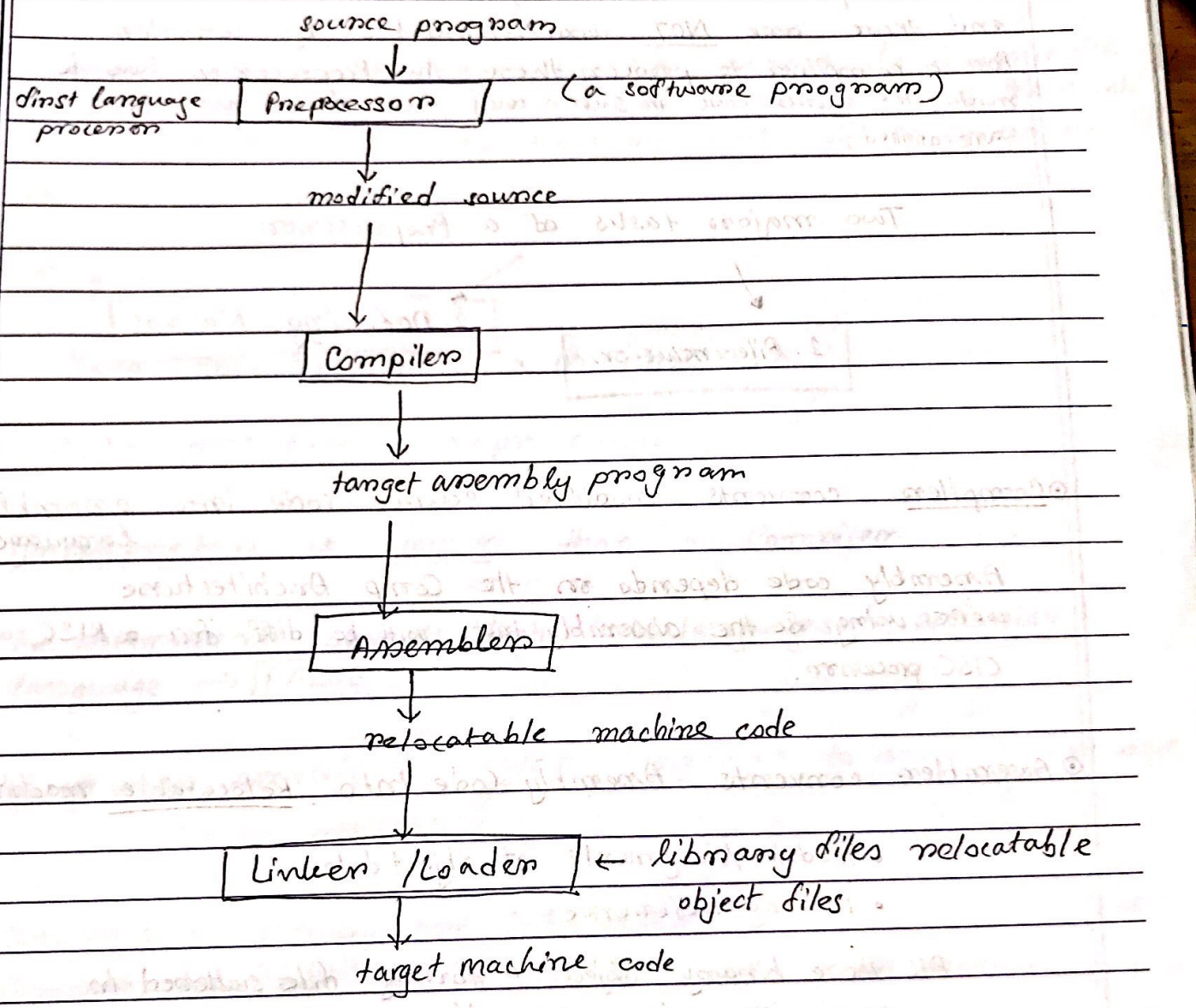
However, Compilers convert the code all at once, save it, then run it. ↳ (C, C++) Java

Whereas, interpreters translate the code one line at a time, as it is run. ↳ (Python, Ruby) JavaScript

* Interpreter is slower
* The interpreter converts HLL to Assembly Language ⇒ FALSE
* ~~The interpreter~~
* The compiler converts HLL to Assembly Language ⇒ TRUE
* Java does not have a preprocessor.

- A Language Processing System is a system that actually converts the source program written in one language into a target machine code consisting of 1s and 0s by incorporating different language processors such as preprocessors, compilers, assemblers, linkers and loaders.

Q Steps of a Language Processing System.



(asks the Preprocessor to include the contents of standard io. header file into our program)

⊙ instructions preceded by a hash symbol are Preprocessor directives / Preprocessor instructions and these are NOT understandable by Compiler. For a compiler to process these, the Preprocessor has to modify the source code in such a way that these instructions are omitted.

Two major tasks of a Preprocessor

1. File inclusion
2. Defining Macros

⊙ Compiler converts modified source code into assembly language.

Assembly code depends on the Comp. Architecture we're using. So the assembly inst. will be diff for a RISC and CISC processor.

⊙ Assembler converts Assembly code into Relocatable machine code.

- modules / fragments of object files
- library reference

All these binary object & library files scattered in hard disk, after Assembler converted A. code to M. code

⊙ Linker is a software program that links all these library & object files into one single separate executable file. keeping in mind the order of instructions.

(Linker links relocatable object files into executable file)

Sumary: { * Compilers
* Compilers Vs Interpreters
* Steps of LP System.

@Loaders loads executable machine code into main memory for execution.

Loaders will take the executable file produced by Linkers in the hard disk and it will load them into the memory by mapping them into memory addresses, after that those segments will be loaded for execution

Quiz - 01

1. Java does not have a preprocessor.

   C, C++ does have a preprocessor

2. Interpreter is slower than a Compiler

3. The interpreter converts higher level languages to assembly language → FALSE

4. The compiler converts higher-level languages to assembly language.
   ↳ TRUE

Compiler runs faster than Interpreter.
Compilers take less time to execute than interpreters because the run time is slower.
Interpreter doesn't have any compile time as the code never compiles, it directly runs.

"Phases of Compiler"

July 12th, 2020

Week 1, Lecture - 2

Phases of a Compiler (LSSICT)

- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- Intermediate code generation
- Code optimization
- Code generator

built-in keyword → (lexeme / meaningful sequence of characters)

Lexical Analyzer/scanner will find lexime (variable, no., operator) and generate tokens for the same.
This token stream fed as input into Syntax Analyser/Parser. which will generate a syntax tree basing on a grammatical structure. or CFG
Semantic Analyzer checks for inconsistency in meaning. (array size can never be declared as a double)

ICG → (3 address code generated called Intermediate Representation)

(Machine-independent code optimizers)

MICO's task → to reduce the no. of lines or to reduce the no. of variables in 3-address code to decrease Compilation Time and increase Efficiency.

Code Generator → generates Assembly Instructions which is hardware-specific.

(Machine-dependent code optimizers)

MDCO → to reduce no. of lines + variables in assembly code. (optional)